



Cover Art By: Tom McKeith

ON THE COVER



6 The Best Just Got Better — Robert Vivrette

Mr Vivrette describes a good portion of the important changes made to the VCL for Delphi 4, including a host of new UI capabilities such as action lists, docking, owner-drawn menus, anchors, constraints, enhanced scroll bars, and extended mouse control, i.e. support for that little wheel.

FEATURES



12 The API Calls

Delphi and TAPI: Part III — Major Ken Kyler and Alan C. Moore, Ph.D. Kyler and Moore polish off their TAPI series by adding some features, and then wrapping all the functionality into a non-visual VCL component you can simply drop onto a form.



18 OP Tech

Procedure Variables — Bill Todd

Using procedure variables, you can call routines by assigning the address of the routine to a variable, then using the variable to call the routine. Mr Todd makes sense of this powerful, flexible technique.



22 On the 'Net

Multicasting — John Penman

Need to develop an Internet "push" application in Delphi? Mr Penman provides everything you need to get started — the background, the savvy, and working client and server multicast programs.



31 Algorithms

The Shape of Data — Rod Stephens

This article demonstrates how to apply the method of linear least squares to find a line or curve that best fits a set of points. It's a bit heavy on the calculus, but Mr Stephens makes it easy to follow.



36 In Development

Monitor Your NT Apps — Craig Dunn

Do you or your users need to know how your app is doing in the field? Mr Dunn shows us how to put the Windows NT Performance Monitor to work by creating a performance extension DLL.

REVIEWS



42 Async Professional 2.5

Product Review by Alan C. Moore, Ph.D.

47 ImageLib Corporate Suite 3.05

Product Review by Bill Todd

DEPARTMENTS

2 Delphi Tools

5 Newline

50 From the Trenches by Dan Miser

51 File | New by Alan C. Moore, Ph.D.





devSoft Announces ICK 1.0

devSoft Inc. announced the release of *ICK (Internet Commerce Kit) 1.0*, a developer's toolkit for secure access and manipulation of Internet data. The toolkit includes native Internet and intranet development components for development environments, including Delphi, C++Builder, Visual Basic, Visual C++, and others.

ICK includes an HTTPS component for securely transferring data and an

XML component for parsing and transformation of retrieved data. Standard FTP and HTTP controls are also included, as well as a NetDial control for managing dialup connections.

The ICK package comes in three editions: ActiveX controls, C++ classes, and native Borland C++Builder VCLs. The controls use internal multithreading for all blocking calls to avoid "freezing" during waiting calls. The communication components

use Microsoft's Win32 Internet library internally. The controls include properties to enable their use in environments that don't support events, such as Active Server Pages. All components have intuitive interfaces, complete Help files, and sample applications.

devSoft Inc.

Price: US\$245 per developer (no royalties).

Phone: (919) 493-5805

Web Site: <http://www.dev-soft.com>

Radiant Releases RAD Objects 1.3

Radiant Data Systems announced the release of *RAD Objects 1.3*, a suite of components created for client/server developers using Delphi and C++Builder.

Compatible with all INPRISE and many third-party controls, RAD Objects 1.3 provides transaction management without additional coding.

RAD Objects 1.3's cache table implementation allows data to be cached locally in a persistent cache by setting a few properties. SQL parsing enables SQL property separation into individual SELECT, FROM, WHERE, and GROUP BY clauses.

Additional features include Global Focus

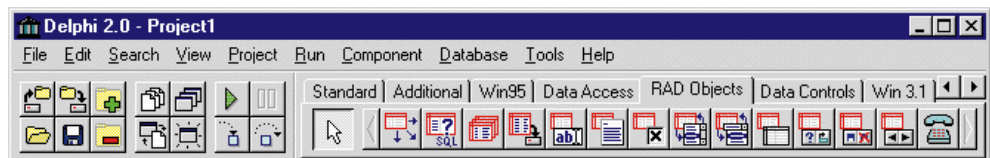
Tracking and client/server-friendly dbNavigator, dbLookupCombo, and dbLookupList controls optimized to use very little bandwidth.

Radiant Data Systems

Price: US\$250

Phone: (888) 840-7550

Web Site: <http://www.radiantdata.com>



HREF Announces WebHub 1.5

HREF Tools Corp. announced the availability of *WebHub 1.5*, a VCL for Delphi. New features benefit Webmasters and VCL programmers during development and "live" production, enhancing security, performance, and usability.

For Webmasters, WebHub 1.5 features an advertisement rotation component; a security component for managing user names, passwords, and account status for sites requiring login/logout; a debug mode for tracing through WebHub-HTML macro calls; and an HTML

editor with color syntax highlighting and code templates.

For VCL programmers, WebHub 1.5 offers reusable, "snap-together" panel technology for the creation of custom Web application servers with instant, full-featured user interfaces; a WebHub Wizard that creates complete projects from shared panels, forms, and data modules; a *SendFileIIS* method to make downloadable files available with dynamic names and/or content for secure software delivery; enhanced security; transparent support for cookie-

based session numbers; a *TWebdataform* component for automatic creation of data-entry forms based on database records; support for HTTP head/get requests; and dataset cloning for enabling single-user coding techniques in multi-user Web applications.

HREF Tools Corp.

Price: WebHub VCL Developer Package, US\$365 for a single developer; Hub licenses, US\$95 to US\$1,955 per additional server, depending on number of application instances.

Phone: (707) 542-0844

Web Site: <http://www.href.com>

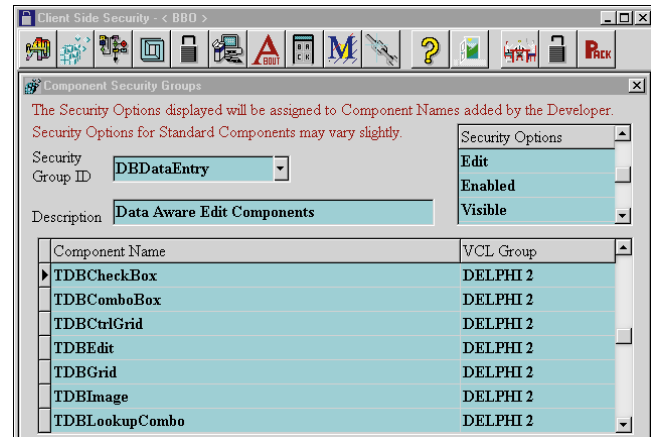


R&E Releases Client Side Security 1.1

R&E Systems, Inc. has released version 1.1 of *Client Side Security* for Delphi 2 and 3 and C++Builder. Client Side Security 1.1 is a client-based, database-independent security system for developers and end-users.

Developers can create demonstration systems that terminate based on expiration date, number of days, or number of iterations. Developers can also make designated application modules or functionality invisible. At a later time, the module or functionality can be sold to an existing customer and made visible by providing an unlock key.

In addition, developers can set up end-user security and



sell it with the application. End-user security then becomes the responsibility of the customer's security administrator.

Client Side Security 1.1 is a point-and-click development environment that runs inde-

pendently of Delphi or C++Builder and requires very little coding.

R&E Systems, Inc.

Price: US\$249; US\$399 with source.

Phone: (515) 279-6223

Web Site: <http://www.mesystems.com>

FileNET Introduces Panagon Capture

FileNET Corp. announced *Panagon Capture*, its component software for capturing an organization's documents anywhere across the enterprise. Panagon Capture leverages Microsoft component and Component Object Model (COM) technologies to customize applications.

Panagon Capture software provides components that improve document capture speed and efficiency, including scanning, automated batch and document separation, document assembly,

automatic indexing, bar-code recognition, image enhancement, and quality assurance. It captures and stores all document types, including images, fax, text, HTML forms, and video.

The software's modular design delivers a customizable solution, allowing document capture components to be included or removed depending on application processing requirements. Its architecture enables multiple components of the same type (such as scan, assembly, bar-

code processing, and index processing) to be included and operate in parallel to improve document throughput. Additionally, each Panagon Capture component can operate independently of the document scanning station to provide multi-phased pipeline processing for high-volume applications.

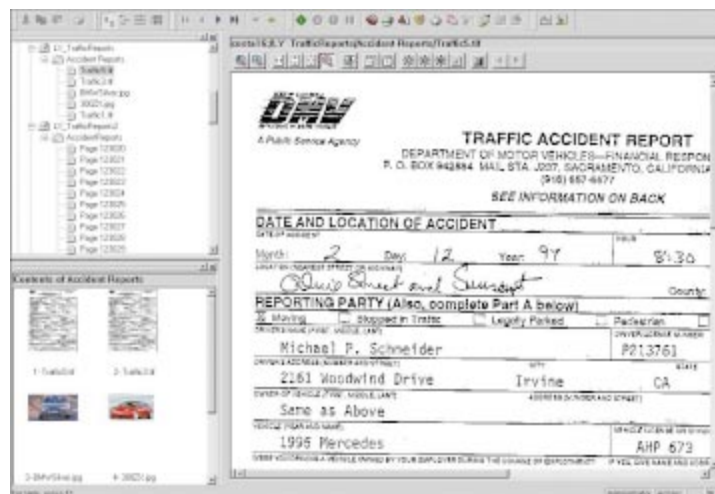
Panagon Capture's architecture is based on Microsoft's Object Linking Embedded (OLE) automation technology that coordinates the linking of all Panagon Capture components. The product includes ActiveX controls that allow rapid development using a range of development tools, including Delphi, Visual Basic, and C++.

FileNET Corp.

Price: From US\$1,000 to US\$25,000; tier-based on document capture volumes.

Phone: (714) 966-3400

Web Site: <http://www.filenet.com>



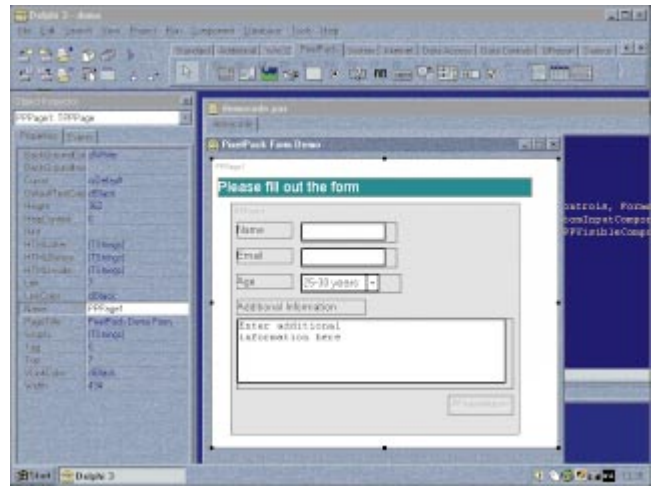


Femte Gear Offers PixelPack

Femte Gear Internet Software announced *PixelPack*, a set of components for Delphi 3 for creating on-the-fly HTML in CGI applications. You can draw a Web page using the PixelPack components inside the Delphi environment just as you would normally create forms and dialogs, and your Web application is ready to output the same pages as HTML.

PixelPack offers complete handling of input forms, Delphi-compatible Help files with examples, independence from CGI-engines or other packages, and the ability to create custom components.

PixelPack components include *TPPDropdownList*, *TPPForm*, *TPPImage*,



TPPJavaApplet, *TPPPanel*, *TPPRadioButton*, *TPPStandardComponent*, *TPPString*, *TPPSubmitButton*, *TPPTemplate*, *TPPTextArea*, *TPPTextInputField*, *TPPCheckBox*,

TPPImageMap, *TPPPage*, *TPPTable*, and *TPPListBox*.

Femte Gear Internet Software
Price: US\$159.95
Phone: +45 43 66 18 09
Web Site: <http://www.femte-gear.dk/pixelpack>

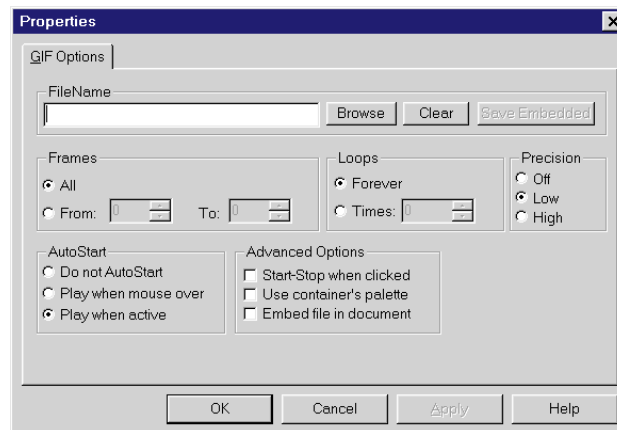
Paul Mace Announces GIF Control

Paul Mace Software, Inc. announced the *MAS GIF Control*, an ActiveX control that plays animated GIF content in executable applications, PowerPoint 97 documents, and any application that sup-

ports ActiveX. Offering fast playback, minimal overhead, accurate display, and a large feature set, the control has the ability to embed the animated GIF file inside a document. It supports the GIF87a and

GIF89a specifications, including delay, inter-frame transparency, and interlacing.

Designed to allow developers using PowerPoint — or most OCX94 compatible software — the ability to add animated GIFs to their documents without any programming or macros, the MAS GIF Control can be programmed through Delphi 3, VBA, Visual Basic 5.0, or any container meeting the OCX94-96 specification.



Paul Mace Software, Inc.
Price: US\$59
Phone: (800) 944-0191 or (541) 488-2322
Web Site: <http://www.pmace.com>

Ingeniering Introduces Wanda the Wizard Wizard

Ingeniering Inc. announced *Wanda the Wizard Wizard*, which allows developers to build, run, and maintain wizards that walk Windows users through complex procedures.

With Wanda, developers can debug wizards using the run-

time emulator; link the run-time engine (RTE) to their applications; instruct the RTE to run a wizard; and query the RTE about user selections and data entry items.

The royalty-free, run-time engine works with Delphi,

Visual Basic, C/C++, and any other language that can call a DLL in a 32-bit environment.

Ingeniering Inc.
Price: US\$199.95
Phone: (734) 662-4646
Web Site: <http://www.ingeninc.com>



September 1998



JEDI Project Finds New Home

Keith Anderson, President of AirSwitch Corp., announced that the purescience.com server will be going offline, so it's time to move all JEDI list accounts to the delphi-jedi.org server. As you read this message, you will no longer be able to send mail to any JEDI related lists on the purescience.com server. The correct list addresses now take the form JEDI@delphi-jedi.org, JEDI-API@delphi-jedi.org, etc. In addition, all LISTSERV commands should be sent to the AIRSWITCH server, LISTSERV@airswitch.com.

To subscribe to a list, send a message to LISTSERV@airswitch.com with a message of SUBSCRIBE JEDI and/or SUBSCRIBE JEDI-API. The URL for the AirSwitch site is <http://www.airswitch.com>, and Keith Anderson can be reached at keith@airswitch.com.

INPRISE Announces Borland Delphi 4

New York, NY—INPRISE Corp. announced Borland Delphi 4, a new version of its rapid application development tool for Windows. Designed to help corporations deliver large-scale “enterprise-class” business applications, Delphi 4 integrates client, middleware, and database development. Delphi 4 includes one-step support for CORBA and COM, as well as the Microsoft Transaction Server and the Oracle8 database server. Delphi is available in three versions: ClientServer Suite, Professional, and Standard.

Delphi 4 enhancements include MIDAS (Multi-Tier Distributed Application Services), which enables organizations to communicate and share enterprise data through multi-tier applications; the AppBrowser IDE, which simplifies the process of reading, writing, and browsing code; an advanced project manager, which compiles projects to multiple targets; new debugging technologies, including a module view, event logs,

INPRISE Expands Business Solutions Partner Program

New York, NY—INPRISE Corp. announced new benefits and services for members of its Business Solutions Program (BSP), which is designed to support commercial application providers, tools vendors, systems integrators, and training companies in offering enterprise computing solutions based on INPRISE products.

INPRISE consolidated and integrated the former Visigenic VIP and Borland partner programs. The new BSP provides greater access to technology, training, information, and sales support. It also opens new

data watch breakpoints, debug inspector, local variable inspection, integrated DLL debugging, and remote debugging; and language enhancements, including method overloading, dynamic arrays, and 64-bit integers.

In addition, Delphi 4 includes enhancements for Windows development, including docking, toolbars, and Windows 98 common controls. Delphi 4 also includes two Windows NT services that allow developers

Borland International Stockholders Approve Name Change to INPRISE Corp.

Scotts Valley, CA — Stockholders of Borland International, Inc. approved a proposal to officially change the name of the company to INPRISE Corp. The vote was taken on June 5, 1998 at the company's annual stockholders' meeting.

The new name reflects the company's transition from a developer of desktop software applications to a vendor in the corporate market for products and services used to

to build applications that run in the background and automatically open at the start up of the operating system.

At press time, the estimated street prices for Delphi 4 were as follows: Delphi 4 Client/Server Suite, US\$2,499; Delphi 4 Professional, US\$799; and Delphi 4 Standard, US\$99.95.

For more information, call INPRISE Corp. at (800) 233-2444 or visit the INPRISE Web site at <http://www.inprise.com>.

build and manage large-scale, or “enterprise,” software systems. The stock symbol for the company has changed from BURL to INPR on the National Market tier of the NASDAQ Stock Market. The name change does not require current holders of Borland stock to surrender stock certificates. Instead, when certificates are presented for transfer, new certificates bearing the INPRISE name will be issued.

markets to partner organizations, delivering INPRISE's CORBA technology to former Borland partners and extending development and management tools to former Visigenic partners.

Baltic Solutions Acquires Programmers' Guild's Products

Klaipeda, Lithuania — Baltic Solutions announced that it acquired the DesignerForms and Animated SystemTray Icon products, which were previously developed and marketed by Programmers' Guild. Baltic Solutions will continue to develop/support them in the future.

The BSP is segmented by partner type and level of participation.

For more information and directions on how to apply to the INPRISE BSP, visit <http://www.inprise.com/programs/BSP>.

Effective May 25, 1998, sales, upgrades, and support for Animated SystemTray Icon and DesignerForms will be supplied by Gintaras Pikelis of Baltic Solutions.

For more information, visit the Baltic Solutions Web site at <http://www.balticsolutions.com>.



ON THE COVER

Delphi 4 / VCL

By Robert Vivrette

The Best Just Got Better

The Delphi 4 VCL

By the time you read this, Delphi 4 will be on store shelves. In the **July, 1998** *Delphi Informant*, Cary Jensen gave us a great sneak peek at many of its new features, but there are plenty more to discuss. This article will focus on changes to Delphi's Visual Component Library (VCL).

INPRISE has prepared a fairly basic list of the things that have been changed in the VCL. These include action lists, docking support, constraint resizing, and a few others. However, there's plenty they don't mention, and they're just as handy to us hard-core Delphi developers as the changes cited.

In this article, I'm going to run through the changes to the VCL, including those documented by INPRISE, as well as the hidden gems. Please note that this article is based on pre-release versions of Delphi 4. Consequently, there's always a possibility that a feature described here may not make it into

the shipping version of Delphi 4, or may not work as described.

Action Lists

We've all run into this situation at one time or another: You're developing an application, and have various ways to call a particular routine. For example, you might have a **Save Document** command under your **File** menu, as well as a toolbar button that performs the same action. Suppose you don't have a document loaded, rendering the **Save Document** command meaningless. If you wanted to disable/gray-out the menu item, you would also want to do the same to the toolbar button. The problem might be worse in circumstances where you have four or five ways of accessing the same action. Before Delphi 4, you would typically have to write a procedure that would enable or disable all the related buttons, menu items, check boxes, and so on. A maintenance headache!

INPRISE's solution was to create action lists. In a nutshell, action lists are a way of tying together controls that perform similar tasks. Once they're all tied together in this way, making changes to the appropriate action list item automatically passes the changes to the components that are linked to that action. For example, start a new application and add a MainMenu component to the form. Double-click the MainMenu component, and add menu items. (I simply imported the File menu template.) Then, add an ActionList component to the form. The results will resemble what is shown in **Figure 1**.

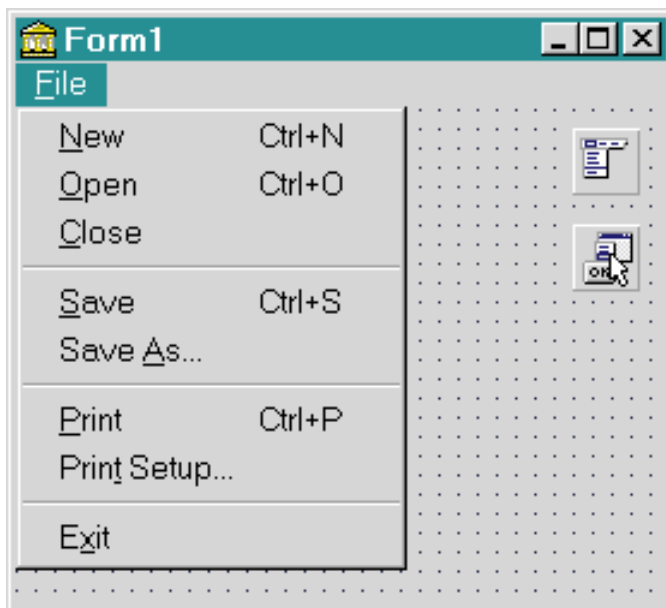


Figure 1: Building our action list example.

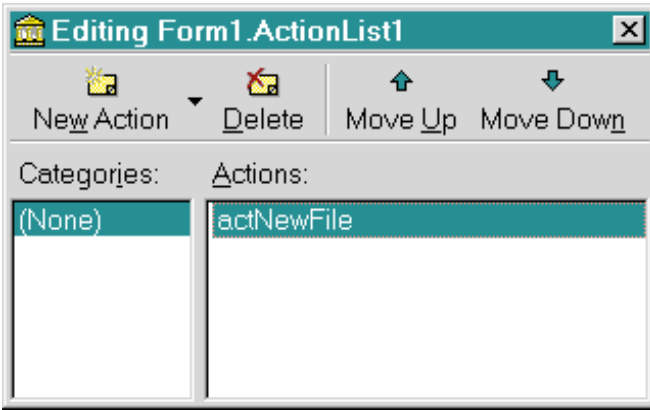


Figure 2: The ActionList editor allows you to enter various action items and group them by category.

