July 1998, Volume 4, Number 7

Cover Art By: *Tom McKeith*

Delphi 4 Hits the Road

Distributed Programming Goes RAD

Extended Systems Announces Advantage Database Server 5.0

Extended Systems, Inc. announced *Advantage Database Server 5.0*, a scalable DBMS that brings client/server benefits to mobile, network, and Internet database applications.

Enhancements to Advantage Database Server 5.0 include Read Ahead Record Caching, which reduces network traffic by reading future records and server-based optimized (bitmapped) filters. In addition, the composite protocols feature combines certain protocols that previously occurred back-to-back.

Developers currently using the Advantage Database Server can use the snap-in features to instantly see increased performance without any application re-writes or changes to existing hardware. The Advantage Management APIs allow developers to incorporate server management functionality into current applications. A separate Management Utility that can be run from anywhere on the network is provided.

Advantage client libraries (sold separately as Advantage Client Kits) can replace existing database drivers with fully compatible Advantage drivers. Advantage Client Kits feature a native component solution for Delphi 1, 2, and 3 and C++Builder.

Extended Systems, Inc.
Price: From US$249 for two-user version to US$7,495 for 1,000-user version. Advantage Client Kit prices range from US$99 to US$299.
Phone: (800) 235-7576 ext. 5030
Web Site: http://www.advantage-database.com

Wise Solutions Announces Wise Installation System 6.0

Wise Solutions, Inc. announced the availability of *Wise Installation System 6.0*, a distribution/deployment tool for Windows-based applications. Wise Installation System provides the ability to create a single file that will install one or many applications at the same time. Version 6.0 includes functionality that reduces the time needed to create installations and distribute them to end-user desktops. Version 6.0 also includes updates to the Installation Expert that includes Wise's Click-and-Script technology. It also offers FoxPro, ODBC 3.0, and BDE 4.51 support in its run-time options.

Wise Installation System 6.0 is available as a stand-alone product or as part of the Wise Installation System Enterprise Edition, which includes SmartPatch, WebDeploy, and SetupCapture technologies. SmartPatch updates software applications by creating an installation that only includes changes from the previous version of the software. WebDeploy allows users to install applications on their desktops with a single click from an Internet/intranet site. SetupCapture allows users to automate installations by creating a Wise version of an existing installation.

In addition, Enterprise Edition includes an integrated debugger for analyzing complicated installations.

Wise Solutions, Inc.
Price: Stand-alone version, US$299; Enterprise Edition, US$699.
Phone: (800) 554-8565
Web Site: http://www.wisesolutions.com

Moss Micro Announces ActiveSales 3.0

Moss Micro, Inc. announced *ActiveSales 3.0*, a closed-loop sales and marketing information system for sales professionals, sales managers, administrators, and developers.

Sales professionals can share contacts, activities, notes, product interest, and quotes with sales team members to optimize sales cycles and close opportunities. They can access sales and marketing information from Office 97 applications, and create written customer correspondence, electronic communications, and information presentations.

Sales managers using ActiveSales 3.0 can reduce development and administrative costs of large system implementations, identify the indicators of leading sales professionals, and use management-level views and drill-down capabilities of customizable, online charts and graphs that display real-time information. Examples are sales cycle analysis, revenue analysis, territory analysis, and timeline opportunities.

Administrators and developers can reduce development time with business components that can be written in Delphi, Visual C++, Visual J++, Java, or Visual Basic. They can also accelerate and streamline initial application roll-out, and accelerate administrative tasks with components that maintain critical information, including product, price list, competitor, user, exchange rate, language, territory, and security information.

Moss Micro, Inc.
Price: ActiveSales client, US$2,500; ActiveSales server, US$25,000.
Phone: (800) 608-6585
Web Site: http://www.mossmicro.com

Delphi TOOLS

New Products and Solutions

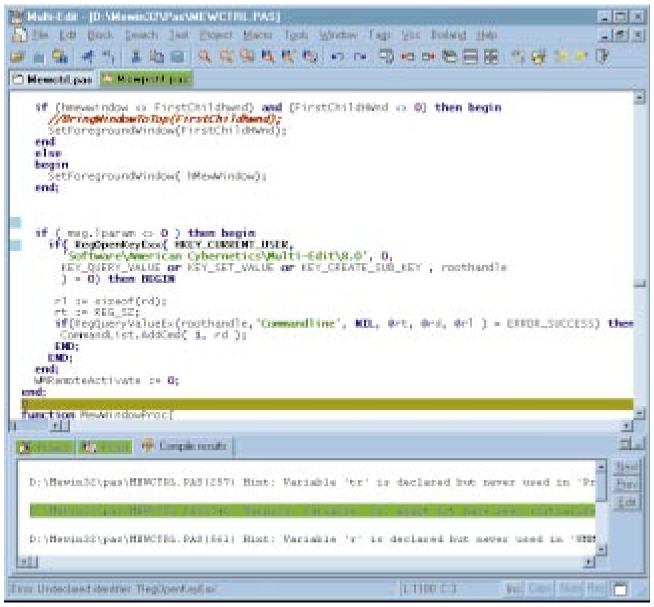
American Cybernetics Announces Multi-Edit 8

American Cybernetics, Inc. announced *Multi-Edit 8*, the company's programming text editor. Multi-Edit 8 features 32-bit performance, an improved interface, tabbed access to editing windows, and a Results window that keeps dialogs and results accessible in a tabbed pane.

Internet programming features in Multi-Edit 8 support HTML with embedded scripts, such as JavaScript and VBScript. It also supports PERL, Java, and other Internet languages. Multi-Edit 8 offers project and site management tools, which include built-in FTP functions.

Developers can keep Multi-Edit and their IDEs in sync with IDE Integration for Delphi, C++Builder, and Watcom C/C++. Programmers can switch between Multi-Edit and their IDE, with both environments reflecting editing changes.

American Cybernetics, Inc.
Price: US$129 for new users; US$69 to upgrade.
Phone: (800) 899-0100
Web Site: http://www.multiedit.com


Multi-Edit - [D:\Mewin32\Pas\MEWCTRL.PAS]
File Edit Block Search Text Project Macro Tools Window Tags Vcs Borland Help
Mewctrl.pas
Mewprint.pas
Compile results

```
  if (hmewindow <> FirstChildhwnd) and (FirstChildHwnd <> 0) then begin
    //BringWindowToTop(FirstChildHwnd);
    SetForegroundWindow(FirstChildHWnd);
  end
  else
  begin
    SetForegroundWindow( hMewWindow);
  end;


  if ( msg.lparam <> 0 ) then begin
    if( RegOpenKeyEx( HKEY_CURRENT_USER,
      'Software\American Cybernetics\Multi-Edit\8.0', 0,
      KEY_QUERY_VALUE or KEY_SET_VALUE or KEY_CREATE_SUB_KEY , roothandle
      ) = 0) then BEGIN

     rl := sizeof(rd);
     rt := REG_SZ;
     if(RegQueryValueEx(roothandle,'Commandline', NIL, @rt, @rd, @rl ) = ERROR_SUCCESS) then
      CommandList.AddCmd( 1, rd );
     END;
    END;
  end;
  WMRemoteActivate := 0;
end;

function MewWindowProc(
```

```
D:\Mewin32\pas\MEWCTRL.PAS(257) Hint: Variable 'tr' is declared but never used in 'Pr
[illegible]
D:\Mewin32\pas\MEWCTRL.PAS(561) Hint: Variable 'r' is declared but never used in 'HSH
```

```
Error: Undeclared identifier 'RegOpenKeyEx'     L:1100 C:3
```

ZieglerSoft Releases ZieglerCollection one

ZieglerSoft announced the availability of *ZieglerCollection one*, a collection of Delphi 16/32-bit components.

ZieglerCollection one offers functions, procedures, and components not provided in Delphi. The collection includes TzMinMax, TzBigLabel, Tz3DLabel, TzAngleLabel, TzTabListBox, TzBlendPaint, TzTileMap, TzLed, TzSegmentClock, TzGauge, TzSlideBar, TzFrame, TzMovePanel, TzTitleBar, TzHint, TzVerSpilt, TzHorSplit, TzMouseSpot, TzCalc, TzColorBtn, TzPanelMeter, and many more.

ZieglerCollection one comes with source code and is available for all versions of Delphi and version 1 of C++Builder.

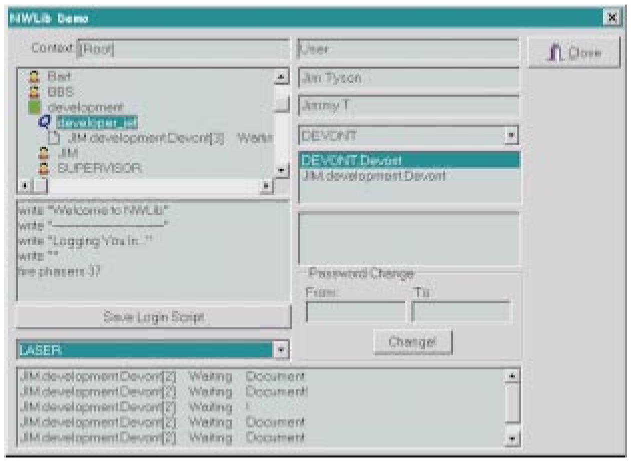
ZieglerSoft
Price: US$52 for e-mail version; US$73 for diskette version.
Phone: +45 9811 3772
Web Site: http://www.zieglersoft.com

Devont Ships NWLib

Devont Software, Inc. shipped *NWLib*, a collection of Netware-aware visual controls for Delphi and C++Builder. NWLib allows developers to drag and drop visual controls to integrate Netware services into their applications.

In addition to the visual controls that integrate Netware facilities, such as user lists, print queue and job management, NDS object browsing, and Bindery object management, NWLib offers over 200 functions that turn complex operations into a single line of code. The entire Netware API is covered, from obtaining user connection information to creating NDS objects and extending the NDS Schema.

Devont Software, Inc.
Price: Standard Edition, US$95; Enterprise Edition, US$225.
Web Site: http://www.devont.com



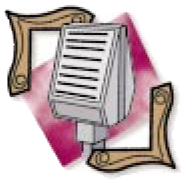
Bart
BBS
development
developer_jet
JIM.development.Devont[3]   Waiti
JIM
SUPERVISOR

User
Jim Tyson
Jimmy T
DEVONT
DEVONT.Devont
JIM.development.Devont

```
write "Welcome to NWLib"
write "————————————"
write "Logging You In..."
write ""
fire phasers 3?
```

LASER

```
JIM.development.Devont[2]   Waiting   Document
JIM.development.Devont[2]   Waiting   Document!
JIM.development.Devont[2]   Waiting   !
JIM.development.Devont[2]   Waiting   Document
JIM.development.Devont[2]   Waiting   Document
```



Borland Becomes INPRISE Corporation

San Francisco, CA — Reflecting its growing focus on enterprise computing, Borland International, Inc. announced a new corporate name, INPRISE Corp. INPRISE combines Borland's heritage of development environments with object component technology to help customers develop, deploy, and manage distributed enterprise applications. INPRISE plans to continue to use the Borland brand name in association with its family of application development tools.

Several factors, including growth and profitability in enterprise distributed computing markets; integration of Visigenic Software, Inc.; growing partnerships with system integrators, such as Arthur Andersen, Cambridge Technology Partners, Cap Gemini, Compuware, EDS, and others; and business relationships with enterprise IT suppliers, such as IBM, Sun Microsystems, SAP, Oracle, Hitachi, Netscape, Novell, and Microsoft, were elemental to the shift in the company's focus and direction.

INPRISE unveiled elements of the company's strategy for delivering end-to-end solutions, including the Inprise Application Server, designed to provide a complete solution for simplifying the development, deployment, and management of middle-tier business logic in a distributed application environment.

INPRISE also outlined plans for providing middleware bridges that allow applications based on existing distributed architectures to interact with the Inprise Application Server.

Other elements of INPRISE's strategy include strengthened sales organization and the expansion of technical support, training, and consulting services operations worldwide.

For more information, visit http://www.inprise.com.

Ensemble Systems Links Rational Rose to Delphi and JBuilder

Richmond, BC, Canada — Ensemble Systems, Inc. announced that Delphi and JBuilder developers can integrate object modeling and round-trip engineering with Rational Rose, using Ensemble's Rose Delphi Link (RDL) and Rose JBuilder Link (RJBL). Ensemble cooperated with INPRISE and Rational Software Corp. as a technology partner under the Rational Rose Link Program to develop RDL and RJBL.

Using Rational Rose 98, Delphi and JBuilder developers can develop object-oriented software design, then generate the code framework for the system using RDL or RJBL. Developers can then complete the implementation with Delphi or JBuilder.

For partially or fully developed systems, RDL and RJBL's round-trip engineering capabilities allow developers to access the existing code to produce an up-to-date model of the system in Rose.

Trial versions of Rose Delphi Link and Rose JBuilder Link are available from Ensemble's Web site at http://www.ensemsys.com.

INPRISE Introduces AppCenter

Scotts Valley, CA — INPRISE announced AppCenter, an enterprise application management tool that enables corporations to model, monitor, and manage their distributed applications. AppCenter will initially support applications built with Entera, INPRISE's RPC-based middleware, with support for CORBA and DCOM applications planned for later this year.

AppCenter allows corporate development and operations teams to deploy and manage large-scale, mission-critical applications. It features a Java-based user interface, which provides the capabilities to create and configure new applications, as well as manage and monitor existing ones.

AppCenter supports the IBM AIX, Sun Solaris, HP-UX, and Windows NT platforms.

For more information, visit http://www.inprise.com/appcenter/.

INPRISE to Expand Research and Development Activities in Singapore

Singapore — INPRISE announced plans to expand its research and development activities in Singapore. The new initiatives, supported through a grant from Singapore's National Science and Technology Board (NSTB), will focus on the development of localized versions of INPRISE's enterprise products, as well as versions of INPRISE's intelligent middleware for additional computing platforms.

The NSTB is the driving force behind Singapore's effort to advance long-term economic growth. Established by the Singapore government in 1991, the NSTB is a statutory board under the Ministry of Trade and Industry.

VisiBroker Surpasses 30 Million Licenses

INPRISE announced that deployed licenses for its VisiBroker ORB (Object Request Broker) surpassed 30 million in 1997. Industry partnerships with Oracle, Hitachi, Novell, Netscape, Informix, Silicon Graphics, and others have been attributed with VisiBroker's success. INPRISE's partners use VisiBroker for Java and VisiBroker for C++ to provide support for CORBA (Common Object Request Broker Architecture) and CORBA's Internet Inter-ORB Protocol (IIOP) in their product lines.

table of CONTENTS

By Cary Jensen, Ph.D.

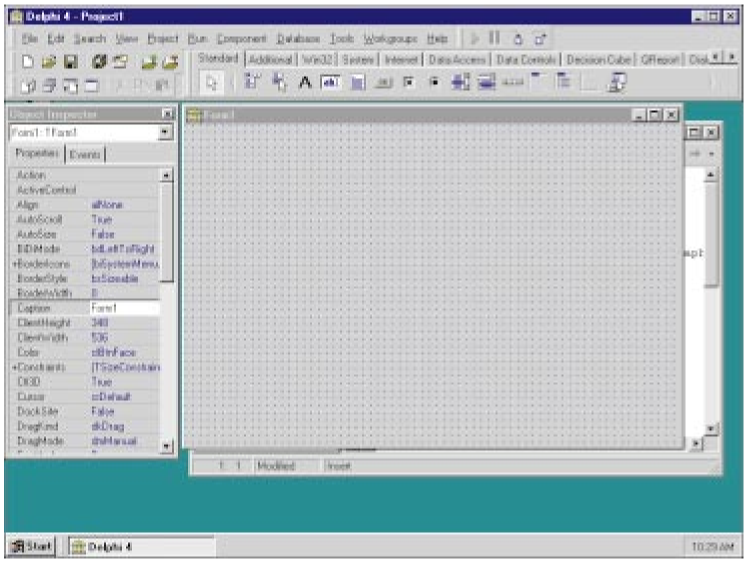
Delphi 4

The New Version Focuses on Ease-of-Development and Distributed Computing

Each new version of Delphi has presented developers with an impressive host of new features and enhancements. Delphi 4 is no exception. In this latest version, INPRISE (*nee* Borland) has managed to add something for everyone. From the bread-and-butter application developer, to the component designer, the new features of Delphi 4 make this a "must have" upgrade.

In a very real sense, every aspect of Delphi has been touched. The most obvious changes are those to the development interface, including the new look of the main window and Code Editor. Other, not so apparent changes, permeate the entire product, from additions to the *TObject* class (the base class for all objects in Object Pascal), to major enhancements to the Object Pascal language.

This first look at Delphi 4 is intended as an overview of the new and improved features of this latest version. Keep in mind, however, that this article is based on a pre-release version of this product. Consequently, there is always a possibility that a feature described here may not ultimately make it into Delphi 4, or may not work as described. On a similar note, not all of these features will appear in all versions of Delphi 4. Specifically, MIDAS (Multi-tier Development Application Services) enhancements will be found only in Client/Server and Enterprise editions, while other features, such as the IDE (Integrated Development Environment) facelifts, are destined for all versions.


Delphi 4 - Project1
File Edit Search View Project Run Component Database Tools Workgroups Help
Standard Additional Win32 System Internet Data Access Data Controls Decision Cube QReport
Object Inspector
Form1: TForm1
Properties Events
Form1
1: 1 Modified Insert
Start Delphi 4
10:29 AM

```
Action
ActiveControl
Align          alNone
AutoScroll     True
AutoSize       False
BiDiMode       bdLeftToRight
+BorderIcons   [biSystemMenu,
BorderStyle    bsSizeable
BorderWidth    0
Caption        Form1
ClientHeight   348
ClientWidth    536
Color          clBtnFace
+Constraints   [TSizeConstrain
Ctl3D          True
Cursor         crDefault
DockSite       False
DragKind       dkDrag
DragMode       dmManual
```

Figure 1: Delphi 4 sports a new-look IDE that includes tear-away toolbars.

IDE Changes

Upon loading Delphi 4 for the first time, there's no question you're using an updated product. As shown in Figure 1, the main window sports a new organization of the speedbuttons, Component palette, and menu. Each of these elements are contained within tear-away toolbars, which permits you to reposition them within the main window, or even tear them off to create floating toolbars. Even Delphi's main menu can be enabled as a floating toolbar.

Displaying the Editor reveals the next major IDE enhancement. As shown in Figure 2, the Editor window now consists of two parts: the Module Explorer and the Code Editor. The Module Explorer provides an overview of the symbols in the unit displayed in the Code Editor. Furthermore, double-clicking an icon in the Module Explorer takes you to the relevant code declaration in the associated unit.

You can press Ctrl ⇧Shift E to quickly navigate between the Module Explorer and Code Editor.

At first you might be alarmed about the space consumed by the Module Explorer. This is not a concern, however, as the Module Explorer (like many of the other windows in Delphi 4), can be dragged out of its docked location within the Editor window and either docked into another window, or left as a floating window. For example, if you prefer, you can dock the Object Inspector and the Module Explorer together in a single Tools window.

In addition to obvious physical changes in the Editor window, Delphi 4 provides three major new features that add convenience to your code-writing efforts: Class Completion, Method Navigation, and the Code Browser.

Class Completion. Class Completion is useful for anyone who regularly declares new methods and properties in class **type** declarations. Based on a method declaration you enter into your class **type** declaration, Class Completion generates a method implementation in the corresponding implementation section of your unit.

For example, enter the following method declaration into the **type** declaration for a class named *TOrder*:

```
function IsValid(NewValue: Integer): Boolean;
```

Next, invoke Class Completion by pressing Ctrl ⇧Shift C. Delphi responds by generating the following method stub in the **implementation** section of the unit:

```
function TForm1.IsValid(NewValue: Integer): Boolean;
begin

end;
```

Likewise, when entering a property declaration, Code Completion will generate a write access method, a field for the storage of the property, and implement the write access method where it assigns the write access method's *Value* parameter to the generated field. For example, consider the following class declaration where a property name and type have been entered:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
    property Order: Integer;
  end;
```

If you press Ctrl ⇧Shift C while your cursor is still in the class declaration, Class Completion will modify the **type** declaration as follows:

```
type
  TForm1 = class(TForm)
  private
    FOrder: Integer;
    procedure SetOrder(const Value: Integer);
    { Private declarations }
  public
    { Public declarations }
    property Order: Integer read FOrder write SetOrder;
  end;
```

In addition, it will add the following implementation of the write access method to the unit's **implementation** section:

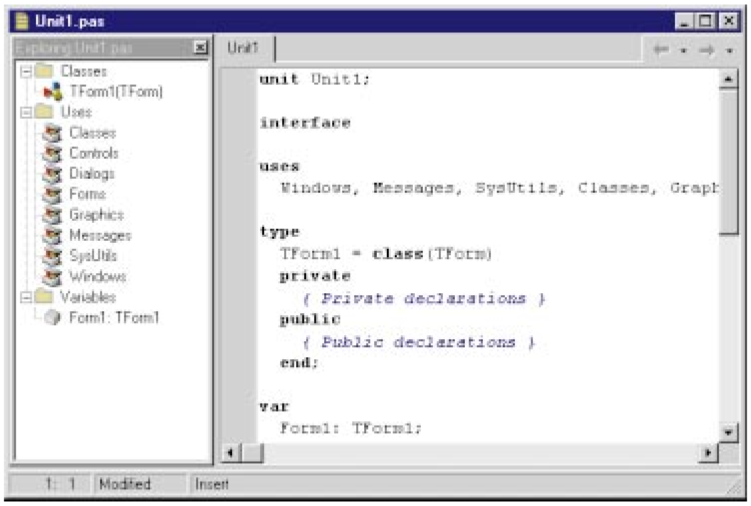
```
procedure TForm1.SetOrder(const Value: Integer);
begin
  FOrder := Value;
end;
```

Class Completion can be used on single method or property declarations, or on an entire class declaration. That is, if you enter multiple method and/or property declarations into a class declaration, pressing Ctrl ⇧Shift C will generate all the necessary declarations and implementations in a single step.

Method Navigation. Method Navigation permits you to quickly navigate to the various parts of your method declarations. For example, if your cursor is positioned on a method declaration in a **type** declaration, pressing Ctrl ⇧Shift ↓ will reposition your cursor in the method's implementation.

Similarly, from a method implementation, press Ctrl ⇧Shift ↑ to quickly move to the method declaration in the appropriate class declaration.

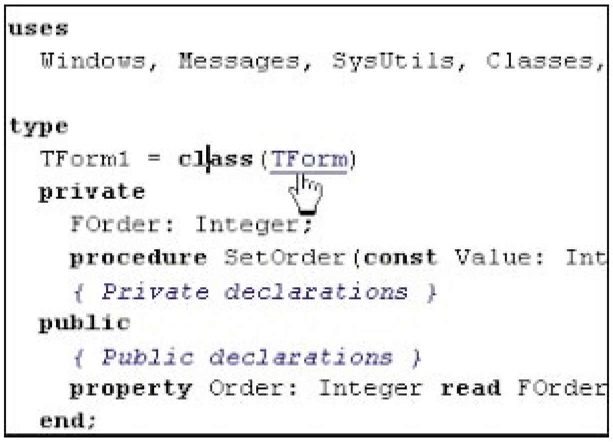
Code Browser. The Code Browser permits you to quickly open the unit in which a particular symbol is declared, or to open a unit listed in a **uses** clause. To use the Code Browser, press Ctrl with the mouse pointer over a declaration. After a brief pause, the mouse pointer will change to a hand shape, and the symbol will be underlined. Then, if you click the symbol (keeping Ctrl depressed), the Code Browser opens the unit associated with the symbol in the Code Editor. For example, while holding Ctrl down, move your mouse over



```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Grapt

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

var
  Form1: TForm1;
```

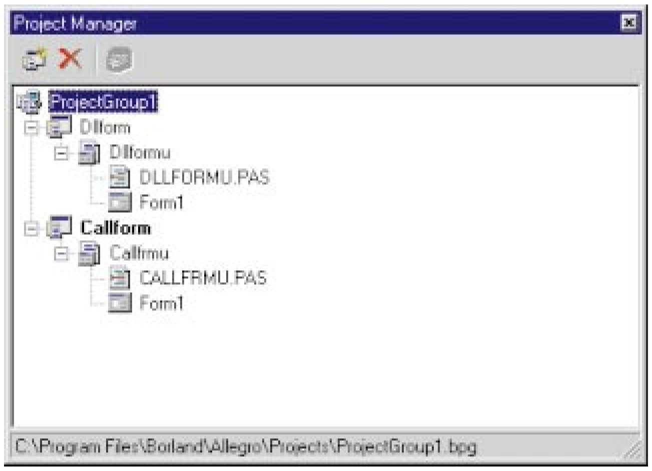
Figure 2: The Editor window initially contains both the Module Explorer and Code Editor.

the *TForm* reference in a class declaration, as shown in Figure 3. Once the word *TForm* appears underlined, click it to load the Forms unit in the Code Editor and position the cursor at the *TForm* declaration.