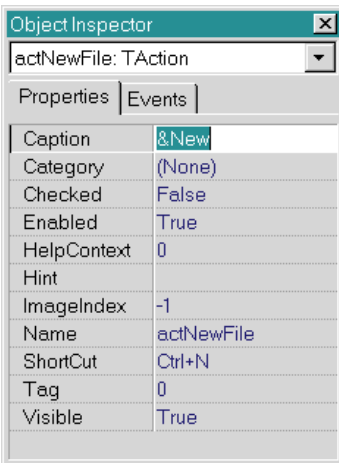


Figure 3: The properties of a new TAction object displayed in the Object Inspector.

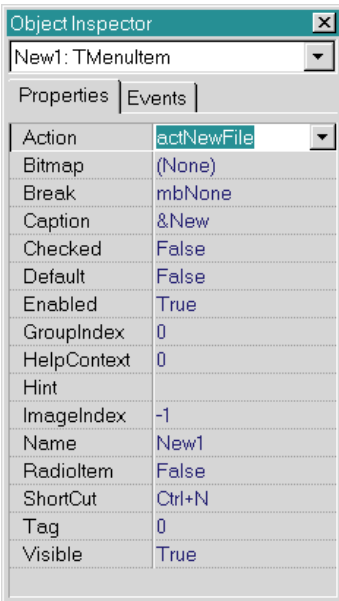


Figure 4: Setting a MenuItem's Action property to the name of the previously defined ActionItem, actNewFile.

The objective of this example is to make the File | New command use an action list item. By double-clicking the ActionList component, you'll see an ActionList editor that allows you to enter various action items and group them by category (see Figure 2). When you select New Action, the Object Inspector will display the TAction object properties (see Figure 3). For the purpose of our example, this action will be responsible for the "New File" process in the computer, so we'll give the action item a name of actNewFile. We'll also define its Caption property as &New, and its ShortCut property as CTRL+N.

Now we need to tell the File | New menu item to use this particular action. Returning to the MainMenu Item editor (double-click the MainMenu component), set this menu item's Action property to actNewFile (as shown in Figure 4).

If we run the application at this point, the New menu item is disabled and grayed-out because we haven't defined an action for this menu item.

Returning to actNewFile, we click on its Events tab and see that it has three properties: OnExecute, OnHint, and OnUpdate. The OnExecute method is where we want the working part of this action. It's the code associated with the user creating a new file, regardless of where in the application the user initiated the request. To keep things simple, I double-clicked OnExecute, and entered a ShowMessage statement in the method template it created (see Figure 5). This way, I will know when the action took place.

If you run the application now, you'll see that selecting File | New triggers the associated action, namely, showing the message we specified.

No big deal, right? Don't worry, it gets better. Suppose you were to add Button and CheckBox components to the form, and set their Action properties to actNewFile. As soon as this is done, they immediately connect some of their primary properties to that action. First, you'll see that their captions are all the same (see Figure 6).

The real time savings occurs when you want to enable or disable some function in the application. For example, assume that we don't want to allow the user to create a new file, and want to disable all the associated menu items, buttons, etc. Simple! All you need to do is disable the action item (actNewFile) to which these controls are connected. By disabling this action, all controls using it as their action item will also be disabled (see Figure 7).

Action lists can do a lot more. By setting an action item's Caption, Checked, Enabled, HelpContext, Hint, ImageIndex, ShortCut, and Visible properties, components that are linked to that action item, and who share the same properties, will also reflect those changes.

Docking

Docking is a wonderful new mechanism that permits users to drag components on a form, and dock them on some other region of the form. A good example of docking can be seen in the Delphi 4 IDE itself. Each of the toolbars at the top of the

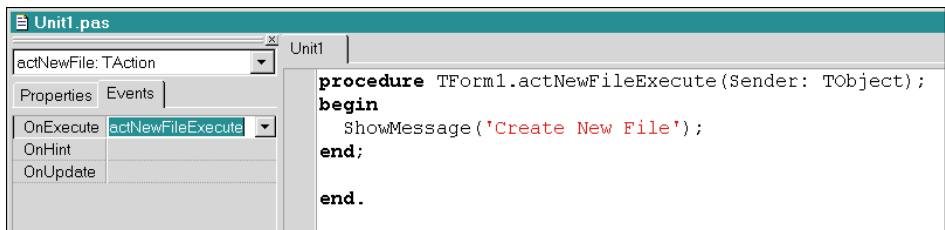


Figure 5: Entering a ShowMessage statement for the OnExecute method.

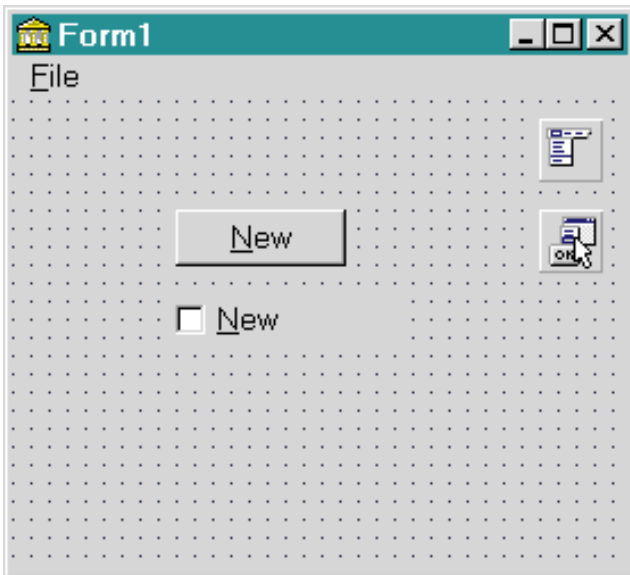


Figure 6: This form's Button and CheckBox Action properties are set to *actNewFile*, connecting some of their primary properties to that action.

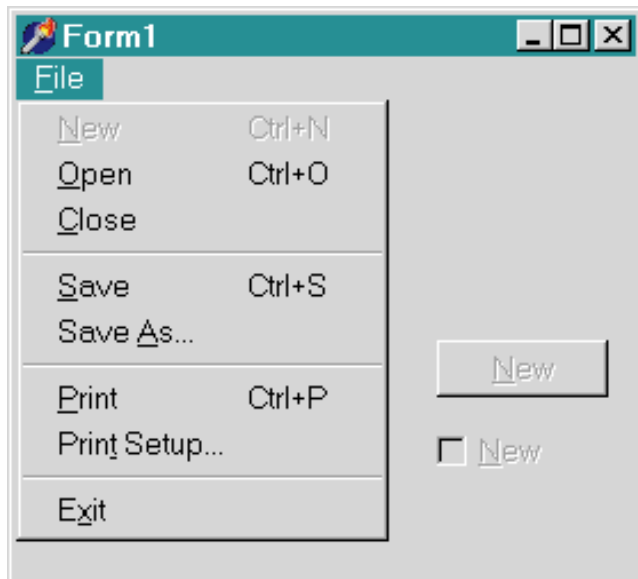


Figure 7: By disabling *actNewFile*, controls that use it as their action item will also be disabled.

application can be dragged and docked to other positions within the IDE's main window. Microsoft Office 97 introduced this kind of dockable toolbars. If the user drags the toolbar outside of the allowable docking site, the toolbar can become its own free-floating toolbar window. This free-floating toolbar can then be docked back into the old site — or any other location.

Support for this kind of docking has been added to Delphi 4. Although much of it will be handled automatically by simply setting a few properties, there are a wealth of methods and events you can use to customize the behavior of docking in your application. All descendants of *TWinControl* can now act as docking sites, meaning they are

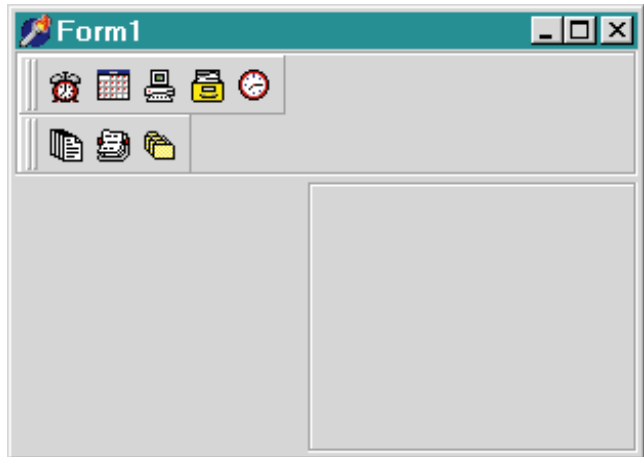


Figure 8: Two Toolbar components in a Panel component on a form.

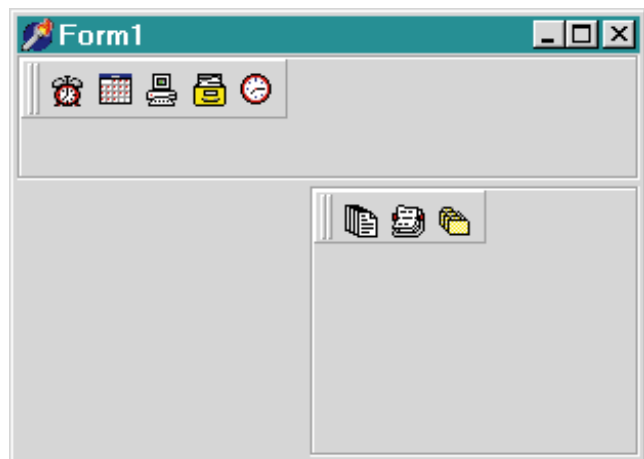


Figure 9: Users can drag either Toolbar to another dock site (another Panel in this case), ...

locations where dockable controls can be attached. Descendants of *TControl* can now act as dockable objects.

Let's take a brief look at how docking works. [Figure 8](#) shows a small window with two toolbars. The beveled areas of the form are *ControlBar* components that are used as a special kind of docking site for *Toolbar* components. In this example, the two *ControlBar* panels each have their *DockSite* property set to *True*. The two *Toolbars* have their *DragKind* property set to *dkDock*. When the application is compiled and run, the user is able to drag either *Toolbar* to another position dock site, or outside of both dock sites to become a free-floating window (see [Figure 9](#)).

In this example, I used *Toolbar* and *ControlBar* components, which behave a little differently as far as docking is concerned. However, the principles apply across all descendants of *TWinControl* (for dock sites), and *TControl* (for dockable controls). For example, you could place a *Panel* on a form, set it as a dock site, and then add a memo control telling it that it's dockable. The result will be a memo control that can dock and align to the inside of the *Panel*, or can be removed and docked elsewhere — or not docked at all (see [Figure 10](#)).

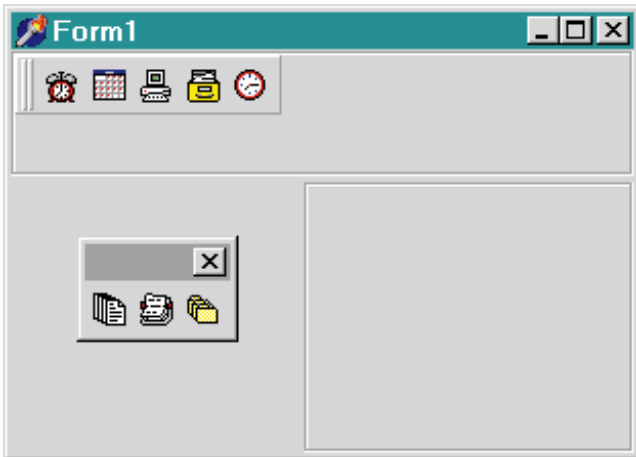


Figure 10: ... or outside of both dock sites to become a free-floating window.

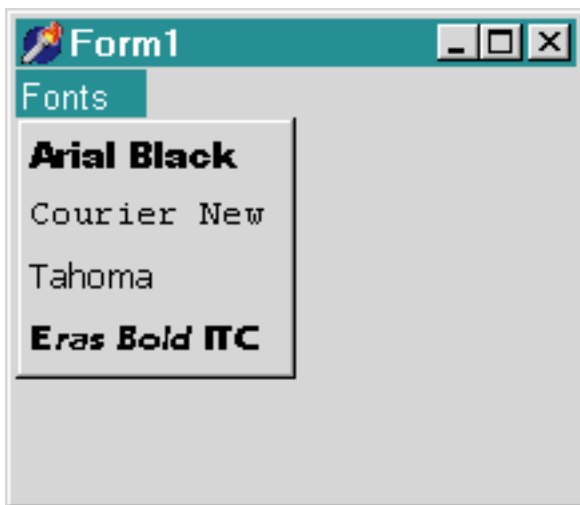


Figure 11: Delphi menus now feature owner-drawn support. This simple example paints each **Fonts** menu item using the actual font.

As I mentioned, Delphi 4 support for docking provides a great deal of control over docking behavior. Dragged controls can query a potential dock site to see if it's allowed to be a child of that site (through the *OnGetSiteInfo* and *OnDockOver* events). There are also events such as *OnDockStart*, *OnDockDrop*, *OnUndock*, and *OnDockEnd* that allow custom behavior to occur at various stages of a drag-and-dock process. You can even tell a control the particular class of *TWinControl* that can host it, if it's not docked anywhere, and is instead a free-floating window.

Owner-drawn Menus

Over the past few years, I've seen a lot of third-party code for implementing owner-drawn menus. With Delphi 4, INPRISE provides built-in owner-drawn support for both *TMainMenu* and *TPopupMenu*, as well as the menu items that reside within these menus.

To create an owner-drawn menu, simply start a new application and drop a *TMainMenu* on the form. Next, set its

OwnerDraw property to *True*. This property serves as a kind of "master switch" that tells the menu's associated menu items that they need to measure and draw themselves.

Next, add a few menu items to the menu. For the purpose of this example, I decided to allow these menu items to show different fonts. To accomplish this, I made the caption of each menu item the name of a different font in my system. Then, I tied all the menu items to the same *OnDrawItem* event (located on the *TMenuItem* Events tab), and added the following code:

```
procedure TForm1.DrawItem(Sender: TObject;
  ACanvas: TCanvas; ARect: TRect; Selected: Boolean);
begin
  with ACanvas do begin
    FillRect(ARect);
    Font.Name := (Sender as TMenuItem).Caption;
    Font.Size := 8;
    TextOut(ARect.Left + 2, ARect.Top + 2,
      (Sender as TMenuItem).Caption);
  end;
end;
```

When any of the menu items attempt to draw themselves, they call this method. Doing so simply clears the space on the menu item's canvas using *FillRect*, then sets the font name equal to the menu item's caption. Lastly, we use *TextOut* to write out the menu item's caption. When you run this code, the results will resemble [Figure 11](#).

There is, of course, much more to owner-drawn menus. Each menu item also has an *OnMeasureItem* event, allowing it to specify its custom size. Also, each menu item has a new *Bitmap* property that allows you to provide a small graphic associated with that menu item. An *ImageIndex* property allows a menu item to use a specific .BMP image maintained by its parent *TMainMenu* or *TPopupMenu*.

Owner-drawn menus will make it much easier to add attractive, useful interfaces to Delphi applications. For example, if you need to permit a user to choose a color, rather than showing a menu with color names, you can make those menu items owner-drawn, and draw them as color bars instead.

Anchors

The *Anchor* property is also new to Delphi 4, and is available for all descendants of *TControl*. It's used to specify how a control will position itself within its parent control when the parent is resized. An *Anchor* property is a set of four Boolean values, one each for a control's left, top, right, and bottom edges. For each of a control's four edges, if *Anchor* is set to *True*, the control will keep that edge relative to the parent.

Without *Anchor*, if you drop a *Panel* on a form, and then resize the bottom-left corner of the form, the *Panel* stays in place. This is because its position is determined by the origin of the window, which isn't moving. However, developers often need controls to resize in specific ways to take advantage of a main form that's getting bigger or smaller. Using the *Align* property can

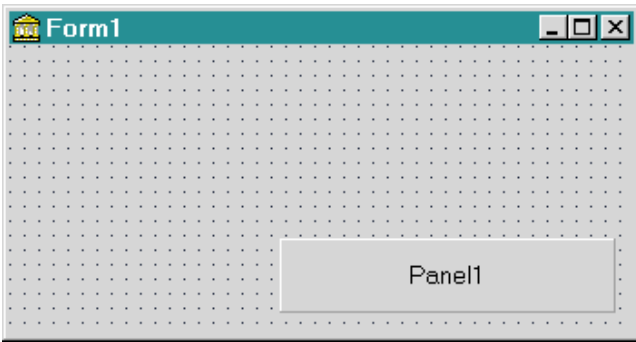


Figure 12: A Panel component with its new *Anchor* properties set relative to the lower-right corner of the form (see Figure 13).

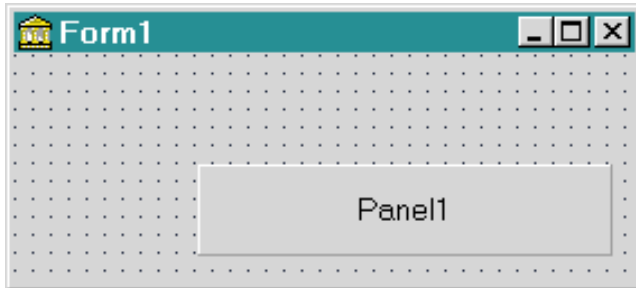


Figure 13: The result of resizing the form shown in Figure 12 up and to the left; the Panel keeps its position relative to the lower-right corner of the form.

sometimes solve this problem, but often, custom code must be written for the form's *OnResize* event to properly place controls.

In Figure 12, I placed a Panel on a form and set its *akRight* and *akBottom* anchors to True, and the *akTop* and *akLeft* anchors to False. The result of taking the lower-left corner of the form and resizing it up and to the left is shown in Figure 13. As you can see, the Panel maintained its position relative to the lower-right edge of the form, i.e. according to its *Anchor* values.

Using *Anchor*, you can add some pretty interesting capabilities that just aren't available with the *Align* property.

Constraints

All descendants of *TControl* now contain a *Constraints* property. This property holds four sub-properties — *MaxHeight*, *MaxWidth*, *MinHeight*, and *MinWidth* — that define the largest and smallest height and width to which the control can be sized. In most cases, you'll use this to control the dimensions of a form. Although the capability applies to many other controls, it's more difficult to see it in action.

As a demonstration, place a Panel component on a form and set its *Height* and *Width* to 200. Then set its *MinWidth* constraint to 100 and its *MaxWidth* constraint to 200. Next, add a scroll bar to the form, and set its minimum value to 0 and its maximum value to 300. On the scroll bar's *OnChange* event, set the Panel's *Width* property to the current position of the scroll bar. When the application is run, the Panel's width remains within these limits (as set by its constraints), even if the scroll bar generates values below 100 and above 200.

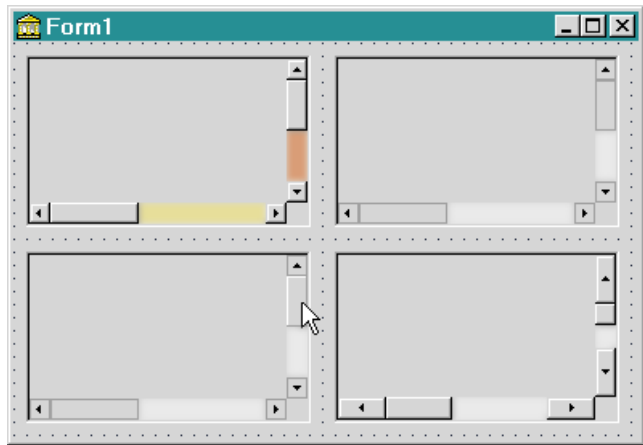


Figure 14: These *TScrollBar* components demonstrate some of the new scroll bar properties.

Scroll Bars

Speaking of scroll bars, a number of new properties have been added to the scroll bars found in the *TScrollBar* component, as well as on forms (the *HorzScrollBar* and *VertScrollBar* properties). These scroll bars now have a *Style* property that allows you to alter its basic appearance. The *ssRegular* style uses the normal 3D appearance. The *ssFlat* style shows the scroll bar only in a flat, 2D representation. The *ssHotTrack* style is initially flat, but, as the mouse passes over elements of the scroll bar (i.e. thumb track and scroll buttons), these elements pop up into a 3D look.

In addition, you can now specify different-sized buttons or thumb bars, and modify the color of the scroll bar track. Figure 14 shows four *TScrollBar* components with some of these properties set in different ways. Notice in the lower-left example that the mouse cursor is over the thumb bar, and is raised. This bar achieves this effect using its *ssHotTrack* style.

Extended Mouse Support

Some of the newer mice available for PCs include a wheel between the right and left mouse buttons. Until now, this added control to the mouse has gone unsupported. With Delphi 4, however, there is now mouse-wheel support added to forms and scroll boxes (technically, descendants of *TScrollingWinControl*). There are now *OnMouseWheel*, *OnMouseWheelUp*, and *OnMouseWheelDown* events that allow interaction with these new mouse devices.

In addition, there is a global *Mouse* object (much like the *Screen* object) that holds information relating to the mouse, i.e. which window currently has the mouse capture, the position of the mouse cursor, whether there is a wheel present, etc.

Multiple Monitor Support

Microsoft Windows 98 and Windows NT 5.0 will add support for multiple monitors. Thankfully, Delphi 4 adds support for the MultiMonitor APIs present in these new versions of Windows. INPRISE has implemented these capabilities by adding a *Monitors* property to the *Screen* global object. This

allows the user to query all the available monitors in the system, and to see each one's capabilities (dimensions, color depth, etc.).

When a form is created in Delphi, you can use its new *DefaultMonitor* property to specify on which monitor the form appears. Possible values for this property are:

- *dmDesktop*: No attempt is made to position the form on a specific monitor.
- *dmPrimary*: The form is positioned on the first monitor listed in the global *Screen* object's *Monitors* property.
- *dmMainForm*: The form appears on the same monitor as the application's main form.
- *dmActiveForm*: The form appears on the same monitor as the currently active form.

Additionally, the *Position* property of a form differentiates between *poScreenCenter* and the new *poDesktopCenter* value. With *poScreenCenter*, the form is positioned in the center of the screen, but, in multi-monitor applications, the form may be moved so that it falls entirely on a single monitor. With *poDesktopCenter*, the form is positioned in the center of the screen without making adjustments due to multiple monitors.

BeforeDestruction / AfterConstruction

When working with Delphi versions 1, 2, and 3, some developers were confused as to when they could safely set properties in a newly-constructed object, or clean up things before an object was destroyed. For example, when developing a component, some properties — such as the control's initial width, or child controls in the component's *Controls* array — are not yet defined in a *Create* constructor. Essentially, component developers needed a spot that says, "Call me when you're completely constructed and I can access all your properties;" or the opposite: "Call me just before you start destroying yourself." Although there were work-arounds, they were under-documented and sometimes confusing.

INPRISE solved this problem in Delphi 4 by adding two protected methods to the *TObject* class: *BeforeDestruction* and *AfterConstruction*. *BeforeDestruction* is automatically called immediately before an object's destructor; *AfterConstruction* is called immediately after an object's constructor. Any classes that descend from *TObject* can override these methods to perform actions that shouldn't occur in the constructor or destructor. For example, custom forms generate *OnCreate* or *OnDestroy* events by overriding these methods. INPRISE indicates this is especially important for developers writing components for both Delphi and C++Builder, to avoid problems with the differences in constructor and destructor behavior between Object Pascal and C++.

Conclusion

While this article by no means presents all the new VCL features of Delphi 4, it does hit many of the highlights. The extensions to basic object classes, such as action lists, constraints, anchors, and of course, docking, will help

Delphi developers cut down on the amount of code they need to maintain, while keeping up with new — and rapidly changing — user-interface features. ▲

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





THE API CALLS

Delphi 2, 3 / TAPI / Line Communications

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



Delphi and TAPI

Part III: Wrapping Up Telephony

In the first two articles of this series (July and August, 1998), we discussed issues related to implementing telephony functionality using TAPI. In Part I, we examined some of the basic TAPI functions in detail, demonstrating how to use them to initiate and manage phone calls. In Part II, we showed techniques for finding the capabilities of a TAPI implementation and monitoring changes to the COMM port.

In this final installment, we're going to do quite a bit more. First, we're going to add some enhancements to our call manager: the ability to take incoming calls, and the ability

to make calls from a pulse phone. Then, we're going to wrap the TAPI functionality we've developed into a class, so it can be called easily from any application. Finally, we're going to transform our call manager into a non-visual component, so it can be simply dropped onto a form from the Component palette. Let's begin with the enhancements.

Answering Incoming Calls

There are several changes we must make to our call-manager application to enable it to answer incoming calls. First, we need to change the *dwPrivileges* parameter of the *LineOpen* function from *LINECALLPRIVILEGE_NONE* to *LINECALLPRIVILEGE_OWNER*. To provide flexibility, we must also add a new Boolean parameter, *AcceptCalls*, to the *OpenLine* method:

```
function OpenLine(var APort: THandle;
  var OpenResult: Longint; AcceptCalls:
  Boolean): string;
```

Now when we open the line and Windows receives notification of an incoming call, it will notify our application. We can respond by answering the call or passing it to another telephony-enabled application. Before we look into how we answer the call (which is fairly simple), we need to examine the notification mechanism. Do you remember our callback function? It's the one we've been using since the first article of this series:

```
procedure LineCallBack(hDevice, dwMessage, dwInstance,
  dwParam1, dwParam2, dwParam3 : DWORD); stdcall;
begin
  TapiMessages.Clear;

  with TapiMessages do begin
    case dwMessage of
      LINE_CALLSTATE:
        begin // Reports asynchronous responses.
          case dwParam1 of
            ...
            LINECALLSTATE_OFFERING:
              begin
                Add('LCB (LINE_CALLSTATE): The call is ' +
                  'being offered to the station. ');
                if dwParam3 <> LINECALLPRIVILEGE_OWNER then
                  Add('Cannot accept call because we ' +
                    'don't have owner privileges. ');
                else
                  begin
                    Add('Trying to accept incoming call');
                    lineAccept(ACall, nil, 0);
                  end;
                end;
            LINECALLSTATE_ACCEPTED:
              begin
                Add('LCB (LINE_CALLSTATE): The call was ' +
                  'offering and has been accepted. ');
                if MessageDlg('Accept the call?',
                  mtConfirmation, [mbYes, mbNo],
                  0) = mrYes then
                  lineAnswer(ACall, nil, 0);
                end;
            ... { other cases }
          end;
        end;
    end;
  end;
```

Figure 1: We need to modify the first two case blocks that handle the dwMessage parameter of the TAPI message.

```
procedure LineCallBack(hDevice, dwMessage, dwInstance,
  dwParam1, dwParam2, dwParam3: DWORD); stdcall;
```

When Windows senses an incoming call, it sets the first parameter of this function, *hDevice*, to a handle it provides for our use. Before we do anything, however, we must ensure that we have owner privileges. To do this, we check the value of the last parameter, *dwParam3*; it should be `LINECALLPRIVILEGE_OWNER`. If it isn't, we should exit and let another installed telephony application manage the call.

We're going to do all this under the `LINECALLSTATE_OFFERING` and `LINECALLSTATE_ACCEPTED` cases of *dwMessage* (near the top of the long `case` statement); this is the message Windows sends via our callback function for an incoming call (see [Figure 1](#)). Once we have the handle, and are sure we have owner privileges, we can relate to the incoming call.

There are two TAPI functions we can use when an incoming call is detected: *lineAccept* and *lineAnswer*. Both are declared in the TAPI.pas unit as follows:

```
function lineAccept(hCall: HCALL; lpsUserUserInfo: LPCSTR;
  dwSize: DWORD): LONG; stdcall;
function lineAnswer(hCall: HCALL; lpsUserUserInfo: LPCSTR;
  dwSize: DWORD): LONG; stdcall;
```

There is a subtle difference between them. The first function, *lineAccept*, merely informs other applications that our application is accepting responsibility for the incoming call; the second, *lineAnswer*, physically answers the call by asking the modem to take the phone off the hook and begin tone negotiation for a data connection. Another function of interest is *lineHandoff*, which provides other applications the opportunity to deal with the incoming call. The first two functions, *lineAccept* and *lineAnswer*, are asynchronous (i.e. they don't necessarily return control immediately), while *lineHandoff* is synchronous (i.e. it returns control immediately).

As shown in the previous `case` statement, we can answer a call in the offering state (`LINECALLSTATE_OFFERING`), or in the accepted state (`LINECALLSTATE_ACCEPTED`). To keep things simple, we're going to work with just the *lineAnswer* function, and not the *lineHandoff* function. Within the offering `case` block, however, you could give users an opportunity to accept a call (moving to the accepted block), or simply pass it on to another application.

Because we're dealing with simple voice calls, we can set the last two parameters of *lineAnswer* to `nil` and 0 (zero), respectively. The other functionality we've added is the ability to make calls from a phone that uses pulse dialing. Let's see what makes that possible.

Pulse Dialing

By default, when modems dial a phone number, they generally send tones, not pulses. Most phones in the United States use tones; in other parts of the world, however, pulse phone systems (associated with rotary dial phones) are still common. (Alan, who lives in rural Kentucky, happens to have one of the older pulse lines, which was one of the motivations for wanting to include this feature.)

If you examine our extended call manager, you'll notice that we added a couple of new check boxes; one to choose whether to accept incoming calls, the other to choose between pulse and tone dialing (see [Figure 2](#)). How does a modem know it's supposed to dial using pulses instead of tones? Simple! If the dialable number starts with the letter "p," it should be dialed with pulses. Here's the line of code that makes a pulse call:

```
ErrNo := LineMakeCall(FLine, @FHCall,
  PChar('p'+PhoneNumber), FCountryCode, @TheLineCallParams)
```

We'll examine this code in more detail when we discuss the structure of our TAPI interface class — specifically, different ways of triggering pulse or tone dialing. First, we need to discuss the general approach we took in building the class.

Doing It with Class

Even before we added the new features we've been discussing, we transformed the code that accompanied the last article in significant ways. We wanted to clearly separate the user interface code from the TAPI-specific code in preparation for creating a TAPI component. In the previous two articles, we demonstrated more than just the basic TAPI techniques; we showed how to use Delphi to create a prototype of an application that uses these techniques. Delphi is particularly suited for creating such prototypes; the Delphi literature contains countless examples. While our call-manager prototype provides an excellent vehicle for demonstrating basic TAPI functionality,

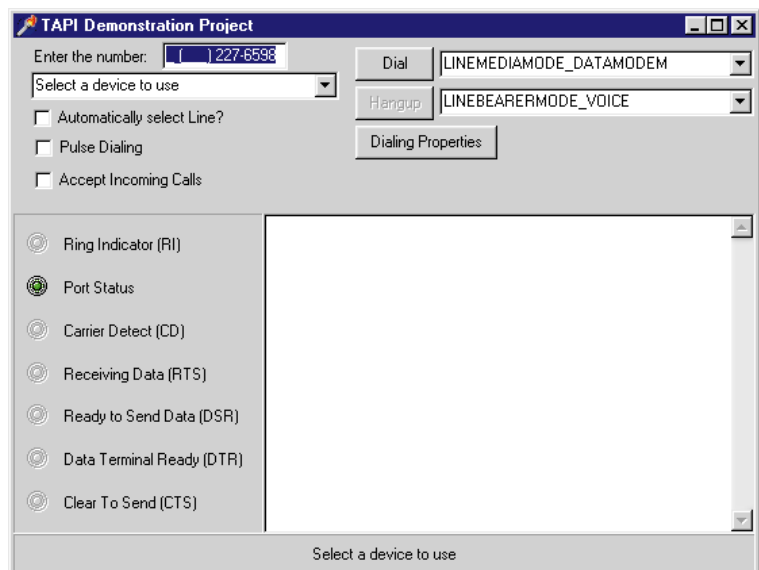


Figure 2: The augmented call-manager interface, with pulse dialing and incoming call handling.

it's flawed in one important respect: it was bound to a particular Delphi form, and thus, to a particular user interface.

Because we want to use this TAPI functionality in multiple applications, we jumped to the next level, wrapping those TAPI elements we wanted to use in a reusable class and a non-visual component. Each can be easily used in any application we write. To demonstrate this, we've included four sample applications with this article: two that use the new TAPI class, and two that use the new TAPI component. The first in each group is the full-blown call manager we've been developing throughout these articles (again, see [Figure 2](#)). The second is a simple dialer without any of the frills (see [Figure 3](#)). You can download the sample files, the class, and the component files they use; see end of article for details.

If you compare the new files to those from Part II, you'll notice many changes. In particular, we've moved all the TAPI-specific code to a TAPI-interface class, keeping only the form-specific code in the test unit. Here are some of the highlights.

First, we changed *ShutdownCallManager* and *LineCallBack* from methods to local routines. We changed the private variables in the earlier version to properties in the interface class. More importantly, we added *TStringList* parameters to most of the previous methods so we could send information back to the form to display in the memo control. (By separating the interface class from the form, a developer could also show this information in a RichEdit control, a ListBox, or any control that uses a *TStrings* derivative.) So, while we had many lines of code like this in last month's prototype:

```
Memo.Lines.Add('LineInitialize was successful');
```

they are now written as:

```
S := 'LineInitialize was successful';
...
InitResults.Add(S);
```

Where *S* is a local string variable and *InitResults* is the *TStringList* parameter to the *TapiInitialize* method. It's passed back to the *Form.Create* method of the main calling unit in the following statements:

```
if not TapiInterface.TapiInitialize(TempStringList) then
begin
  { Error handling code. }
  Memo.Lines.Assign(TStrings(TempStringList));
  Memo.Invalidate;
end;
```

Similar techniques are used in other routines. For example, we use another local string list, *InitResults*, to store the results when we call *TapiInitialize*.

There was one area where we wanted to add flexibility; we didn't want our new class to necessarily be bound to the

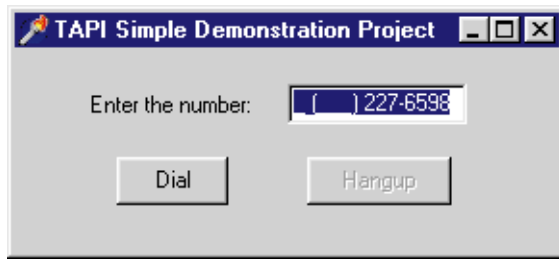


Figure 3: A simple program based on the TAPI class and component.

form we've been developing. In our prototype, we do a lot of communicating between the main form unit and the TAPI interface unit. For example, whenever a new phone number is entered, or a different line device is selected, those changes are sent to the interface unit, and one of its properties is changed. However, if we want to use a simpler user interface, we're stuck with the prospect of stripping out a lot of unnecessary code. Fortunately, there's an easier way.

By creating a new compiler directive, *WithForm*, and testing for its presence, we can conditionally skip or include the code in the TAPI interface unit that's tied to the form unit. Another advantage in dividing the class code from the form code is that it makes creating the component easier. (As you'll notice when we create the TAPI component, we abandoned such checking, and simplified other end-user details as well.)

Now, let's take a look at what was involved in that task of creating the new class. One of the issues we didn't need to worry about in the code we wrote previously was naming conventions, but classes or components we wish to make available to others must have unique names. We needed to come up with something more original than "MyTapi!" We decided to prepend "kkam" (our initials) to each class and component.

In laying the groundwork to create a TAPI component, we also added a number of properties to our class. Particularly with the TAPI component, these properties make it easy to set a number of values at design time. Let's take a look at that component.

A TAPI Component

Separating the *kkamTapiInterface* class from the user interface code improves the structure greatly, and makes it much easier to read the code. There's still a drawback, however: We have to write too much code in our form unit. Instead, we should take advantage of Delphi's component architecture. Having several public and published properties available makes our work considerably easier (see [Figure 4](#)).

Looking back at our earlier prototype, there's also a certain lack of elegance in how we sent strings of information back to our form. Again, Delphi provides a better way to handle this task: *event properties*. Most of the new Delphi events we've written (see [Figure 5](#) for a complete list) enable our component users to easily access the string lists generated in

Property	Scope	Purpose
<i>TAPI_Initialized</i>	public	A Boolean value indicating if TAPI has been successfully initialized.
<i>Dev</i>	public	Integer identifying the line device the calling application and TAPI will use to make a call.
<i>LineIsOpen</i>	public	A Boolean property indicating if the line is open.
<i>PhoneNumber</i>	public	Stores the phone number string we get from the calling application.
<i>MediaMode</i>	public	A DWORD indicating the media mode.
<i>AutoSelectLine</i>	published	A Boolean property indicating if the application should automatically select a line.
<i>AnswerCalls</i>	published	A Boolean property indicating if the application wants owner privileges.
<i>PulseDialing</i>	published	A Boolean property indicating if the application will use pulse dialing.

Figure 4: Key properties of the *kkamTAPI* component.

Event Property	Purpose
<i>OnCreateCallManager</i>	Triggered when a call is initiated through creation of the call manager.
<i>OnShutdownManager</i>	Triggered when the call manager is shut down.
<i>OnDestroyCallManager</i>	Triggered when the call manager object is freed.
<i>OnEnumerateDevices</i>	Triggered when TAPI determines available devices and their capabilities; can be used to populate a combo box.
<i>OnTriggerCommEvent</i>	Triggered when the <i>CommThread</i> or callback function detects a change in status that the calling application might want to display.
<i>OnTAPIInit</i>	Triggered when TAPI is initialized.
<i>OnOpenLine</i>	Triggered when a communications line is opened.
<i>OnDial</i>	Triggered when the phone is dialed (after the call manager is opened).
<i>OnPhoneNumberChange</i>	Triggered when the stored phone number is changed by the application user.
<i>OnChangeMediaMode</i>	Triggered when the media mode changes.
<i>OnChangeBearerMode</i>	Triggered when the bearer mode changes.
<i>OnCommThreadEvent</i>	Used to change the status of the <i>CommThread</i> externally (<i>taSuspend</i> , <i>taResume</i> , <i>taTerminate</i>).

Figure 5: Event properties of the *kkamTAPI* component.

our TAPI component. Let's look at the anatomy of one of these event properties and follow the process.

We're going to work with one of the essential methods we introduced in earlier articles, the *EnumerateDevices* method, which collects and sends a list of available line devices back to a combo box on our main form. By using an event property, we give the component user the option of listing, or not listing, these devices.

Here's the process. First, we need to create a new type for all the events that will send back string list information to the form:

```
type
  TTapiUpdateEvent = procedure(
    Sender: TObject; UpdateInfo: TStringList) of object;
```

Next, we need to declare instances of this type for each TAPI event for which we want to collect information. Let's follow one example through the entire process. In the **interface** section of our *TkkamTAPI* component declaration, we declare the *TTapiUpdateEvent* as follows:

```
FOnEnumerateDevices: TTapiUpdateEvent;
```

In the **protected** section, we declare a method to trigger this event as follows:

```
procedure TriggerEnumerateDevicesEvent; virtual;
```

Now, the event will appear in the Object Inspector and will be easily accessible to developers. Next, we declare the new event property itself in the **published** section:

```
property OnEnumerateDevices: TTapiUpdateEvent
  read FOnEnumerateDevices write FOnEnumerateDevices;
```

Now we need to tie things together. Here's the method we use to trigger this event:

```
procedure TkkamTAPI.TriggerEnumerateDevicesEvent;
begin
  if assigned(FOnEnumerateDevices) then
    FOnEnumerateDevices(Self, DeviceList);
end;
```

The final step is actually triggering the event code. Ironically, we need to go back and change the structure of the *EnumerateDevices* function again, returning it to its original form with no parameters. Because there is no longer a need to send a *TStringList* back to the form as a parameter (we're going to use our new event to do that), we simply make this a local variable. Throughout the function, we build the string list as we did in previous versions. However, when we get to the last line, we insert the following code:

```
TriggerEnumerateDevicesEvent;
```

Now if a developer using our component wants to list these results in a combo box, as we did in previous versions, he or

she can select the event in the Object Inspector and write some code like this:

```
procedure TTapicallManager.kkamTAPI1EnumerateDevices(
  Sender: TObject; UpdateList: TStringList);
begin
  cboxDivices.Items.Assign(UpdateList);
  cboxDivices.Update;
end;
```

Let's take a look at a somewhat different, but equally elegant, event property — one that will manage the communications thread we introduced in Part II.

Elegant Threads

You'll recall that we used our thread class, *TCommThread*, to monitor modem events and to reflect these in modem status lights (bitmaps). Again, because we won't necessarily want to incorporate this in every TAPI application, we have an excellent candidate for an event property. While the process is similar to the one described for the *OnEnumerateDevices* event property, the purpose and the details are quite different. First, we need to declare a new type, so that we can send a message back indicating the type of COMM port activity found in the thread:

```
TCommEvent = (tceDSR, tceDTR, tceCTS, tcePORT);
```

Next, we declare the event type, as before, using our new type as its main parameter:

```
TCommEventProc = procedure(Sender: TObject;
  ACommEvent: TCommEvent; AStatus: Integer) of object;
```

Again, we declare a private instance of our event class:

```
FOnTriggerCommEvent: TCommEventProc;
```

Finally, we declare the trigger procedure (in the **protected** section) and the property itself (in the **published** section) in the following two statements:

```
procedure TriggerCommEvent(Sender: TObject;
  ACommEvent: TCommEvent; AStatus: Integer);
property OnTriggerCommEvent: TCommEventProc
  read FOnTriggerCommEvent write FOnTriggerCommEvent;
```

Here's the method of the *TCommStatus* class where we trigger this event:

```
procedure TCommStatus.TriggerTapiCommEvent(Sender: TObject;
  CommEvent: TCommEvent; Status: Integer);
begin
  TkkamTAPI(TheOwner).TriggerCommEvent(Sender, CommEvent,
  Status);
end;
```

When a *CommEvent* is detected in our thread, we need to communicate that to the main TAPI class, which owns an instance of the *CommThread* object. To pull this off, we

need to get a pointer to the *TkkamTAPI* instance that the *TCommStatus* instance can recognize. We do this by declaring a private variable of type *TComponent*. Then, in *TCommStatus*'s constructor (which has an *Owner* parameter, of course), we include the following statement to keep a copy of the owner:

```
TheOwner := Owner;
```

We're still not quite finished. The *TCommStatus.TriggerTapiCommEvent* method calls the *TkkamTAPI(TheOwner).TriggerCommEvent(Sender, CommEvent, Status)* method. The process involves a whole chain of triggering methods. To understand what is actually happening, let's look at the origin of the chain of events in *TCommStatus*'s *Execute* method, and the final destination in the form unit that sets an event handler to use this information. First, here is the *Execute* method (compare it with the method we used last month):

```
procedure TCommStatus.Execute;
var
  dwEvent: DWord;
  dwStatus: DWord;
begin
  dwEvent := 0;
  SetCommMask(ThePort, EV_DSR or EV_CTS or SETDTR);
  repeat
    WaitCommEvent(THandle(ThePort), dwEvent, nil);
    GetCommModemStatus(THandle(ThePort), dwStatus);
    case dwEvent of
      EV_DSR: TriggerTapiCommEvent(Self, tceDSR, green);
      SETDTR: TriggerTapiCommEvent(Self, tceDTR, green);
      EV_CTS: TriggerTapiCommEvent(Self, tceCTS, green);
    end;
  until Terminated;
end;
```

Here's the event handler:

```
procedure TTapicallManager.kkamTAPI1TriggerCommEvent(
  Sender: TObject; ACommEvent: TCommEvent;
  AStatus: Integer);
begin
  case ACommEvent of
    tceDSR: SetBitmap(DSR, AStatus);
    tceDTR: SetBitmap(DTR, AStatus);
    tceCTS: SetBitmap(CTS, AStatus);
    tcePORT: SetBitmap(PORT, AStatus);
  end;
end;
```

Note that while the *CommThread* class doesn't manage the PORT indicator, the callback function does; that's why it's included. We've already described the elements that connect these two methods.

To review the complex journey we've just finished, the *CommThread* instance within our *kkamTAPI* component monitors and reports on modem events, sending the information back through the *kkamTAPI* component to the calling application, which can then reflect those events in the modem lights we introduced last month.