```
uses
  Windows, Messages, SysUtils, Classes,

type
  TForm1 = class(TForm)
  private
    FOrder: Integer;
    procedure SetOrder(const Value: Int
    { Private declarations }
  public
    { Public declarations }
    property Order: Integer read FOrder
  end;
```

Figure 3: The Code Browser permits you to quickly view units associated with symbols in your code.

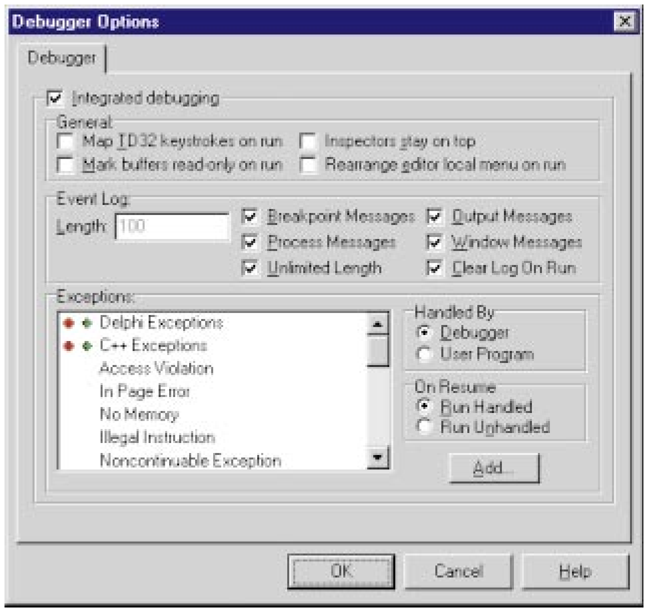


Project Manager

ProjectGroup1
Dllform
Dllformu
DLLFORMU.PAS
Form1
Callform
Callfrmu
CALLFRMU.PAS
Form1

C:\Program Files\Borland\Allegro\Projects\ProjectGroup1.bpg

Figure 4: The new Project Manager permits you to manage projects as well as project groups.



Debugger Options

Debugger

[✓] Integrated debugging

General
[ ] Map TD32 keystrokes on run
[ ] Inspectors stay on top
[ ] Mark buffers read-only on run
[ ] Rearrange editor local menu on run

Event Log
Length: 100
[✓] Breakpoint Messages
[✓] Output Messages
[✓] Process Messages
[✓] Window Messages
[✓] Unlimited Length
[✓] Clear Log On Run

Exceptions:
Delphi Exceptions
C++ Exceptions
Access Violation
In Page Error
No Memory
Illegal Instruction
Noncontinuable Exception

Handled By
(•) Debugger
( ) User Program

On Resume
(•) Run Handled
( ) Run Unhandled

Figure 5: The Debugger Options dialog box permits you to configure the event log, as well as define how exceptions are handled.

Project Manager. Another addition to Delphi 4's IDE is an improved Project Manager. The Project Manager has been enhanced to permit you to group two or more projects together. This feature is especially useful when the projects are related. For example, you may have a project that defines a DLL, and another an EXE that calls the DLL. Using a *project group*, you can conveniently manage these two projects together.

Furthermore, a project group permits you to quickly and easily navigate between the various modules of any projects in the group. Even when the Project Manager is used for a single project, its tree view displays the various units and form files of your project. Simply double-click a unit or form in the Project Manager to view it. The new Project Manager is shown in Figure 4.

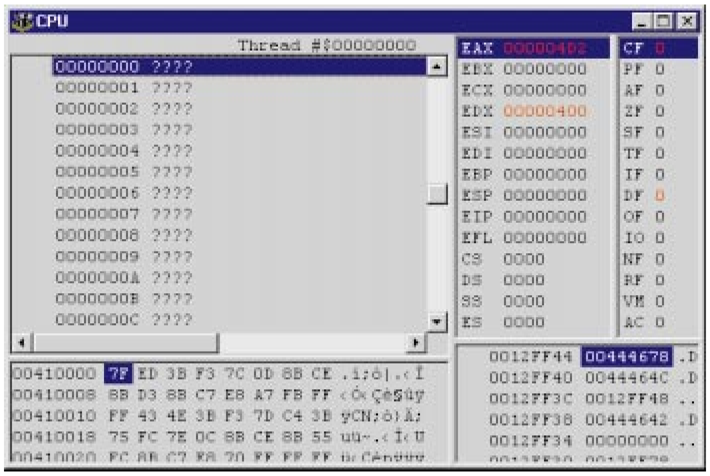
Form designer. The designer has also been improved to provide you with additional help while designing forms, data modules, etc. For example, simply pause your mouse over an object to see its name and class displayed in a floating help window. In addition, anytime you resize an object, a floating window appears and displays the object's width and height, and during a drag operation, the window displays the object's top-left coordinates.

Debugger. Even bigger enhancements are found in the tools you use for code debugging. The biggest of these is the Event Log, a window that you configure to trace events during program execution, including process loading and termination, breakpoint messages, and windows messages. Furthermore, using the Debugger Options dialog box (see Figure 5), you can control how individual exceptions are handled.

Two windows previously available for debugging are now improved. The CPU window, which was only available if you added the appropriate Windows registry entry, is now a standard part of the debugger. In addition, it has been enhanced significantly to provide additional information not previously shown, including the contents of the CPU registers. An example of how the new CPU window looks while a project is running is shown in Figure 6.

Likewise, the module window has been enhanced to include the source code paths for loaded modules (binary executables such as EXEs and DLLs). Another pane in the module window lists module function entry points, as well as global symbols if the module was compiled with debug information.

Object Repository. Another area within the IDE with extensive enhancements is the Object Repository. Here you will find a number of valuable new Wizards, including the Project Group Wizards, a Resource DLL Wizard, an MTS Data Module Wizard, a Service Wizard, a Service Application Wizard, and a CORBA Data Module Wizard. These Wizards are shown in Figure 7, which displays the New page of the Delphi 4 Object Repository. Note that because of the number of new Wizards, not all Wizards on the New page of the Object Repository appear in Figure 7. To access the Web Server Module Wizard, for example,



```
CPU
                         Thread #$00000000   EAX 00000402   CF 0
   00000000 ????                             EBX 00000000   PF 0
   00000001 ????                             ECX 00000000   AF 0
   00000002 ????                             EDX 00000400   ZF 0
   00000003 ????                             ESI 00000000   SF 0
   00000004 ????                             EDI 00000000   TF 0
   00000005 ????                             EBP 00000000   IF 0
   00000006 ????                             ESP 00000000   DF 0
   00000007 ????                             EIP 00000000   OF 0
   00000008 ????                             EFL 00000000   IO 0
   00000009 ????                             CS  0000       NF 0
   0000000A ????                             DS  0000       RF 0
   0000000B ????                             SS  0000       VM 0
   0000000C ????                             ES  0000       AC 0

00410000 7F ED 3B F3 7C 0D 8B CE .í;ó|.‹Î       0012FF44 00444678 .D
00410008 8B D3 8B C7 E8 A7 FB FF ‹Ó‹Çè§ûÿ       0012FF40 0044464C .D
00410010 FF 43 4E 3B F3 7D C4 3B ÿCN;ó}Ä;       0012FF3C 0012FF48 ..
00410018 75 FC 7E 0C 8B CE 8B 55 uü~.‹Î‹U       0012FF38 00444642 .D
                                                0012FF34 00000000 ..
```

Figure 6: The new CPU window contains information not previously available for debugging.

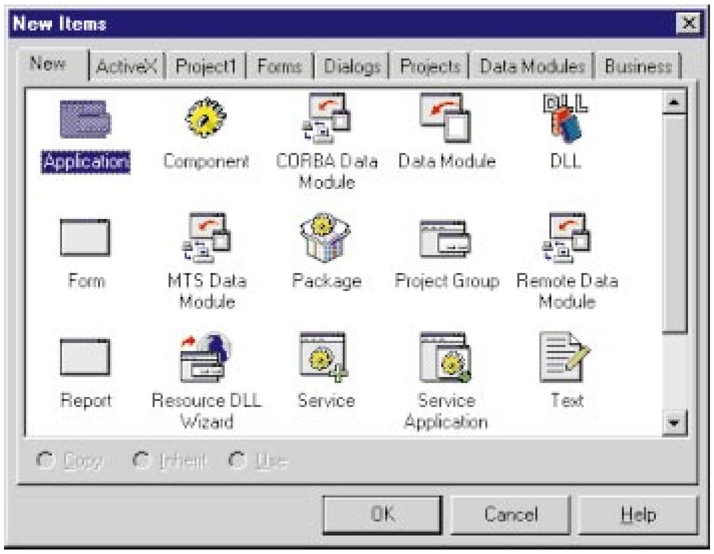
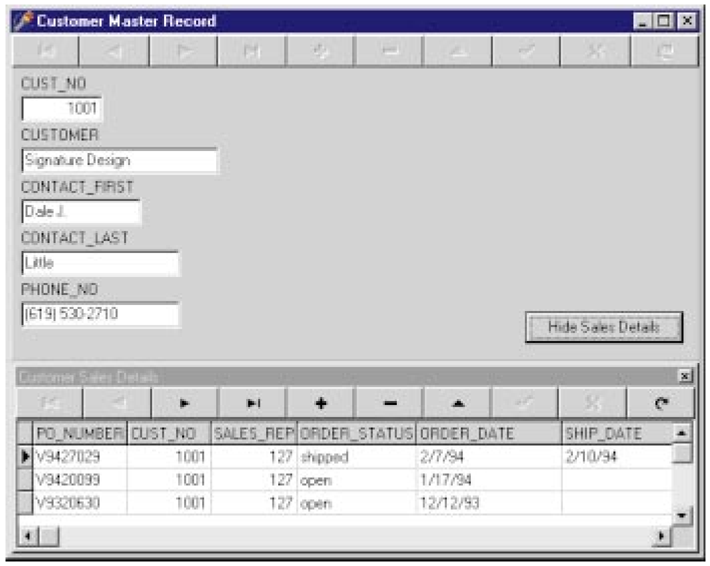


Figure 7: Delphi 4 introduces a large number of new Wizards, including those for CORBA and Microsoft Transaction Server.



Customer Master Record

CUST_NO
1001
CUSTOMER
Signature Design
CONTACT_FIRST
Dale J.
CONTACT_LAST
Little
PHONE_NO
(619) 530-2710

PO_NUMBER	CUST_NO	SALES_REP	ORDER_STATUS	ORDER_DATE	SHIP_DATE
V9427029	1001	127	shipped	2/7/94	2/10/94
V9420099	1001	127	open	1/17/94	
V9320630	1001	127	open	12/12/93	

Figure 8: A master-detail relationship displayed using a docked window for detail records.

you will need to scroll within the New page of the Object Repository.

VCL Changes

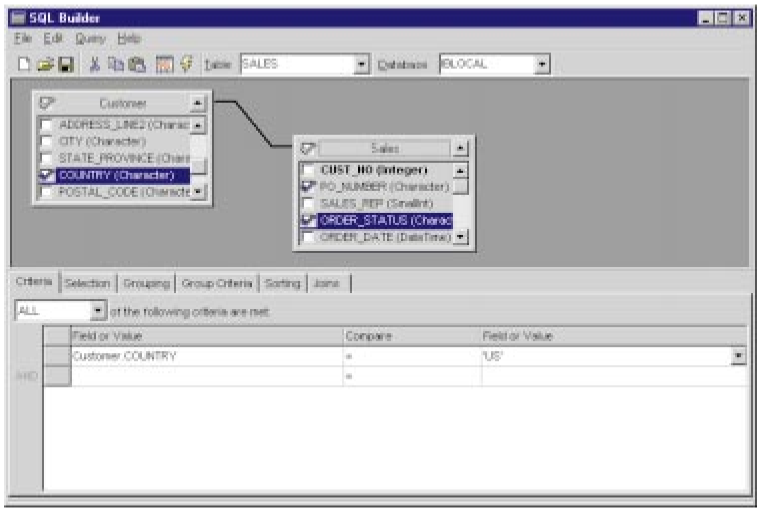
Numerous changes have been made to the Delphi 4 VCL (Visual Component Library). Among these are the *TActionList* component, which simplifies the process of synchronizing labels, bitmaps, and event handlers across buttons and menus.

Internet components. In addition, the Component palette's Internet page now includes a large number of new Internet-related components based on the Net Masters Fastnet tools. While additional information on these components was not available at the time of this writing, these new components appear to be entirely Object Pascal-based, as opposed to being ActiveX controls like the NetManage Internet controls available in earlier versions of Delphi. Consequently, these controls make it much easier to deploy Internet-based applications.

Object enhancements. Besides the new components, there have been some significant updates to the object hierarchies in Delphi 4. For example, the *TObject* class now includes *AfterConstruction* and *BeforeDestruction* methods, both of which are protected. Since every object in Delphi inherits these methods, object developers now have these two additional methods that they can override, if necessary, to provide for more effective component creation and release.

In addition, both the *TControl* and *TWinControl* classes now support drag-and-dock operations. Any *TWinControl* descendant can be designed to be a potential dock site. Likewise, any *TControl* descendant can be configured to be dockable. Among other uses, with these features you can let a user dock one window within another, similar to the docking provided within Delphi's IDE. For example, in a database application you may display customer information on one form, and the sales to that customer in another (these sales constitute detail records associated with the selected customer in the customer form). If you want, you can design these forms so the sales detail form can be docked into the customer master form. An example of such a relationship created with Delphi 4 is shown in Figure 8.

Another feature introduced into the object hierarchy at the *TControl* level is the *Constraints* property. This property permits you to define maximum and minimum dimensions for a control, as well as to define vertical and horizontal anchor points. One purpose for the *Constraints* property would be to prevent an aligned control from becoming so small that a contained control is hidden. The *TControl* class also enhances the *AutoSize* property for some descendants, so that re-scaling operations apply to both component size as well as contained font properties.



	Field or Value	Compare	Field or Value
	Customer.COUNTRY	=	'US'
AND		=	

Figure 9: The new SQL Builder simplifies the process of creating SQL queries.

If you need to create ActiveX controls, you'll be glad to learn that ActiveX support has also been improved. The ActiveX controls you create with Delphi 4 can now include Visual Basic data interfaces.

Database Enhancements

Delphi's data access components have been improved to support features new to the latest release of the Borland Database Engine (BDE). These include improved support for Access 97 and support for Oracle 8 extensions to SQL, including ADTs (Abstract Data Types), arrays, references, and nested tables. Even the *ClientDataSet* class has been improved. While many of these improvements apply to MIDAS applications, other improvements have been introduced to make this component even better suited for those stand-alone applications that don't use the BDE. For example, the *ClientDataSet* class now supports a wider range of filter expressions, as well as maintained aggregates (Sum, Count, Average, and so forth).

Another big enhancement specific to database development is the inclusion in the Client/Server edition of the new SQL Builder, which replaces the Query Builder provided in earlier versions. SQL Builder provides a more intuitive interface for building SQL queries, and simplifies the process of defining selection criteria, groupings, sorts, joins, and aggregates (see Figure 9).

Multi-tier Enhancements

One of the major additions to Delphi 3 was support for multi-tier application development. As you can imagine, with INPRISE's commitment to enterprise computing, this area of Delphi received a great detail of additional attention. Delphi 4 provides even more control over distributed applications using OLE Enterprise and DCOM (Distributed Component Object Model) technologies. In addition, Delphi 4 provides support for both CORBA (Common Object Request Broker Architecture) and MTS (Microsoft Transaction Server) applications, further extending Delphi's multi-tier support across the network.

Some specific MIDAS changes include enhancements in the *TClientDataSet* class and the related connection classes on the client side. For example, *ClientDataSet* objects can now define parameters that can be used by the application server, can more easily refresh and resync with the server, and can more conveniently use a server's interface. Also, the *TRemoteServer* and *TMIDASConnection* classes have been replaced by the *TCOMConnection* and *TOLEConnection* classes, respectively. These classes descend from the new *TDispatchConnection* class, which defines the common interface for all connection components. (*TRemoteServer* and *TMIDASConnection* are retained for backward-compatibility).

Language Enhancements

While not usually the most glamorous to discuss, language enhancements can be the source of great added power for developers. And fortunately for Delphi developers, INPRISE is free to improve Object Pascal however it likes. (Vendors of tools for other languages, e.g. C++, are often at the mercy of standards committees whose constituent members may have conflicting interests.)

In previous versions of Delphi, Object Pascal has gained some notable additions, including ANSI strings (long strings) and Interfaces. However, it can be argued that Delphi 4 introduces more enhancements to the Object Pascal language than any previous release. These additions include dynamic arrays, method overloading, default parameter values in methods, interface delegation to object properties, and two new 64-bit data types. Each of these is discussed briefly in the following section.

Dynamic arrays. A dynamic array is one whose dimensions can be defined and redefined at run time, without losing data. The dynamic arrays in Delphi 4 can be multi-dimensional, and the dynamic nature of these arrays allows for non-rectangular matrices. For example, the first dimension of a two-dimensional array can be an array of 100 elements, while the second dimension can be an array of five elements. Additional support is provided for copying dynamic arrays.

Method overloading. With the addition of method overloading to Object Pascal, component developers can now declare two or more methods to have the same name. The only requirement is that all methods with the same name do not have the same number and types of parameters, and, if the method is a function method, they must all return a value of the same type.

The advantage of method overloading is that you can create two or more methods that perform essentially the same task, albeit differently (based on the parameter values they are passed), and retain the same name for consistency. An obvious example of this is the ability to overload the *Create* constructor. In past versions of Delphi, a component with multiple constructors had to provide different names for each constructor. For example, the *Exception* class declares no less than eight constructors, including *Create*, *CreateFmt*, *CreateRes*, and so on. Using overloading, it would be possible to declare eight different constructors — all named *Create*.

When creating overloaded methods, each method with the same name must be identified by the **overload** keyword. For example, the following two constructor declarations are valid within a single class **type** declaration:

```
constructor Create(AOwner: TComponent); overload; override;
constructor Create(AOwner: TComponent; Text: string); overload;
```

Default parameters. Like overloading, the addition of default parameters to Object Pascal applies to the declaration of methods. In Delphi 4, you can define a default value for the last parameter in a parameter list. (There must be two or more parameters to use this feature.) When a method declared using this new syntax is called, the final parameter can be omitted in the method invocation, in which case the default value is used. For example, consider the following method declaration:

```
procedure Run(Enabled: Boolean; Interval: Integer = 1000);
```

When this method is called, it can be called using either two parameters, or only the first. When the last parameter is omitted, the value, 1000, is automatically assigned to the formal parameter interval.

Interface enhancements. Without a doubt, the most important language feature introduced in Delphi 3 was interfaces. If you are versed in using interfaces, you will be pleased to learn that Delphi 4 provides an additional tool for implementing and using interfaces in classes. You can now declare a property of a class to be either a class type that implements an interface, or an interface type. This is done by following the property declaration with the new **implements** directive, followed by an interface type. Only one property in each class can be declared to implement a given interface, although a given property may be declared to implement two or more interfaces.

This is demonstrated in the following **type** block which declares a class, *TSomeObject*, that includes a property that can reference any object that implements the *IDispatch* interface:

```
type
  TSomeObject = class(TObject, IDispatch)
  private
    FOLEObject: IDispatch;
  public
    property OLEObject: IDispatch read FOLEObject
      write FOLEObject implements IDispatch;
  // The rest of the class declaration goes here.
  end;
```

As shown in the preceding code example, the class that includes this property must be declared to implement the interface type of the property (*IDispatch* in this case). From within the implementation of a class that makes use of a property that implements an interface, a reference to an object that implements that interface can be assigned to the interface property. This will cause that implementation to be returned if an instance of the class containing the property is assigned to another interface reference, i.e. a variable declared to be other than that interface type. In the previous code example, an object that implements the *IDispatch* interface can be assigned to the *OLEObject* property of an instance of *TSomeObject*. As a result, if the *TSomeObject* instance is subsequently assigned to an interface reference, the implementation provided by the object assigned to the *OLEObject* property will be returned.

As previously mentioned, the property type declared to implement an interface can be a class type (as opposed to an interface type), as demonstrated in the preceding example. When the delegate property is a class, the compiler will search methods declared in the property's class type to satisfy the interface's method implements first, before searching the class in which the delegate property is declared. At run time, after the delegate property has been assigned an instance of its declared class, an instance of the class containing the delegate property declaration can be assigned to an interface reference. Subsequent calls to the methods of this interface using the interface reference may call either methods of the delegate property's implementation or method implementations found in the instance of the class containing the delegate property, depending on where the compiler discovered the method implementations.

When the type of an interface delegate property is a class type, method name resolution can be used to change which method implementations the compiler will assign to the interface methods. For more information on using interfaces, see the article "Run-time Type Information" in the June, 1998 issue of *Delphi Informant*.

New data types. Finally, Object Pascal now defines two new 64-bit data types. The Real type, which was previously 48 bits, is now 64 bits like the Double type. If your code relies on a 48-bit Real type, Delphi 4 provides you with the:

{$REALCOMPATIBILITY ON}

compiler directive to use the previous Real size.

For reference to very large integers, Object Pascal now supports the Int64 type, which is a 64-bit signed integer. Constants declared too large to fit into the 32-bit Integer type are treated as Int64 integers automatically by the compiler.

Conclusion

As mature as Delphi has become over its three short years, it is hard to imagine that INPRISE could add much more to this highly regarded development tool. But as you can see, Delphi 4 introduces numerous enhancements and new features. To put it simply, if you are going to write 32-bit Windows applications, you should do so with Delphi 4. ∆

Cary Jensen is President of Jensen Data Systems, Inc., a Houston-based database development company. He is author of more than a dozen books, including *Delphi in Depth* [Osborne McGraw-Hill, 1996]. He is also 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 the Jensen Data Systems Web site at http://idt.net/~jdsi. You can also reach Jensen Data Systems at (281) 359-3311, or via e-mail at cjensen@compuserve.com.

Delphi 3 / WinSock 2

By John Penman

WinSock 2

Part II: Multi-threading

In last month's article, we introduced the WinSock 2 API (see *Delphi Informant*, June 1998). We examined two functions: *WSAEnumProtocols* and *WSAEnumNameSpaceProviders*. With these functions, we created an application to demonstrate transport protocol independence and name space providers.

In this article, we'll create two applications to demonstrate multi-threading with WinSock 2 functions: One is a server that delivers a file on demand; the other is a client that requests a file. We use a home-grown protocol that we call Simple File Transfer Protocol (SFTP) to perform uni-directional file transfers (from the server to the client). In spite of the similarity in their names, the SFTP protocol is not the same as FTP, because of the following differences:

- SFTP uses both User Datagram Protocol (UDP) and Transmission Control Protocol (TCP), whereas FTP uses TCP only.
- SFTP has a much lower overhead than FTP, because it uses UDP for messages. FTP uses TCP for both messages and file transfer.
- SFTP has no robust error checking.
- SFTP is uni-directional, i.e. the client can only retrieve a file from the server. FTP is bi-directional, i.e. the client can upload and download files from the server.

SFTP, or the Poor Man's FTP

Before dealing with the WinSock 2 APIs that drive SFTP, we must explain the protocol in detail. In brief, here's the protocol in operation:

1) Server listens on UDP port XXX.
2) Client sends a request to the server on UDP port XXX.
3) Server replies to the client.
4) Server sends data on TCP port YYY to the client.
5) Server resumes listening on UDP port XXX for more requests.

When the server starts, it listens on a UDP port for a message from the client. UDP is a connectionless transport protocol that doesn't perform any integrity checking on packets of data in transit; therefore, it has a low overhead. Packets sent over UDP can either expire silently, arrive out of order, or become duplicated. In spite of the limitations, UDP is a highly efficient protocol for messages. We must assign a number to the UDP port that is known to the client. The SFTP client sends a request message in the following format:

```
UserName:Password:MachineName:PortNo:FileName
```

A colon separates each token for easy parsing by the server. The server then decodes the message to retrieve the tokens sequentially, comparing each token with its database. If a token doesn't match, the server sends back an error message. For example, if there is no match with the `UserName` token, the server sends back the following message:

```
'ERR'
```

The server does no further processing of the message, and resumes listening for more requests. If the `UserName` token matches, the server processes the next token, `Password`. This process of matching continues until the message is successfully done, or until there is a mismatch. On completion, the server sends an "OK" string that tells the client to prepare to receive the data on its data port, designated by the `PortNo` token.

The data port uses TCP to transfer a file from the server to the client. Unlike UDP, TCP is a reliable transport protocol that performs error checking and manages data integrity, which is absolutely essential for data. (Using TCP is like sending registered mail; you become aware of any problems, such as non-receipt, when it happens.) After connecting to the data port on the client side, the server sends a stream of data, which the client stores in a file. At the end of the data streaming, the server closes its data port (by closing the socket), signaling to the client to close the file and data port. We will demonstrate the SFTP protocol from the client's perspective.

The Front End

After mastering the essentials of the SFTP protocol, we can focus on implementing the SFTP client. Figure 1 shows the SFTPClient in the IDE. Before we request a file, we need to start WinSock 2. Clicking the **Load** button initiates the *Start* function, which attempts to load WinSock 2. When WinSock 2 starts running, the SFTPClient application enables the **Get File!** button and disables the **Load** button. The application does this to prevent the loading of multiple copies of the WinSock 2 DLL. SFTPClient populates the *edMachineName* control by calling the *gethostname* API to retrieve the name of the machine. Sometimes a call to *gethostname* can fail because there is no name associated with the machine. In this case, we must enter a fictitious name.

WinSock 2 stays in memory until we click the **Close** button, which calls the WinSock 2 API *WSACleanUp* to unload WinSock 2.

We use the Edit controls *edMachineName*, *edUserName*, *edPassword*, *edFileName* and *edPortNo* to capture the tokens we require to build a request message. The *edHostName* control stores the name of the server. If *edHostName* contains no host name, SFTPClient won't allow us to proceed. We store the tokens as fields of the *Request* record, which we pass to the *thrdMsg* thread like this:

```
thrdMsg := TMsgThrd.Create(Request, TimeOutValue);
```

We set the time-out in the *edTimerSetting* control, which we pass in *TimeOutValue* to regulate the timer in *thrdMsg*. Because we use UDP, it's possible that a packet can become lost in transit, so it's prudent to time-out the application (otherwise it could wait forever). In contrast to TCP, using UDP is like sending a letter in the mail, which could become lost in transit.

Handling Events on a Thread

In the *CMsgThrd* unit (available for download; see end of article for details), we derive *TMsgThrd* from the *TThread* class. Before spinning the thread, we use the *Create* constructor to set up the timer, the request string, the message socket, and the WinSock 2 events (see Figure 2).

We call the following function to set up a socket for sending and receiving messages using UDP:

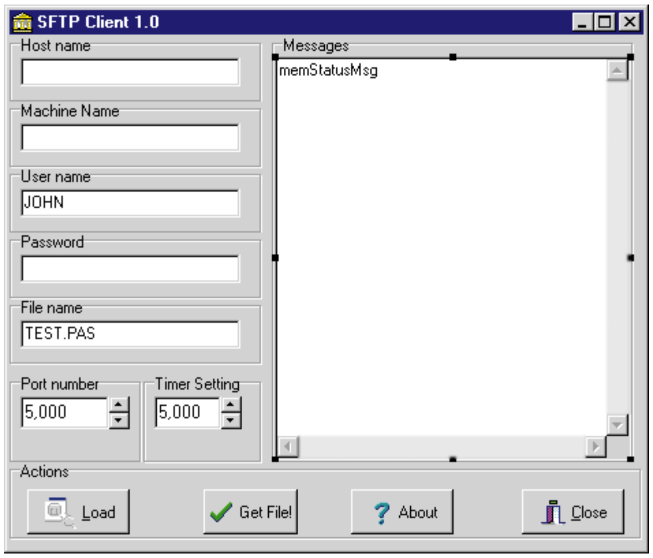


Figure 1: SFTPClient in the Delphi 3 IDE.

```
sktMsg := WSASocket(AF_INET, SOCK_DGRAM,
                    IPPROTO_UDP, nil, 0, 0);
```

WSASocket is the WinSock 2 version of the socket API, which has the capability to create an overlapped socket. Using overlapped sockets may enhance the data throughput. Under certain circumstances, however, overlapped sockets may not always work correctly on Windows 95. I believe the forthcoming Windows 98 implements the overlapped socket I/O feature correctly. For now, however, we'll do without this feature.

As we are not going to use overlapped I/O, we set the last parameter to zero in *WSASocket*. The first parameter specifies the address type, the second specifies the use of datagrams (discrete data packets), and the third specifies the UDP transport protocol. The fourth parameter is a pointer to the WSAPROTOCOL_INFO structure (which we examined in last month's article), but for the sake of simplicity, we have set this to **nil**. The fifth and last parameters specify the socket identifier and the socket attribute, respectively, which we can ignore.

We want a notification whenever we get a reply from the server. To do this, we create *EventMsg* and *EventData*, which are generic event objects of the type *WSAEvent*. Then we call the Win32 API *CreateEvent* function to get handles to these event objects:

```
CreateEvent(lpEventAttributes: PSecurityAttributes;
            bManualReset: LongBool;
            bInitialState: LongBool; lpName: PChar);
```

In Delphi, we call the API like this:

```
EventMsg := CreateEvent(nil, False, False, nil);
```

We set the first parameter to **nil** to accept the default security settings. The second parameter is *False* to tell Windows to reset the event object automatically. The third

```
constructor TMsgThrd.Create(Requests: TRequest;
                            TimerSetting: Integer);
var
  Res: Integer;
begin
  inherited Create(True);

  FreeOnTerminate := True;
  OnTerminate     := OnMsgThrdDone;
  TimeOutValue    := TimerSetting;
  Done            := False;

  // Set the Timer.
  ResTimer          := TTimer.Create(nil);
  ResTimer.Interval := TimeOutValue;
  ResTimer.OnTimer  := OnTimeOut;
  ResTimer.Enabled  := False;

  // Decode Request record to build the request message.
  Request := Requests;
  with Request do begin
    RequestMsg := ConCat(UserName,':', Password, ':',
                  MachineName, ':', Port, ':', FileName);
  end;
  State := stMsg;

  // Set up the message socket.
  sktMsg := WSASocket(AF_INET, SOCK_DGRAM, IPPROTO_UDP,
                      nil, 0, 0);
  if sktMsg = SOCKET_ERROR then begin
    Msg := Concat('Failed to create socket Error ',
                  IntToStr(WSAGetLastError));
    Synchronize(Update);
    State := stError;
    Done  := True;
    Exit;
  end;

  // Create events.
  EventMsg := CreateEvent(nil, False, False, nil);
  if EventMsg = WSA_INVALID_EVENT then begin
    Msg := Concat('Failed to create Message Event. Error ',
                  IntToStr(WSAGetLastError));
    Synchronize(Update);
    State := stError;
    Done  := True;
    Exit;
  end;
  EventData := CreateEvent(nil,False,False,nil);
  if EventData = WSA_INVALID_EVENT then begin
    Msg := Concat('Failed to create Data Event. Error ',
                  IntToStr(WSAGetLastError));
    Synchronize(Update);
    State := stError;
    Done  := True;
    Exit;
  end;

  // Set up notification.
  Res := WSAEventSelect(sktMsg, EventMsg,
                        FD_READ or FD_WRITE);
  if Res = SOCKET_ERROR then begin
    Msg := Concat('WSAEventSelect call failed for socket ',
                  IntToStr(sktMsg), '. Error ',
                  IntToStr(WSAGetLastError));
    Synchronize(Update);
    State := stError;
    closesocket(sktMsg);
    Done := True;
    Exit;
  end;

  // Fire the thread.
  Resume;

  // Resolve a host name before sending a request message.
  Resolve;
end;
```

Figure 2: The class constructor.

parameter is also *False*, which sets the *EventMsg* in a non-signaled state. As we do not require the event object to have a name, we set the last parameter to **nil**. On success, *CreateEvent* returns a valid handle. Otherwise, if *CreateEvent* returns an invalid value (WSA_INVALID_EVENT), SFTPClient sets the *State* variable to *stError* and abandons the message thread.

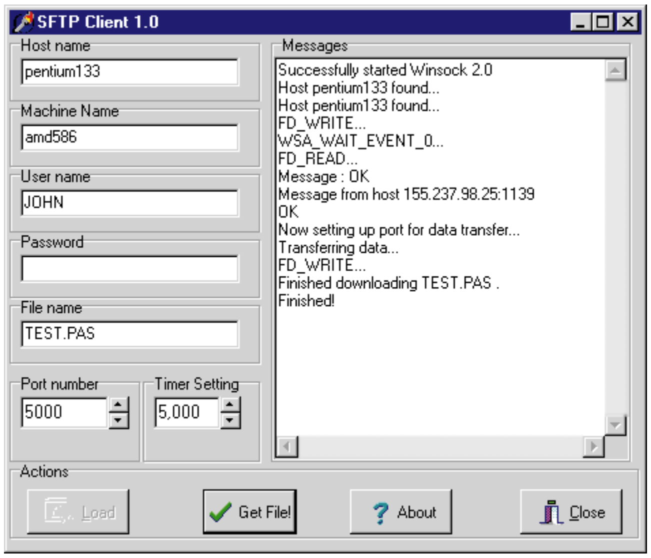
After we set the event objects — *EventMsg* and *EventData* — we must use *WSAEventSelect* to associate read or write events with *EventMsg* on the message socket, *sktMsg*. This function is similar to *WSAAsyncSelect*, but instead of posting a message to a window, *WSAEventSelect* sets the associated event object and records the network event that occurs. We call *WSAEventSelect* like this:

```
Res := WSAEventSelect(sktMsg, EventMsg,
                      FD_READ or FD_WRITE);
```

The function associates the *EventMsg* event object with two selected network events: FD_READ and FD_WRITE, on a socket, *sktMsg*.

Like all WinSock APIs, we must check the result of every operation. On success, *WSAEventSelect* returns a zero; otherwise, it returns SOCKET_ERROR. When this happens, SFTPClient uses the *Synchronize* function to display a message from the message thread to the user interface. Figure 3 shows SFTPClient in action.

WSAEventSelect doesn't operate in isolation; it works with *WSAWaitForMultipleEvents* and *WSAEnumNetworkEvents* (we'll explore these APIs when we discuss handling network events). Then, we call *Resume* to fire the *thrdMsg* thread, kicking the *Execute* method into action. The *Execute* method has a **repeat** loop that repeats until *WSAWaitForMultipleEvents* returns an event object when a network event occurs.



Host name: pentium133
Machine Name: amd586
User name: JOHN
Password:
File name: TEST.PAS
Port number: 5000
Timer Setting: 5,000

Messages

```
Successfully started Winsock 2.0
Host pentium133 found...
Host pentium133 found...
FD_WRITE...
WSA_WAIT_EVENT_0...
FD_READ...
Message : OK
Message from host 155.237.98.25:1139
OK
Now setting up port for data transfer...
Transferring data...
FD_WRITE...
Finished downloading TEST.PAS .
Finished!
```

Figure 3: SFTPClient in action.

After spinning the thread, we call *Resolve* to determine the IP address from *Request.HostName*. On success, *Resolve* calls *SendMsg* to send the request to the server. On failure, we exit from the message thread. *Resolve* doesn't handle IP-dotted addresses at all; we would need to add the *gethostbyaddr* API to the code.

SendMsg allocates memory to *Buff* (of type PWSABUF, a WinSock 2 data structure) to hold the request message in the *Msg* variable. The following code fragment shows how this is done:

```
try
  Buff     := AllocMem(SizeOf(Buffers));
  Buff.Buf := Buffers;
  Buff.Buf := PChar(Msg);
  Buff.Len := SizeOf(Buffers);

  { Rest of code follows. }

finally
  FreeMem(Buff, SizeOf(Buffers));
end;
```

As mentioned earlier, we prevent SFTPClient from waiting forever for a reply that may never come, and we enable the timer, *ResTimer*. When the time-out value exceeds, *ResTimer* calls its *OnTimeOut* event handler to shut down the message thread gracefully.

The API to send the message as a datagram is:

```
WSASendTo(s: TSocket; lpBuffers: PWSABUF;
  dwBufferCount: Integer; lpNumberOfBytesSent: PDWORD;
  dwFlags: Integer; lpTo: PSockAddr; iToLen: Integer;
  lpOverlapped: POverlapped;
  lpCompletionRoutine: PWSAOVERLAPPED_COMPLETION_ROUTINE);
```

which we call in Delphi like this:

```
Res := WSASendTo(sktMsg, Buff, BuffCount, @NoBytesSent,
                 Flags, @HostAddr, Len, nil, nil);
```

We won't examine the parameter list in detail, but note that the *Buff* parameter contains the request and the *HostAddr* parameter contains the address of the server. Because we aren't using the overlapped I/O feature, we set *lpOverlapped* and *lpCompletionRoutine* to **nil**.

Calling *WSASendTo* triggers a network event, FD_WRITE, to wake up *SAWaitForMultipleEvents*. When SFTPClient receives a reply, it generates a network event, FD_READ, which notifies *WSAWaitForMultipleEvents*. The formal definition of this function is:

```
WSAWaitForMultipleEvents(cEvents: Integer;
  lphEvents: PHandle; fWaitAll: LongBool;
  dwTimeOut: Integer; fAlertable: LongBool);
```

The *cEvents* parameter specifies the number of events, which is two (*EventMsg* and *EventData*). The second parameter, *lphEvents*, points to an array of event object handles. Because we want immediate notification of an event, we set *f WaitAll* to *False*. As we have no idea of when we get an event notification, we set *dwTimeOut* to WSA_INFINITE. We set the last parameter, *fAlertable*, to *False*, because we have no completion routine to call when *WSAWaitForMultipleEvents* returns. In Delphi, we call the function like this:

```
WaitStatus := WSAWaitForMultipleEvents(NumEvents,
                @EventArray, False, WSA_INFINITE, True);
```

The API returns whenever an event object becomes signaled or an error has occurred. We determine the value of *WaitStatus* in a **case** statement:

```
repeat
  WaitStatus := WSAWaitForMultipleEvents(NumEvents,
                  @EventArray, False, WSA_INFINITE, False);
  if State = stError then
    Exit;

  case WaitStatus of
    WSA_WAIT_FAILED: begin
      Done := True;
      Msg := Concat('WSA_WAIT_FAILED ... Error ',
                    IntToStr(WSAGetLastError));
      Synchronize(Update);
      State := stError;
    end;
    WAIT_IO_COMPLETION: begin
      Msg := 'WAIT_IO_COMPLETION...';
      Synchronize(Update);
    end;
    WSA_WAIT_EVENT_0: begin
      Msg := 'WSA_WAIT_EVENT_0...';
      Synchronize(Update);
      HandleSocketEvent;
    end;
  end;

until Done;
```

When there is an FD_READ or FD_WRITE event, we call *HandleSocketEvent* to handle the network event. The *HandleSocketEvent* function calls another WinSock 2 API, *WSAEnumNetworkEvents*, to enumerate which network event has occurred. We call the function like this:

```
Res := WSAEnumNetworkEvents(sktMsg, EventMsg, @Buffers[0]);
```

To retrieve the network event, we typecast the *Buffers* parameter with the following statement:

```
lpNetworkEvents := PWSANETWORKEVENTS(@Buffers[0]);
```

Then, we de-reference the *lpNetworkEvents* structure to determine which network event has occurred:

```
// Decipher Network events.
with lpNetworkEvents^ do begin
  // Is this an FD_READ event?
  if (lNetworkEvents and FD_READ) = FD_READ then begin
    if iErrorCode[1] = WSAENETDOWN then begin
      Msg := 'Network down...';
      Synchronize(Update);
    end;
    Size := SizeOf(HostAddr);
    Msg := 'FD_READ...';

    { Rest of code. }

  end;
end;
```

Getting the Goods

We disseminate the *Buff* parameter to retrieve the message. If the message contains the "OK" string, we create a new thread, *thrdData*, to handle the data transfer:

```
thrdData := TDataThrd.Create(StrToInt(Request.Port),
                             Request.FileName);
```

We pass the port number in the first parameter and the file name in the second. In the *CDataThrd* unit (see Listing One, beginning on page 16), the *Create* constructor suspends the *thrdData* thread so we can create a listening socket for the incoming data stream. To do this, we perform these steps:

1) Call *WSASocket* to create the socket that uses TCP.
2) Populate the *DataAddr* data structure, including the assignment of the local port.
3) Call bind to associate the listening socket with the local address.

Then we call the thread's *Resume* method, which activates *Execute* to call *GetFile*. Using the listen API, *GetFile* "listens" for an incoming connection on the local port. When a connection arrives, we call another WinSock 2 API, and choose to "accept" the connection. Inside the **repeat** loop, we call *WSAReceive* to read the data stream, and *Write* (a method of *FileStream*, derived from the *TFileStream* class) to store the data. The reading and storing of the data stream continue until the data streaming ends.

When the data stream ends, the *thrdData* thread calls the event procedure, *OnDataThrdDone*, to terminate itself. To re-enable the thread-safe **Get File!** button, the event procedure calls *EnableBtn* via *Synchronize*. The *thrdMsg* thread also terminates. Note that I've hard wired the C drive in the previous code, which you will need to adjust for your machine.

This completes the client application; now we'll take a brief look at the server, SFTPServer.

SFTPServer

Much of what we have said about SFTPClient applies to SFTPServer, with a few exceptions that I will describe. I mentioned that the server would compare the tokens received from the client with its database. For brevity (instead of using a database), SFTPServer compares the tokens extracted from the client's request with those we entered in the Edit controls. Figure 4 shows these controls in the front end.

Like SFTPClient, we click the **Load** button to load WinSock 2. Then, we click the **Start** button to create the message thread that will handle any messages from the client. However, unlike SFTPClient, the server's *thrdMsg* exists for the lifetime of the application.

thrdMsg's constructor sets up a listening socket by calling the *WSASocket* and bind APIs. SFTPServer uses the same mechanisms, as demonstrated in SFTPClient to create the network event objects, and to monitor and intercept network events.

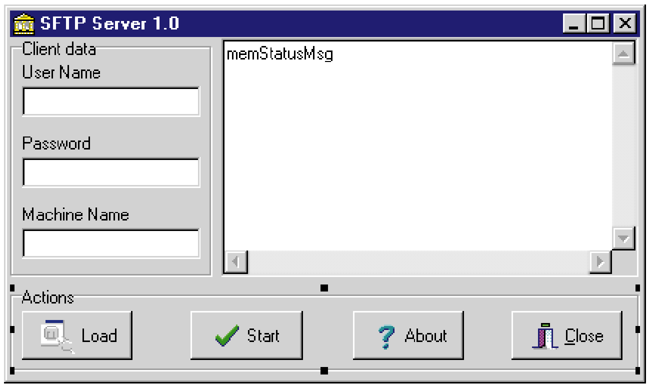


Figure 4: SFTPServer in the IDE.

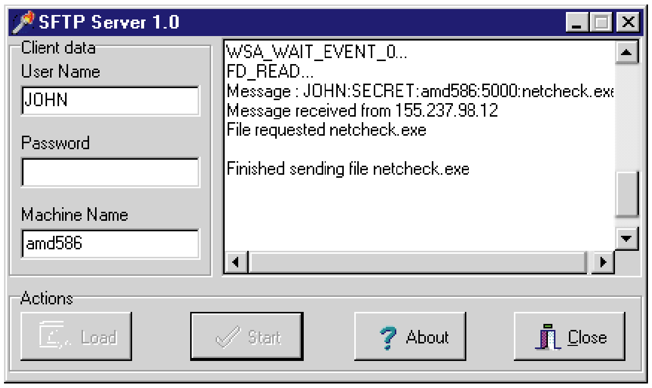
When SFTPServer receives a message on UDP port 2589, it generates an FD_READ event that invokes the *ParsedOK* function to decipher the message. If the message is deciphered correctly (i.e. all the tokens match what we entered in the Edit controls), then SFTPServer calls *SendMsg* to send the "OK" message back to the client. SFTPServer creates the *thrdData* thread to handle the transfer of data to the client.

The *thrdData.Create* constructor creates a new socket to send a file over TCP and initializes some variables before calling *Resume*. In the *Execute* procedure, *SendFile* sends the file. *SendFile* calls the *WSAConnect* API to connect with SFTPClient on the designated TCP port. Upon connection, SFTPServer uses *FileStream.Read* (*FileStream* is derived from *TFileStream*) to read the data from a file, and *WSASend* to send a stream of bytes. These actions take place in a **repeat** loop. After sending the file, *Execute* calls *Terminate* to kill the data thread. Figure 5 shows the dialog box after SFTPServer transfers a file.

Using the Example SFTPClient and SFTPServer Applications

There are two zip files in this month's archives, SFTPCLIENT.ZIP and SFTPSERVER.ZIP, for the client and server applications, respectively. Install the client on one machine and the server on another on your LAN or somewhere on the Internet. Make sure you have WINSOCK2.PAS (included in SFTPCLIENT.ZIP) in the path; I suggest you put the unit in the Delphi \Lib directory.

For the client and server, change the hard-wired drive mapping (in the *GetFile* method in the *CDataThrd* and



User Name: JOHN
Password:
Machine Name: amd586

```
WSA_WAIT_EVENT_0...
FD_READ...
Message : JOHN:SECRET:amd586:5000:netcheck.exe
Message received from 155.237.98.12
File requested netcheck.exe

Finished sending file netcheck.exe
```

Figure 5: SFTPServer in action.

SDataThrd units) to suit your system, and compile the SFTPClient and SFTPServer projects. Start up SFTPClient first, and enter the user name and password. If, on startup, the call to *gethostname* fails, enter any machine name in the *edMachineName* control. Armed with the details, start up SFTPServer and enter the details of SFTPClient, such as user name, password, and machine name on which SFTPClient resides. Then load WinSock 2 in both applications. If an application fails to load, perhaps it's because WinSock 2 isn't on your Windows 95 system (Windows NT 4.0 has WinSock 2 already installed). To get instructions on where to download the WinSock 2 SDK, aim your browser at http://www.sockets.com/winsock2.htm.

Looking Ahead

We have achieved the following:

- Using some of WinSock 2-specific APIs to implement a uni-directional file transfer
- Using these APIs with multi-threading
- Designing and using our own protocol
- Using UDP and TCP in the same application

In the next article, we'll examine the exciting world of multicast ("push technology") and how to create simple multicast server and client applications. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUL\DI9807JP.

John Penman is the owner of Craiglockhart Software, which specializes in providing Internet and intranet software solutions. John can be reached on the Internet at jcp@craiglockhart.com.

Begin Listing One — SFTP Client Data Thread Unit

```
unit CDataThrd;

interface

uses
  Classes, Windows, Winsock2;

type
  TDataThrd = class(TThread)
  protected
    sktListenData,
    sktData: TSocket;
    DataAddr: TSockAddrIn;
    Msg, NewFileName: string;
    procedure Execute; override;
    procedure OnDataThrdDone(Sender: TObject);
    procedure GetFile;
    procedure Update;
  public
    constructor Create(PortNo: Integer; FileName: string);
  end;

var
  thrdData: TDataThrd;

implementation

uses
  Dialogs, Main, CMsgThrd, SysUtils;

{ TDataThrd }
constructor TDataThrd.Create(PortNo: Integer;
                             FileName: string);
var
  Res: Integer;
begin
  // Create thread in suspended state, so we can set
  // important variables.