Before we leave this topic, we'd like to pass along one caveat concerning the creation of custom events. In *Developing Custom Delphi 3 Components* [Coriolis Group Books, 1997], Ray Konopka states, "The single most important issue regarding custom events is that events are optional." The event's triggering code should not be "dependent on the existence of an associated event handler," or on a user responding to the event in a certain way. We take that approach here. While any form that uses this component to initiate a phone call will call the *Dial* method (which in turn will call the *CreateCallManager* method), only those applications that assign an *OnCreateCallManager* event will receive a string list of results from this component.

As we pointed out earlier with our non-visual TAPI component (and our TAPI class), we have many options. We could recreate our fully functional call-manager application by dropping the required visual component onto a form, copying the Modem Lights enabling code, dropping our new TAPI component onto the form, setting various properties, and calling various methods. If you study the sample projects we've included with this article, you'll note that this code is much simpler than any of the previous form units we've presented. Our TAPI component is doing its job, and doing it well!

We can get even simpler. [Figure 3](#) shows another program based on this component. This time, we used only four visual components, our TAPI component, and set a few properties. We've really reduced the amount of code we have to write; we simply need to initialize TAPI, provide handlers for our *Dial* and *Hangup* buttons, and shut down TAPI. That's it.

Conclusion

This series of articles is quickly coming to an end. Of course, there is a great deal more we could do. In terms of implementing the functionality in the TAPI.pas unit, we've only scratched the surface. Fortunately, much of the additional functionality is similar in structure to what we've been working with in this series. Note how easy it was to add the new features we introduced in this article. We could also wrap one of the two forms we've been working with as a dialog box component.

We would like to acknowledge some of the source materials and individuals that provided invaluable help in writing this series of articles. First, the TAPI.pas conversion itself, first written by Alex Staubo and modified by Brad Choate as part of Project JEDI (see Alan's "[Symposium](#)" column in the [December, 1997 Delphi Informant](#)). In writing the *CommStatus* thread, Keith Anderson (keith@airswitch.com) donated the *GetPortHandle* function, which provided the port handle we needed to pass to *SetCommMask*, *GetCommModemStatus*, and any other low-level Comm functions. In crafting the component, Alan found the Component Development Kit (CDK-3 beta) indispensable

in constructing the basic outline. Finally, of all the books we studied, we found *Communications Programming for Windows 95* [Microsoft Press, 1996] by Charles A. Mirho and Andre Terrisse especially helpful.

If you want to build a communications program from scratch, we've provided the basic foundation. If you would rather work with an existing library, we recommend TurboPower's Async Professional, which we both use (see Alan's review of the new version on [page 42](#) in this issue of *Delphi Informant*). In closing, we wish you the best of communications. [▲](#)

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\SEP\DI9809AM.

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.





By Bill Todd

Procedure Variables

Call Different Routines Dynamically at Run Time

Procedure variables let you call procedures, functions, and methods by assigning the address of the routine you wish to call to a variable, then using the variable to call the routine. What makes this useful is that it allows you to vary which procedure gets called at run time. To make things easy, assume for the balance of this article that — unless otherwise stated — the term “procedure” applies to procedures, functions, and methods.

Declaring Procedural Variables

The first step in working with procedural variables is to declare them, and the best way to do that is by declaring a procedural type. The following examples apply to procedures and functions, but not to methods. For example:

```
type
  TSomeProc = (SomeInt: Integer);
  TSomeFunc = (SomeString: string): Boolean;

var
  AProc: TSomeProc;
  AFunc: TSomeFunc;
```

The first type, *TSomeProc*, defines a type that points to a procedure which takes a single integer parameter. The second, *TSomeFunc*, defines a pointer to a function that takes a single string parameter and returns a Boolean value. While you can declare a procedural

```
type
  TSquareFunc = function(AValue: Integer): Integer;

var
  SquarFunc: TSquareFunc;
  I: Integer;

function SquareIt(AValue: Integer): Integer;
begin
  Result := AValue * AValue;
end;

procedure UseSquareIt;
begin
  SquareFunc := SquareIt;
  I := SquareFunc(2);
end;
```

Figure 1: Using a procedural variable.

variable directly (i.e. without first declaring a type), you won't be able to pass the procedural variable as a parameter. For example:

```
var
  SomeProc: procedure(ADate: TDateTime);
```

is a valid declaration for a variable that points to a procedure that takes a single *TDateTime* parameter. However, you can't pass this variable as a parameter to another procedure because there's no way to specify the parameter type.

Moreover, while the declaration:

```
procedure MyProc(AProc: TSomeProc);
```

is valid, the declaration:

```
procedure MyProc(AProc: procedure(ADate:
  TDateTime));
```

is not. Because one of the most useful ways to use a procedural variable is to pass it as a parameter, you'll usually want to declare a type for the variable.

Declaring Method Pointers

The procedural variables in the previous examples contain a single 32-bit value that is the memory address of the entry point of the function or procedure to which the variable points. A pointer to a method needs more information.

Because a method is part of an object, the compiler not only needs to know the address of the method, but also the address of the specific instance of the object whose method is being called. Therefore, two 32-bit addresses are required for a method pointer. The first contains the address of the method, and the second contains the address of the instance of the object. You tell the compiler that you're declaring a method pointer by adding the keywords **of object** to the end of the declaration. To convert the declaration shown earlier from a procedure pointer to a method pointer, change it as follows:

```
type
  TSomeProc = (SomeInt: Integer) of object;
  TSomeFunc = (SomeString: string): Boolean of object;
```

Using Procedural Variables and Method Pointers

To use a procedural variable, assign the procedure to which you want it to point to the variable (see [Figure 1](#)). The procedure *UseSquareIt* first assigns the address of the function *SquareIt* to the procedural variable *SquareFunc*, and then uses the variable to call the function. Once you have assigned the value of a procedure to a variable, you call the procedure by using the variable name exactly as though it was the name of the procedure.

Using a method pointer is almost identical. The best examples of method pointers in Delphi are the events that are part of most Delphi components. The events that appear on the Events page of the Object Inspector in Delphi are nothing more than properties whose type is a method pointer. Consider the *OnClick* event handler of a button. If you look at the events for the *TButton* class in the online Help, you'll see that the *OnClick* event is defined as:

```
property OnClick: TNotifyEvent;
```

so this "event" is simply a property named *OnClick* whose type is *TNotifyEvent*. If you look in the Classes unit in the VCL source code, or at *TNotifyEvent* in the online Help, you'll find it's declared as:

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

so *TNotifyEvent* is the name of a method pointer type that points to a method, which is a procedure, and takes a single parameter, which is a pointer to an object of type *TObject*, or one of its descendants.

If events are really just properties, why do they appear on the Events page of the Object Inspector instead of the Properties page? The Object Inspector displays any property whose type is a method pointer on the Events page, and all other properties on the Properties page.

Because an event is really a method pointer, the code in [Figure 2](#) shows how to assign a value to a method pointer, assuming a form, *Form1*, contains a button, *Button1*. This code works

```
procedure Form1.MyOnClickHandler(Self: TObject);
begin
  MessageBeep(0);
end;

procedure Form1.AssignEvent;
begin
  Button1.OnClick := MyOnClickHandler;
end;
```

Figure 2: Assigning a method pointer.

```
procedure Form1.SetHandler(Flag: Integer);
begin
  case Flag of
    1: Button1.OnClick := ClickHandler1;
    2: Button1.OnClick := ClickHandler2;
    3: Button1.OnClick := ClickHandler3;
  else
    Button1.OnClick := nil;
  end;
end;
```

Figure 3: Changing event handlers at run time.

because *AssignEvent* is a method of *Form1*, and both the button and the method *MyOnClickHandler* are part of the *TForm1* type declaration. If *AssignEvent* was not a method of *Form1*, the assignment would have to be changed to:

```
Form1.Button1.OnClick := Form1.MyOnClickHandler;
```

The fact that events are method pointers is very powerful; it means you can change the event handler that's assigned to any event any time you want to while your application runs. You can also temporarily disconnect an event handler so that no code is executed when an event occurs, as shown in [Figure 3](#). If the value of *Flag* is anything other than 1, 2, or 3, the *OnClick* event (which is really a method pointer) is set to **nil** so that no event handler will be called when the button's *OnClick* event fires.

Normally, when a procedural variable appears in a statement, the compiler assumes you want to call the procedure to which the variable points. As you've seen in the preceding examples, this is not true when the procedural variable appears on the left side of the assignment operator. When a procedural variable appears on the left side of an assignment, the compiler knows that the right side of the assignment must supply a procedural value (a pointer to a procedure).

However, there are some cases where the compiler cannot tell what you want to do. Consider the following code:

```
type
  TMathFunc = function: Integer;

var
  SquareFunc: TMathFunc;

procedure CompareIt;
begin
  if SquareFunc = SquareIt then
    MessageBeep(0);
end;
```

This code raises a question. Does the author want to call the function pointed to by the procedural variable *SquareFunc*, and call the function *SquareIt* and compare the values returned by the two functions to see if they're equal? Or does the author want to compare the address of *SquareIt* to the address contained in *SquareFunc* to see if the procedural variable *SquareFunc* points to the function *SquareIt*?

The compiler will, in fact, call both functions and compare the returned values. If you want to compare addresses, change the code to:

```
if @SquareFunc = @SquareIt then
  MessageBeep(0);
```

The “address of” operator, @, tells the compiler to convert the argument to a pointer instead of calling the function. If you need to get the memory address where the procedural variable resides, use @@, e.g. @@SquareFunc.

Procedural variables are assignment compatible if all the following conditions are met:

- Both must have the same number of parameters.
- Both must have the same calling convention (**pascal**, **cdecl**, **stdcall**, and **safecall**).
- The parameters in corresponding positions must have the same type.
- For functions, the return values must have the same type.

Procedure variables and method pointers are never compatible. Additionally, the constant **nil** is compatible with any procedural variable or method pointer.

Other Uses

How else can you use procedural variables? Suppose you need to create a generic way for users to run reports in your program. You would like to display a dialog box that prompts the user for the starting and ending page number, the number of copies, and whether to send the report to the screen or the printer.

That part is easy. But what if some of your reports require some code to be run before the report is printed, and some require code to be run after the report is printed? Suddenly, the routine isn't generic anymore.

A solution to the problem is to pass the pre- and postprocessing routines as procedural variables (see [Figure 4](#)). This procedure takes four parameters:

- 1) the procedure to call before the report is run,
- 2) the procedure to call after the report is run,
- 3) the parameters for the preprocessing procedure, and
- 4) the parameters for the postprocessing procedure.

For flexibility, a single variant is used as the parameter for both the preprocessing and postprocessing routines. Because a variant can also be an array of any type (including variant), it can be used to pass any number of values of any type, so

```
type
  TRptProc = procedure(Params: Variant);

procedure RunReport(
  PreProcessing, PostProcessing: TRptProc;
  PreParams, PostParams: Variant);
begin
  RptDlg := TRptDlg.Create(Application);
  try
    with RptDlg do begin
      ShowModal;
      Rpt.Copies := StrToInt(CopiesEdit.Text);
      Rpt.StartPage := StrToInt(StartPageEdit.Text);
      Rpt.EndPage := StrToInt(EndPageEdit.Text);
      if Assigned(PreProcessing) then
        PreProcessing(PreParams);
      Rpt.Print;
      if Assigned(PostProcessing) then
        PostProcessing(PostParams);
    end;
  finally
    RptDlg.Free;
  end;
end;
```

Figure 4: Passing procedures as parameters.

that all of the pre- and postprocessing routines don't have to have the same number and type of parameters.

The procedure creates the dialog form and shows it modally. When the user closes the dialog box, the values from the dialog box's edit controls are assigned to the report object's properties. Next, the statement:

```
if Assigned(PreProcessing) then
  PreProcessing(PreParams)
```

calls the preprocessing procedure, if there is one. The *Assigned* function returns True if the procedural variable *PreProcessing* isn't equal to **nil**, and False if it's equal to **nil**. Finally, the report is printed and the postprocessing routine, if any, is called.

This technique can be used to make many routines generic that otherwise could not be. Another example is a custom file import routine that could be made generic, except that the structure of the table the data is imported into is not the same. The solution is to load the values from the input record into a variant array. A procedure variable that points to a routine that will insert the imported values into a table is passed as a parameter to the import routine and the import routine calls it, passing the variant array that contains the imported values as a parameter.

Yet another example would be to make the sorting routines from Rod Stephens' article “[Sorts of All Types](#),” in the [January, 1998 Delphi Informant](#), more flexible. Suppose you want to sort arrays of Pascal records. You can modify any of the sort routines to do this, but now they are not generic (because you'll have to change the code that compares members of the array to use the appropriate field in the particular record to determine which array element is greater). The solution is to pass the sort routine a procedural variable that points to a function that returns -1 if

the first record is less than the second, 0 if they are equal, and 1 if the first record is greater. Now all the code specific to the array being sorted is isolated in the function that is passed to the sort routine.

Conclusion

Procedure variables and method pointers will let you change the routine that is called by your code dynamically at execution time. This lets you control which event handler responds to any event, at any given time, and allows you to write generic routines in many situations where you otherwise could not. To learn more about procedural variables, see chapters 4, 5, and 6 in the *Delphi 3 Object Pascal Language Guide*. ▲

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





By John Penman



Multicasting

When You'd Rather Push Than Pull

Push media is one of the hottest topics on the Internet. Much of it is marketing hype; however, the technology that lies behind the hype — *multicast* — is solid, practical, and ready now.

As I write this, most content providers use point-to-point (i.e. *unicast*) to push media. For example, when you use FTP, you are using unicast to download a file sequentially from the server. There may be other users waiting in the queue to download the same file. Some content providers are using unicast to push data to each client (there may be thousands) one by one. Obviously, this can take up considerable bandwidth to propagate the same data.

Conversely, multicast is bandwidth-friendly because it provides content to a group of clients at the same time. The only requirement for clients is to “tune in” to receive

the content. The standard analogy is that of a radio station that transmits its programs to thousands of listeners at the same time. In a similar way, multicast enables the data to be broadcast to many users at once (see [Figure 1](#)).

So multicast is the enabling technology that allows the sender to “push” data to receivers. This is opposite behavior of using our Web browsers to retrieve, or “pull,” information from the Internet. Moreover, receiving information this way is better than receiving it by radio or TV, because you can customize how you want to receive the information.

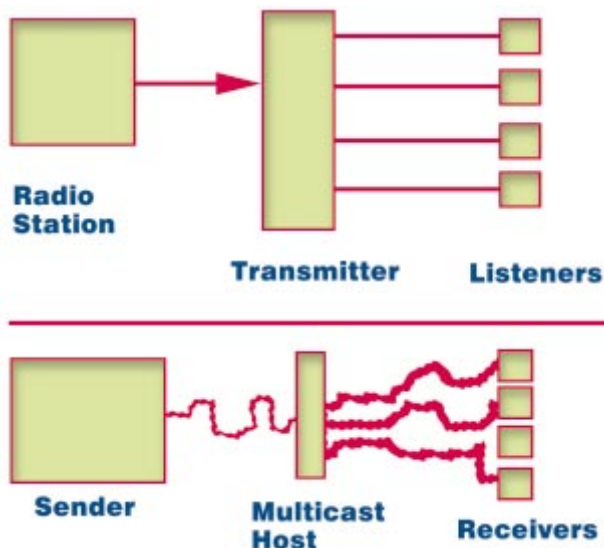


Figure 1: Sketch of how multicast works (using a radio analogy).

There are two data propagation models for multicast: one-to-many and many-to-many. Examples of the one-to-many model are:

- software updates
- stock exchange news updates
- weather forecasts
- real-time measurement from a single source to multiple recipients
- database synchronization

Examples of the many-to-many model are:

- real-time video conferencing
- document sharing and editing
- virtual reality
- gaming

As you can see, the possibilities for multicast are limitless.

Multicast Mechanics

Multicast uses the User Datagram Protocol (UDP) for sending discrete packets. Unlike TCP, which manages data integrity, UDP does not keep track of data sent between the sender and the receiver. This means that packets of data could arrive out of order, be duplicated, or simply disappear into the void. Because of this apparent handicap, well-known services that use TCP (such as FTP, NNTP, and SMTP) are not amenable to multicast. Institutions like the Internet Engineering Task Force (IETF), universities, and software houses are researching ways of making multicast available to these services. There's already a product on the market that offers reliable multicast file transfer.

A multicast sender sends data to a multicast address (typically that of a multicast host), which, in turn, broadcasts the data to any multicast client that may be listening to that multicast address. In essence, the multicast host or router is a transmitter of data. The multicast address ranges from 224.0.0.0 through 239.255.255.255. Any host can send data to a multicast address. However, only receivers that request data from a multicast address can receive.

The sender and receiver applications we'll develop follow these essential steps. Before the host (the sender) can broadcast, it must perform the following steps:

- 1) create a UDP socket
- 2) set the Time To Live value (TTL) to reach the intended audience
- 3) send
- 4) close the UDP socket on completion

Before the client (the receiver) can receive the data, it must do the following:

- 1) create a UDP socket
- 2) bind to a port number that corresponds to a "channel"
- 3) join the multicast group
- 4) receive
- 5) close the UDP socket on completion

Although UDP provides the transport of data between the sender and the receiver(s), it's the Internet Group Management Protocol (IGMP) that provides the framework for multicast. Steve Deering's RFC1112 document, "Host Extensions for IP Multicasting," establishes the design of IGMP.

Multicast and WinSock 2

In the world of Windows networking, multicast has been available on some implementations of WinSock 1.1 stacks (although it wasn't part of the WinSock 1.1 specification). We use the BSD sockets API, like *setsockopt* in WinSock 1.1 and 2. Multicast is part of the WinSock 2 specification, which has APIs specific to multicast. We'll learn briefly how to use one of these multicast-specific APIs in WinSock 2 later in the article.

Multicast and Delphi

Generally, it's simple to build multicast client applications with Delphi; it's simpler still with multicast servers. Using

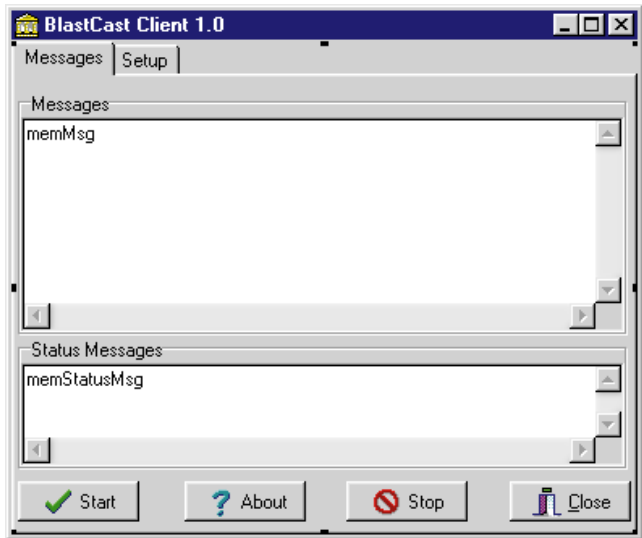


Figure 2: BlastCast Client's Messages page at design time.

the Delphi interface units *Winsock.pas* and *Winsock2.pas*, you can develop simple multicast applications easily.

I'll demonstrate this by developing two simple multicast applications: BlastCast Server and BlastCast Client. The BlastCast Server application broadcasts a message to all PCs on the local network that have the BlastCast Client up and running. Although these are simple applications, you can use them — for example — to communicate between an irate project manager and his minions. Both applications use multithreading. (For more details on using WinSock 2 with threads, see part two of John Penman's "WinSock 2" series in the *July, 1998 Delphi Informant*.)

Tuning the BlastCast Client Application

Figure 2 shows the BlastCast Client application in the IDE. The *tsMsg* TabSheet control has two Memo controls: *memMsg* displays messages pushed by the server; *memStatusMsg* reports status and error messages from the application. Before you can receive a broadcast, you need to initialize important variables in the *tsSetUp* TabSheet control (see Figure 3). You assign the data to the port number and the multicast address.

It's crucial to use the same port number and multicast address that the BlastCast Server uses. I set the port to 9,500, but it's purely arbitrary; you can use any number. The multicast address, 234.5.6.7, is also arbitrary, but the restriction is that the address must lie in the range of 244.0.0.0 through 239.255.255.255. There's one CheckBox control that you can tick if you wish BlastCast Client to use *WSAJoinLeaf*, a WinSock 2-specific API function for multicast.

We'll leave this for now. Before spinning the thread, the application stores the configuration in the *Options* record.

Listening to Your Favorite Station

To start listening for broadcast messages from BlastCast Server, click the **Start** button. It performs two tasks: first,

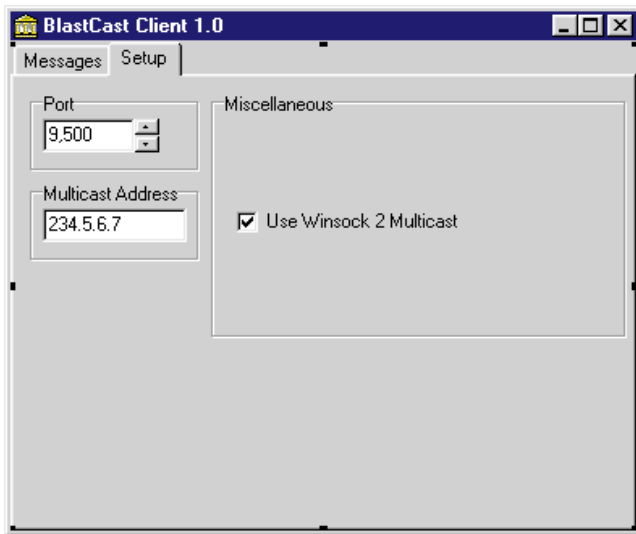


Figure 3: BlastCast Client's Setup page.

it spins a thread to listen for “push” data; second, it minimizes the application's window, preserving your screen's valuable footprint. The following two statements show how:

```
thrdListen := TListenThrd.Create(Options);
frmMain.WindowState := wsMinimized;
```

This piece of code is shown in [Listing One \(beginning on page 27\)](#). The constructor, *TListenThrd.Create* (see [Listing Two beginning on page 28](#)), performs several preliminary steps:

- 1) Calling WinSock 2. If WinSock 2 fails to load, the program aborts.
- 2) Assigning the *OnTerminate* property to the *CleanUp* event handler.
- 3) Creating a UDP socket using the *socket* function, or a *WSASocket* if you checked the CheckBox control on the Setup page.
- 4) Creating an event object, *EventMsg*.
- 5) Calling *WSAEventSelect* to associate *EventMsg* with the *FD_READ* network event.
- 6) Setting up socket attributes for multicast.

Because we covered steps 1 through 5 in detail in a previous article (again, see part two of Penman's “WinSock 2” series in the [July, 1998 Delphi Informant](#)), we'll concentrate on step 6. In step 6, the constructor makes several calls to *setsockopt* (a WinSock API function) to prepare the socket for multicast. In Delphi, the formal definition of *setsockopt* is:

```
function setsockopt(s: TSocket; level, optname: Integer;
  optval: PChar; optlen: Integer): Integer; stdcall;
```

The first call sets the *sktBlast* socket to be reusable for all messages it receives. Otherwise, WinSock will generate a *WSAEADDRINUSE* fault. To enable this attribute, you set the *ReUseFlag* to True:

```
// Always reuse socket.
ReUseFlag := True;
sktRes := setsockopt(sktBlast, SOL_SOCKET, SO_REUSEADDR,
  @ReUseFlag, SizeOf(ReUseFlag));

if sktRes = SOCKET_ERROR then
begin
  StatusMsg := Concat('Call to setsockopt failed! Error ',
    IntToStr(WSAGetLastError));
  Synchronize(DisplayStatusMsg);
  closesocket(sktBlast);
  Synchronize(ChangeWS);
  Exit;
end;
```

If the call to *setsockopt* fails, the constructor calls *WSAGetLastError* to display the cause of the error, close the socket, and leave.

On success, the construct calls the *bind* function to associate the socket with the port number the application requires for multicast. Then the application “joins” the multicast group by calling *JoinSession*. The method fills the fields of the *Multicast* record, which *setsockopt* uses as a fourth parameter. The third parameter (*IP_ADD_MEMBERSHIP*) notifies WinSock that the socket (*sktBlast*) is a member of the multicast group. This fragment from *JoinSession* follows:

```
with Multicast do begin
  // Multicast address.
  imr_multiaddr.s_addr := inet_addr(MultiCastAddr);
  // Any interface address.
  imr_interface.s_addr := INADDR_ANY;
end;

sktRes := setsockopt(sktBlast, IPPROTO_IP,
  IP_ADD_MEMBERSHIP, PChar(@Multicast), SizeOf(Multicast));

if sktRes = SOCKET_ERROR then
begin
  StatusMsg := Concat(
    'Call to setsockopt failed! Error ',
    IntToStr(WSAGetLastError));
  Synchronize(DisplayStatusMsg);
  Result := False;
end;
```

The call to *setsockopt* must succeed before the application can receive any multicast datagrams. Finally, the constructor calls *Resume* to execute the listening thread.

Using the WinSock 2 Multicast API

What follows will only apply if you checked the CheckBox control in the Setup TabSheet (again, see [Figure 3](#)). The constructor uses the *WSASocket* API instead of a socket. The procedure, *JoinSession*, calls *WSAJoinLeaf* like this:

```
sktTemp := WSAJoinLeaf(sktBlast, @RemoteAddr,
  SizeOf(RemoteAddr), nil, nil, nil,
  nil, JL_RECEIVER_ONLY);
```

The parameters set to *nil* are structures that pertain to callee, caller data, and quality of service (QOS), which are beyond the scope of this article. The parameter that concerns us is the last one, which specifies whether the socket can only

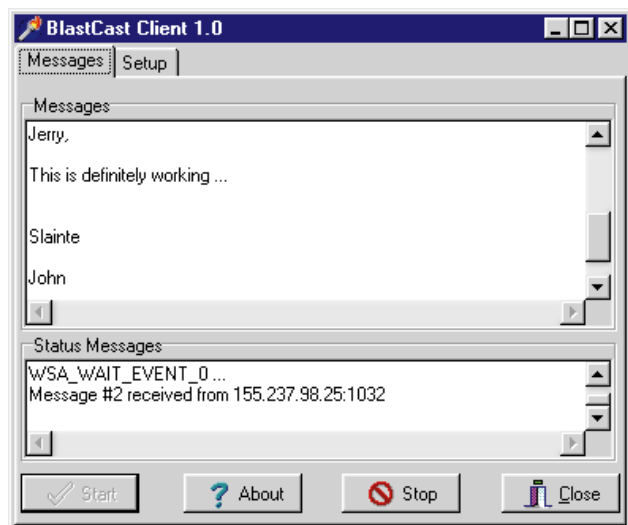


Figure 4: BlastCast Client in action.

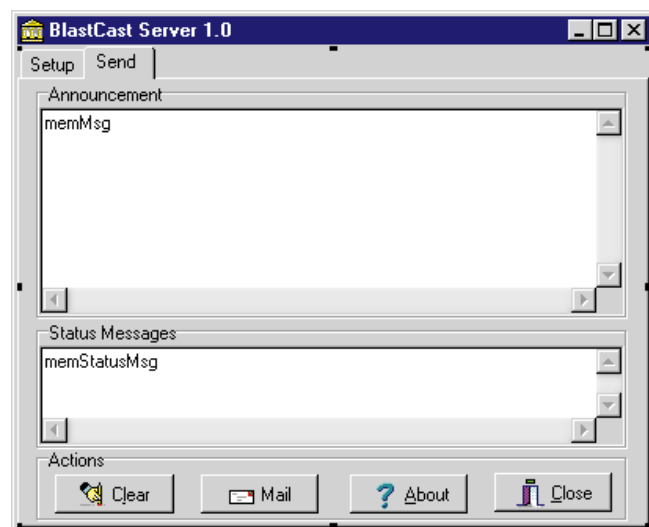


Figure 5: BlastCast Server's Send page at design time.

receive, only send, or do both. The `JL_RECEIVER_ONLY` value only allows receives.

Listening for Messages

The BlastCast Client application is ready to accept messages. It uses the same approach as the SFTPClient application to listen for messages on a thread (again, see part two of Penman's series in the *July, 1998 Delphi Informant*). Although we covered this topic in detail, I'll sketch an outline of what happens.

When a message arrives, inside the *Execute* method *WSAWaitForMultipleEvents* wakes up from its blocked state, and calls *HandleSocketEvent* to handle the network event, `FD_READ`. In turn, *HandleSocketEvent* calls *GetMsg* to process the data pushed by the BlastCast Server application.

After *WSARecvFrom* gets the data, *GetMsg* calls *ChangeWS* to send a beep and to change the BlastCast Client's window state to alert the user of a fresh message. To update the window state safely from the *thrdListen* thread, the

application calls *Synchronize* with *ChangeWS* as its parameter. You can see the result of receiving the message in [Figure 4](#).

Channel Hopping?

When you want to leave the multicast group to join another group, or switch off, click the **Stop** button to call *thrdListen.CloseSession*, as the following code shows:

```
procedure TfrmMain.bbbtnStopClick(Sender: TObject);
begin
  // Leave the multicast group.
  if thrdListen <> nil then
    begin
      thrdListen.CloseSession;
      bbbtnStart.Enabled := True;
      bbbtnStop.Enabled := False;
    end;
end;
```

The *thrdListen.CloseSession* calls *LeaveSession* to leave the multicast group and kill the thread. After initializing the *Multicast* record, *LeaveSession* calls the *setsockopt* API like this:

```
sktRes := setsockopt(sktBlast, IPPROTO_IP,
  IP_DROP_MEMBERSHIP, PChar(@Multicast),
  SizeOf(Multicast));
```

The second parameter, `IP_DROP_MEMBERSHIP`, tells BlastCast Client to leave the group.

At this time, version 1 of the IGMP protocol regards this operation as a NOOP (NO OPERATION), which makes this step unnecessary. Version 2 of IGMP may recognize this operation. In any case, it's always good policy to clean up before leaving.

After this call, *thrdListen.CloseSession* sets *Done* to `True`, and calls *WSASetEvent* to set the *EventMsg* object. This action signals the end of the thread. Note that when we use the *WSAJoinLeaf* API, there is no corresponding API to leave the group. Calling the *closesocket* API function (in *CleanUp*) performs this task.

The BlastCast Server Application

The BlastCast Server application is very similar to BlastCast Client, but I'll sketch out some features. [Figure 5](#) shows the BlastCast Server in the IDE. The *tsSend* page contains two Memo controls — *memMsg* and *memStatusMsg* — for entering messages and reporting errors, respectively.

Before you can broadcast data beyond the subnet, you must adjust the BlastCast Server's Time To Live (TTL), i.e. the number of hops (multicast routers) between the sender and the receiver. The default is 1, which allows only BlastCast Client applications on the local subnet to receive the broadcast. To broadcast beyond the local network, you would need to increase the TTL, which you can do on the Setup page (see [Figure 6](#)).

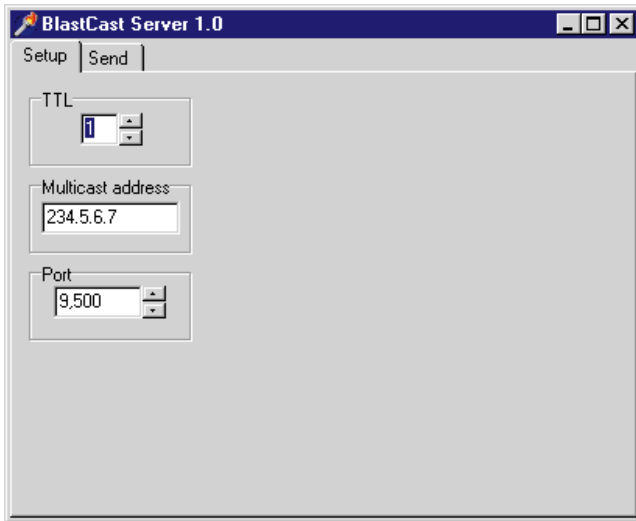


Figure 6: BlastCast Server's Setup page.

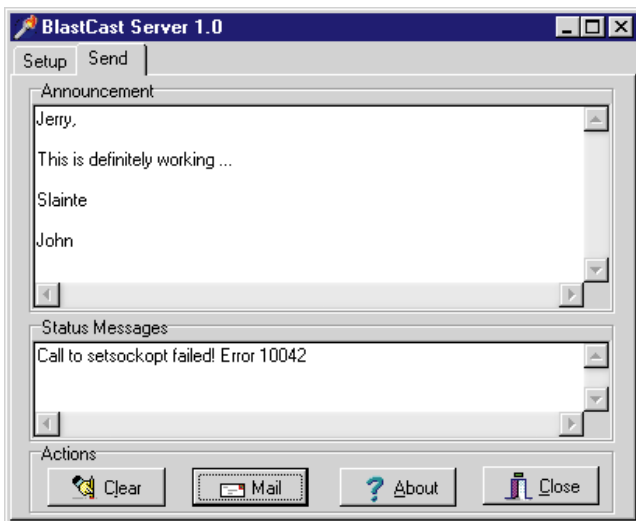


Figure 7: BlastCast Server in action.

After entering your message, click the **Mail** button to send it. Like BlastCast Client, the action creates a thread to send the message:

```
SendMsgThrd := TSendMsgThrd.Create(memMsg, Options);
```

Unlike BlastCast Client's thread, however, this thread is destroyed once the message has been sent. (In BlastCast Client, the thread is persistent until you click the **Stop** button.) BlastCast Server creates a new thread for every message and destroys the thread after sending. It's unnecessary to use multithreading for BlastCast Server, but the code illustrates another approach to multithreading.

The first parameter in `TSendMsgThrd.Create` (see [Listing Three beginning on page 29](#)) passes the Memo control (`memMsg`) to the thread. The second parameter passes the `Options` record containing configuration details like the port number, TTL, and multicast address.

In the constructor, BlastCast Server follows the same steps that BlastCast Client takes to create and bind a UDP sock-

```
procedure TSendMsgThrd.BlastMsg;
var
  Size      : Byte;
  sktRes    : Integer;
  WorkMsg   : array [0..MAXGETHOSTSTRUCT-1] of Char;
  Buffer     : PChar;
begin
  with RemoteAddr do begin
    sin_family      := AF_INET;
    sin_addr.s_addr := inet_addr(pchar(MCOptions.Address));
    sin_port        := htons(MCOptions.Port);
  end;

  Size := Memo.GetTextLen;
  Inc(Size);
  Buffer := nil;

  try
    GetMem(Buffer, Size);
    Memo.GetTextBuf(WorkMsg, SizeOf(WorkMsg));
    sktRes := sendto(sktBlast, WorkMsg, SizeOf(WorkMsg), 0,
                    TSocketAddrIn(RemoteAddr),
                    SizeOf(RemoteAddr));
    if sktRes = SOCKET_ERROR then
      begin
        Msg := Concat('Call to sendto failed! Error ',
                     IntToStr(WSAGetLastError));
        Synchronize(DisplayMsg);
        closesocket(sktBlast);
        Exit;
      end;
  finally
    FreeMem(Buffer, Size); // Frees buffer memory.
  end;
end;
```

Figure 8: BlastCast Server's `TSendMsgThrd.BlastMsg` procedure.

et. In addition, BlastCast Server joins the multicast group by a call to `setsockopt` with the `IP_ADD_MEMBERSHIP` parameter. According to RFC1112, it's unnecessary for a sender that doesn't receive data to join a multicast group. However, the Microsoft implementation of multicast requires this step.

The next step is to call `setsockopt` again to set the TTL:

```
sktRes := setsockopt(sktBlast, IPPROTO_IP,
                    IP_MULTICAST_TTL, PChar(@MCOptions.TTL),
                    SizeOf(MCOptions.TTL));
```

The final step before executing the thread via `Resume` is to disable the loopback feature. In Microsoft's implementation, the call to `setsockopt` with the `IP_MULTICAST_LOOP` parameter always fails (hence the display of the error message in [Figure 7](#)).

BlastCast Server calls `BlastMsg` (see [Figure 8](#)) to send the contents of the Memo control. This code has a glaring weakness. It's possible that the contents of the Memo control are larger than the `WorkMsg` array can handle. (It can only handle 1,024 characters, which is not sufficient to send the contents at one blast.) The work-around to this problem is to use a **repeat** loop.

After sending the message, the thread dies, and BlastCast Server is ready to blast another message.

Using BlastCast Client and BlastCast Server Applications

BlastCast Client and BlastCast Server are available for download (see end of article for details). Install BlastCast Client on several machines on the local network, and BlastCast Server on one machine. Your network must be running TCP/IP. If you wish to put BlastCast Server on a network on the other side of the gateway, you'll need to increase its TTL value.

Make sure you have the Winsock2 unit (which is included with BlastCast Client) on the path. I suggest you put the unit in the Delphi's \Lib directory. Providing you have WinSock 2 on those machines, you should be able to start sending and receiving messages.

If there's no WinSock 2 on your machines, point your browser at <http://www.sockets.com/winsock2.htm> to get information on where to obtain the WinSock 2 SDK.

Conclusion

You've seen how easy it is to build simple multicast applications. We built a sender application that sends data, and a receiver application to receive data. Both applications conform to the one-to-many data distribution model. It's a short step to expand these applications to peer-to-peer chat programs that use the many-to-many data propagation model.

Earlier, I mentioned that because multicast uses UDP for the transmission of data, it would seem to rule out the use of multicast for services like FTP. However, in the near future, I will demonstrate how to deploy our own multicast client and server for a well-known service. To find out more about multicast, check out these two sites:

- <http://ipmulticast.com> provides links to additional research and commercial initiatives in multicast
- <http://www.mbone.com>

Let the big push begin! 

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM98\SEP\DI9809JP.

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 — Main.pas (from BlastCast Client)

```
unit main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Buttons, ComCtrls, Winsock2,
```

```
Mask;

const
  MultiCastAddr = '234.5.6.7';
  BlastCastPort = 9500;
  BlastCastTTL = 2;
  LoopBack = True;
  ReUse = True;
  UseWS2 = False;

type
  TfrmMain = class(TForm)
    pcBlastClient: TPageControl;
    tsMsg: TTabSheet;
    tsSetUp: TTabSheet;
    bbtnStart: TBitBtn;
    bbtnStop: TBitBtn;
    bbtnAbout: TBitBtn;
    gbPort: TGroupBox;
    edPort: TEdit;
    udPort: TUpDown;
    gbMultiCastAddr: TGroupBox;
    bbtnExit: TBitBtn;
    gbUseWS2MC: TGroupBox;
    ckbWS2: TCheckBox;
    gbStatusMsg: TGroupBox;
    memStatusMsg: TMemo;
    gbMessages: TGroupBox;
    memMsg: TMemo;
    edMCAddr: TEdit;
    procedure bbtnStopClick(Sender: TObject);
    procedure bbtnStartClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure bbtnExitClick(Sender: TObject);
    procedure bbtnAboutClick(Sender: TObject);
  private
  end;

var
  frmMain: TfrmMain;

implementation

{$R *.DFM}

uses
  ListenThrd, About;

procedure TfrmMain.bbtnStopClick(Sender: TObject);
begin
  // Leave the multicast group.
  if thrdListen <> nil then
  begin
    thrdListen.CloseSession;
    bbtnStart.Enabled := True;
    bbtnStop.Enabled := False;
  end;
end;

procedure TfrmMain.bbtnStartClick(Sender: TObject);
var
  Position: Integer;
  WorkStr: string;
begin
  with Options do begin
    WorkStr := edPort.Text;
    Position := Pos(',', WorkStr);
    if Position > 0 then
      delete(WorkStr, Position, 1);

    Port := StrToInt(WorkStr);
    Address := edMCAddr.Text;
    UseWS2 := ckbWS2.Checked;
```

```

end;

bbtnStart.Enabled := False;
bbtnStop.Enabled := True;
thrdListen := TListenThrd.Create(Options);
frmMain.WindowState := wsMinimized;
end;

procedure TfrmMain.FormCreate(Sender: TObject);
begin
  memMsg.Clear;
  memStatusMsg.Clear;
  edPort.Text := IntToStr(BlastCastPort);
  udPort.Position := BlastCastPort;
  edMCAddr.Text := MultiCastAddr;
  ckbWS2.Checked := UseWS2;

  with Options do begin
    Port := StrToInt(edPort.Text);
    Address := edMCAddr.Text;
    UseWS2 := ckbWS2.Checked;
  end;
end;

procedure TfrmMain.bbtnExitClick(Sender: TObject);
begin
  Close;
end;

procedure TfrmMain.bbtnAboutClick(Sender: TObject);
begin
  try
    frmAbout := TfrmAbout.Create(nil);
    frmAbout.ShowModal;
  finally
    frmAbout.Free;
  end;
end;

```

End Listing One

Begin Listing Two — *TListenThrd.Create*

```

constructor TListenThrd.Create(Options: TMCOptions);
var
  sktRes: Integer;
begin
  inherited Create(True);
  MCOptions := Options;
  FreeOnTerminate := True;
  Started := Start;
  if not Started then
    begin
      StatusMsg := 'Cannot load Winsock 2.0!';
      Synchronize(DisplayStatusMsg);
      Exit;
    end;

  OnTerminate := Cleanup;
  MsgNo := 0;
  with MCOptions do begin
    // Get a UDP socket.
    if UseWS2 then
      begin
        // Convert address string to a value.
        RemoteAddr.sin_family := AF_INET;
        AddrStrSize := 16;
        sktRes := WSAStrToAddress(Pchar(MultiCastAddr),
          AF_INET, nil, @RemoteAddr,
          @AddrStrSize);
        if sktRes = SOCKET_ERROR then

```

```

begin
  StatusMsg := Concat(
    'Call to WSAStrToAddress failed! Error ',
    IntToStr(WSAGetLastError));
  Synchronize(DisplayStatusMsg);
  Exit;
  end;
  sktBlast := WSASocket(AF_INET, SOCK_DGRAM,
    IPPROTO_UDP, nil, 0,
    WSA_FLAG_OVERLAPPED or
    WSA_FLAG_MULTIPOINT_C_LEAF or
    WSA_FLAG_MULTIPOINT_D_LEAF);

  end
else
  sktBlast := socket(AF_INET, SOCK_DGRAM, 0);

if sktBlast = SOCKET_ERROR then
  begin
    StatusMsg := Concat(
      'Failed to create UDP socket! Error ',
      IntToStr(WSAGetLastError));
    Synchronize(DisplayStatusMsg);
    Exit;
  end;

  // Create an event.
  EventMsg := CreateEvent(nil, False, False, nil);
  if EventMsg = WSA_INVALID_EVENT then
    begin
      StatusMsg := Concat(
        'Failed to create Message Event. Error ',
        IntToStr(WSAGetLastError));
      Synchronize(DisplayStatusMsg);
      Done := True;
      Exit;
    end;

  // Now set up notification.
  sktRes := WSAEventSelect(sktBlast, EventMsg, FD_READ);
  if sktRes = SOCKET_ERROR then
    begin
      StatusMsg := Concat(
        'Call to WSAEventSelect failed for socket ',
        IntToStr(sktBlast), '. Error ',
        IntToStr(WSAGetLastError));
      Synchronize(DisplayStatusMsg);
      closesocket(sktBlast);
      Done := True;
      Exit;
    end;

  // Set up socket options for multicast.
  // Always reuse socket.
  ReUseFlag := True;
  sktRes := setsockopt(sktBlast, SOL_SOCKET,
    SO_REUSEADDR, @ReUseFlag,
    SizeOf(ReUseFlag));
  if sktRes = SOCKET_ERROR then
    begin
      StatusMsg := Concat(
        'Call to setsockopt failed! Error ',
        IntToStr(WSAGetLastError));
      Synchronize(DisplayStatusMsg);
      closesocket(sktBlast);
      Synchronize(ChangeWS);
      Exit;
    end;

  // Name the socket and assign the local port number
  // to receive on.
  with LocalAddr do begin
    sin_family := AF_INET;
    sin_addr.s_addr := htonl(INADDR_ANY);