  inherited create(True);
  FreeOnTerminate := True;
  OnTerminate := OnDataThrdDone;
  sktListenData := WSASocket(AF_INET,SOCK_STREAM,
                             IPPROTO_TCP, nil, 0, 0);
  if sktListenData = INVALID_SOCKET then begin
    Msg := Concat(
            'Failed to create listening socket! Error ',
            IntToStr(WSAGetLastError));
    Synchronize(Update);
    State := stError;
    Exit;
  end;

  with DataAddr do begin
    sin_port := htons(PortNo);
    sin_family := AF_INET;
    sin_addr.s_addr := INADDR_ANY;
  end;

  Res := bind(sktListenData, DataAddr, SizeOf(DataAddr));
  if Res = SOCKET_ERROR then begin
    Msg := Concat('Failed to create listening socket! ',
                  IntToStr(WSAGetLastError));
    Synchronize(Update);
    State := stError;
    closesocket(sktListenData);
    Exit;
  end;

  NewFileName := FileName;
  // Start the data thread.
  Resume;
end;

// Execute the data transfer in the background.
procedure TDataThrd.Execute;
begin
  GetFile;
end;

// Retrieve the file.
procedure TDataThrd.GetFile;
var
  Buffers: array[0..MAXGETHOSTSTRUCT-1] of Char;
  Done: Boolean;
  Data: PWSABUF;
  Flags, NoBytesRecv, Res, Size: Integer;
  FileStream: TFileStream;
begin
  Res := listen(sktListenData,1);
  if Res = SOCKET_ERROR then begin
    Msg := Concat('Call to listen failed. Error ',
                  IntToStr(WSAGetLastError));
    Synchronize(Update);
    closesocket(sktListenData);
    State := stError;
    Exit;
  end;

  Size    := SizeOf(DataAddr);
  sktData := WSASocket(AF_INET,SOCK_STREAM, IPPROTO_TCP,
                       nil, 0, 0);
  sktData := accept(sktListenData, DataAddr, Size);

  if sktData = INVALID_SOCKET then begin
    Msg := Concat('Call to accept failed. Error ',
                  IntToStr(WSAGetLastError));
    Synchronize(Update);
    State := stError;


    closesocket(sktListenData);
    closesocket(sktData);
    Exit;
  end;

  closesocket(sktListenData);
  // Set the Data buffer and FileStream to avoid warning
  // messages from the compiler.
  Data       := nil;
  FileStream := nil;
  try
    Data     := AllocMem(SizeOf(Buffers));
    Data.buf := Buffers;
    Data.len := SizeOf(Buffers);
    Flags    := 0;
    // Create a new file for writing.
    try
      FileStream := TFileStream.Create('C:\' + NewFileName,
                                       fmOpenWrite);
      Done := False;
      repeat
        Res := WSARecv(sktData, Data, 1, @NoBytesRecv,
                       @Flags, nil, nil);
        if Res = SOCKET_ERROR then begin
          FileStream.Free;
          Msg := Concat('Call to WSARecv failed. Error ',
                        IntToStr(WSAGetLastError));
          Synchronize(Update);
          State := stError;
          closesocket(sktData);
          Exit;
        end;

        if NoBytesRecv = 0 then
          Done := True
        else
          FileStream.Write(Data.Buf^, NoBytesRecv);
      until Done;

      closesocket(sktData);
      Msg := Concat('Finished downloading ',
                    NewFileName,' .');
      Synchronize(Update);
    finally
      FileStream.Free;
    end;
  finally
    FreeMem(Data, SizeOf(Buffers)-1);
  end;
end;

procedure TDataThrd.Update;
begin
  frmMain.memStatusMsg.Lines.Add(Msg);
end;

procedure TDataThrd.OnDataThrdDone(Sender: TObject);
begin
  Terminate;
  Msg := 'Finished!';

  Synchronize(Update);
  frmMain.bbtnGetFile.Enabled := True;
end;

end.
```

End Listing One

By Ray Lischner

Palettes Made Plain

Demystifying Microsoft Windows Palettes

Palettes are one of the least understood aspects of Windows graphics programming. In some ways, Delphi's ease-of-use contributes to this problem; it usually manages all palette issues automatically. Sometimes, however, you need more control over your application's palette. Therefore, whether you write components, games, or client/server applications, you should understand how palettes work in Windows and in Delphi.

Introducing Palettes

A *palette*, in programming terms, is a small array of colors, similar to a painter's palette. The painter has a vast number of colors at their disposal, but working with all those colors is unwieldy. Instead, a painter selects a few colors to work with, and puts only those colors on the palette. A computer graphics palette is similar. When using a palette, your program must display only colors selected from the palette — and no others.

Memory considerations. The most common use for palettes is to save memory. For example, a 1024x1024-pixel bitmap with 24 bits of color information per pixel takes up 3MB of memory. A palette can reduce that by two-thirds, to 1MB. The way a palette works is to keep a table of 256 colors, and each pixel in the bitmap refers to one of the palette entries. Each palette entry has a full 24 bits of color information — 8 bits each for the colors red, green, and blue. Because the bitmap contains only palette indices, it's smaller than the bitmap that contains full color information. An index into a 256-color palette takes up 8 bits instead of 24, making the bitmap one-third the size of the original bitmap.

The drawback to using a palette is that you have a limited number of colors from which to choose. In the previous example, the bitmap contains over a million pixels, and each pixel might have a different color. When using the palette, though, the bitmap is limited to 256 colors. The saving grace is that the human eye cannot distinguish a million colors, so you don't need the full color information for every pixel. On the other hand, the eye *can* identify more than 256 colors, so using a palette slightly compromises image quality in exchange for a smaller bitmap size.

Microsoft defines several different bitmap formats that Windows supports. A bitmap might have a palette with only 16 colors (4 bits per pixel) or 256 colors (8 bits per pixel). If you don't mind the extra size, you can also create bitmap files that store full color information for each pixel.

Video adapters. Windows also uses a palette for certain video adapters. Just as you might want to save disk space in a bitmap file, saving video memory is also important. Video RAM costs more than normal system RAM, so anything to reduce the amount of RAM your video card needs can help reduce its cost. Thus, many video adapters use palettes.

For example, suppose your video card has 1MB RAM. At 640x480 pixels, it can store 24 bits of color information per pixel. But that's not many pixels on the screen. Run Delphi, open the Object Inspector, a form, and the source window, and try to make all these windows visible at the same time. The screen is too small for that much information. I like to use 1024x768 pixels, which gives me more screen real estate, but only 8 bits per pixel. You can't fit any reasonable color in only 8 bits, so the video adapter uses a 256-entry color palette. In the palette, the video card typically uses 18 bits per pixel (6 bits each for red, green, and blue), but there's no reason why your card couldn't use more or fewer bits in the palette. The important thing is that each pixel you see doesn't have a real color stored in the video RAM, just an index into the video card's palette. The palette stores the real color information. For this reason, a palette is also known as a *color lookup table*.

Windows manages the video card's palette, so the more common term for the display palette is the *system palette*. The system palette stores all the colors that you see on your monitor. If you could change an entry in the system palette, every pixel that displays that entry's index would immediately change color, but Windows protects the system palette against inadvertent changes. Sudden changes to the system palette can be distracting and annoying. With the proper setup, however, you can make abrupt changes to the system palette. If you do it carefully, you can achieve some interesting effects, as you will learn later in this article.

Video drivers. Because everything that Delphi displays must go through your video adapter, you must be sure you're using the most up-to-date video drivers for your video card. The most common source of problems when using palettes is not your program, but the video drivers. If you notice any strange video behavior, the first thing you should check is the version and date of your video drivers. Get the most recent drivers from the vendor, and try your program again.

Palettes and Microsoft Windows

Now that you know the basics of color palettes, the next step is to learn how Windows deals with palettes. The first lesson is that Windows uses a palette only for video cards that display 256 colors (8 bits per pixel). Plain VGA displays a fixed set of 16 colors, and you cannot change those colors, so Windows doesn't need or use a palette. With more than 8 bits per pixel, Windows stores the full color information for each pixel, whether it's 15 bits, 16 bits, or 24 bits per pixel.

Windows lets your application use palettes even if the video card does not. This makes your programming job easier. The same code works with any video card, whether or not it uses a system palette. There are times, though, when you need to know whether the system uses a real palette. To do that, check the video adapter's *RasterCaps* capability by calling the Windows API function *GetDeviceCaps* and testing for the *Rc_Palette* capability, as shown here:

```
if (GetDeviceCaps(Canvas.Handle, RasterCaps) and
    Rc_Palette) <> 0 then
  UsesPalette := True;
```

Notice that the first argument to *GetDeviceCaps* is the handle of a canvas. The canvas should be a form or other control's canvas, not a bitmap's canvas. You might call *GetDeviceCaps* in the form's *OnCreate* handler, for example. Even if the video card doesn't use a palette, your program might need to work with bitmap palettes, so don't put away this article yet.

Sharing the palette. If your video card uses a system palette, you know it can display, at most, 256 colors at one time. Windows reserves 20 of those colors for its own use. The remaining 236 colors are shared by every application. If one program uses all 236 colors, the other programs are left out in the cold. To ensure that every program gets a fair shot at displaying its own colors, Windows has a set order in which applications contend for the system palette.

Windows always starts with the foreground application. This program gets to fill the system palette with as many colors as it wants. Then, Windows asks every other application to fill the rest of the system palette, going backward in z-order. When an application tries to fill the system palette with its colors, it's known as *realizing a palette*. If an application has multiple windows open, Windows starts with the active window, then goes to the window behind that, and so on, going backward through all the application's windows. Then it proceeds to the next application.

When an application realizes its palette, it only gets to set colors to palette slots that haven't been filled. If this isn't possible, Windows maps the requested palette entry to the closest matching color. Eventually, every application gets to realize its palette, but the last application gets the last pick of the system palette entries. If a foreground application grabbed most of the palette entries, background applications might not look right. Windows does its best to match requested colors with colors already in the palette, but its success depends entirely on the colors used by the foreground application.

Figure 1 illustrates how palette realization and color matching works. The first application gets to fill the first few slots of the system palette (after the standard Windows colors) with its choice of colors. When the next application realizes its palette, Windows matches its request to colors already in the palette, and creates new palette entries for unmatched colors. The last application matches what colors it can, and creates as many new entries as it can, but the system palette fills before Windows can realize the last few colors. In this case, Windows does its best to match the colors against similar colors that are already in the system palette.

The reserved 20 colors guarantees that the basic colors (black, red, cyan, yellow, white, etc.) are always available. Even if another program is displaying 236 finely graded shades of puce, your application in the background will be able to display the basic Windows colors correctly. Thus, you should always try to stick with these basic colors. Figure 2 shows the standard colors, including literals for the color names that Delphi defines.

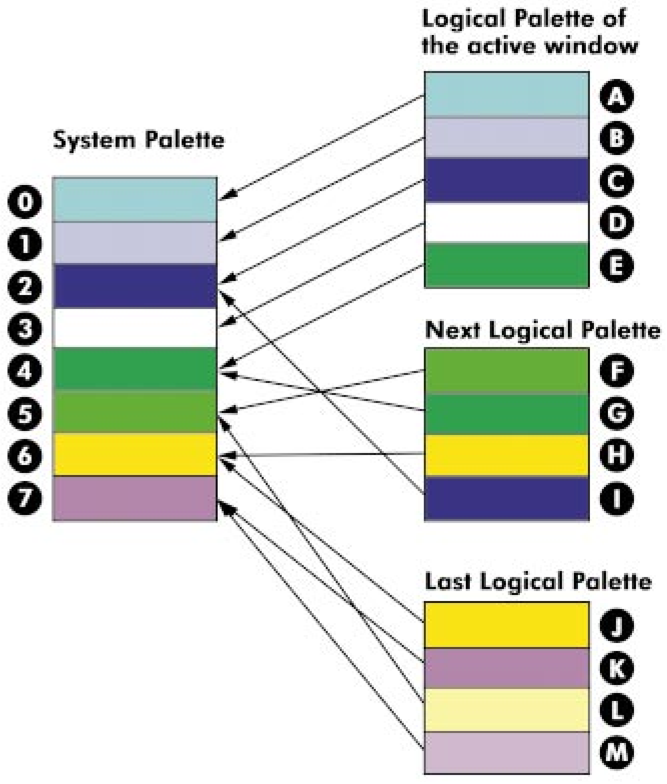
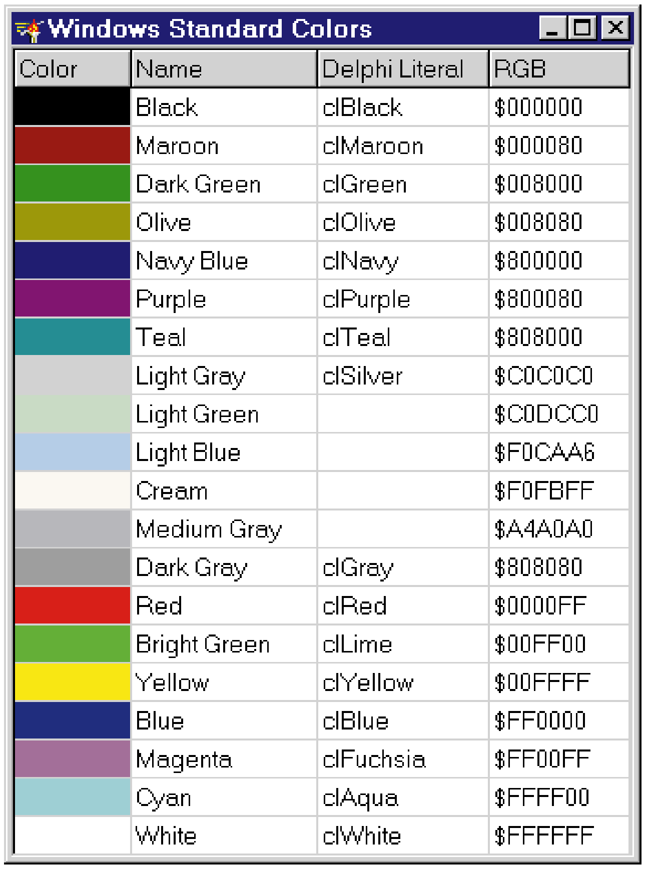


Figure 1: Mapping palettes to the system palette.


Windows Standard Colors

Color	Name	Delphi Literal	RGB
⬛	Black	clBlack	$000000
🟥	Maroon	clMaroon	$000080
🟩	Dark Green	clGreen	$008000
🟨	Olive	clOlive	$008080
🟦	Navy Blue	clNavy	$800000
🟪	Purple	clPurple	$800080
🟦	Teal	clTeal	$808000
⬜	Light Gray	clSilver	$C0C0C0
🟩	Light Green		$C0DCC0
🟦	Light Blue		$F0CAA6
⬜	Cream		$F0FBFF
⬜	Medium Gray		$A4A0A0
⬜	Dark Gray	clGray	$808080
🟥	Red	clRed	$0000FF
🟩	Bright Green	clLime	$00FF00
🟨	Yellow	clYellow	$00FFFF
🟦	Blue	clBlue	$FF0000
🟪	Magenta	clFuchsia	$FF00FF
🟦	Cyan	clAqua	$FFFF00
⬜	White	clWhite	$FFFFFF

Figure 2: The 20 standard Windows colors.

When you switch applications, Windows starts with a fresh system palette and gives the new foreground application a chance to realize its palette. This can change all the colors displayed, which might cause an annoying flicker. This happens most often when the previous or new foreground application uses most or all of the system palette. You can do your part to avoid this flicker by using as little of the system palette as your application allows.

One way you can minimize the number of colors your application needs is to test how many distinct colors the video card can display. Most 256-color video adapters use a color resolution of 18 bits per pixel, or 6 bits each for red, green, and blue. For example, this means the video card can display *at most* 64 (or 2^6) distinct shades of gray. To make a complete gradation of all possible shades of gray, you need to take up only 64 entries in the system palette, not all 236. To test the color resolution, ask for the *ColorRes* capability from *GetDeviceCaps*.

Palette Handles

To use a palette in Windows, you must create the palette; Windows then gives you a handle for the palette. To create a palette, start with a logical palette, which is the array of colors that make up the palette. You need to allocate a *TLogPalette* record dynamically because of the unusual way this type is defined. It uses an old C hacker's trick, namely, an array of one element. You can allocate a larger array using *GetMem*, but the record definition prevents you from using range checking. After you set up the logical palette, call the Windows API function *CreatePalette*, which returns a palette handle.

The listing in Figure 3 shows an example of creating a logical palette that displays 64 distinct shades of gray. It then creates a palette handle from the logical palette. Once you've created the palette handle, you can free the logical palette. Keep the palette handle until you're done using it, then use the form's *OnDestroy* handler to destroy the palette handle by calling *DeleteObject*.

You can also set a flag for each palette entry to do special things with the palette. One useful option is *Pc_Explicit*, which lets you get the actual color that the Windows system palette stores. You can use this information to display the system palette, which is a helpful debugging tool. When you're writing a program that sets its own palette, you can examine the system palette to learn whether your application is setting up the palette correctly. The PalView project that accompanies this article demonstrates how to view the system palette (see end of article for download details).

You don't have to call *CreatePalette* to get a palette handle. When Delphi loads a bitmap that contains a palette, it automatically creates a palette handle for the bitmap. You can refer to the

```
procedure TForm1.FormCreate(Sender: TObject);
var
  LogPal: PLogPalette;
  Gray: Byte;
  I: Integer;
  BitsPerPixel: Integer;
begin
  // Determine the number of bits per pixel.
  if (GetDeviceCaps(Canvas.Handle, RasterCaps) and
      Rc_Palette) <> 0 then
    BitsPerPixel := GetDeviceCaps(Canvas.Handle, ColorRes)
  else
    BitsPerPixel := GetDeviceCaps(Canvas.Handle,Planes) *
                    GetDeviceCaps(Canvas.Handle,BitsPixel);

  // Divide by 3 to get the number of distinct shades of
  // each color element: red, green, blue. Then determine
  // the number of colors.
  NumShades := 1 shl (BitsPerPixel div 3);

  // Allocate the logical palette. The LogPal record
  // already has room for one color, so add enough memory
  // for the remaining colors.
  GetMem(LogPal, SizeOf(LogPal) + (NumShades-1) *
                 SizeOf(TPaletteEntry));
  try
    LogPal.palVersion := $300;  // Required by Windows.
    LogPal.palNumEntries := NumShades;
    for I := 0 to Pred(NumShades) do begin
      // Use a linear gray scale for simplicity. In a real
      // graphics program, you should use a more
      // sophisticated gray scale because the human eye
      // does not respond linearly.
      Gray := I * 255 div NumShades;
      {$R- turn off because TLogPalette is defined poorly.}
      LogPal.palPalEntry[I].peRed   := Gray;
      LogPal.palPalEntry[I].peGreen := Gray;
      LogPal.palPalEntry[I].peBlue  := Gray;
      LogPal.palPalEntry[I].peFlags := 0;
      {$R+}
    end;
    Palette := CreatePalette(LogPal^);
    if Palette = 0 then
      RaiseLastWin32Error;
  finally
    FreeMem(LogPal);
  end;
end;
```

Figure 3: Creating a palette with 64 shades of gray.

TBitmap.Palette property to get the bitmap's palette handle. When using a bitmap's palette, remember that the bitmap and its palette are intimately related. If you use the bitmap with a different palette, Windows will not display the bitmap correctly. A common mistake in Delphi programs is to copy a bitmap, using, say, *TCanvas.CopyBitmap*, when the source and destination canvases use different palettes. Unless you know that they use the same palette, you should call *StretchDIBits* instead.

Working with bitmap palettes is easier if you understand the difference between a device-dependent bitmap (DDB) and a device-independent bitmap (DIB). Delphi 1 and 2 always use DDBs, but Delphi 3 will use a DIB if possible. The details of working with DDBs and DIBs, however, is beyond the scope of this article.

Delphi and Palettes

Once you have a palette handle, then what? Delphi can use the palette handle and automatically realize that palette for you. When your application becomes active, Windows asks it to realize its palette. When another application becomes active, Windows tells your application that it's no longer active, and it can realize its palette as a background application. Delphi handles these details for you. All you need to do is override your form's *GetPalette* method to return a palette handle. By default, this function returns zero, which means your form doesn't have a palette.

```
procedure TForm1.FormPaint(Sender: TObject);
var
  I: Integer;
  Rect: TRect;
  Top: Integer;
  OldPal: HPalette;
begin
  // Tell Windows which palette to use when drawing
  // the rectangles.
  OldPal := SelectPalette(Canvas.Handle,GetPalette,False);
  try
    // Fill a rectangle for each horizontal stripe. The
    // horizontal limits are fixed, and update the top and
    // bottom in the loop.
    Rect.Left := 0;
    Rect.Right := ClientWidth;

    // To avoid gaps in coverage, increment Top as
    // the top of the next stripe.
    Top := 0;
    for I := 1 to NumShades do begin
      Canvas.Brush.Color := PaletteIndex(I - 1);
      Rect.Top := Top;
      // The next top is the current bottom.
      Top := I * ClientHeight div NumShades;
      Rect.Bottom := Top;
      Canvas.FillRect(Rect);
    end;
  finally
    // Always restore the old palette.
    SelectPalette(Canvas.Handle, OldPal, True);
  end;
end;
```

Figure 4: Painting with the gray-shade palette.

The listing in Figure 4 shows an example of a form that displays a color gradient as its background. Notice how it calls *PaletteIndex* to ensure the colors it paints are in the palette. You have four ways to specify a color in Delphi. The first way is to use one of Delphi's special color names, such as *clRed*, *clWhite*, or a system color name, such as *clWindowText*. Another is to use an explicit red, green, and blue combination, by calling the *RGB* function. If you want to make sure you use a color from your palette, call *PaletteRgb* with the same arguments. If you know which palette entry you want, you can use *PaletteIndex*.

If you are writing a new control, you can override the *GetPalette* method for the control. However, if a form has more than one control that defines its own palette, you will have problems, because each control is trying to realize its palette as the main window's palette. As a component writer, you never know what other controls might be on the form. Thus, you should do your best not to use the system palette. If you must, try to use as few colors as possible. Otherwise, you might be in a situation where another control gets all of its colors in the system palette, but your control gets no palette entries. When that happens, your control will use its own palette indices, but those indices will refer to colors from the other control's palette. The results are uglier than you can imagine.

If you decide that your component or form must use its own palette, say, for a color gradient, you can create the palette and return the handle from *GetPalette*, but you need to take one more step when painting your control. Although Delphi automatically arranges for Windows to use your palette, you still need to select the palette before using it. In your control's *Paint* method, or form's *OnPaint* handler, call the Windows API function *SelectPalette*. You must pass a canvas handle as the first argument. A canvas has exactly one palette, from which it draws its colors. When you select your palette, the canvas deselects the previous palette. You must restore the old palette before your *Paint* method returns. Use a **try..finally** block to guarantee the palette is properly restored. The listing in Figure 5 shows a simple gradient component's *Paint* method, illustrating this use of *SelectPalette*. Figure 6 depicts a form using the gradient control.