```



```

    sin_port      := htons(MCOptions.Port);
end;

// Bind it!
sktRes := bind(sktBlast, LocalAddr,
              SizeOf(TSockAddrIn));
if sktRes = SOCKET_ERROR then
begin
    StatusMsg := Concat('Call to bind failed! Error ',
                       IntToStr(WSAGetLastError));
    Synchronize(DisplayStatusMsg);
    closesocket(sktBlast);
    Exit;
end;

if UseWS2 then
begin
    // Swap host to network order.
    with RemoteAddr do begin
        sin_family := PF_INET;
        sktRes := WSAshton(sktBlast, MCOptions.Port,
                          @sin_port);
        if sktRes = SOCKET_ERROR then
            begin
                StatusMsg := Concat(
                    'Call to WSAshton failed! Error ',
                    IntToStr(WSAGetLastError));
                Synchronize(DisplayStatusMsg);
                closesocket(sktBlast);
                Exit;
            end;
        end;
    end;
end;

// Join the multicast group.
if not JoinSession then
begin
    closesocket(sktBlast);
    Exit;
end;

StatusMsg := 'Started listening thread ...';
Synchronize(DisplayStatusMsg);

end; // with MCOptions.

Resume;

end;

```

End Listing Two

Begin Listing Three — TSendMsgThrd.Create

```

constructor TSendMsgThrd.Create(MsgMemo: TMemo;
    Options: TMCOptions);
var
    sktRes: Integer;
begin

    inherited Create(True);
    FreeOnTerminate := True;
    OnTerminate := Cleanup;
    Started := Start;

    if not Started then
        begin
            Msg := 'Cannot load Winsock 2.0!';
            Synchronize(DisplayMsg);
            Exit;
        end;

    Memo := TMemo.Create(nil);

```

```

Memo := MsgMemo;
MCOptions := Options;
sktBlast := socket(AF_INET, SOCK_DGRAM, 0);

if sktBlast = INVALID_SOCKET then
    begin
        Msg := Concat(
            'Error creating datagram socket! Error ',
            IntToStr(WSAGetLastError));
        Synchronize(DisplayMsg);
        Exit;
    end;

// Bind the datagram socket.
with LocalAddr do begin
    sin_family := AF_INET;
    // Any old interface.
    sin_addr.s_addr := htonl(INADDR_ANY);
    sin_port := 0;
end;

sktRes := bind(sktBlast, LocalAddr, SizeOf(TSockAddrIn));
if sktRes = SOCKET_ERROR then
    begin
        Msg := Concat('bind failed! Error ',
                     IntToStr(WSAGetLastError));
        Synchronize(DisplayMsg);
        closesocket(sktRes);
        Exit;
    end;

// Join the multicast group using setsockopt.
with Multicast do begin
    // IP multicast address of group.
    imr_multiaddr.s_addr :=
        // Local IP address of interface.
        inet_addr(Pchar(MCOptions.Address));
    imr_interface.s_addr := INADDR_ANY;
end;

sktRes := setsockopt(sktBlast, IPPROTO_IP,
                    IP_ADD_MEMBERSHIP, pchar(@multicast),
                    SizeOf(multicast));

if sktRes = SOCKET_ERROR then
    begin
        Msg := Concat('setsockopt failed! Error ',
                     IntToStr(WSAGetLastError));
        Synchronize(DisplayMsg);
        closesocket(sktBlast);
        Exit;
    end;

// Set IP TTL by using setsockopt.
sktRes := setsockopt(sktBlast, IPPROTO_IP,
                    IP_MULTICAST_TTL, pchar(@MCOptions.TTL),
                    SizeOf(MCOptions.TTL));
if sktRes = SOCKET_ERROR then
    begin
        Msg := Concat('setsockopt failed! Error ' +
                     IntToStr(WSAGetLastError));
        Synchronize(DisplayMsg);
        closesocket(sktBlast);
        Exit;
    end;

// Disable loopback.
LoopBackFlag := False;
sktRes := setsockopt(sktBlast, IPPROTO_IP,
                    IP_MULTICAST_LOOP, pchar(@LoopBackFlag),
                    SizeOf(LoopBackFlag));
if sktRes = SOCKET_ERROR then
    begin

```

```
Msg := Concat('Call to setsockopt failed! Error ',  
             IntToStr(WSAGetLastError));  
Synchronize(DisplayMsg);  
end;  
  
Resume;  
  
end;
```

End Listing Three





ALGORITHMS

Delphi 1, 2, 3, 4 / Linear Least Squares / Gaussian Elimination



By Rod Stephens

The Shape of Data

Object Pascal Implementations of Linear Least Squares

Many applications generate data values that should lie on a straight line, but, due to error or inaccuracy in the input data, they do not. Analyzing the data is easier if the program can find a line that fits the data well, even if the points don't lie on it exactly. The line can give the user an idea of what the data might look like under ideal circumstances.

This article shows how a program can use the method of linear least squares to find the line or polynomial curve that best fits a set of points. This method uses some calculus and a little linear algebra. Some of the equations that follow are a bit intimidating, but don't be dismayed; the equations are mostly multiplication and addition, so they're long, but simple.

Minimizing Error

Suppose a program needs to fit a line to a series of data points $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$. The general equation for a line is $y = m * x + b$ for some constants m and b . The method of *linear least squares* picks the values of m and b that makes the line lie as close to the points as possible. It minimizes the sum of the vertical distances squared between the line and the points.

For example, suppose (x_i, y_i) is one of the data points. The vertical distance from that point to the line is the difference between y_i and the y value of the line for the x value x_i . In other words, $y_i - (m * x_i + b)$. The square of this distance is $(y_i - (m * x_i + b))^2$.

The method of linear least squares minimizes the sum of all of these terms for each data point. You can think of this sum as the error between the line and the points. You can write the error using the following equation, where Σ means to sum the values for each i :

$$E = \Sigma (y_i - (m * x_i + b))^2$$

To pick the values of m and b that minimize this function, take the partial derivatives of E with respect to m and b , set them equal to zero, and solve the two resulting equations for m and b . This may seem confusing, but it's actually straightforward. The error equation is just a simple sum of poly-



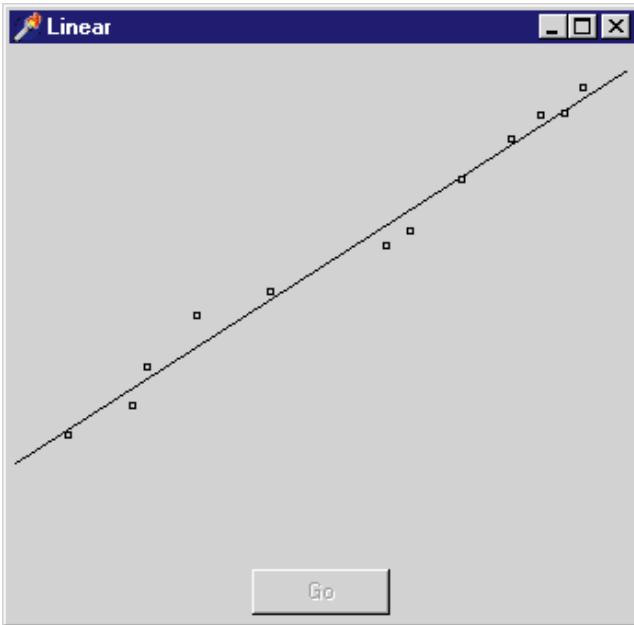


Figure 1: The example project Linear displaying a linear least squares fit.

nomials, so it's easy to differentiate. If your calculus is a little rusty, you can take my word for this step. The partial derivatives are:

$$\begin{aligned}\frac{\delta E}{\delta b} &= \sum 2 * (y_i - (m * x_i + b)) * (-1) \\ &= -2 * \sum (y_i - m * x_i - b) \\ &= -2 * (\sum y_i - m * \sum x_i - b * \sum 1) \\ \frac{\delta E}{\delta m} &= \sum 2 * (y_i - (m * x_i + b)) * (-x_i) \\ &= -2 * \sum (y_i * x_i - m * x_i^2 - b * x_i) \\ &= -2 * (\sum y_i * x_i - m * \sum x_i^2 - b * \sum x_i)\end{aligned}$$

These equations are a little easier to work with after making these substitutions:

$$\begin{aligned}S_1 &= \sum 1 \\ S_x &= \sum x_i \\ S_y &= \sum y_i \\ S_{xx} &= \sum x_i^2 \\ S_{xy} &= \sum x_i * y_i\end{aligned}$$

Then, the partial derivatives become:

$$\begin{aligned}\frac{\delta E}{\delta b} &= -2 * (S_y - m * S_x - b * S_1) \\ \frac{\delta E}{\delta m} &= -2 * (S_{xy} - m * S_{xx} - b * S_x)\end{aligned}$$

Setting the partial derivatives equal to zero and rearranging a little gives:

$$\begin{aligned}S_y &= m * S_x + b * S_1 \\ S_{xy} &= m * S_{xx} + b * S_x\end{aligned}$$

Solving these equations for the two unknowns, m and b , gives:

$$\begin{aligned}m &= (S_1 * S_{xy} - S_x * S_y) / (S_1 * S_{xx} - S_x^2) \\ b &= (S_{xx} * S_y - S_x * S_{xy}) / (S_1 * S_{xx} - S_x^2)\end{aligned}$$

```
procedure LeastSquares(PtX, PtY: array of Integer;
  NumPts, max_x, max_y: Integer;
  var x1, y1, x2, y2: Integer);
var
  S1, Sx, Sy, Sxx, Sxy, m, b : Single;
  I : Integer;
begin
  // Calculate the least squares sums.
  S1 := 0;
  Sx := 0;
  Sy := 0;
  Sxx := 0;
  Sxy := 0;
  for i := 1 to NumPts do begin
    S1 := S1 + 1;
    Sx := Sx + PtX[i];
    Sy := Sy + PtY[i];
    Sxx := Sxx + PtX[i] * PtX[i];
    Sxy := Sxy + PtX[i] * PtY[i];
  end;

  // Make sure the line isn't vertical.
  if ((S1 * Sxx - Sx * Sx) = 0) then
  begin
    x1 := PtX[1];
    x2 := x1;
    y1 := 0;
    y2 := max_y;
  end
  else
  begin
    // Calculate m and b.
    m := (S1 * Sxy - Sx * Sy) / (S1 * Sxx - Sx * Sx);
    b := (Sxx * Sy - Sx * Sxy) / (S1 * Sxx - Sx * Sx);
    // Calculate the line's end points.
    x1 := 0;
    y1 := Round(m * x1 + b);
    x2 := max_x;
    y2 := Round(m * x2 + b);
  end;
end;
```

Figure 2: Calculating linear least squares.

These equations are still rather intimidating, until you remember that the (x_i, y_i) are the data points and the program knows them. Then, values like $S_x = \sum x_i$ are just sums of known values, so they're easy to calculate.

For example, if the points' x values are stored in the PtX array, a Delphi program could calculate S_x using the following code fragment:

```
var
  PtX : array [1..NumPts] of Single;
  Sx : Single;
  i : Integer;
begin
  Sx := 0;
  for i := 1 to NumPts do
    Sx := Sx + PtX[i];
```

After it calculates the values S_1, S_x, S_y, S_{xy} , and S_{xx} , the program can compute the values of m and b that provide the closest fitting line.

The example project, Linear, shown in [Figure 1](#), uses the code in [Figure 2](#) to find linear least squares fits (the examples cited in this article are available for download; see end

ALGORITHMS

of article for details). The *LeastSquares* procedure takes as input parameters arrays containing the x and y coordinates of the data points. It also takes as parameters the number of points, and the maximum x and y values the program will need to display. It returns, through parameters $x1$, $y1$, $x2$, and $y2$, the coordinates of a line the program can draw to represent the linear least squares fit.

Higher Degrees

Sometimes data values may lie along a curve other than a line. For example, you might suspect the point should lie along the degree 2 polynomial:

$$y = a_0 + a_1 * x + a_2 * x^2$$

for some values a_0 , a_1 , and a_2 . You may even have reason to believe the points lie along a curve of a higher degree.

The method of linear least squares generalizes for higher degree polynomials in a straightforward manner. You still write an error equation, take partial derivatives with respect to the variables a_i , and solve for the a_i values. The equations are longer, but they're not any more complicated.

```
function PolynomialLeastSquares(PtX, PtY: array of Integer;
    NumPts, degree: Integer;
    var poly_coeff: array of Double) : Boolean;
var
    x_sum      : array [1..40] of Double;
    coeff      : array [0..20, -1..20] of Double;
    value, y0, y1 : Double;
    i, j, d     : Integer;
begin
    // Calculate the sums x^i.
    for i := 0 to 2 * degree do begin
        // Initialize this sum to zero.
        x_sum[i] := 0;
        // Add the contributions from each point.
        for j := 1 to NumPts do
            x_sum[i] := x_sum[i] + Power(PtX[j], i);
        end;

        // Calculate the coefficients for the equations.
        // coeff(i, j) is the ith equation's jth coefficient.
        // coeff(i, -1) is the constant term.
        for i := 0 to degree do begin
            // Calculate the constant term.
            value := 0;
            for j := 1 to NumPts do
                value := value + Power(PtX[j], i) * PtY[j];
            coeff[i, -1] := value;
            // Calculate the other coefficients.
            for j := 0 to degree do
                coeff[i, j] := -x_sum[i + j];
            end;

            // Solve the equations using Gaussian elimination.
            for i := 0 to degree do begin
                // Use the ith equation to eliminate the ith
                // coefficient from all but the ith equation.
                value := coeff[i, i];

                // If value is 0, no polynomial will work.
                if (value = 0) then
                    begin
```

The general equation for a degree D polynomial is:

$$y = a_0 + a_1 * x + a_2 * x^2 + \dots + a_D * x^D$$

Using this curve to approximate the points, the error function is given by:

$$E = \sum (y_i - (a_0 + a_1 * x_i + a_2 * x_i^2 + \dots + a_D * x_i^D))^2$$

The partial derivative of E with respect to the variable a_i is:

$$\begin{aligned} \delta E / \delta a_j &= \sum 2 * (y_i - (a_0 + a_1 * x_i + a_2 * x_i^2 + \dots + a_D * x_i^D)) * (-x_i^j) \\ &= -2 * \sum (y_i * x_i^j - a_0 * x_i^j + a_1 * x_i^{j+1} + a_2 * x_i^{j+2} + \dots + a_D * x_i^{j+D}) \end{aligned}$$

Again, this seemingly complicated equation is simpler than it appears. All the x_i and y_i terms are known data values. That makes this a simple linear equation involving the a_i values.

Calculating all the partial derivatives gives $D + 1$ linear equations with $D + 1$ unknowns a_0, a_1, \dots, a_D . Now, the program simply needs to solve the equations for the a_i values, and it's done.

Process of Elimination

To use the method of linear least squares, a program must

```
    // Return True to indicate failure.
    Result := True;
    Exit;
end;

// Normalize the ith equation so the ith
// coefficient is 1.
for j := -1 to degree do
    coeff[i, j] := coeff[i, j] / value;
coeff[i, i] := 1;

// Eliminate the ith coefficient from all
// other equations.
for j := 0 to degree do begin
    if (j <> i) then
        begin
            value := coeff[j, i];
            // If value = 0, it's already ok.
            if (value <> 0) then
                for d := -1 to degree do
                    coeff[j, d] := coeff[j, d] - coeff[i, d] * value;
                coeff[j, i] := 0;
            end;
        end;
end; // End processing all coefficients.

// Now the ith equation includes only two non-zero terms:
// the constant term coeff(i, -1) and the ith term
// coeff(i, i) which equals 1. Plugging these values in
// and solving for Ai gives Ai = -coeff(i, -1).

// Copy the results into the poly_coeff array for return.
for i := 0 to degree do begin
    if (coeff[i, i] > 0) then
        poly_coeff[i] := -coeff[i, -1]
    else
        poly_coeff[i] := 0;
    end;

    Result := False;
end;
```

Figure 3: Calculating polynomial least squares fits.

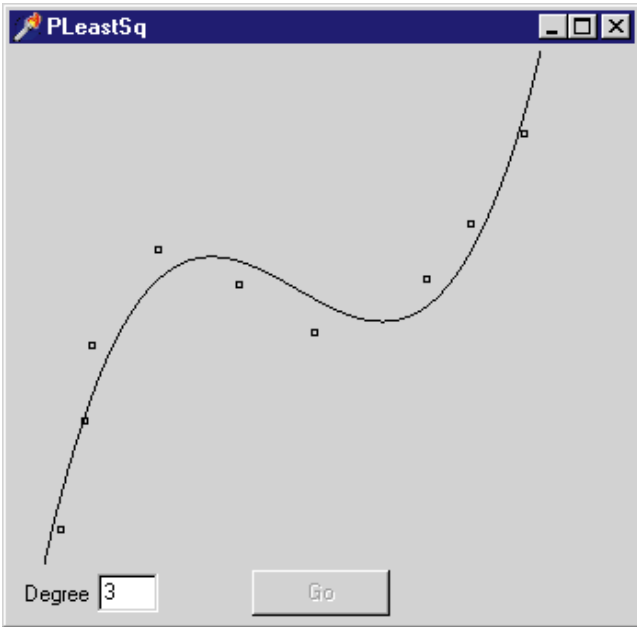


Figure 4: The example project, PLeastSq, displaying a degree 3 polynomial linear least squares fit.



Figure 5: A high-degree curve may fit better, but a low-degree curve may show the shape of the data more accurately.

solve two equations with two unknowns. This is fairly easy. Solving this new, larger system of equations is also easy, using *Gaussian elimination*.

The program should start by placing the equations' coefficients in a two-dimensional array. The first dimension represents the equation number, and the second represents a term in the equation. The Delphi code is a bit simpler if the equation's constant term is given the index -1. For example, suppose the partial derivative with respect to A_0 looks like this:

$$c_{-1} + c_0 * a_0 + c_1 * a_1 + \dots + c_D * a_D = 0$$

Then, the first few entries in the array would be:

$$\begin{aligned} \text{coeff}[0, -1] &= c_{-1} \\ \text{coeff}[0, 0] &= c_0 \\ \text{coeff}[0, 1] &= c_1 \end{aligned}$$

The program starts the Gaussian elimination by dividing all the entries in the first row by $\text{coeff}[0, 0]$. Dividing every element in the row by the same value is equivalent to dividing every term in the corresponding equation by the same value. Because both sides of the equation are divided by the same value, the equation remains true. Dividing each entry by $\text{coeff}[0, 0]$ also makes the $\text{coeff}[0, 0]$ entry equal to 1.

Next, the program subtracts a multiple of the coefficients in the first row from the other rows so they contain zero in their first positions. For instance, the first coefficient in the second row is $\text{coeff}[1, 0]$. Because $\text{coeff}[0, 0]$ is now 1, subtracting $\text{coeff}[1, 0]$ times $\text{coeff}[0, 0]$ from $\text{coeff}[1, 0]$ gives:

$$\text{coeff}[1, 0] - \text{coeff}[1, 0] * \text{coeff}[0, 0] = 0$$

The second row's other coefficients are calculated by subtracting the same multiple of the corresponding coefficient. For example, $\text{coeff}[1, 1]$ becomes:

$$\text{coeff}[1, 1] - \text{coeff}[1, 1] * \text{coeff}[0, 1]$$

The program repeats this process for each of the other rows, so row 0 (zero) is the only row with a non-zero entry in position 0.

Next, the program repeats the entire process for every other coefficient. It uses row 1 to eliminate the other rows' coefficients in position 1, row 2 to eliminate coefficients in position 2, and so forth. Eventually, the i th equation will contain a 1 in position i . The array's $\text{coeff}[i, -1]$ entry will be the only other non-zero entry in the row, so the array will look like this:

b_0	1	0	...	0
b_1	0	1	...	0
:	:	:	:	:
b_D	0	0	...	1

These values correspond to the system of equations:

$$\begin{aligned} b_0 - a_0 &= 0 \\ b_1 - a_1 &= 0 \\ &: \\ b_D - a_D &= 0 \end{aligned}$$

It's easy to solve these equations for the a_i values:

$$\begin{aligned} a_0 &= -b_0 \\ a_1 &= -b_1 \\ &: \\ a_D &= -b_D \end{aligned}$$

The example project PLeastSq uses the code shown in [Figure 3](#) to find polynomial least squares fits. The *PolynomialLeastSquares* function takes as input parameters arrays containing the x and y coordinates of the data points, the number of data points, and the degree of the polynomial it should create. It returns, through the *poly_coeff* array, the polynomial's coefficients, a_n .

[Figure 4](#) shows the project displaying a degree 3 polynomial fitting a set of data points.

Selecting the Best Curve

Polynomial curves have a couple of important properties you should keep in mind when using them. First, you can find parameters for a degree D polynomial that passes exactly through any $D + 1$ points, as long as no two points have the same x value. For example, you can find a degree 1 polynomial (line) that passes through any two points. There is no reason to use a higher degree polynomial than necessary (e.g. it would be silly to fit two data points with a degree 9 polynomial).

A high-degree curve may fit a set of points closely, but it may also hide important low-degree features in the data. [Figure 5](#) shows two curves fitting six data points. The degree 5 curve fits the data closely, passing exactly through each of the points. The degree 1 curve more correctly shows the overall linear shape of the data.

For this reason, you should use high-degree curves only when you have good reason to believe the data values were generated by a high-degree process. If you're unsure of the nature of the data, start with curves of low degree. Then, increase the degree of the curve until you get a reasonable fit.

Conclusion

The method of linear least squares is a powerful curve-fitting technique. It lets you discover the underlying shape of a set of data values while ignoring minor fluctuations, giving you insight into your data that might otherwise be difficult to find. [▲](#)

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM98\SEP\DI9809RS.

Rod Stephens is the author of several books, including *Custom Controls Library* [1998], *Visual Basic Algorithms* [1998], and *Visual Basic Graphics Programming* [1997], all from John Wiley & Sons. He also writes algorithm columns in *Visual Basic Developer* and *Microsoft Office & Visual Basic for Applications Developer*. He can be reached at RodStephens@vb-helper.com, or see what else he's up to at <http://www.vb-helper.com>.





By Craig Dunn

Monitor Your NT Apps

Creating and Using Performance Monitor Extension DLLs

It used to be sufficient to deliver software on a floppy disk in a plastic bag. Now, users demand slick packaging, logo-compliance with various standards, and easy configuration, installation, and management. This last element of managing and monitoring an application is often overlooked in the development process. By using Delphi to create simple extensions to the Windows built-in performance monitoring infrastructure, however, we can easily incorporate monitoring capability into any application.