```
procedure TGradient.Paint;
var
  I: Integer;
  X, Y: Integer;      // Current position on canvas.
  Rect: TRect; // Rectangle for filling 1 band in gradient.
  OldPal: HPalette;  // Old palette.
  Red, Green, Blue: Byte;
begin
  Rect := ClientRect;
  Y := 0;
  X := 0;
  OldPal := SelectPalette(Canvas.Handle,GetPalette,False);
  try
    for I := 0 to NumColors-1 do begin
      {$R-}
      with LogPalette.palPalEntry[I] do
        Canvas.Brush.Color :=
          PaletteRgb(peRed, peGreen, peBlue);
      {$R+}
      if Orientation = goVertical then begin
        Rect.Top := Y;
        Y := MulDiv(ClientHeight, I + 1, NumColors);
        Rect.Bottom := Y;
        end
      else begin
        Rect.Left := X;
        X := MulDiv(ClientWidth, I + 1, NumColors);
        Rect.Right := X;
      end;
      Canvas.FillRect(Rect);
    end;
  finally
    SelectPalette(Canvas.Handle, OldPal, False);
  end;
end;
```

Figure 5: A gradient control's *Paint* method.

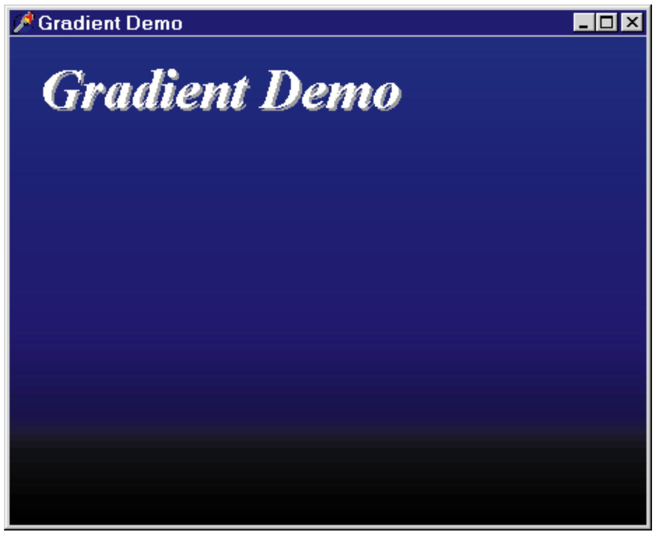


Figure 6: Using the gradient control.

Palette Tricks

There's one case where it makes sense for a control to seize at least part of the palette. Earlier in this article, I wrote that a palette can help you achieve interesting effects. The most common palette trick is to animate the palette. In palette animation, you deliberately change colors in the system palette to change the colors on the display without repainting the form or control. Palette animation requires some extra work, though.

To start with, you must inform Windows of your intentions. When you create the palette, you must reserve the palette entries that you will animate. Windows will prevent other applications from using those palette entries, even if they have the same colors. This prevents your application's tricks from interfering with the colors in other applications. To reserve a palette entry, set *peFlags* to *Pc_Reserved* when creating the logical palette.

```
// Shift the colors to animate the palette. Just rotate all
// the colors in the palette by one step.
procedure TMovingGradient.ShiftColors;
var
  Tmp: TPaletteEntry;
  I: Integer;
begin
  Assert(LogPalette <> nil);

  {$R-}
  Tmp := LogPalette.palPalEntry[0];
  for I := 0 to LogPalette.palNumEntries-2 do
    LogPalette.palPalEntry[I] :=
      LogPalette.palPalEntry[I+1];
  LogPalette.palPalEntry[LogPalette.palNumEntries-1]:=Tmp;
  {$R+}
end;

// Animate the control by shifting the colors and then
// telling Windows to use the new colors.
procedure TMovingGradient.Animate(Sender: TObject);
var
  OldPal: HPalette;
begin
  GetPalette; // Make sure the palette has been created.
  ShiftColors;

  OldPal := SelectPalette(Canvas.Handle, Palette, False);
  try
    AnimatePalette(Palette, 0, LogPalette.palNumEntries,
                   @LogPalette.palPalEntry[0]);
  finally
    SelectPalette(Canvas.Handle, OldPal, False);
  end;
end;
```

Figure 7: A sliding gradient effect using palette animation.

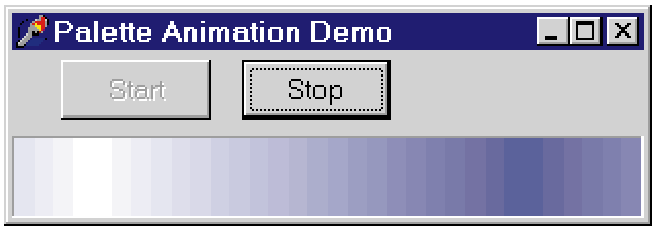


Figure 8: An illustration of the sliding gradient effect.

The *AnimatePalette* API function changes colors in the reserved entries of the system palette. The listing in Figure 7 shows how you can use this trick to make a sliding gradient effect. Figure 8 shows what it looks like at run time.

Most of the palette functions work the same on systems with palettes as on systems without palettes. Windows and Delphi hide the differences from you, except with *AnimatePalette*. You can call *AnimatePalette* on any system, but if the computer isn't running in 256-color mode, Windows will not have a system palette.

Conclusion

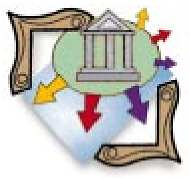
Delphi simplifies the use of palettes, but you may still need to understand how to use them correctly. Bitmaps use palettes to save disk space, and video adapters use palettes to save video memory. Delphi manages the Windows system palette, which Windows uses if the video card displays only 256 colors. You can use palettes in a custom control or in an application. You can even play tricks with palettes, such as using palette animation.

Delphi takes care of most of the burden of using palettes. Your component or application just needs to override the *GetPalette* method to return a palette handle. You can get a palette handle by creating a logical palette with *CreatePalette*, or you can return the palette handle of a bitmap by using the *TBitmap.Palette* property.

When using palettes, you must realize that the palette entries are a scarce resource, and every application competes for those precious few colors. Windows reserves 20 colors for its own use, leaving 236 palette entries to be shared among all applications. If your application needs only 20 extra slots, make sure you use only 20 — don't gobble up all 236. Windows does its best to meet the conflicting demands of every application, but when a new application comes forward, it gets priority in the system palette, possibly causing other applications to lose their old palette entries. This causes a distinct flicker as Windows repaints every application with its new colors. You can minimize this distraction by using standard colors whenever possible, and keeping your application's or component's palette demands to a minimum. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUL\DI9807RL.

Ray Lischner is the author of *Secrets of Delphi 2* [Waite Group Press, 1996] and *Hidden Paths of Delphi 3* [Informant Press, 1997]. He teaches computer science at Oregon State University and provides consulting and training in Delphi, Java, and Smalltalk. Catch his presentation at the 9th Annual Borland Conference, and learn how easy it is to customize Delphi's IDE. You can contact Ray at delphi@tempest-sw.com.



By Binh Ly

COM Callbacks

Part II: Connectable Objects

Understanding Connectable Objects, also known as *connection points*, is really not that difficult. What makes it sound complex is the introduction of several technical terms, such as sinks, sources, incoming interfaces, outgoing interfaces, etc. The purpose of this article (the second of a two-part series) is to familiarize you with these terms, and provide an implementation of the connection points methodology using the Chat application from last month's article as an example.

The basic idea behind connection points is for client objects to be able to easily communicate with server objects, and vice versa, using a standard protocol. The protocol works this way: A client object asks a server object if it knows anything about a particular callback interface; the server object can then either say it does or it doesn't. If the server object says it does, the client can then say "Here's my callback interface. Sign me up and let's start talking to each other." From that point, the client and the server can start calling each other's methods.

```
IConnectionPointContainer = interface
  function EnumConnectionPoints(
    out enum: IEnumConnectionPoints): HResult;
  function FindConnectionPoint(
    const iid: TIID; out cp: IConnectionPoint): HResult;
end;

IConnectionPoint = interface
  function GetConnectionInterface(out iid: TIID): HResult;
  function GetConnectionPointContainer(
    out cpc: IConnectionPointContainer): HResult;
  function Advise(const unkSink: IUnknown;
                  out dwCookie: Longint): HResult;
  function Unadvise(dwCookie: Longint): HResult;
  function EnumConnections(
    out enum: IEnumConnections): HResult;
end;
```

Figure 1: The *IConnectionPointContainer* and *IConnectionPoint* interfaces.

What's nice about this idea is that it's extremely flexible. The client doesn't need to know any specifics about a particular server object being able to support a particular callback interface. The client would simply ask any imaginable server object if it supports an interface that the client can implement, and, if the server object says yes, the client can say "Hey! lets make a deal." Otherwise, the client can say "No deal!" and move on to asking another server object.

In COM parlance, the callback interface (from the server's point of view) is called an *outgoing interface* because it's an interface that is implemented in the client to be used by the server, as opposed to an incoming interface, which is implemented in the server to be used by the client. A server object that supports at least one outgoing interface is called a *Connectable Object*. For the client to connect a callback interface to a Connectable Object, the Connectable Object must implement a connection point for that specific interface. It is not uncommon for a server object to support more than one outgoing interface; thus, it may need to implement several connection points for various clients.

In this connection points model, the Connectable Object is often called a *source,* and the client object that implements the callback interface is called a *sink*.

Connection Points Objects and Interfaces

COM objects implement connection points by using the *IConnectionPointContainer* and *IConnectionPoint* interfaces (see Figure 1). A client object that wishes to connect to a server object first queries the server object for an *IConnectionPointContainer* interface.

If the server passes back a valid *IConnectionPointContainer*, the client then calls the *FindConnectionPoint* method passing into the *iid* parameter the interface ID (IID) of the outgoing interface that it implements. If the server supports that interface, it passes back an *IConnectionPoint* pointer in the *cp* parameter, which serves as the connection point for the client to use.

At this point, the client has established that the server can support its sink interface. The only remaining thing to do is to pass a pointer to that interface to the server. Using the *cp* parameter, the client now calls *IConnectionPoint.Advise*, passing into *unkSink* a pointer to its sink interface's *IUnknown*. *Advise*'s *dwCookie* parameter is passed back from the server to the client, which serves as a unique ID that the client needs when it later disconnects from the connection point. *IConnectionPoint*'s *UnAdvise* method allows the client to disconnect its sink interface as a means to terminate its connection from the server. *UnAdvise* takes into the *dwCookie* parameter the unique ID the client got earlier by calling *Advise*.

Essentially, a connection point container (*IConnectionPointContainer*) is just a list of all connection points that a server object supports. This means that a server object can support a virtually unlimited number of outgoing interfaces, each defined by a specific connection point. A connection point is also capable of supporting an unlimited number of clients. As long as the server can pass back a unique value into the *dwCookie* parameter of *IConnectionPoint.Advise*, any number of clients can connect to a single connection point, where each of them is uniquely identifiable using the *dwCookie* value.

We won't go further into the details of the rest of the methods in the connection points interfaces because they're not very important for our purposes. You can always look them up on MSDN online at http://www.microsoft.com/msdn/.

You're probably thinking that implementing all these connection points interfaces will take more work than the hand-coded callback interface manager we studied last month. Well, you may be surprised to know that Delphi already provides the basic connection points implementations in its *TConnectionPoints* and

```
procedure TChatChannel.Initialize;
begin
  inherited;
  FChatUsers := TConnectionPoints.Create(Self);
  FChatEventSinks := FChatUsers.CreateConnectionPoint(
                       IChatEvent, ckMulti, nil);
end;
```

Figure 2: Initialization for *ChatChannel*'s connection points.

TConnectionPoint classes, which are in the Axctrls unit (*TConnectionPoints* implements *IConnectionPointContainer* and *TConnectionPoint* implements *IConnectionPoint*). With the help of these classes, we're now ready to implement our Chat application using the connection points methodology.

Implementing the Server

For our chat server, we'll need to implement a connection point for the *IChatEvent* sink interface, and a connection point container that contains this single *IChatEvent* connection point. Because *ChatChannel* is the Connectable Object in our server, we'll implement the connection points objects in *TChatChannel*.

Figure 2 shows the initialization code for *ChatChannel*'s connection points. *FChatUsers* is of type *TConnectionPoints*, and is the object that implements *IConnectionPointContainer*. After creating *FChatUsers*, we create the object that implements the *IChatEvent* connection point. This is accomplished by using the *TConnectionPoints.CreateConnectionPoint* method. *CreateConnectionPoint* accepts three parameters:

1) The interface ID of the sink interface that this connection point supports;
2) This connection point's "kind" (*ckMulti* if it supports multiple clients, or *ckSingle* if it supports one client); and
3) An event handler, *OnConnect:TConnectEvent*, that gets triggered every time a client connects to, or disconnects from, this connection point, i.e. when the client calls *IConnectionPoint.Advise* or *IConnectionPoint.UnAdvise*, respectively. In our case, we pass **nil** as the argument because we don't need to do anything special as clients connect or disconnect from *ChatChannel*.

The next step is to implement *ChatChannel*'s *BroadcastMessage* method. If you recall, *BroadcastMessage* is used by *ChatChannel* to broadcast an incoming chat message to all *IChatEvent* clients that are connected to the channel. We previously implemented this by simply cycling through our *TChatUsers* class' items and calling each client's *IChatEvent.GotMessage* method manually. This time, we'll do it using *TConnectionPoint*.

Figure 3 shows our new *BroadcastMessage* implementation. Although it looks complex at first, it really is just an implementation of a simple concept. COM's way of traversing the sink clients of a connection point is through the use of an enumerator. Enumerators are common in COM; this is a good introduction if you haven't seen one before. Basically, an enumerator provides a fairly standard way of accessing collection object items by providing the user with an interface where the user can simply say:

- *Reset* — move to first item in the collection,

- *Next* — move to next item in the collection, or
- *Skip* — move to next item skipping a specified number of items.

The idea is that the user can traverse the entire collection using a call to *Reset* followed by repetitive calls to *Next* until it fails, in which case there are no more available items in the collection.

IEnumConnections is a COM enumerator designed specifically to iterate a connection point's sink interface items. We first get an *IEnumConnections* pointer from a connection point using the *IConnectionPoint.EnumConnections* method. *EnumConnections* uses its single parameter, *Enum*, to return the *IEnumConnections* enumerator. We then use *Enum*'s *Next* method to iterate all sink interface items using a **while** loop. *Next* accepts three parameters:

1) The count of the next consecutive connection point items to retrieve.
2) An array of Delphi's *TConnectData* structure (the length of this array must be pre-allocated by the caller, and must have at least enough space specified in the previous parameter. If the previous parameter is 1, this parameter can just be a single record of the *TConnectData* structure).
3) A pointer to a Longint variable containing the number of items successfully retrieved as a result of the *Next* call.

Taking a close look at *TConnectData*, you'll note it's very similar to our *TChatUser* class, which contains the sink interface and its corresponding unique ID. Calling *Next* fills this *TConnectData* structure with the proper information from each client item in the connection point. Because a connection point stores the sink interface's *IUnknown*, we have to cast it back (*ConnectData.pUnk* as *IChatEvent*) before we can call the *GotMessage* method. Note that we test the return value of *Next* against S_OK so we can determine if we've exhausted all the items in the connection point. After the last item, calling *Next* will return an HResult error code rather than S_OK (see Figure 4).

```
procedure TChatChannel.BroadcastMessage(
  const UserName, Message: WideString);
var
  Enum: IEnumConnections;
  ConnectData: TConnectData;
  Fetched: Longint;
begin
  OleCheck((FChatEventSinks as IConnectionPoint).
    EnumConnections(Enum));
  while Enum.Next(1, ConnectData, @Fetched) = S_OK do begin
    (ConnectData.pUnk as IChatEvent).GotMessage(
                                 UserName, Message);
    ConnectData.pUnk := nil;
  end;
end;
```

Figure 3: The *ChatChannel* object's *BroadcastMessage* procedure.

```
IEnumConnections = interface
  function Next(celt: Longint; out elt;
             pceltFetched: PLongint): HResult;
  function Skip(celt: Longint): HResult;
  function Reset: HResult;
  function Clone(out enum: IEnumConnections): HResult;
end;

TConnectData = record
  pUnk: IUnknown;
  dwCookie: Longint;
end;
```

Figure 4: The *IEnumConnections* interface and *TConnectData* structure.

```
function TChatChannel.ObjQueryInterface(const IID: TGUID;
                                     out Obj): Integer;
begin
  Result := inherited ObjQueryInterface(IID, Obj);
  if not Succeeded(Result) then
    if FChatUsers.GetInterface(IID, Obj) then
      Result := S_OK;
end;
```

Figure 5: Delegating *IConnectionPointContainer* for *TChatChannel*.

One last thing: We have to somehow make the client think that *ChatChannel* actually implements *IConnectionPointContainer* because, after all, the client is going to be asking *ChatChannel* for this interface. By using *FChatUsers*, we indirectly implement *IConnectionPointContainer* using a contained object; we're going to extend *ChatUser*'s *IUnknown.QueryInterface* implementation slightly to delegate returning the *IConnectionPointContainer* interface from within *FChatUsers*. Figure 5 shows how this is accomplished by overriding the *ObjQueryInterface* virtual method, which is *TComObject*'s, and ultimately *TChatChannel*'s, *IUnknown.QueryInterface* implementation.

We've completed our server's implementation. We're now ready to take a look at how the client actually connects to our *ChatChannel* object.

Implementing the Client

Implementing the client side of the connection points methodology is simpler than implementing the server. As we studied previously, all the client needs to do is to ask the server object for the *IConnectionPointContainer* interface, call the *FindConnectionPoint* method to retrieve the connection point it's interested in, and call the connection point's *Advise* method to connect to the server.

To disconnect from the server, the client simply follows the previous steps, but it will need to call *UnAdvise* instead of *Advise*. Figure 6 shows how our client connects to, and disconnects from, *ChatChannel*. The rest of the implementation, such as the *TChatEvent* class and how the client handles chat messages, remains the same as we've studied previously.

For the sake of completeness, I have also implemented *TChatChannel*'s *ConnectUser* and *DisconnectUser* methods using the same procedures previously mentioned. This should make our previous version of chat client still able to connect to our connection points version of chat server. Figure 7 shows the implementations of *ConnectUser* and *DisconnectUser* for the *TChatChannel* server object.

Conclusion

This two-part series has demonstrated two methods of implementing call-backs in your Delphi 3 applications. Last month's topic was a hand-coded callback interface manager; this month, we covered Connectable Objects, i.e. connection points.

This article has shown the basic ideas behind the COM connection points methodology. As we've seen, connection points provide an extremely flexible, standard, and generic way for implementing server-object to client-object communication. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUL\DI9807BL.

```
procedure TfrmMain.ConnectUser;
var
  cpChatUsers: IConnectionPointContainer;
  cpChatEventSinks: IConnectionPoint;
begin
  if (FChatConnection = nil) then begin
    FChatConnection := CoChatConnection.Create;

    { Standard connect code for connection points. }
    cpChatUsers := FChatConnection.ChatChannel as
      IConnectionPointContainer;
    cpChatUsers.FindConnectionPoint(
      IChatEvent, cpChatEventSinks);
    cpChatEventSinks.Advise(FChatEvent as IUnknown,
FUserId);
  end;  { if }
end;

procedure TfrmMain.DisconnectUser;
var
  cpChatUsers: IConnectionPointContainer;
  cpChatEventSinks: IConnectionPoint;
begin
  if (FChatConnection <> nil) then begin
    { Standard disconnect code for connection points. }
    cpChatUsers := FChatConnection.ChatChannel as
      IConnectionPointContainer;
    cpChatUsers.FindConnectionPoint(IChatEvent,
                                    cpChatEventSinks);
    cpChatEventSinks.UnAdvise(FUserId);

    FChatConnection := nil;
  end;
end;
```

Figure 6: Connecting to, and disconnecting from, *ChatChannel*.

```
function TChatChannel.ConnectUser(
  const Callback: IChatEvent;
  var UserId: Integer): WordBool;
var
  cpChatUsers: IConnectionPointContainer;
  cpChatEventSinks: IConnectionPoint;
begin
  { Standard connect code for connection points. }
  cpChatUsers := Self as IConnectionPointContainer;
  cpChatUsers.FindConnectionPoint(IChatEvent,
                                  cpChatEventSinks);
  cpChatEventSinks.Advise(Callback as IUnknown, UserId);
  Result := True;
end;

function TChatChannel.DisconnectUser(
  UserId: Integer): WordBool;
var
  cpChatUsers: IConnectionPointContainer;
  cpChatEventSinks: IConnectionPoint;
begin
  { Standard disconnect code for connection points. }
  cpChatUsers := Self as IConnectionPointContainer;
  cpChatUsers.FindConnectionPoint(IChatEvent,
                                  cpChatEventSinks);
  cpChatEventSinks.UnAdvise(UserId);
  Result := True;
end;
```

Figure 7: *ChatChannel*'s *ConnectUser* and *DisconnectUser* methods.

Ever since Delphi 1 debuted, Binh Ly has found Windows programming to be extremely rewarding and a lot of fun. Binh currently works as a Systems Analyst at Brickhouse Data Systems, Inc., developing core functionality for Brickhouse's Business Object Architecture (BOA) application development framework. Binh can be reached at bly@brickhouse.com.

By Major Ken Kyler and Alan C. Moore, Ph.D.

Delphi and TAPI

Part I: An Introduction to Telephony Programming

Line communications in general, and telephony in particular, have come a long way since Windows 3.x. In the days of 16-bit Windows development, you had to work at a low level, directly with the serial port. In 32-bit communications programming, there are new application programming interfaces (APIs) that make the work considerably easier. The four essential ones are the Win32 Communications API, the WAVE API, the Messaging API (MAPI), and the Telephony API (TAPI).

TAPI, in particular, has added a great deal of convenience, both for the programmer and the user. In this article, we'll examine some of the basic TAPI functions in detail, showing you how to use them to initiate and manage phone calls. Next month, we'll build on the basic information presented here and discuss more advanced TAPI techniques. Let's begin with a brief overview of 32-bit Windows communications and TAPI's role.

Windows 95 Communications APIs

While the new APIs help a lot, communications programming for Windows 95 is hardly trivial. We need to understand how the communications process works, and we need to understand the role of each of these APIs. First, we'll examine the role of each of the communications APIs.

The Win32 Communications API is used to send or receive data in a streaming, interactive mode. It's particularly useful for setting up the configuration and sending error-sensitive, non-time-critical data. For example, you would use this API for editing shared documents, playing multi-player games, and similar interactive situations. The WAVE API is used to configure, and send or receive error-tolerant, time-sensitive audio data in real-time. MAPI is used to send and receive files, faxes, e-mail, or any other non-interactive, message-based transmissions. Our focus, TAPI, is used to connect, control, and disconnect telephone calls.

It's important to understand that these four APIs are not mutually exclusive; they're often used in conjunction depending on the task. TAPI, however, is at the heart of it all; it is "Call Central" in Windows 95. This series of articles is designed to give you a basic understanding of how to use TAPI to place (dial) a simple telephone call and to provide you with some basic tools on which to build. While Borland/INPRISE didn't include a unit for TAPI, programmers involved with Project JEDI (the Delphi API Library) have converted the C++ headers for us (see the sidebar "Project

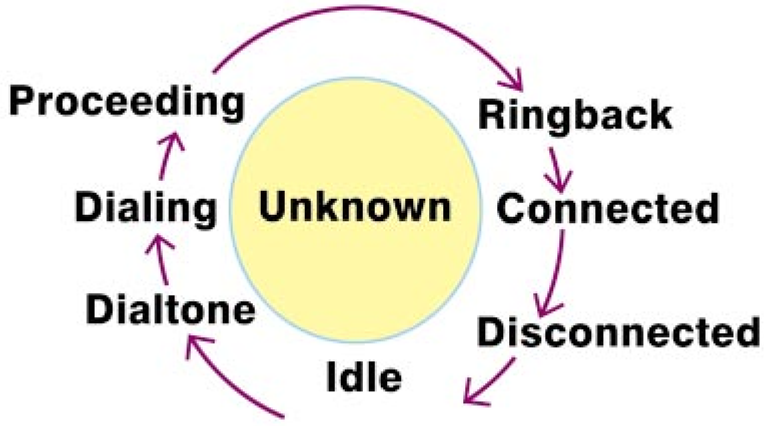


Figure 1: These are the steps involved in an outgoing call.

JEDI" on page 30). That conversion, TAPI.pas, is included with the sample project accompanying this article (see end of article for download details).

Now that we've examined the role of each of the essential APIs, let's take a look at what happens during the life-cycle of a phone call.

The various steps involved in the life-cycle of a telephone call apply to both voice and data calls (see Figure 1):

- Outgoing calls begin life in the IDLE state.
- When you pick up the handset, the call enters the DIALTONE state (assuming the phone is plugged in and the line isn't dead).
- When you start entering a phone number, the call enters the DIALING state.
- After you finish dialing the number, but before the other phone starts ringing, the call is in the PROCEEDING state.
- It enters the RINGBACK state when the called phone starts ringing ...
- ... unless the phone is busy, in which case it enters the BUSY state (if there are teenagers around, this is the normal state).
- As soon as the called party picks up the phone, the call enters the CONNECTED state.
- If you hang up, the call enters the DISCONNECTED state and immediately transitions back to the IDLE state.

When making data, fax, or modem calls, the call may not enter the CONNECTED state immediately when the called modem answers the call. First the modems will negotiate the transfer parameters and protocols, i.e. generate the weird sounds you hear when two modems connect. During this time, the call may enter the UNKNOWN state, or it may remain in the RINGBACK state. Modems have a very difficult time detecting CONNECTED, BUSY, and RINGBACK states on interactive voice calls due to variation in standards between phone switches.

As with outgoing calls, incoming calls start in the IDLE state (see Figure 2). When the phone starts ringing, it enters the OFFERING state. If two modems are trying to connect, the call enters an ACCEPTED state when the call is answered, and they immediately start negotiating transfer protocols and parameters. Once the negotiation is completed, the call enters the CONNECTED state. Hang up and the call enters the DISCONNECTED state, then returns to the IDLE state.

Getting Started

All Windows 95 communications revolve around a call manager, which is also referred to as a telephone service provider. The default call-manager application is "Phone Dialer" (see Figure 3). You can create a custom call manager; in fact, you must if you want to use any of the advanced TAPI functions.

There are two basic ways to initiate a telephone call using TAPI. You can use the function *TapiRequestMakeCall*, or you can use the more elaborate *LineMakeCall*. *TapiRequestMakeCall* handles everything for you by invoking the default call manager. The call manager then handles everything else. The demonstration project included with this article shows both methods. *LineMakeCall* is a different story altogether. As you'll see, it's anything but trivial.

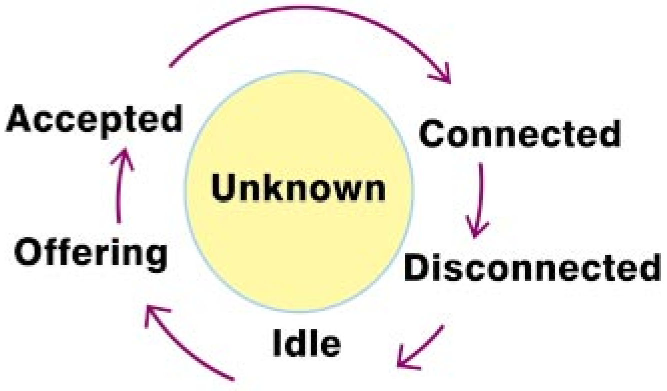


Figure 2: These are the steps involved in an incoming call.

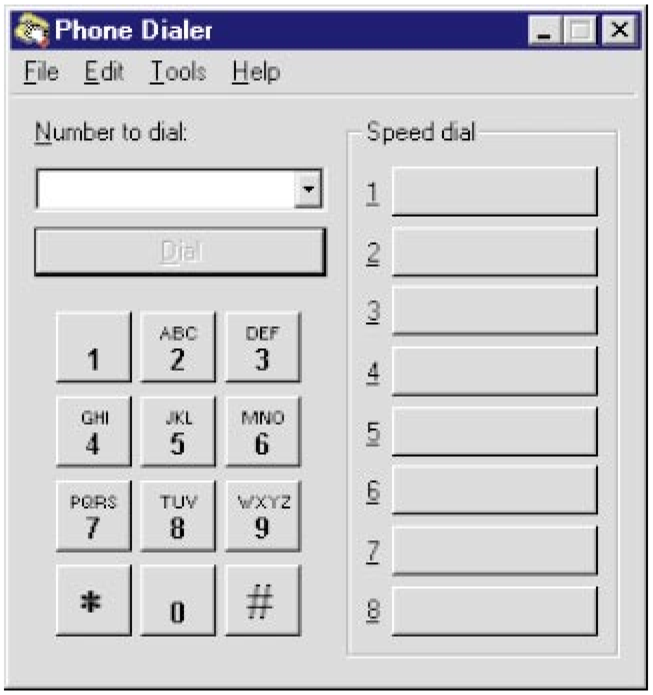


Figure 3: Phone Dialer, an application's default call manager.

The Hard Way

The call manager must do a lot of work. It must create a line, place the call, handle the asynchronous callbacks, and dispose of resources. However, if you're willing to go to the trouble of creating a custom call manager, you will have the basis for a very powerful telephony application. Let's take a look at each of the basic steps previously mentioned: initializing TAPI, placing the call, handling the callbacks, and hanging up and disposing of resources.

Step 1: Initializing TAPI

Initializing TAPI is a two-step process: First you call *LineInitialize*, then *LineNegotiateAPIVersion*. *LineInitialize* initializes TAPI for use by the call-manager application. It initializes the application's use of TAPI.DLL, registers the application's callback mechanism, and returns the number of logical line devices available to the application (using the *lpdwNumDevs* parameter). If you're new to all of this, and are wondering what a line device is, the *Win32 Programmer's Reference* provides this definition: "A line device is a physical device, such as a fax board, a modem, or an ISDN card, that is connected to an actual telephone line. Line devices support telephonic capabilities by allowing applications to send or receive information to or from a telephone network. A line device is the logical representation of a physical line device, one of the two device classes supported by TAPI."

Simply stated, a line device is the *dwDeviceID* parameter required in all *Line...* functions. For example, the TAPI.pas file defines *LineInitialize* as:

```
function LineInitialize(lphLineApp: LPHLINEAPP;
  hInstance: HINSTANCE; lpfnCallback: TLINECALLBACK;
  lpszAppName: LPCSTR; lpdwNumDevs: LPDWORD): LONG;
```

The first parameter, *lphLineApp*, is a handle for this usage instance of TAPI. The second parameter, *hInstance*, is the instance of the call-manager application. The third parameter, *lpfnCallback*, is the address of the callback function. The callback function is used to receive the asynchronous responses from the line. The fourth parameter, *lpszAppName*, is the name of the application originating or accepting a call. If you leave it null, Windows will use the file name of the calling application by default. The last parameter, *lpdwNumDevs*, is the number of line devices available; *LineInitialize* returns this value. (If you want to see what line devices are available on your computer, open the Control Panel and double-click on **Modems**.)

LineInitialize returns zero if successful. If *LineInitialize* returns LINEERR_REINIT, then the application should try *LineInitialize* again. LINEERR_REINIT is a non-fatal error; it simply means "try again." See the *Win32 Programmer's Reference* for an explanation of the other error values. You should always check the return value of *LineInitialize* to determine if there are any configuration problems with the telephony system.

LineNegotiateAPIVersion tells the call-manager application which version of TAPI to use. Because it has several versions — each with different capabilities — TAPI is one of the few Windows APIs that requires version negotiation. Windows 95 uses TAPI version 1.4, and Windows NT uses version 2.0. Microsoft has recently released a beta of version 2.1. Newer versions include added or changed features/functions. This function allows the developer to use a particular set of functions and ensure that future versions of TAPI will use those functions.

LineNegotiateAPIVersion is defined as:

```
function LineNegotiateAPIVersion(hLineApp: HLINEAPP;
  dwDeviceID: DWORD; dwAPILowVersion: DWORD;
  dwAPIHighVersion: DWORD; lpdwAPIVersion: LPDWORD;
  lpExtensionID: LPLINEEXTENSIONID): LONG;
```

The first parameter, *hLineApp*, is the TAPI usage handle returned in the call to *LineInitialize*. The second parameter, *dwDeviceID*, specifies the device to be used. If you intend to use more than one device, you must negotiate the API version for each device. Recall that *LineInitialize* returns the number of devices available in the *lpdwNumDevs* parameter. The third parameter, *dwAPILowVersion*, specifies the lowest version of TAPI to use. The fourth parameter, *dwAPIHighVersion*, specifies the highest version of TAPI to use. Windows will select the highest version in this range that is supported, and return it in the fifth parameter, *lpdwAPIVersion*. The last parameter, *lpExtensionID*, identifies TAPI extensions that the driver supports. At present, UNIMODEM doesn't support any extension, so you can ignore, but not omit, this parameter. *LineNegotiateAPIVersion* returns zero if successful.

Unless you explicitly shut down TAPI, you must initialize it only once. If you terminate calls with *LineShutDown*, however, you will need to initialize TAPI again (as we'll see in Step 4).

Project JEDI

The basic TAPI conversions we used in the example program accompanying this article were originally written by Alex Staubo and further developed by a Project JEDI Team (Team One). As I wrote in the December, 1997 *Delphi Informant* "Symposium," Project JEDI, the Delphi API Library, began in response to a query in the COBB DDJ-Thread mailing list: "Why does there have to be such a long waiting period for a new technology's API to become available in Delphi?" Project JEDI is the answer to that question: There doesn't have to be and we can do something about it. Expressed in Delphi terms, Project JEDI can be defined as:

```
ProjectJEDI := TDelphiAPI.Create(DelphiCommunity);
```

A lot has happened since December's "Symposium" piece was written. There are now six complete teams, several of which are in the midst of conversions, including TAPI, MAPI, Direct X V5, and the ODBC/SQL API. Soon there will be ten teams, with more on the horizon. Each team includes converters, testers, and Help-file writers. Ken Kyler and I are part of Team Eight (although TAPI was done by Team One), Ken as a tester and me as a Help-file writer. Most of the other administrative people are involved with teams as well. Everyone's getting their hands dirty!

To find out more about Project JEDI, or to get involved, visit our Web site at http://www.delphi-jedi.org.

— Alan C. Moore, Ph.D.

dwPrivileges Value	Meaning
LINECALLPRIVILEGE_NONE	Application makes outgoing calls.
LINECALLPRIVILEGE_MONITOR	Application monitors incoming/outgoing calls.
LINECALLPRIVILEGE_OWNER	Application owns incoming calls of the types specified in *dwMediaModes* (Figure 5).
LINECALLPRIVILEGE_MONITOR	Application owns incoming calls of the types specified in *dwMediaModes*, but ...
LINECALLPRIVILEGE_OWNER	... if it can't be a call's owner, it wants to be a monitor.

Figure 4: TAPI Call Privileges (*dwPrivileges*) affect how much control an application has.

dwMediaModes Value	Meaning
LINEMEDIAMODE_UNKNOWN	Application handles unclassified calls of unknown media type.
LINEMEDIAMODE_INTERACTIVEVOICE	Application handles interactive voice (human user) calls.
LINEMEDIAMODE_AUTOMATEDVOICE	Voice energy is present on the call. (Voice handled locally by automated application.)
LINEMEDIAMODE_DATAMODEM	Application handles data modem calls.
LINEMEDIAMODE_G3FAX	Application handles group 3 fax calls.
LINEMEDIAMODE_TDD	Application handles TDD (Telephony Devices for the Deaf) calls.
LINEMEDIAMODE_G4FAX	Application handles group 4 fax calls.
LINEMEDIAMODE_DIGITALDATA	Application handles digital data calls.
LINEMEDIAMODE_TELETEX	Application handles teletex calls.
LINEMEDIAMODE_VIDEOTEX	Application handles videotex calls.
LINEMEDIAMODE_TELEX	Application handles telex calls.
LINEMEDIAMODE_MIXED	Application handles ISDN mixed media calls.
LINEMEDIAMODE_ADSI	Application handles ADSI (Analog Display Services Interface) calls.
LINEMEDIAMODE_VOICEVIEW	Call's media mode is VoiceView.

Figure 5: TAPI's Media Modes (*dwMediaModes*) affect different kinds of calls.

Step 2: Placing the Call

After initializing TAPI, open a line with *LineOpen* and dial the appropriate number with *LineMakeCall*. *LineOpen* allows you to specify what communications functions your application can perform, including answering calls, monitoring calls, or initiating calls. It's defined as follows:

```
function LineOpen(hLineApp: HLINEAPP; dwDeviceID: DWORD;
  lphLine: LPHLINE; dwAPIVersion: DWORD;
  dwExtVersion: DWORD; dwCallbackInstance: DWORD;
  dwPrivileges: DWORD; dwMediaModes: DWORD;
  const lpCallParams: LPLINECALLPARAMS): LONG;
```

The first parameter, *hLineApp*, is a handle to the usage instance of TAPI. It was returned by the first call to *LineInitialize* in Step 1. The second parameter, *dwDeviceID*, is an integer value that identifies which logical device to open. The third parameter, *lphLine*, is the handle to the line opened by Windows. The fourth parameter, *dwAPIVersion*, is the API version returned in *LineNegotiateAPIVersion* in Step 1. The fifth parameter, *dwExtVersion*, is the extension version number the application and service provider will use (normally zero). The sixth parameter isn't used by TAPI, so we'll ignore it (it's normally set to zero). The seventh parameter, *dwPrivileges*, tells the call manager how to handle calls on the line. See Figure 4 for the values that can be used. This information, and that in Figures 5 and 6, is based on the *Win32 Programmer's Reference* (see that file for details).

The eighth parameter, *dwMediaModes*, tells Windows what types of calls will be answered or monitored. Please note that this affects incoming calls only. Therefore, it's possible to have 14 applications handling calls — one for each type.

The last parameter, *lpCallParams*, is a pointer to a *TLineCallParams* structure. This is only used if LINEMAPPER is used; otherwise this parameter is ignored and must be set to **nil**. This sets the parameters of the "line." (We won't be working with this now, but in our next article on TAPI, we'll show you how to use it.) *LineOpen* returns zero if successful, or a negative error number if an error occurs. See the *Win32 Programmer's Reference* for the possible return values and a discussion of the error causes.

What have we done so far? We've opened the line, and the application is waiting for us to dial a number. To place the call, we use the *LineMakeCall* function. That's over-simplifying things a bit; first, we must "create" a call. Calls can be managed: You can put them on hold, forward them, and so on. In TAPI, calls and lines are managed separately. The function we use, *LineMakeCall*, is defined as follows:

```
function LineMakeCall(hLine: HLINE; lphCall:LPHCALL;
  lpszDestAddress: LPCSTR; dwCountryCode: DWORD;
  const lpCallParams: LPLINECALLPARAMS): LONG;
```

The first parameter is the handle to the line opened with *LineOpen*. The next parameter, *lphCall*, is a pointer to the call handle. The next parameter, *lpszDestAddress*, is the phone number to dial. The data type is PChar and the phone number must be in canonical format (see the sidebar "Canonical Address

Message	Meaning
LINE_ADDRESSSTATE	Status of an address on an open line has changed. (Use *LineGetAddressStatus* to get current status of the address.)
LINE_CALLINFO	A certain call's information has changed. (Use *LineGetCallInfo* to get current call information.)
LINE_CALLSTATE	Status of specified call has changed. (Used often; see states in Figures 1 and 2.) Use *LineGetCallStatus* to get detailed information about call's status.
LINE_CLOSE	Specified line device has been forcibly closed; associated handles are no longer valid.
LINE_CREATE	New line device has been created.
LINE_DEVSPECIFIC	Provides device-specific information about events related to line, an address, or a call.
LINE_DEVSPECIFICFEATURE	Provides device-specific information about events related to line, an address, or a call.
LINE_GATHERDIGITS	Current buffered digit-gathering request has been terminated or is canceled.
LINE_GENERATE	Current digit or tone generation has been terminated or canceled.
LINE_LINEDEVSTATE	A line device's state has changed. Use *GetLineDevStatus* to get new status.
LINE_MONITORDIGITS	Digit has been detected. Controlled with the *LineMonitorDigits* function.
LINE_MONITORMEDIA	A call's media mode has changed. Controlled with the *LineMonitorMedia* function.
LINE_MONITORTONE	Tone has been detected. Controlled with the *LineMonitorTones* function.
LINE_REPLY	Reports the results of asynchronously completed function calls.
LINE_REQUEST	Reports a new request from another application.

Figure 6: Callback messages used by TAPI.

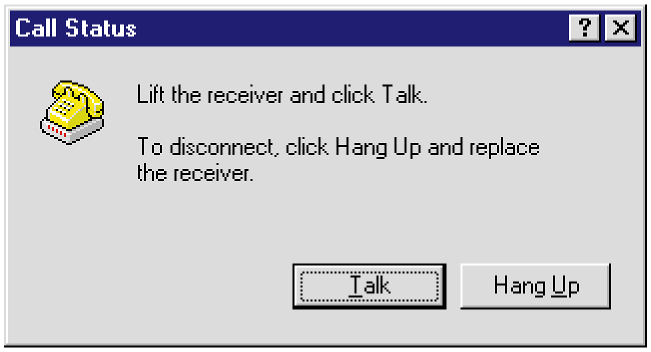

Call Status
Lift the receiver and click Talk.
To disconnect, click Hang Up and replace the receiver.
Talk
Hang Up

Figure 7: Use the Call Status dialog box to get connected.

Formats" on page 33). The fourth parameter, *dwCountryCode*, is the country in which the caller resides (0 for the US).

Lastly, *lpCallParams*, is a pointer to a *TLineCallParams* structure. This specifies the characteristics of the call. We also saw this structure in *LineOpen* where it was used to set the parameters for the "line." Here we set the parameters for the "call." If you're making a data call, you must create this structure and fill the *dwMediaMode* value with LINEMEDIAMODE_DATAMODEM. (We'll discuss this more in the next article.) Otherwise, set it to **nil**. When this value is passed as **nil**, *LineMakeCall* automatically places an interactive voice call regardless of what privileges were set when the line was opened.

Interactive voice calls will display a dialog box like the one shown in Figure 7. When you click on **Talk**, the dialog box disappears and a "connected" message is sent to the callback regardless of the actual state of the call. It also tells the modem to connect the call to the handset. Clicking **Hang Up** does exactly what you think: It disconnects the call, after which the call usually enters the IDLE state. *LineMakeCall* returns zero if successful, or a negative error number if an error occurs (see the *Win32 Programmer's Reference* for details).

It's been pretty easy so far, right? Now things get more involved.

LineMakeCall is asynchronous. The call is complete when the *dwParam2* parameter of the corresponding LINE_REPLY callback message returns zero. If you have a telephone extension to dial, you follow a call to *LineMakeCall* with one to *LineDial*. You must wait for *LineMakeCall* to complete asynchronously before using *LineDial*. *LineDial* is defined as:

```
function lineDial(hCall: HCALL; lpszDestAddress: LPCSTR;
  dwCountryCode: DWORD): LONG;
```

The first parameter is the handle to the open call returned by *LineMakeCall*. The second parameter is the number to call (data type: PChar; canonical format). The third parameter is the country code (zero if you're calling from the US). You can make as many calls as necessary to *LineDial*. If you're going to use *LineDial*, you must terminate *lpszDestAddress* with a semicolon in both *LineMakeCall* and *LineDial*. This indicates that more dialing will be done using *LineDial*. The next step involves callback functions.

Step 3: Handling the Callbacks

The callback function handles the messages from asynchronous operations such as *LineMakeCall*. Keep this in mind when writing callbacks: All callbacks occur in the application's context. The callback must reside in a dynamic link library (DLL) or the application module:

```
TLINECALLBACK = procedure(hDevice, dwMessage, dwInstance,
                          dwParam1, dwParam2, dwParam3: DWORD);
```

The *hDevice* parameter specifies a handle to a line device or a call associated with the callback. The nature of this handle (line handle or call handle) can be determined by *dwMessage*. The *dwMessage* parameter specifies a line or call-device message. The *dwInstance* parameter specifies the callback instance data passed back to the application in the callback. Note that this is not interpreted by TAPI. Lastly, the parameters *dwParam1*, *dwParam2*, and *dwParam3* specify parameters for the message. Again, see Figure 6 for the possible message events.

Here, we'll be concerned with three callback messages: LINE_LINEDEVSTATE, LINE_CALLSTATE, and LINE_REPLY. LINE_LINEDEVSTATE has one very important return message, LINEDEVSTATE_REINIT, that indicates the configuration of a line device has changed. To become aware of this change, the application must reinitialize its use of TAPI. Generally, this message is sent when the telephony system has been changed through the Windows Control Panel.

Two other messages are related to PCMIA cards: LINEDEVSTATE_INSERVICE and LINEDEVSTATE_OUTOFSERVICE. They are sent when a PCMIA card is plugged in or removed.

Canonical Address Formats

Phone number formats vary a bit from one country to another. Fortunately, Windows 95 provides a standard international format called the *canonical address format* that can represent any phone number, anywhere in the world. TAPI functions use this format, so it's important to understand it. In the sample project accompanying this article, we've encapsulated this format in a MaskEdit control to force adherence to its requirements.

What are the requirements and the structure of the canonical address format? The format always begins with the plus character (+) followed by the digits identifying the country, the area code (if there is one), and the local number. If there's an area code, it must be in parentheses. As an example, here's the mask we used for the MaskEdit control in this project:

```
!9 \(999\) 000-0000;1;_
```

You'll notice there's no plus sign at the beginning. The program appends that character to the beginning of the string. Also note the literal parenthetical characters, `'\('` and `'\)'`, to force adherence to the area code format. The above mask would translate into an actual phone number as follows:

_ (800) 123-4567

Now, at least, we can send phone numbers between Windows API functions. Dialing a phone number often involves more, however. You may have to add digits like 1, 8, or 9 for long distance, or to get an outside line. So we also need a dialable address — one that will be recognized by the phone system we're hooked into. The sample project handles those details for us, as the comments in it show.

— *Alan C. Moore, Ph.D.*

The LINE_CALLSTATE message is sent to the callback function when the status of the call has changed. Many LINE_CALLSTATE messages will be received during the duration of a call. Four deserve special attention: LINECALLSTATE_IDLE, LINECALLSTATE_DISCONNECTED, LINECALLSTATE_RINGBACK, and LINECALLSTATE_BUSY. As stated previously, the BUSY signal and the RINGBACK are very difficult to determine for interactive voice calls. Therefore you should not rely on receiving LINECALLSTATE_RINGBACK and LINECALLSTATE_BUSY to spawn critical processes when using interactive voice calls. The LINECALLSTATE_DISCONNECTED message may or may not be detected, but disconnected calls always receive LINECALLSTATE_IDLE.

Recall that we said *LineMakeCall* completes asynchronously? LINE_REPLY returns zero in *dwParam2* if *LineMakeCall* was successful. *LineMakeCall* isn't complete until this message is received. A value of zero in *dwParam2* indicates success; any negative value indicates that *LineMakeCall* failed.

Step 4: Hanging Up and Disposing of Resources

Now you're done chatting with your friends and it's time to hang up. This is the easiest part — if you only have one line open. *LineShutDown* will terminate all calls and shut down TAPI. It's a well-behaved function and cleans up all resources associated with any calls:

```
function LineShutDown(hLineApp: HLINEAPP): LONG;
```

The parameter passed is the handle to the open line returned in the call to *LineInitialize* in Step 1. This terminates the TAPI instance; any further calls must reinitialize TAPI. However, if you have several calls open and being managed by the same application, or if you plan on making more calls, then you should close each one separately. You can do this in one of two ways: 1) call *LineDrop* followed by *LineDeallocateCall*; or, 2) call *LineClose*. *LineClose*, like *LineShutDown*, will free all open resources. These functions are straightforward. See the *Win32 Programmer's Resource* for more information.

The Demonstration Project and a Preview

As you have no doubt seen, even working at the most basic level with TAPI is hardly a trivial matter. To help you better understand the process, we've included a demonstration project (see Listing Two beginning on page 34). Take a few minutes to read through it. It's a bare-bones project designed to show how to make an interactive voice call. The assumption is you will only be placing outgoing, interactive-voice telephone calls.

Next month we'll explain some of the more intricate TAPI functions, such as how to use LINEMAPPER in *LineOpen*, how to get device capabilities, how to enumerate available devices, and how to get modem lights. See you then.

References

- *Communications Programming for Windows 95*, Charles A. Mirho and Andre Terrisse, Microsoft Press, 1996.
- *Windows 95 Multimedia and ODBC API Bible*, Richard J. Simon, Tony Davis, John Eaton, R. Murray Goertz, Waite Group Press, 1996.

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUL\DI9807AM.

Major Ken Kyler is the Air National Guard Systems Analyst for the Defense Integrated Military Human Resources System (DIMHRS). He has been programming with Delphi for two years. He is also a free-lance technical writer with articles published in several Delphi magazines. You can reach him at KylerK@PR.OSD.MIL.

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

Begin Listing Two — TAPI Example Application

```
unit TAPIU_1;

interface

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

type
  TForm1 = class(TForm)
    Memo: TMemo;
    Panel1: TPanel;
    ePhoneNum: TMaskEdit;
    btnDial: TButton;
    Label1: TLabel;
    btnHangup: TButton;
    rbDefault: TRadioButton;
    rbCallManager: TRadioButton;
    procedure btnDialClick(Sender: TObject);
    procedure FormCloseQuery(Sender: TObject;
                             var CanClose: Boolean);
    procedure FormCreate(Sender: TObject);
    procedure btnHangupClick(Sender: TObject);
  private
    function  TapiInitialize: Boolean;
    procedure CreateCallManager;
  public
    { ShutdownCallManager is called from CallBack function,
      so ShutdownCallManager must be declared public. }
    function  ShutdownCallManager: Boolean;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

uses
  Tapi;

var
  FLineApp: HLINEAPP;
  FNumDevs,
  FVersion: Integer;
  FExt: TLINEEXTENSIONID;
  FLine: HLINE;
  FLineCallParams: TLineCallParams;
  FLineOpen: Boolean;
  FHCall: HCALL;
  FCountryCode: Integer;
  // Value of device ID to initialize, 0..NumDevs
  FDev: Integer;

const
  HiVer = $00020000;  // Highest API version wanted (2.0).
  LoVer = $00010004;  // Lowest API version accepted (1.4).

// Handle messages from the line.
procedure LineCallBack(hDevice, dwMessage, dwInstance,
  dwParam1, dwParam2, dwParam3 : DWORD); stdcall;
begin
  with Form1, Memo.Lines do begin
    case dwMessage of
      // Report asynchronous responses.
      LINE_CALLSTATE: begin
        case dwParam1 of
          LINECALLSTATE_IDLE: begin
            Add('LCB (LINE_CALLSTATE): ' +
                'The call is idle; no call exists.');
            ShutdownCallManager;
          end;
          LINECALLSTATE_OFFERING:
            Add('LCB (LINE_CALLSTATE): ' +
                'Call is being offered to the station.');
          LINECALLSTATE_ACCEPTED:
            Add('LCB (LINE_CALLSTATE): ' +
                'The call was offered and accepted.');
          LINECALLSTATE_DIALTONE:
            Add('LCB (LINE_CALLSTATE): ' +
                'The call is receiving a dial tone.');
          LINECALLSTATE_DIALING:
            Add('LCB (LINE_CALLSTATE): Dialing ' +
                Form1.ePhoneNum.Text);
          LINECALLSTATE_RINGBACK:
            Add('LCB (LINE_CALLSTATE): ' +
                'The call is receiving ringback.');
          LINECALLSTATE_BUSY: begin // Difficult to detect.
            case dwParam2 of
              LINEBUSYMODE_STATION:
                Add('LCB (LINE_CALLSTATE): ' +
                    'Busy; called party's station busy.');
              LINEBUSYMODE_TRUNK:
                Add('LCB (LINE_CALLSTATE): ' +
                    'Busy; trunk or circuit is busy.');
              LINEBUSYMODE_UNKNOWN:
                Add('LCB (LINE_CALLSTATE): ' +
                  'Busy; specific mode is unknown.');
            LINEBUSYMODE_UNAVAIL:
              Add('LCB (LINE_CALLSTATE): ' +
                  'Busy; specific mode is unavailable.');
          else
            Add('LCB (LINE_CALLSTATE): ' +
              'Call receiving unidentifiable busy tone.');
          end;
          ShutdownCallManager;
        end;
        LINECALLSTATE_SPECIALINFO:
          Add('LCB (LINE_CALLSTATE): ' +
              'Special information sent by network.');
        LINECALLSTATE_CONNECTED:
          Add('LCB (LINE_CALLSTATE): ' +
              'Call established and connection made.');
        LINECALLSTATE_PROCEEDING:
          Add('LCB (LINE_CALLSTATE): ' +
              'Dialing completed; call proceeding.');
        LINECALLSTATE_ONHOLD:
          Add('LCB (LINE_CALLSTATE): ' +
              'The call is on hold by the switch.');
        LINECALLSTATE_CONFERENCED:
          Add('LCB (LINE_CALLSTATE): Call is a member ' +
              ' of a multi-party conference call.');
        LINECALLSTATE_ONHOLDPENDCONF:
          Add('LCB (LINE_CALLSTATE): Call is on hold ' +
              'while being added to a conference.');
        LINECALLSTATE_DISCONNECTED: begin
          Add('LCB (LINE_CALLSTATE): ' +
              'The line has been disconnected.');
          case dwParam2 of
            LINEDISCONNECTMODE_NORMAL:
              Add(#9 + 'A "normal" disconnect request.');
            LINEDISCONNECTMODE_UNKNOWN:
              Add(#9 +
              'Unknown reason for disconnect request.');
            LINEDISCONNECTMODE_REJECT:
              Add(#9 + 'Remote user rejected the call.');
            LINEDISCONNECTMODE_PICKUP:
              Add(#9 + 'Call picked up from elsewhere.');
            LINEDISCONNECTMODE_FORWARDED:
              Add(#9 + 'Call was forwarded by switch.');
            LINEDISCONNECTMODE_BUSY:
              Add(#9 +'Remote user's station is busy.');
            LINEDISCONNECTMODE_NOANSWER:
              Add(#9 +
              'Remote user's station does not answer.');
            LINEDISCONNECTMODE_BADADDRESS:
              Add(#9 +'Destination address in invalid.');
            LINEDISCONNECTMODE_UNREACHABLE:
              Add(#9 +
                  'Remote user could not be reached.');
            LINEDISCONNECTMODE_CONGESTION:
              Add(#9 + 'The network is congested.');
            LINEDISCONNECTMODE_INCOMPATIBLE:
              Add(#9 + 'Remote user's station ' +
                  'equipment is incompatible');
            LINEDISCONNECTMODE_UNAVAIL:
              Add(#9 + 'Reason for the disconnect ' +
                  'is unavailable');
          end;
        end;
        LINECALLSTATE_UNKNOWN:
          Add('LCB (LINE_CALLSTATE): ' +
              'The state of the call is not known.');
      end;
    end;
    LINE_LINEDEVSTATE:
      case dwParam1 of  // Incomplete list.
        LINEDEVSTATE_RINGING:
          Add('LCB (LINE_LINEDEVSTATE): ' +
              '(Ringing) Ring, ring, ring...');
          LINEDEVSTATE_CONNECTED:
            Add('LCB (LINE_LINEDEVSTATE): Connected...');
          LINEDEVSTATE_DISCONNECTED:
            Add('LCB (LINE_LINEDEVSTATE): Disconnected.');
          LINEDEVSTATE_REINIT:
            // Line device has changed or been modified.
            if (dwParam2 = 0) then begin
              Add('LCB (LINE_LINEDEVSTATE): ' +
                  'Shutdown required');
              ShutdownCallManager;
            end;
        end;
      LINE_REPLY:
        if (dwParam2 = 0) then
          Add('LCB (LINE_REPLY): ' +
              'LineMakeCall completed successfully')
        else
          Add('LCB (LINE_REPLY): LineMakeCall failed');
    end;
  end;
end;
// -------------------------------------------------------------

procedure TForm1.btnDialClick(Sender: TObject);
var
  ErrNo: longint;
  S: string;
begin
  if rbCallManager.Checked then begin
    btnDial.Enabled := False;   // Disable dial button.
    btnHangup.Enabled := True;  // Enable hangup button.
    CreateCallManager;
  end
  else begin  // Use default call manager.
    // Not necessary to disable the buttons, the default
    // call manager handles everything.
    ErrNo := TapiRequestMakeCall(PChar(ePhoneNum.Text), '',
                                 'Some person', '');
    case ErrNo of
      0: S := 'success!'; // Success.
      TAPIERR_NOREQUESTRECIPIENT: S :=
        'TAPIERR_NOREQUESTRECIPIENT';
      TAPIERR_INVALDESTADDRESS: S :=
        'TAPIERR_INVALDESTADDRESS';
      TAPIERR_REQUESTQUEUEFULL: S :=
        'TAPIERR_REQUESTQUEUEFULL';
      TAPIERR_INVALPOINTER: S := 'TAPIERR_INVALPOINTER';
    else
      S := 'unknown value (' + IntToStr(ErrNo) + ')';
    end;
    Memo.Lines.Add('TapiRequestMakeCall returned: ' + S);
  end;
end;

procedure TForm1.FormCloseQuery(Sender: TObject;
                                var CanClose: Boolean);
begin
  // If the custom call manager was used,
  // make sure to free resources.
  if FLineOpen then
    CanClose := ShutdownCallManager;
end;

function TForm1.TapiInitialize: Boolean;
var
  ErrNo: Longint;
  S: string;
begin
  Result := False;
  // Initialize TAPI for use by custom call manager.
  FDev := 0;
  FCountryCode := 0;
  FVersion := 0;
  ErrNo := LineInitialize(@FLineApp, MainInstance,
                          LineCallback, '', @FNumDevs);
  case ErrNo of
    0: Memo.Lines.Add('LineInitialize was successful');
    LINEERR_INVALAPPNAME: S := 'LINEERR_INVALAPPNAME';
    LINEERR_OPERATIONFAILED: S :='LINEERR_OPERATIONFAILED';
    LINEERR_INIFILECORRUPT: S := 'LINEERR_INIFILECORRUPT';
    LINEERR_RESOURCEUNAVAIL: S :='LINEERR_RESOURCEUNAVAIL';
    LINEERR_INVALPOINTER: S := 'LINEERR_INVALPOINTER';
    LINEERR_REINIT: S := 'LINEERR_REINIT - (try again)';
    LINEERR_NODRIVER: S := 'LINEERR_NODRIVER';
    LINEERR_NODEVICE: S := 'LINEERR_NODEVICE';
    LINEERR_NOMEM: S := 'LINEERR_NOMEM';
    LINEERR_NOMULTIPLEINSTANCE: S :=
      'LINEERR_NOMULTIPLEINSTANCE';
  else
    S := 'Unknown Reason (' + IntToStr(ErrNo) + ')';
  end;

  // Show how many devices available.
  Memo.Lines.Add('Devices available: '+IntToStr(FNumDevs));

  if (ErrNo <> 0) then begin
    Memo.Lines.Add(
     'LineInitialize failed with error: ' + S);
    Exit;
  end
  else
    ErrNo := LineNegotiateAPIVersion(FLineApp, FDev, LoVer,
                                     HiVer, @FVersion, @FExt);

  if (ErrNo <> 0) then begin
    case ErrNo of
      LINEERR_BADDEVICEID: S := 'LINEERR_BADDEVICEID';
      LINEERR_NODRIVER: S := 'LINEERR_NODRIVER';
      LINEERR_INCOMPATIBLEAPIVERSION: S :=
       'LINEERR_INCOMPATIBLEAPIVERSION';
      LINEERR_OPERATIONFAILED: S :=
       'LINEERR_OPERATIONFAILED';
      LINEERR_INVALAPPHANDLE: S :=
       'LINEERR_INVALAPPHANDLE';
      LINEERR_RESOURCEUNAVAIL: S :=
       'LINEERR_RESOURCEUNAVAIL';
      LINEERR_INVALPOINTER: S := 'LINEERR_INVALPOINTER';
      LINEERR_UNINITIALIZED: S := 'LINEERR_UNINITIALIZED';
      LINEERR_NOMEM: S := 'LINEERR_NOMEM';
      LINEERR_OPERATIONUNAVAIL: S :=
       'LINEERR_OPERATIONUNAVAIL';
      LINEERR_NODEVICE: S := 'LINEERR_NODEVICE';
    else
      S := 'Unknown Reason (' + IntToStr(ErrNo) + ')';
    end;
    LineShutDown(FLineApp);
    Memo.Lines.Add(
     'LineNegotiateAPIVersion failed with error: ' + S);
  end
  else
    Result := True;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  FLineOpen := False;
  btnHangup.Enabled := False;    // Disable hangup button.
  if not TapiInitialize then begin
   Memo.Clear;
    Memo.Lines.Add('Failed to initialize TAPI');
    btnHangup.Enabled := False;  // Disable hangup button.
    btnDial.Enabled := False;    // Disable dial button.
  end;
end;

procedure TForm1.btnHangupClick(Sender: TObject);
begin
  if rbCallManager.Checked then
    if not ShutdownCallManager then
      Exit;
end;

procedure TForm1.CreateCallManager;
var
  S: string;
  ErrNo: longint;
begin
  Memo.Clear;
  // If a line is open, no need to initialize TAPI.
  if not FLineOpen then
    if not TapiInitialize then begin
      Memo.Lines.Add('Failed to initialize TAPI');
      Exit;
    end;
  // Open a line and get the line handle.
  ErrNo := LineOpen(FLineApp, FDev, @FLine, FVersion, 0, 0,
                    LINECALLPRIVILEGE_NONE,
                    LINEMEDIAMODE_INTERACTIVEVOICE, nil);
  case ErrNo of
    0: S := 'Line is open';  // Success, so drop through.
    LINEERR_ALLOCATED: S := 'LINEERR_ALLOCATED';
    LINEERR_BADDEVICEID: S := 'LINEERR_BADDEVICEID';
    LINEERR_INCOMPATIBLEAPIVERSION: S :=
      'LINEERR_INCOMPATIBLEAPIVERSION';
    LINEERR_INCOMPATIBLEEXTVERSION: S :=
      'LINEERR_INCOMPATIBLEEXTVERSION';
    LINEERR_INVALAPPHANDLE: S := 'LINEERR_INVALAPPHANDLE';
    LINEERR_INVALMEDIAMODE: S := 'LINEERR_INVALMEDIAMODE';
    LINEERR_INVALPOINTER: S := 'LINEERR_INVALPOINTER';
    LINEERR_INVALPRIVSELECT: S :='LINEERR_INVALPRIVSELECT';
    LINEERR_NODEVICE: S := 'LINEERR_NODEVICE';
    LINEERR_LINEMAPPERFAILED: S :=
      'LINEERR_LINEMAPPERFAILED';
    LINEERR_NODRIVER: S := 'LINEERR_NODRIVER';
    LINEERR_NOMEM: S := 'LINEERR_NOMEM';
    LINEERR_OPERATIONFAILED: S :='LINEERR_OPERATIONFAILED';
    LINEERR_RESOURCEUNAVAIL: S :='LINEERR_RESOURCEUNAVAIL';
    LINEERR_STRUCTURETOOSMALL: S :=
      'LINEERR_STRUCTURETOOSMALL';
    LINEERR_UNINITIALIZED: S := 'LINEERR_UNINITIALIZED';
    LINEERR_REINIT: S := 'LINEERR_REINIT';
    LINEERR_OPERATIONUNAVAIL: S :=
      'LINEERR_OPERATIONUNAVAIL';
  else
    S := 'LineOpen returned an unknown value of ' +
         IntToStr(ErrNo);
  end;

  Memo.Lines.Add('LineOpen reports: ' + S);
  if (ErrNo <> 0) then
    Exit
  else
    FLineOpen := True;

  { Create and fill the LineCallParams structure; mandatory
    for data calls, optional for voice calls. }
  with FLineCallParams do begin
    dwTotalSize := sizeof(FLineCallParams);
    dwBearerMode := LINEBEARERMODE_VOICE;
    dwMediaMode := LINEMEDIAMODE_INTERACTIVEVOICE;
  end;

  // Now place the call.
  ErrNo := LineMakeCall(FLine, @FHCall,
    PChar(ePhoneNum.Text), FCountryCode, @FLineCallParams);
  case ErrNo of
    0: S := 'LineMakeCall succeeded';  // Success.
    LINEERR_ADDRESSBLOCKED: S := 'LINEERR_ADDRESSBLOCKED';
    LINEERR_BEARERMODEUNAVAIL: S :=
      'LINEERR_BEARERMODEUNAVAIL';
    LINEERR_CALLUNAVAIL: S := 'LINEERR_CALLUNAVAIL';
    LINEERR_DIALBILLING: S := 'LINEERR_DIALBILLING';
    LINEERR_DIALDIALTONE: S := 'LINEERR_DIALDIALTONE';
    LINEERR_DIALPROMPT: S := 'LINEERR_DIALPROMPT';
    LINEERR_DIALQUIET: S := 'LINEERR_DIALQUIET';
    LINEERR_INUSE: S := 'LINEERR_INUSE';
    LINEERR_INVALADDRESS: S := 'LINEERR_INVALADDRESS';
    LINEERR_INVALADDRESSID: S := 'LINEERR_INVALADDRESSID';
    LINEERR_INVALADDRESSMODE: S :=
      'LINEERR_INVALADDRESSMODE';
    LINEERR_INVALBEARERMODE: S :='LINEERR_INVALBEARERMODE';
    LINEERR_INVALCALLPARAMS: S :='LINEERR_INVALCALLPARAMS';
    LINEERR_INVALCOUNTRYCODE: S :=
      'LINEERR_INVALCOUNTRYCODE';
    LINEERR_INVALLINEHANDLE: S :='LINEERR_INVALLINEHANDLE';
    LINEERR_INVALLINESTATE: S := 'LINEERR_INVALLINESTATE';
    LINEERR_INVALMEDIAMODE: S := 'LINEERR_INVALMEDIAMODE';
    LINEERR_INVALPARAM: S := 'LINEERR_INVALPARAM';
    LINEERR_INVALPOINTER: S := 'LINEERR_INVALPOINTER';
    LINEERR_INVALRATE: S := 'LINEERR_INVALRATE';
    LINEERR_NOMEM: S := 'LINEERR_NOMEM';
    LINEERR_OPERATIONFAILED: S :='LINEERR_OPERATIONFAILED';
    LINEERR_OPERATIONUNAVAIL: S :=
      'LINEERR_OPERATIONUNAVAIL';
    LINEERR_RATEUNAVAIL: S := 'LINEERR_RATEUNAVAIL';
    LINEERR_RESOURCEUNAVAIL: S :='LINEERR_RESOURCEUNAVAIL';
    LINEERR_STRUCTURETOOSMALL: S :=
      'LINEERR_STRUCTURETOOSMALL';
    LINEERR_UNINITIALIZED: S := 'LINEERR_UNINITIALIZED';
    LINEERR_USERUSERINFOTOOBIG: S :=
      'LINEERR_USERUSERINFOTOOBIG';
  else
    S := 'LineMakeCall returned an unknown value (' +
         IntToStr(ErrNo) + ')';
  end;
  Memo.Lines.Add('LineMakeCall reports: ' + S);
end;

function TForm1.ShutdownCallManager: Boolean;
var
  S: string;
begin
  Result := False;
  case LineShutdown(FLineApp) of
    0: begin
      S := 'success!';
      { LineShutDown performs the equivalent of LineClose
        so set the line flag to False. }
      FLineOpen := False;
      btnHangup.Enabled := False; // Disable hangup button.
      btnDial.Enabled := True;    // Enable dial button.
      Result := True;
    end;
    LINEERR_INVALAPPHANDLE:  S :='LINEERR_INVALAPPHANDLE';
    LINEERR_NOMEM:           S :='LINEERR_NOMEM';
    LINEERR_UNINITIALIZED:   S :='LINEERR_UNINITIALIZED';
    LINEERR_RESOURCEUNAVAIL: S :='LINEERR_RESOURCEUNAVAIL';
  else
    S := 'Unknown value';
  end;
  Memo.Lines.Add('LineShutDown reports: ' + S);
end;

end.
```

End Listing Two

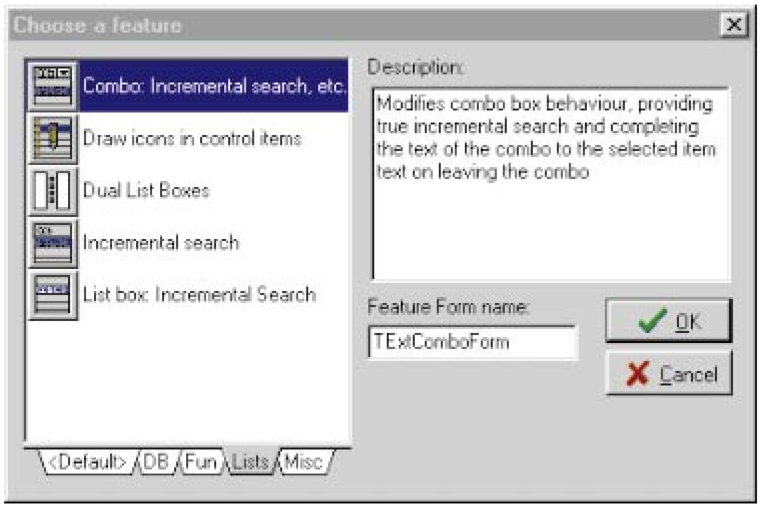
By Alan C. Moore, Ph.D.

Propel

A Powerful Tool for Code Re-use

Code re-use. What does it mean to you? Perhaps it's Object Pascal's underlying object-oriented structure, which allows you to easily create new classes. Or it may be Delphi's component architecture, which enhances rapid application development. Object-Oriented Programming (OOP) and Rapid Application Development (RAD) each have their advantages and disadvantages.

Is it possible to combine the best qualities of each approach? Propel, from Nevrona Designs, attempts to do that. It's built on a solid, object-oriented structure, yet, at the same time, supports rapid development with its convenient component editors. More importantly, it provides a giant step forward in code re-use for Delphi programmers. To make the best use of the product, however, you'll need to have at least a basic understanding of OOP. What exactly does Propel do and how?



Modifies combo box behaviour, providing true incremental search and completing the text of the combo to the selected item text on leaving the combo

TExtComboForm

Figure 1: Propel's Feature selection dialog box, where you choose the particular feature you want to use.

Propel and Features

The central concept of Propel is the *Feature* — a special binding of code to components and/or components to each other. What makes this binding so special is the level of flexibility it provides. (In fact, I wish they had not used the term Feature, but something like FlexComponent instead. Feature is such a commonly used term in software development!)

The best way to describe Propel is to compare it with other tools available in Delphi and from third parties. Consider normal components; you usually just drop them onto a form and set their properties. But what if you need additional properties to enhance or expand their functionality? Then you generally have to subclass the component and add the new properties. Or what if you want to mimic a component's *OnClick* behavior in another component? Again, you're faced with the prospect of subclassing, or pasting the event code, each time you use that component. With Propel, you can accomplish both of these tasks without having to derive a new component or write a lot of code.

We could also consider the situation with compound components. Sometimes called super components, these complexes are built by dropping several components on a Panel, exposing some of their properties, defining certain internal relationships, and compiling the whole thing as a single component. Tools like the Component Development Kit (CDK) from Eagle Software make this easy to accomplish. However, compound components have several limitations: 1) If you wish to expose certain additional properties later, you are back to square one — creating a new compound component; 2) you cannot change the spatial relationship of the sub-components once the compound component has been created; and 3) you must use all of the sub-component.

With Propel, all of this has changed dramatically. You have complete control over all the components included in a Feature: control over their visual placement, connections to other components and events, and whether they appear at all. There are two aspects of Propel that I will examine in depth: using and creating Features.

Using Features

Using Propel's Features is very easy. In fact, it's quite similar to using an ordinary component. You simply drop a *TFeature* component on a form, click on its icon to bring up the Feature selection dialog box (see Figure 1), and select the particular Feature you want to use. In one of the test applications I wrote, I dropped four *TFeature* components on a form and set each one to a different Feature. These Features, among a dozen or so that come with Propel, include one that paints the main form with a gradient, a dual-listbox Feature, and two others that enable incremental keyboard selections within a combobox and a listbox. Figure 2 shows the form once all the Features have been selected.

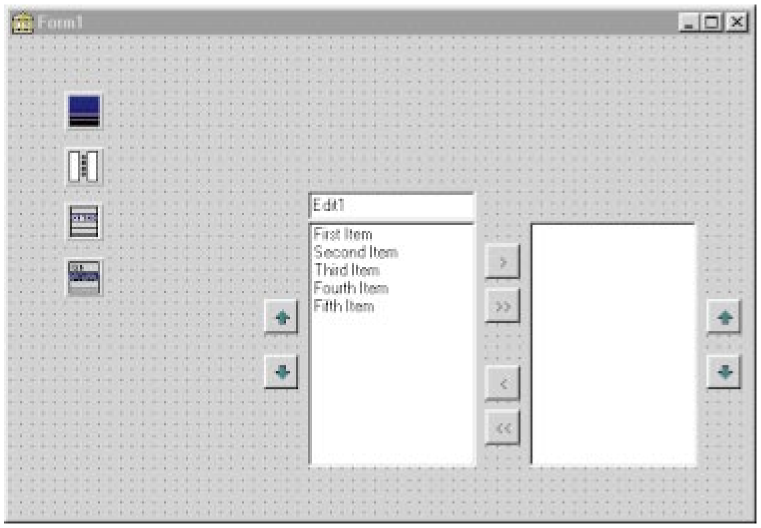
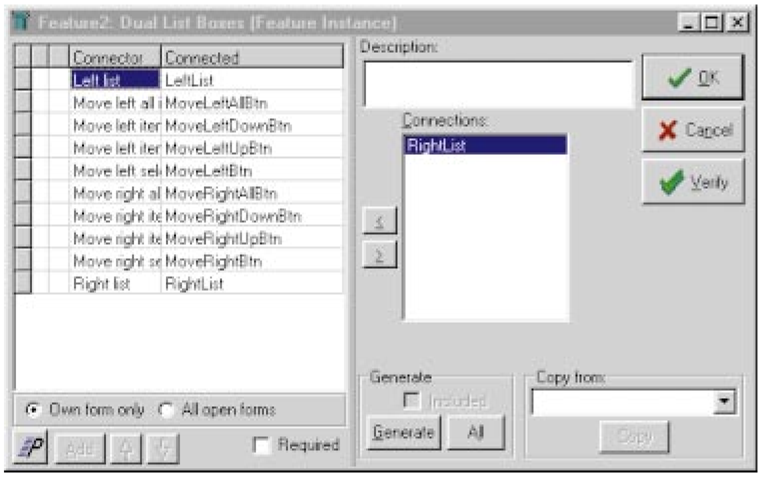


Figure 2: A form with four features selected.



Connector	Connected
Left list	LeftList
Move left all i	MoveLeftAllBtn
Move left iter	MoveLeftDownBtn
Move left iter	MoveLeftUpBtn
Move left sel	MoveLeftBtn
Move right al	MoveRightAllBtn
Move right ite	MoveRightDownBtn
Move right ite	MoveRightUpBtn
Move right se	MoveRightBtn
Right list	RightList

Connections: RightList

Figure 3: The Feature Instance dialog box, where you establish the connections between controls and control their behavior.

Once you've selected your Feature, you control its behavior in the Feature Instance dialog box (see Figure 3). Here, you establish the connections between controls and events; you also can select which controls will be generated on your application's form. (Figure 4 shows the same application at run time, as I am about to click on the down arrow, moving to the next entry in the combo box.)

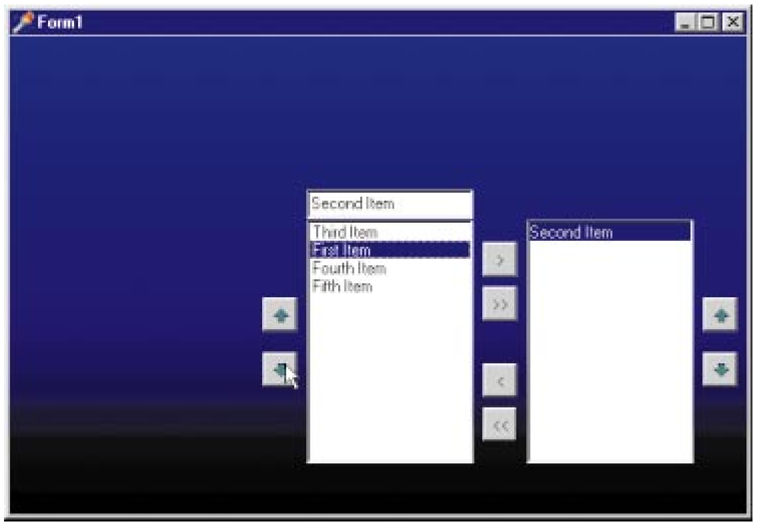


Figure 4: The same application as shown in Figure 2, at run time.

Propel can automatically generate all components of a Feature (similar to component templates but without the duplicate code). You have a lot of control over these components. Those which are part of a Feature instance are completely normal components; unlike subcomponents of a compound component, you can modify their properties, events and position. You can also define such connector components to be optional so you don't have to use the entire Feature's contents.

Because the *TFeature* component is similar to what Ray Lischner calls a metacomponent, it knows about all the other components on a form. So, when you add more components to a form and bring up the Feature Instance dialog box again, the new component(s) will be listed there, ready to be connected to the functionality of the particular Feature.

But Features are much more powerful than any metacomponent I've seen. For example, you can add special properties that are related to the components in the Feature, setting them in the object inspector as you would any property. With array connectors (which we'll be discussing later), you can connect a Feature component to a whole series of components on the form.

Propel's Own Features

As we've already seen, Propel comes with a powerful collection of Features that help to demonstrate the capabilities of the product. Some are quite useful, while a few fall into the novelty category. The dual-listbox Feature (again, see Figure 2) provides enhanced communication between two listboxes, enabling moving a single or groups of file(s) between them. As we've seen already, a listbox Feature facilitates incremental search. Others extend the listbox and the combo box in similar ways.

There are three database-related Features: one provides a means for you to use your own custom buttons, another adds special speed buttons, and a third introduces a really nice bookmarking Feature. There are other interesting Features. One enables a self-drawn graphical listbox; another paints a gradient as the main form's background (see Figure 4). There are several more useful Features and a few amusing ones. I found the hint Feature particularly useful.

Using Features is quite easy and the collection included with Propel is excellent, but what about creating new ones?

Creating Features

Remarkably, creating a new Feature is not much more difficult than using existing Features. The basic steps are as follows:

- Create a new form and give it a unique name (required).
- Drop a *TFeatureDef* component on the form.
- Write the registration code (handled automatically in Delphi 2, Delphi 3, and C++Builder).
- Add components to the form and name them.
- Add default settings of components to the Form's *Feature OnCreate* handler.
- Write component event code to enable the Feature's functionality.
- Define the Feature in the Feature Definition Editor.

The Feature definition form looks like a regular Delphi form. Figure 5 shows an example of a Feature I defined, a status bar with connections to controls on the form. (In this case, I wanted to design a reusable status bar, based on several of the components from the Raize Components.) Note the Def component in the top-left corner. As we'll see, this single component controls the definition of this Feature. There are two array connectors on the form, one to send fly-over hints from any component on the form to the left-most pane on the status panel, the other to control the graphic that will display in the right-most pane.

When you double-click on the Def component, the multi-page dialog box shown in Figure 6 comes up. In the first page, you can provide your Feature with its own icon, a short and longer description, and the page on which it will appear in the *TFeature* component. The second page (see Figure 7) allows you to define the various connector components that are at the heart of your Feature's functionality. If you want, you can provide a short and full description for each connector component.

The last page of the Feature Definition dialog box (see Figure 8) allows you to define new properties to enhance the functionality of your Feature. In this case, I needed to be able to store specific strings and bitmaps for the glyph status pane (the one at the far right).

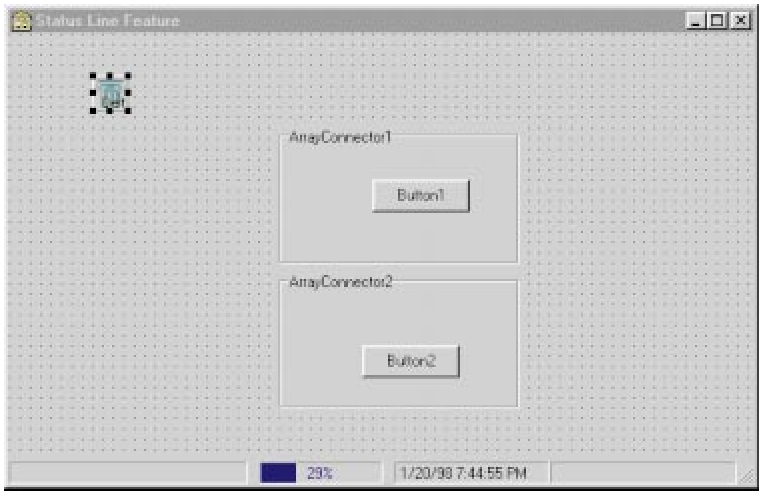
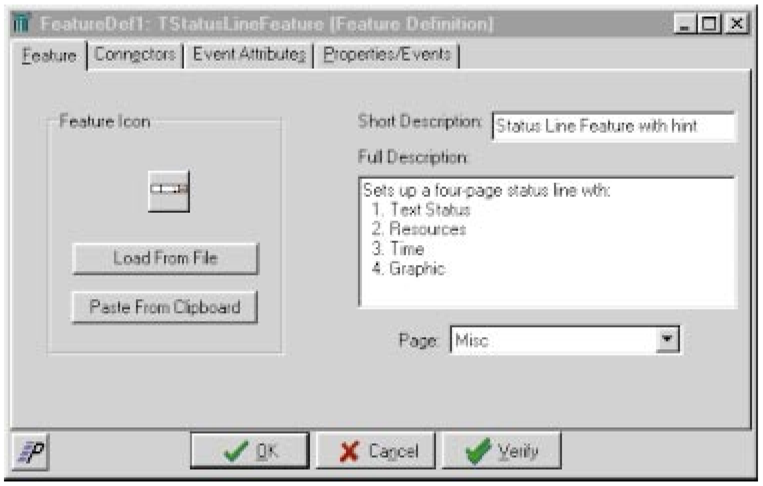


Figure 5: A new status bar feature based on several Raize Components.



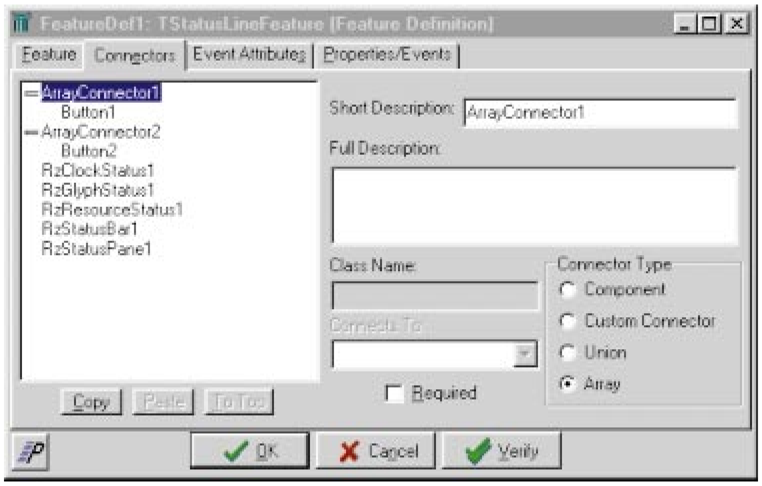
Short Description: Status Line Feature with hint

Full Description:

Sets up a four-page status line with:
1. Text Status
2. Resources
3. Time
4. Graphic

Page: Misc

Figure 6: The first page of the Feature Definition dialog box is where you set a Feature's icon and descriptions.

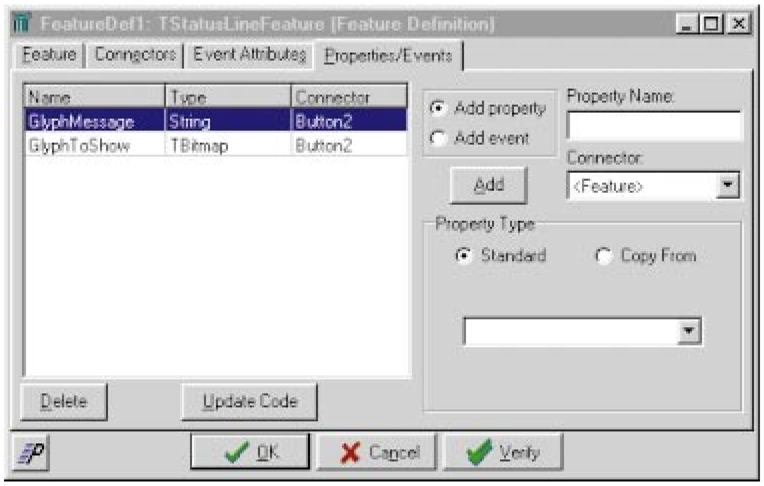


FeatureDef1: TStatusLineFeature [Feature Definition]

- ArrayConnector1
  - Button1
- ArrayConnector2
  - Button2
- RzClockStatus1
- RzGlyphStatus1
- RzResourceStatus1
- RzStatusBar1
- RzStatusPane1

Short Description: ArrayConnector1
Full Description:
Class Name:
Connects To:
Required [ ]
Connector Type: ( ) Component ( ) Custom Connector ( ) Union [✓ Array]

Figure 7: The second page of the Feature Definition dialog box allows you to define connector components.



FeatureDef1: TStatusLineFeature [Feature Definition]

Name	Type	Connector
GlyphMessage	String	Button2
GlyphToShow	TBitmap	Button2

[✓ Add property] ( ) Add event
Property Name:
Connector: <Feature>
Property Type: [✓ Standard] ( ) Copy From

Figure 8: The last page of the Feature Definition dialog box allows you to define new properties for your Feature.

INFORMANT FACT FILE

Propel is a powerful code re-use tool from Nevrona Designs that eliminates much of the need for the more cumbersome code re-use strategies, such as component creation and cutting and pasting of code. Its special components, called Features, are much more flexible than ordinary components, including compound components. Propel has several intuitive dialog boxes for creating new Features and using existing ones. It includes a substantial collection of built-in Features and supports all versions of Delphi and C++Builder. Propel also includes full source code, a manual, an integrated Help file, and a 30-day money-back guarantee.

Nevrona Designs
1930 S. Alma School #B-214
Mesa, AZ 85210-3041

Phone: (888) 776-4765 or (602) 491-5492
Fax: (602) 530-4823
E-Mail: order@nevrona.com
Web Site: http://www.nevrona.com
Price: US$199 plus shipping/handling

According to Propel terminology, when components are added to a Feature form, they are referred to as "connectors" because they play such an active and flexible role. As I indicated previously, connector components are not limited to one-to-one relationships with the components on the form. Propel includes two special kinds of connectors: array and union. Array connectors allow you to connect one type of functionality to a whole series of components; union connectors allow you to choose the one you want to enable from a group of related components.

You can also create "array properties," which allow you to add new properties to a Feature and then apply those properties to a whole series of components on a form. Propel also gives you the ability to easily intercept and react to Windows messages without deriving a new component. Not only can you add new properties (as we've been discussing), you can also add new events and methods. But what about distributing your Features to other developers — some of whom may not own Propel? That's also possible, but there's a caveat: it's rather involved, and the documentation needs to be improved.

Distributing Propel Features

To use Features in a system where Propel is not installed, you must save Propel Features as separate components called customized Feature components. Such Features require the run-time version of Propel, which you can distribute royalty-free. This is also an alternate way of using Features, even with Propel installed. Instead of dropping the main Feature component on a form and then selecting your Feature from the choices there, you can drop your new Feature component on a form and you're ready to define its properties.

Usually, it takes very little additional work to write a customized Feature component. First, you develop the Feature itself as previously described. Then, in the same unit, you derive an instance of *TCustomFeature* and override its *Create* constructor. In the body of the implementation of that constructor, you need to include a statement like this:

```
FeatureFormClass := 'MyFeatureForm';
```

In other cases, things can get trickier. Earlier, I described a status-bar Feature I developed based on Raize Components. I intended to create a new version of this that I could install on the Component palette. However, because it included two array-connectors, I had to take additional steps. Examine the example form (available for download; see end of article for details), paying particular attention to the comments. Note that each of the published properties in the Feature (form) class must be included in the custom Feature class. Because these properties are associated with array connectors, this step is necessary for them to appear with each actual component with which the array connector is associated. I also had to create two instances of the *TFeatureArrayPropObj* class to be a container for the published properties.

Conclusion — Successful Code Re-use

The President of Nevrona Designs, Jim Gunkel, shared his three criteria for a successful code re-use system with me. It must be:

- easy to save to a library;
- easy to retrieve from the library; and
- flexible and adaptable to a variety of programming situations.

In my opinion, Propel meets these criteria very well. It is a very well designed and implemented tool for code re-use. It will be especially useful to consultants and independent developers who tend to write a large number of similar applications. As a bonus, the full source code for the main Propel engine itself and the sample Features is included. Code junkies (like this reviewer) will take delight in studying these files and learning new Object Pascal tricks.

The one area where I encountered some problems was the documentation. While the first part of the manual is well written and clearly explains the purpose of the product, the latter sections, which introduce some of the more advanced topics, are in need of expansion. The entire manual could benefit from a thorough editing, as there are a number of errors and awkward sentences. Nevertheless, I highly recommend this product. The very problems I mentioned gave me a chance to test the technical support, which I found very helpful. By all means, visit the Nevrona Web site at http://www.nevrona.com, download the demo files and free Feature files, and find out for yourself. I think you'll be pleasantly surprised. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUL\DI9807NU.

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

Delphi Will Survive

With the advent of Delphi 4, this month seems like a good time to share my thoughts with you regarding why, in the words of Gloria Gaynor, "[Delphi] will survive." Feel free to share these arguments with your Microsoft friends and zealots who use the Delphi-won't-be-around-for-another-year excuse to try and justify why they won't even look at, much less use, the best development tool around. That particular line of reasoning — which I've heard since Delphi 1 — makes increasingly less sense as each new version comes out.

INPRISE is profitable. INPRISE has now been profitable for three consecutive quarters. This is important because the return to profitability under Del Yocam and the new management team reinforces the industry's assertion that there is room for more than one software company in this world.

INPRISE isn't a small company. By arguing that INPRISE won't be around because it is a "small company," you must first define a "small company." Is INPRISE a small company? If you consider a company that had revenues of $43,000,000 last quarter to be small, then it is. Somehow, we must come up with a magic number that shows which companies are small. The only thing we do know is that Microsoft is the largest, so they are not small. Anyone else, however, can suffer from this label.

History. Even mentioning the name Phillipe Kahn evokes strong reaction within the industry. Nevertheless, his aggressive, entrepreneurial attitude is what really put Borland on the map. Today, INPRISE has clearly established itself as the innovator of software development tools, especially as it pertains to the enterprise market. And with the acquisition of Visigenic, it doesn't look like INPRISE will relinquish that title easily. This acquisition positions INPRISE to spur the growth of distributed objects today much as it did with OOP nine years ago.

Compare Pascal to C++. I don't expect to change anyone's mind about which language is "best." However, there's nothing you can do in C++ that you cannot do in Pascal. The only differences are syntax and convention. If there is a specific feature you find useful in C++ that isn't in Pascal, you can at least find another way to represent it.

Compare Pascal to BASIC. BASIC isn't a true OOP language. No matter how you try, you can't redefine OOP to exclude inheritance. Inheritance is every bit as important to OOP as polymorphism and encapsulation. You also might recall the transition from BASIC to procedural, structured languages like Pascal and C some 15 years ago, as it became clear that C and Pascal could better support a robust development effort. BASIC has grown up considerably since then, but it still has its roots as a Beginner's All-purpose Symbolic Instruction Code.

Community support. Try to get the same level of support on any Microsoft product that you can for Delphi. The sense of community that many Delphi users exhibit is phenomenal. The Borland newsgroups are a perfect example of this. People participating in these newsgroups are eager to help and share information with others.

Delphi per se. Delphi is a viable product all by itself. A development environment that sells over 1,000,000 copies validates this claim, i.e. even if INPRISE did go under, someone would definitely recover Delphi from the flotsam. After all, a tool that makes so many developers so productive *must* survive. I know of one company in the northwest that could give Delphi a good home. Wouldn't that be the ultimate irony? I'd love to see the Delphi-bashers stop, turn on a dime, and say they loved Delphi all along. Who knows?

The best tool. Finally, let me address the general process of selecting a development tool. There are no certainties in this world. If the Department of Justice decided that Microsoft had an unfair competitive advantage, they could effectively close Microsoft down, or at least break them up as they did with AT&T. I certainly don't agree with this, but it is a fact. And if people are going to use a logical approach to selecting their software development tool, they should factor this into the equation. After all, wouldn't that mean that "Microsoft-might-not-be-there-next-year," or that "Microsoft-is-a-small-company?"

If someone selects a development tool other than Delphi based on technical merit, fine, but to ignore Delphi for other reasons is illogical. And after all, isn't logic the basis of software development? ∆

— Dan Miser

Dan Miser is a Design Architect for Stratagem, a consulting company in Milwaukee. He has been a Borland Certified Client/Server Developer since 1996, and is a frequent contributor to Delphi Informant. *You can contact him at http://www.execpc.com/~dmiser.*

table of CONTENTS

Quality First

A Challenge to RAD?

Last summer, I received a fascinating piece of e-mail from Glenn Crouch, a developer from Western Australia who is also a fellow moderator for the COBB Group's DDJ-Thread Internet list server. The message summarized an approach to programming — called the "Quality First Model"— articulated by Bertrand Meyer (IEEE's *Computer Magazine*, May, 1997). At the time, I filed the message for later consideration. When I recently encountered it again during a search for something else, it occurred to me that it might be of interest to readers.

My main question was "What relationship, if any, does Quality First have to RAD (Rapid Application Development)?" As we'll see, it's highly critical of the "speed at any cost" philosophy, and, as Steve McConnell points out in his *Rapid Development*: *Taming Wild Software Schedules* [Microsoft Press, 1996], provides sound recommendations that will save us time in the long run.

The key concept of the Quality First Model seems to be quality in the beginning, quality in the middle, and quality at the end. Meyer is very careful to differentiate between quality and perfection. If our goal is quality, we can be reasonably confident our efforts will meet with success; if we become fixated on some concept of "perfection," we may find it difficult to "produce results in [our] lifetime."

Quality in the beginning. Quality in the beginning means being mindful of the fundamentals, as Steve McConnell recommends. It also means avoiding sloppiness. Delphi is the preeminent RAD tool for Windows developers. But it's a two-edged sword. It's a visual development environment that can greatly enhance our productivity, but its ease-of-use can lull us into a sense of complacency and laziness that can undermine that productivity.

So, we need to be vigilant and avoid sloppiness. We could be sloppy in developing the logic of our application's flow. While that logic can certainly change, it is worth focusing on in the beginning. We could also be sloppy in dealing with bugs, and here Meyer has some practical advice: If we're working in a team environment, we shouldn't rely on others — especially the QA team — to clean up our messes.

The essence of Meyer's Quality First attitude can be summed up in his statement: "Build it so you can trust it. Then don't trust it." So how should we begin? Meyer recommends beginning with the cosmetics, by which he means using correct syntax, writing readable and well-commented code, and sticking to a consistent coding style. Quality in the beginning means to "get everything right from the start and fix it immediately if it is not." Quality comes first; functionality comes with time.

Quality in the middle. It's natural for us to want our applications to be feature-rich, but there's a danger in trying to do too much. McConnell recommends critically evaluating every proposed new feature. Eliminating superfluous features can enable us to do a better job with what is really required. As we build our applications and add needed functionality, we must remain focused on quality. Meyer's advice: Compile often, execute as soon as possible, and execute with full error checking turned on. We're going to make mistakes, and we're going to create bugs. It's better to fix them earlier than later.

In dealing with bugs, Meyer endorses the strategy put forth by Tom Van Vleck in his Web article: "Three Questions about Each Bug You Find" (see http://www.lilli.com/threeq.html). The first question is: "Is this mistake somewhere else?" On occasion, I still find myself repeating the same type of error. The second question, "What bug is hidden behind this one?", recognizes that one bug can prevent a whole section of code from executing. It's possible that more bugs may be lurking. The third question, "What should I do to prevent bugs like this?", can lead to some productivity-enhancing steps, such as acquiring and/or developing tools to help trap common errors.

As we continue to add functionality, Meyer recommends that we "always have a working system" so we can demonstrate the current state of the project to colleagues or potential customers. He also stresses the importance of early testing. An essential aspect of testing is making sure the program is robust. As he puts it, the job of our testers is to "displease us." We must make certain that even the most novice among users can run our application without getting hopelessly lost or causing it to crash.

Quality in the end. If there's quality in the beginning, and if we stay focused on quality as we progress, there will be quality in the end. As Meyer explains, the Quality First Model transforms the question "Can we ship now?" into "Does it do enough to impress the marketplace?" Commitment to quality is a constant; the main variable becomes functionality. Quality First doesn't contradict RAD, particularly as articulated by McConnell. By keeping us focused on robustness and fixing errors early, it increases our productivity and makes RAD workable. ∆

— Alan C. Moore, Ph.D.

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