For those unfamiliar with performance monitoring under Windows NT, Microsoft includes a utility with NT, aptly named Performance Monitor, which allows you to view various real-time statistics and counters pertaining to different pieces of the operating system. And these counters aren't limited to built-in operating system information; third-party drivers and applications can easily extend their own performance data to be intercepted and exposed by Performance Monitor.

This article will explore the entire process of how native performance monitoring works under Windows NT, creating a sample performance extension DLL in Delphi, and adding code to your application to expose performance information to the extension. [Figure 1](#) highlights the items we'll be creating, and the basic flow of data. In addition, we'll demonstrate a unique method of simplifying the registration and installation process of the performance extension DLL.

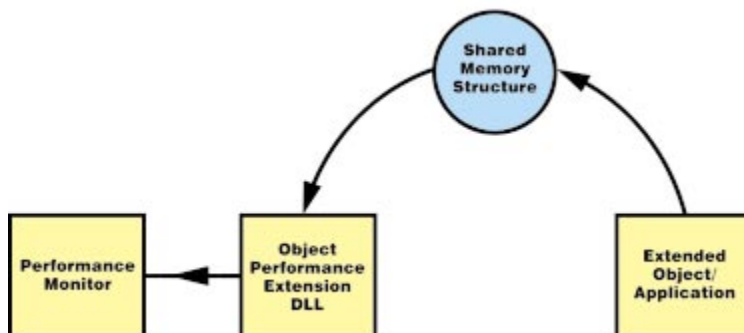


Figure 1: Performance data transport from 100,000 feet.

Monitoring under Windows NT

Performance data under Windows NT is gathered and retrieved using some special properties of the Windows NT registry. The Performance Monitor application accesses performance counters through a special hidden registry handle named HKEY_PERFORMANCE_DATA. Upon opening this handle using standard registry calls, all registered performance extensions are loaded into memory and initialized.

Performance counter requests flow as queries to this registry handle and return as buffers containing structured data, which the Performance Monitor interprets. When a query hits the HKEY_PERFORMANCE_DATA handler, it subsequently calls the collection procedure of the appropriate extension DLL. This collection procedure understands the specifics of how to retrieve the performance data from the application, and returns a structured response containing the requested counters. [Figure 2](#) depicts the typical flow of performance counter data.

In the example we'll be working with, the object or application we're extending is a simulated television. This allows us to demonstrate several different counter types, and provides a common object with familiar data elements. The code accompanying this article includes complete projects for both the television application and the television performance extension DLL (see end of article for download details).

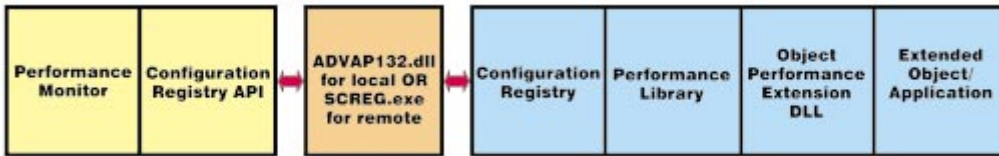


Figure 2: Performance Monitor collection data flow.

Support Files

The WPTelevision.dpr project includes several supporting files that require explanation. First, there is WinPerf.pas, which is essentially a translation of the original C header file from the SDK. SharedMemory.pas is a support unit that was developed for creating and accessing Windows shared memory structures from within Delphi. While this article is not a technical discussion of shared memory details, the topic is covered well in both the [November 1997 issue of Delphi Informant](#) and *Delphi Developer's Handbook*, by Cantù, Gooch, and Lam [SYBEX, 1998]. The TelevisionLIB.pas unit contains declarations of the actual shared memory structure for the television counters, as well as some constant definitions indicating relative offsets of the television counters. These offsets are used for calculating the locations of specific counters within the performance output structure.

The WPTelevision.ini and WPTelevision.h are complex enough to warrant more discussion. WPTelevision.h may strike some as odd since we're coding in Delphi and not C. The existence of a C header file in this project serves to facilitate the registration process. This header file merely contains C equivalents of the offset constants we defined in the TelevisionLIB unit:

```
// WPTelevision.h
#define Television           0
#define TelevisionPower     2
#define TelevisionChannel   4
#define TelevisionClosedCaptioningEnabled 6
#define TelevisionSleepTimer 8
#define TelevisionTimeOn    10
#define TelevisionClosedCaptioningWordsPerSecond 12
```

Note that the constants begin with zero and increment as consecutive even numbers. This is because there are really two elements being registered for each counter: the counter name and its associated help string.

The explanation of the WPTelevision.ini file could probably consume a short article on its own, but here's a brief description. The .INI file is used only during the installation process by the lodctr.exe facility to: 1) establish the registered name and help strings for each counter; and 2) point to a header file containing the relative offsets of each counter.

Figure 3 shows some of WPTelevision.ini from the included project. The [info] section has name/value pairs indicating the name of the performance application and the name of the header file from which to obtain offsets. The [languages] section dictates language constants supported. The [text] section occupies the bulk of the file, and is comprised

of two sets of name/value pairs for each object (i.e. Television) and each counter (i.e. Channel) to be monitored. The names are composed from the object name, counter name, and language constant (allowing support of multiple languages from within a performance extension). The values are the names and help descriptions that actually get displayed to a user. The order of the text name/value pairs is important, as it must synchronize with the order of the counter offsets. Again, this is a simplified description of this file, but Microsoft provides more detailed references on their Web site and MSDN.

The other included project, TVDemo.dpr, uses both the SharedMemory and the TelevisionLIB units. Its main purpose is to simulate the extended application and "drive" the counters so our extension DLL can read some useful data.

The Performance Extension DLL

The basic extension DLL consists of only three primary pieces of functionality: "Open," "Collect," and "Close," implemented by the *OpenData*, *CollectData*, and *CloseData* functions, respectively. In fact, although the task for each is very specific, the naming of these functions in the DLL isn't so rigid. During the installation process, pointers to the library and each of its functions are established in the registry.

The *OpenData* function simply initializes internal structures and establishes whatever communication method will be used between the extension DLL and the extended application. The *CollectData* function triggers the DLL to harvest counter data from the extended application and return it to the caller. The *CloseData* function shuts down communications with the extended application and frees any internal resources. Together, these three functions make up the core of enabling your applications to expose counter data to the outside world.

The OpenData Function

The *OpenData* function (see **Figure 4**) is used primarily to initiate our link to the counter data exposed by the extended application. Because the performance extension DLL can potentially be called multiple times (or instances), we must track the number of open instances so we only initialize the communications link once. **Figure 4** also shows the *OpenCount* globally-scoped variable being checked before entering the primary initialization logic.

```
[info]
drivername=Television
symbolfile=WPTelevision.h
[languages]
009=English
[text]
Television_009_NAME=Television
Television_009_HELP=Television object
...
```

Figure 3: A portion of the WPTelevision.ini file.

In our example, we've chosen to use shared memory mapping as our conduit for counter data from the extended application to the extension DLL. This is a common approach that pro-

```
function OpenData(lpDeviceNames: PWideChar):DWord; stdcall;
var
  Registry: TRegistry;
  keystr: string;
begin
  if (OpenCount = 0) then begin
    SharedMem := TSharedMem.Access(SharedMemName);
    if (SharedMem.Buffer <> nil) then begin
      ACCESS_SMEM := SharedMem.Buffer;
      ...code to get settings from registry...
      DLLInitOK := True;
      Registry.Free; end
    else begin
      Result := GetLastError;
      Exit;
    end;
  end;

  Inc(OpenCount);
  Result := ERROR_SUCCESS;
end;
```

Figure 4: The *OpenData* function.

```
function CollectData(lpwszValue: LPWSTR; lppData: PPointer;
  lpcbBytes: LPDWORD; lpcObjectTypes: LPDWORD):
  DWORD; stdcall;
var
  SpaceNeeded      : Cardinal;
  I                : Integer;
  pPerfCounterBlock : PPerf_Counter_Block;
  pOutputCounters  : POUTPUT_COUNTERS;
  pDataDefinition  : PChar;
begin
  // Before doing anything else, see if Open went OK.
  if (not DLLInitOK) then begin
    // Unable to continue because open failed.
    lpcbBytes^ := DWord(0);
    lpcObjectTypes^ := DWord(0);
    // Yes, this is considered a successful exit.
    Result := ERROR_SUCCESS;
    Exit;
  end;
  SpaceNeeded := SizeOf(TPERF_OBJECT_TYPE) +
    SizeOf(TPERF_COUNTER_BLOCK);
  for i := 0 to NumberOfCounters-1 do
    SpaceNeeded := SpaceNeeded +
      SizeOf(TPERF_COUNTER_DEFINITION) +
      GetSizeOfSpecificCounter(i);
  if (lpcbBytes^ < SpaceNeeded) then begin
    lpcbBytes^ := DWord(0);
    lpcObjectTypes^ := DWord(0);
    Result := ERROR_MORE_DATA;
    Exit;
  end;
  // Get the Response Buffer initialized.
  InitializeOutputStructure;
  // Set an internal pointer to the caller's data buffer.
  pDataDefinition := PChar(lppData^);
  // Copy the initialized Object Type and counter
  // definitions to the caller's data buffer.
  Move(pOutputStructure^, pDataDefinition^, SpaceNeeded);
  // Format and collect data from shared memory.
  ...
  // Update arguments before return.
  lppData^ := Pointer(DWORD(pOutputCounters) +
    SizeOf(TOutput_Counters));
  lpcObjectTypes^ := 1;
  lpcbBytes^ := SpaceNeeded;
  Result := ERROR_SUCCESS;
end;
```

Figure 5: The *CollectData* function.

vides fast interprocess communication with little overhead to either the application or the extension DLL.

A Delphi unit accompanies the sample project to facilitate shared memory usage. Shared memory is opened by calling the *Access* method of the *TSharedMem* object with a string indicating the name of the shared memory block. For this to return a non-*nil* handle, the shared memory must be previously established by the extended application. So, when the extended application isn't running, counter data will be available to the monitoring application, i.e. Performance Monitor.

When the shared memory communication has been established, we simply need to load some internal variables with data from the registry. Each step of the way, we're validating the existence of the data in the registry. If any step fails, the whole initialization fails. First, we open a key specific to the extension DLL that is being initialized. In our case, we're opening:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
  Television\Performance
```

to ascertain certain values set during registration of the DLL. These values, *First Counter* and *First Help*, point to the base indexes within the registry of the counter names and help strings. If the key and both of these values exist, we set *DLLInitOk* and close the registry. If we had a failure at any point, the specific error code would've been returned.

Finally, the *OpenCount* variable is incremented and we return a successful result code to the caller. Overall, the *OpenData* function is fairly simple to implement and is generic enough to be reused by many extension DLLs with little or no code changes.

The *CollectData* Function

The *CollectData* function is the heart of the extension DLL, and is therefore — as you can guess — the most complicated (see Figure 5). To simplify the code, several support functions have been broken out of the main *CollectData* function and will be covered as we come across them.

First, we must ensure the DLL initialized without a hitch. If *DLLInitOk* isn't True, we need to clear out the return parameters and drop out of the function immediately. This will commonly occur when the extended application isn't running, so the return result is still considered a success and not an error.

Second, we need to calculate some space requirements for the structure we're returning to the caller. This is calculated by adding the sizes of the object type and counter block structures, along with the counter definition structure size and actual data element size for each exposed performance counter. We've defined some support functions (*GetSizeOfSpecificCounter* and *GetSizeOfAllCounters*) to assist in these calculations. We already know the size of the buffer passed to the function from *lpcbBytes*. If the calculated size exceeds what the passed-in buffer (*lppData*) can handle, we clear the return parameters and immediately pass back a specific error code indicating

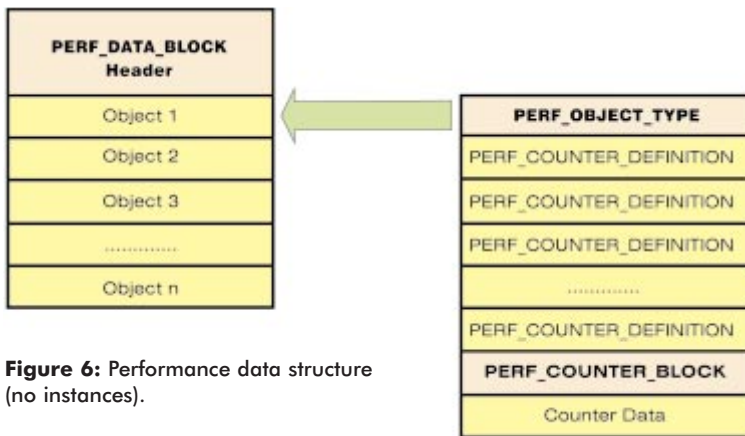


Figure 6: Performance data structure (no instances).

more space is required. Ultimately, the calling application could call *CollectData* repeatedly until a large enough buffer has been passed to the function.

The *InitializeOutputStructure* function (not shown) is called to allocate and initialize the response structure to be returned by *CollectData*. This function, although called with every call to *CollectData*, really only does this allocation one time. First it checks if the output structure is *nil*. If it isn't, then the initialization is done and we can return.

Now we need to calculate the amount of space needed by the output structure and allocate memory for it. This calculation is based on the sizes of the different structures upon which the performance data block is built, as well as the sizes of the individual counter types (see [Figure 6](#)). Once we allocate the right amount of memory, we populate it by simply de-referencing the pointer, setting the value, and then incrementing the pointer for each element in the structure. The rest of the accompanying *InitializeOutputStructure* code is similar and is sufficiently documented and self-explanatory.

Our data structure is in place and initialized. It's time to copy it into the caller's return buffer, *lppData*, so we don't overwrite our initialized data template. First, we typecast the address referenced by *lppData* into a PChar pointer. Then we can perform the copy using the Delphi *Move* procedure. Next, we get to the heart of the *CollectData* function. Here, we'll move a pointer called *pOutputCounters* to point to the location within the output structure that contains the counter data. Then, for each of the counters we'll de-reference the pointer and grab the value from the shared memory structure.

Last, we make sure our function arguments are properly set for return to the caller, and respond with an *ERROR_SUCCESS* result. Although the collect function is probably the most complicated of the three because of the sheer number of elements and structures that must be set up, most of the code is fairly static. The primary changes one would make are to the section where the data is harvested from shared memory and to the support functions that calculate the individual counter sizes.

The *CloseData* Function

The *CloseData* function simply reverses what was set in motion by the *OpenData* function. From the listing in [Figure 7](#), we see that *OpenCount* gets decremented immediately. If this was the last instance using the DLL, we enter our shutdown logic. This proceeds to free the shared memory conduit and de-allocate memory in use by the *pOutputStructure* variable. Again, this function is generic enough for easy reuse.

Self-registration

Registration of the performance DLL involves a number of steps, including numerous registry key and name/value additions, as well as shelling out to a

Microsoft-provided counter registration utility. This registration tells the system where the extension exists, what the base offsets are of the counters, and what the entry point functions are. While registering our new performance extension is far from trivial, with some clever tricks we can make it much simpler.

We've built the registration facility into the extension DLL by exporting two additional functions: *DLLRegisterServer* and *DLLUnregisterServer*. Many COM folks might recognize these known entry points as being used for the automatic registration of in-process COM objects using the *Regsvr32.exe* utility (or the built-in facilities of *InstallShield*, etc.). So, what are we accomplishing by this? We are simply leveraging existing tools and installation features to automate the registration/unregistration of our extension. In a sense, our extension DLL becomes "self-aware," and self-registering.

The *DLLRegisterServer* function sets up our extension for use. It first makes a call to our own *DLLUnregisterServer* function to clear out any registrations that may exist for this extension. (*DLLRegisterServer* and *DLLUnregisterServer* aren't shown. All source for this article is available for download; see end of article for details.) After determining the directory path to the extension DLL using the support function *GetDllInstallPath*, create a registry key under *HKEY_LOCAL_MACHINE* called:

```
SYSTEM\CurrentControlSet\Services\Television\Performance
```

Under this key, we want to add some values to point to the specific function names we used in our DLL for the "Open," "Collect," and "Close" processes. This is how we attain our flexible naming scheme mentioned earlier. The registry points

```
function CloseData: DWord; stdcall;
begin
  Dec(OpenCount);
  if (OpenCount = 0) then begin
    SharedMem.Free;
    if (pOutputStructure^ <> 0) then begin
      FreeMem(pOutputStructure, SizeOf(Integer));
      pOutputStructure := nil;
    end;
    SharedMem := nil;
  end;
  Result := ERROR_SUCCESS;
end;
```

Figure 7: The *CloseData* function.

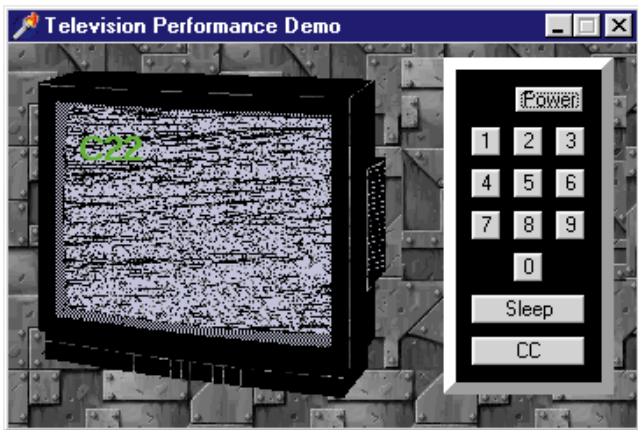


Figure 8: The example TVDemo application at run time.

```

procedure TTelevisionDemo.TVTimerTimer(Sender: TObject);
begin
  ...
  if TVPower then
    SharedMemory^.Power := 1
  else
    SharedMemory^.Power := 0;
  SharedMemory^.Channel := TVChannel;
  if TVCC then
    SharedMemory^.ClosedCaptioningEnabled := 1
  else
    SharedMemory^.ClosedCaptioningEnabled := 0;
  SharedMemory^.SleepTimer := TVSleepTimer;
  SharedMemory^.TimeOn := TVElapsedTime;
  SharedMemory^.ClosedCaptioningWordsPerSecond :=
    TVCCWords;
  ...
end;
    
```

Figure 9: A portion of the TVTimerTimer procedure.

to the function names for each of the known processes. Another value to add is Library, which points to the path and file name of the extension DLL. Now we invoke the *ShellExecute* function to call the *lodctr.exe* utility included with NT. This utility expects a parameter pointing to an INI file. Hence, we pass the location of the *WPTelevision.ini* we examined earlier. This utility inserts the counter names and help strings into the registry and creates new values for First Counter, Last Counter, First Help, and Last Help offsets under our Performance key mentioned earlier. Finally, *DLLRegisterServer* closes the registry and returns a successful result. (*DLLUnregisterServer* merely attempts to undo what we did with the registration function.) Initially, we will make a call to another NT utility called *unlodctr.exe* to remove our counter names and help strings from the registry. This Unload Counter tool accepts an Application Name (recall an entry in the .INI file for such) as a parameter. Once we return from executing the utility, we open the registry and delete the:

```
SYSTEM\CurrentControlSet\Services\Television\Performance
```

key, and its parent key:

```
SYSTEM\CurrentControlSet\Services\Television
```

Finally, we free our registry resource and return success. We've simplified a once-complicated process by creating

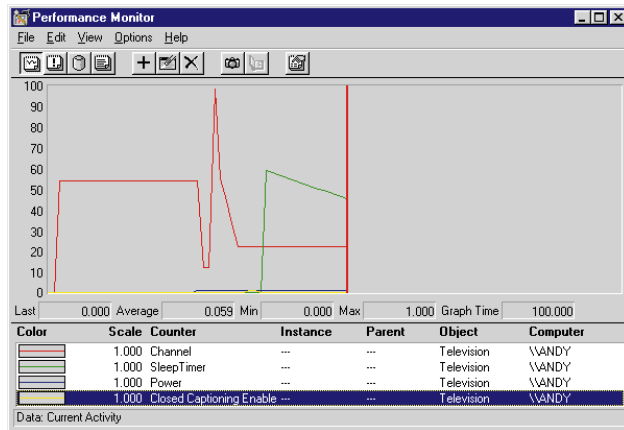


Figure 10: Monitoring TVDemo with Windows NT Performance Monitor.

generic code and taking advantage of known DLL entry points and common tools.

Putting It All Together

Let's see our extension DLL in action. Unzip the files for both projects into a single directory, and build both to obtain TVDemo.EXE and WPTelevision.DLL. From the command prompt and within the directory containing the DLL, run *regsvr32 WPTelevision* to register the extension. Run the TVDemo application to simulate our television "object" (see Figure 8). During the startup, we're creating a shared memory mapping named *Television_SMEM*. Our extension will be looking for this structure during its "Open" function. A Timer component within TVDemo handles the updating of the shared memory data once per second. Updating shared memory data is as simple as setting elements in a record pointer (see Figure 9).

Once the "television" is up and running, we can use the NT Performance Monitor utility to view the counter information. There should be a shortcut to it on the menu under *Start | Programs | Administrative Tools*, or you can find it in *\\Winnt\System32* as *Perfmon.exe*. Select *Edit | Add To Chart* to select some counters to monitor. From the *Object* dropdown list, find the *Television* object. If you don't see *Television* in the list, then TVDemo probably isn't running. Once you select the object, you will see the *Counter* list update just below it. Select each of the television counters and click the *Add* button. Now, click *Done* to close the dialog box and watch the monitoring. At this point, you can play with the television simulator and watch the results on Performance Monitor (see Figure 10).

Conclusion

We now have the ability to include native NT performance monitoring in our Delphi applications. The code has been reworked to the point of being generic and usable as a foundation for your own projects (with minimal tweaking). By adding the capability to monitor your applications using standard Windows tools, you enhance overall manageability and value and add a truly professional touch to your applications. Δ

IN DEVELOPMENT

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\SEP\DI9809CD.

Craig Dunn is a senior software engineer with SMART Technologies, Inc., the global leader in the Enterprise Relationship Management (ERM) marketplace. He has over 17 years of coding experience on a vast assortment of hardware/language platforms in various environments. He has been using Delphi since its inception and has done extensive development with the tool, preferring to use it over other languages within his skill-set, such as C++ and Visual Basic. His current projects include R&D efforts into creating Delphi implementations of emerging Microsoft technologies and heavyweight COM/DCOM object development. He can be reached via e-mail at cdunn@smartdna.com, by voice at (512) 719-9180, or by fax at (512) 719-9176.





By Alan C. Moore, Ph.D.

Async Professional 2.5

A Great Communications Library Gets Better

Async Professional (APRO) from TurboPower is considered by many developers to be the pre-eminent Delphi communications library. Participants in this year's *Delphi Informant* Readers Choice Awards certainly concurred, selecting APRO first in the Best Connectivity Tool category for the second straight year (see the **April, 1998** issue for details).

I reviewed the earlier version (2.0) in the **June, 1997** issue of *Delphi Informant*. In the sidebar, "Async Professional 2.0: Building on a Solid Foundation" (on page 46), I summarize the main features I described in the earlier review. In this review, I'll concentrate on new and improved features. APRO is fully compatible with Delphi 1, 2, and 3, and C++Builder 1 and 3 (with separate Help files and examples for Delphi and C++Builder).

In the previous review, I pointed out that communications capabilities are becoming increasingly important in the world of computers: A fax modem (or a network connection) is now considered standard equipment for a new computer. Bundled with that hardware, you generally find appropriate communications software. I've noticed that, with the

computers I've either purchased or used in the past two years, the communications software is getting considerably more sophisticated. It now includes full-featured telephony capabilities. Mindful of this trend, and responsive to its customers' requests, TurboPower has released a major upgrade to its flagship product so developers using the library can remain on the cutting edge. Let's examine some of the new features and see how some of the older features have been enhanced.

As in previous versions, *TApdComPort* continues to be one of the most important and basic components in the library. It controls the serial port hardware and input/output operations that take place through it. Its 50-plus properties, 11 events, and nearly 100 methods give programmers a very high level of control. This basic component also includes a plethora of exceptions and useful monitoring, error-checking, and debugging features.

Figure 1 shows the new version of the massive TCOM demonstration program, and gives some indication of the amount of control you have over a COM port at run time.

What's New

In version 2.5, the new *TApdWinsockPort* component joins *TApdComPort*. Derived from the older component, and sharing many of its properties and features, it provides a WinSocket port through which you can open a TCP/IP connection, just as you open a standard serial port connection with the *TApdComPort* component.

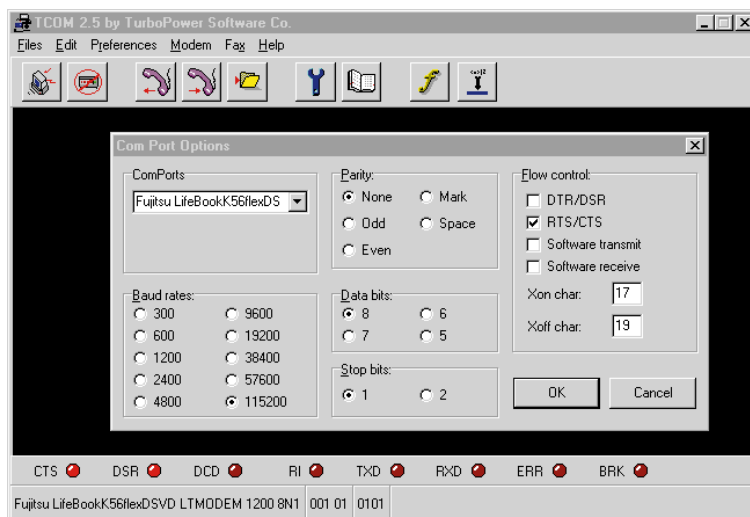


Figure 1: A dialog box in the TCOM demonstration program gives you control of the COM port at run time.

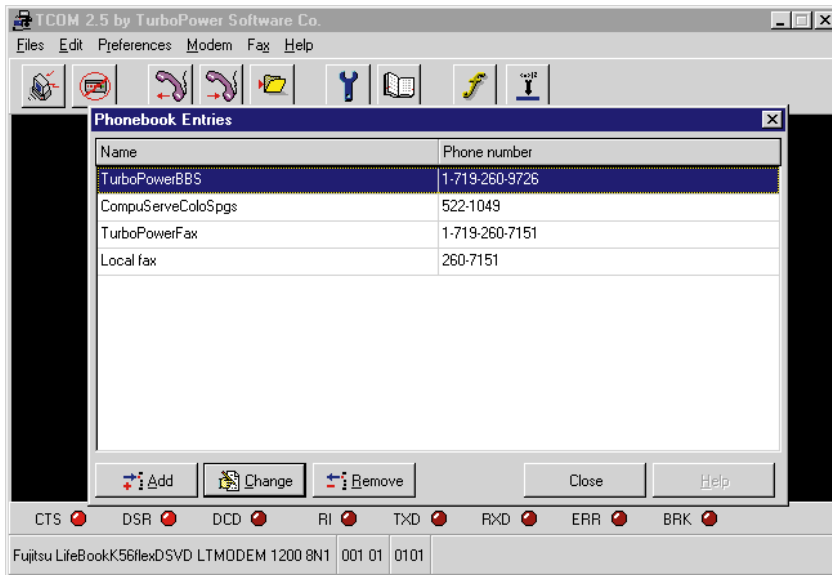


Figure 2: This TCOM dialog box manages a built-in phone number database system.

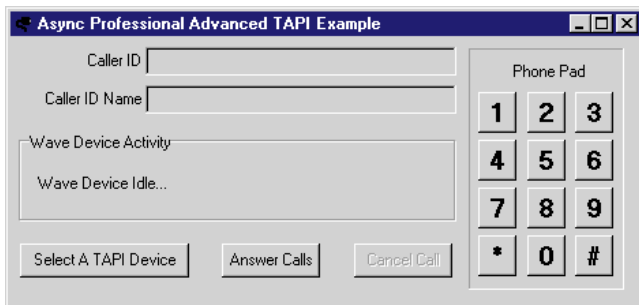


Figure 3: An example TAPI program that uses DTMF to accept and monitor incoming voice calls.

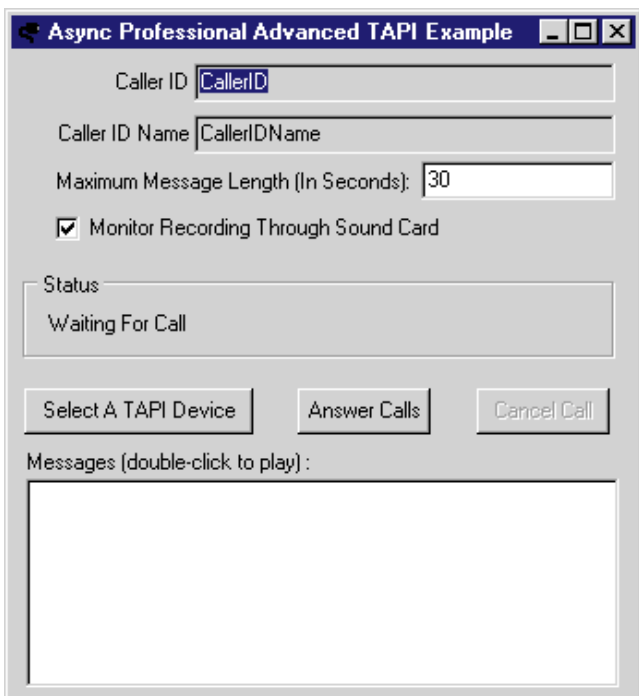


Figure 4: An example TAPI program that uses TAPI .WAV file support to record incoming calls.

There are several important new features in Async Professional 2.5, including support for DTMF (Dual Tone Multiple Frequency) touch tones, playback, and recording .WAV files under TAPI, communications over AT-compatible ISDN lines or through Windows Sockets, packets to manage incoming data, and more. While most of these new features expand considerably the sophistication of applications that can be written with APRO 2.5, others, such as data packets, ease the developer's burden. We'll begin by examining some of the new possibilities.

Communications systems are rapidly changing and expanding. Switchboards with live operators are quickly becoming an anachronism, being replaced with automated telephone systems that prompt the caller to enter one or several digits. With the DTMF and .WAV file support included in this major upgrade, you can now develop applications

that respond to keypad entries and record and playback voice prompts or messages. This gives you the ability to produce a variety of voice mail applications or an entire call management system. APRO's built-in phone number database system (see Figure 2) is helpful in building such applications. Figure 3 shows an example TAPI program that puts the new DTMF and .WAV file support into action. It allows you to accept and monitor incoming voice calls. Note that the buttons on the phone pad are not there to be clicked. They respond to incoming characters being depressed. Another of the several TAPI programs shows a similar user interface, but with other capabilities, including recording of incoming calls (see Figure 4).

APRO's already impressive fax-handling capabilities have been expanded in version 2.5. Now, applications based on APRO can recognize and switch between fax and voice calls. This version also includes a new voice-to-fax handoff feature, which supports building fax-back systems where users can select the document they wish to have faxed back to them. Figure 5 shows another example program that demonstrates these capabilities.

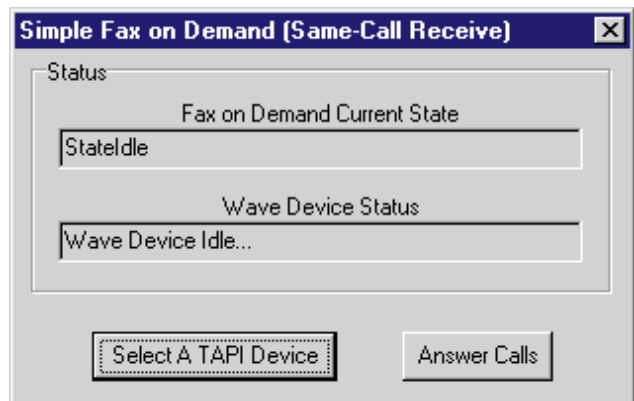


Figure 5: With APRO's new voice-to-fax handoff feature, you can build fax-back systems where users can select the document they wish to have faxed back to them.

Figure 6: The first notebook page of my example program collects minimal registration information ...

Figure 7: ... the second notebook page of my example program determines how to send that information.

Applications using APRO's Fax-Printer Drivers send various types of data directly to a fax modem. These drivers can be manipulated programmatically, making it possible to convert documents with extensive formatting into a faxable form. The new capability to send multiple faxes is impressive and useful.

As mentioned, some of the new features are designed to make a developer's work easier. Evidently, TurboPower received requests to provide additional support for managing data in the incoming stream. Beyond triggers, data packets allow you to indicate specific data you're expecting, and be notified when it arrives. If this particular feature is a requirement of your communication system, data packets could be a great time saver. The new component, *TApdDataPacket*, automatically collects and delivers data from the incoming stream based on properties you can set using the component's property editor. It also includes its own buffering system, so you need not worry about losing incoming data. You can control data packets at design time or run time.

Documentation and Support

One of the major enhancements in version 2.5 is its documentation. As with previous versions, this version includes full source code, many example programs, and an excellent manual describ-

ing each component, its uses, properties, methods, and events. With this version, TurboPower has added a *Developer's Guide*, which presents the components of the library in a systematic way. Based on the excellent demonstration and example programs that have been a hallmark of TurboPower component libraries, this guide provides even the novice communications developer an excellent introduction to this field of programming in general, and this library in particular. While some of the material in the demonstration programs was included in a section of the earlier manual, expanding it — to include the more focused example programs — and putting it in its own volume makes it much more useful. As a big fan of printed documentation, I say, "Bravo, TurboPower!"

INFORMANT
FACT FILE

Async Professional 2.5 is a superb library of Delphi components for handling all major aspects of asynchronous communications, including configuring/using modems, terminal/keyboards emulation, file transfer protocols, and sending/receiving faxes. Version 2.5 adds important new features, including DTMF (Dual Tone Multiple Frequency) touch tones, TAPI .WAV file operations, use of AT-compatible ISDN lines or Windows Sockets, and more. With the addition of an impressive new *Developer's Guide* to its comprehensive manual, a plethora of example programs, full source code, and extensive online Help, it's a model of excellence for programming documentation. More than ever, it's perfect for experienced and novice communications programmers.

TurboPower Software Company
P.O. Box 49009
Colorado Springs, CO 80949-9009

Phone: (800) 333-4160 or (719) 260-9136
Fax: (719) 260-7151
E-Mail: info@turbopower.com
Web Site: <http://www.turbopower.com>
Price: US\$279; upgrades vary from US\$79 to US\$159.

In the area of customer support, TurboPower is also at the head of the pack. It now has online newsgroups for nearly all of its products. I've noticed that participants generally get their questions answered within 24 hours. While TurboPower may not be able to fix every reported bug as quickly as users would like (I know of one or two tough cases), they generally make a good-faith effort, and are able to correct problems as quickly as possible. The frequency with which TurboPower customers recommend these libraries on Internet discussion groups is a testament not only to their power and flexibility, but also to the excellent documentation and support.

A New Demonstration Program

For this review, I wanted to present a new demonstration program that might appeal to developers who may not consider communications programming applicable to them. My model was from the online registration routines I've seen with some of the newer applications I've purchased. I thought that would be the perfect approach.

This simple application consists of a two-page wizard, the first page of which collects minimal registration information (see [Figure 6](#)). The second page, which sends that information back to the vendor (see [Figure 7](#)), appears only after the user has entered his or her name and the product serial number. The user is prompted for, but not required to enter, an e-mail address.

Four APRO components are used: *TApdComPort*, *TApdWinsockPort*, *TApdProtocol*, and *TApdTapiDevice*. Remarkably, little code is needed to send the file. In this first case, I used a modem operated by TAPI to send the file to the TurboPower BBS with the following statements:

NEW & USED

```
ApdTapiDevice1.Dial('1-719-260-9726');  
...  
ApdProtocol1.FileMask := FName;  
ApdProtocol1.StartTransmit;
```

In the second case, I used the WinSock Port component to send the file over an open Internet line:


```
if not ApdWinsockPort1.Open then  
begin  
  ApdWinsockPort1.WsAddress := 'bbs.turbopower.com';  
  ApdWinsockPort1.WsPort := 'telnet';  
  ...  
  ApdProtocol1.FileMask := FName;  
  ApdProtocol1.StartTransmit;
```

The entire project can be downloaded from the Informant Web site (see end of article for details).

Conclusion

Needless to say, all the powerful and developer-friendly features I wrote about in my earlier review continue in this version. Each of the native components include full source code and thorough documentation. Many of the example programs include dialog boxes that can be easily modified to meet many situations. Using APRO's custom classes (where the properties are unpublished), you can create your own new communications components to meet special circumstances.

To maintain our competitive edge as programmers, we must be able to quickly produce robust, feature-rich applications. In the field of communications, Async Professional meets those needs superbly. Particularly with the expanded documentation, experienced and novice programmers alike will be able to quickly get up to speed in this increasingly important field of Windows programming.

With version 2.0, I had very few reservations: the lack of WinSock support and insufficient explanations on some of the example programs. Now, with the new WinSock component and the *Developer's Guide*, those reservations have been eliminated completely, and I couldn't be more pleased. If you're going to be working with Windows communications, and don't want to spend months writing basic routines and components, you should definitely take a look at Async Professional. I think you'll come to the same conclusion I did: This is one superb Delphi library! 

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\SEP\ADI9809NU.

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 and 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.



Async Professional 2.0: Building on a Solid Foundation

Based on its venerable DOS predecessor, Async Professional for Delphi 2 (APRO) provided a complete solution for most communication tasks. This library of native Delphi components and low-level functions included tools for managing the COM port, configuring/using modems, enabling terminal/keyboard emulation, working with a variety of file transfer protocols, and sending/receiving faxes.

With APRO 2.0, there were two ways of working with modems: the *TApdModem* component (and its related dialer components) for the Windows 3.x or Windows NT 3.51 environments and the several TAPI components for the Windows 95 or Windows NT 4.0 environments. It provided several components for setting up and using modems:

- A modem database component, *TApdModemDBase*
- A general purpose modem component, *TApdModem*
- A phone dialer component, *TApdModemDialer*

TApdModemDBase provided an interface to the library's modem database, *AWModem.INI*, enabling developers to manipulate a modem database. The *TApdModemDBase* component provided basic information about a modem, while *TApdModem* provided all the properties, events, methods, and exceptions needed to operate it. All these components were closely integrated.

The Telephony Application Programming Interface (TAPI) that comes with Windows 95 provides a standard interface for working with communications (see the series of articles I wrote with Ken Kyler beginning in the [July, 1998 issue of Delphi Informant](#)).

APRO 2.0 provided several useful TAPI components:

- *TApdTapiDevice* for basic TAPI functions of dialing, answering, and configuring a modem
- *TApdTapiStatus* to display status information on TAPI's activities
- *TApdTapiLog* to log TAPI events

APRO 2.0 also provided excellent terminal support, including several terminal emulation components. The main one, *TApdTerminal*, could be used with or without emulation for a variety of purposes, including retrieving characters from the serial port, sending keystrokes through the serial port, translating escape codes into the colors and formatting they represent, or storing incoming data in a buffer.

The non-visual keyboard emulation component, *TApdKeyboardEmulator*, worked with the *TApdTerminal* component. Finally, the *TApdBPTerminal* component allowed users to view data from CompuServe B+ file transfers. Developers could also derive new terminal or keyboard emulation classes from the base classes, *TApdCustomTerminal* and *TApdKeyboardEmulator*.

One of the oldest elements of serial communication with modems are file transfer protocols. Async Professional for Delphi supported all the major protocols, including ASCII, B+, Kermit, Zmodem, and various forms of Xmodem and

Ymodem with the main component, *TApdProtocol*. This component included some properties and methods that applied to all the protocols, such as *ComPort* and *CancelProtocol*, and protocol-specific properties and methods, such as *KermitRepeatPrefix* and *ZmodemRecover*.

APRO 2.0 provided strong error-checking and logging features. General error-handling properties, events, and methods included *AbortNoCarrier*, *BlockErrors*, *ProtocolError*, *OnProtocolError*, and *WriteFailAction*. Protocol-specific ones were also included. The associated class, *TApdProtocolLog*, provided support for automatically logging a file transfer. The *TApdProtocol* property, *ProtocolLog*, provided the means of creating an instance of *TApdProtocolLog* to keep track of the main component's file transfer activities. The same kind of support was found in several other *TApdProtocol* events and properties, including *OnProtocolLog*, *BytesRemaining*, and *BytesTransferred*.

Fax support has become one of the essential elements of modern asynchronous communications; APRO 2.0 provided extensive support for sending, receiving, and converting faxes. Its *TApdFaxConverter* component enabled users to convert common file formats, such as ASCII, .BMP, .PCX, .DCX, and .TIF, to a compressed (.APF) form to be faxed. A companion component, *TApdFaxUnpacker*, reversed the process, converting received .APF files back to the above-mentioned file formats. The main fax components, *TApdSendFax* and *TApdReceiveFax*, allowed users to send and receive facsimile documents on Class 1, Class 2, and Class 2.0 fax modems; additionally, any of these fax modems could communicate with any other Group 3 fax device. APRO 2.0 also provided a fax printing component, a fax viewing component, and a fax status component. The printer component, *TApdFaxPrinter*, allowed users to send a fax to a Windows printer, or add headers and footers and scale the fax to a specified paper size. The *TApdFaxViewer* component allowed users to view a received fax or .APF file; it included support for scaling, white space compression, drag-and-drop support, and the ability to copy all, or part, of a fax to the Windows Clipboard. The *TApdFaxStatus* component enabled developers to show the progress of a fax transmission. The library also included a small class, *TApdFaxPrinterStatus*, that implemented a standard printer status display and a Fax Printer Driver.

The earlier library included two general groups of non-communications low-level utilities potentially useful in other applications: The first group was made up of timer functions; the second group included three functions for numeric/string conversions. A final, low-level utility, *TApdIniDBase*, provided support for maintaining a Windows .INI file, useful in many communications and non-communications applications.

— Alan C. Moore Ph.D.



NEW & USED

By Bill Todd

ImageLib Corporate Suite 3.05

A Powerful, Mature Image-manipulation Tool

First, ImageLib from Skyline Tools won the **1997 Delphi Informant Readers Choice Award** for Best Imaging Component. Now, version 3.05 brings a host of new features to this already excellent product. If you need to create an imaging application, ImageLib has everything you're likely to need.

In addition to Delphi and C++Builder, ImageLib is also compatible with Borland C++, Microsoft Visual C++, and Microsoft Visual Basic. A 450-page manual provides a complete reference to the VCL components, as well as the underlying DLL functions. You can distribute applications created with ImageLib royalty-free, which makes it an excellent tool for creating commercial, as well as custom, applications. Also, the manual clearly outlines which of the DLLs you need to distribute to support the features you're using. One significant shortcoming of many Delphi add-ins is that you can't internationalize them. Not so with ImageLib. All the strings in each DLL are stored in the DLL's resource for easy translation.

The Components

Installing ImageLib adds 20 components to your Delphi Component palette. ImageLib provides components to display and manipulate static images, as well as multimedia image files, in 23 different formats. There's also a great collection of components that will save you time developing your applications. The core components are *TPMultiImage* and *TPMultiMedia*. *TPMultiImage* displays images in any of the static image formats that ImageLib supports. It also allows you to save an image in any compatible format. *TPMultiMedia* supports all common multimedia file formats, including .AVI, .MOV, .WAV, and .MID. Each of these components has a data-aware twin for

working with images in any format stored in BLOB fields in a database table.

There are also *TIIDocumentImage* and *TIIDBDocumentImage* components, with special features for document management applications, and *TIIWebImage* for working with images for the Web. *TIIWebImage* adds options for progressive display, and the ability to start and stop GIF animation displays, or display every frame in the GIF animation. The *OnProgress* event lets you provide feedback to users when loading a large file.

Within a few minutes, starting from the example in the manual, I was able to:

- create an application that could load an image from disk in any format supported by ImageLib;
- pan the image;
- zoom in on any area of it using a "rubber band;"
- display the effects editor and apply any of 40 image manipulation effects;
- rotate or invert it;
- cut or copy it to the Clipboard;
- paste a new image into my viewer from the Clipboard; and
- scan an image from a scanner.

Now *that's* rapid application development!

Effects Editor

The effects editor, shown in **Figure 1**, is built into the image display components.

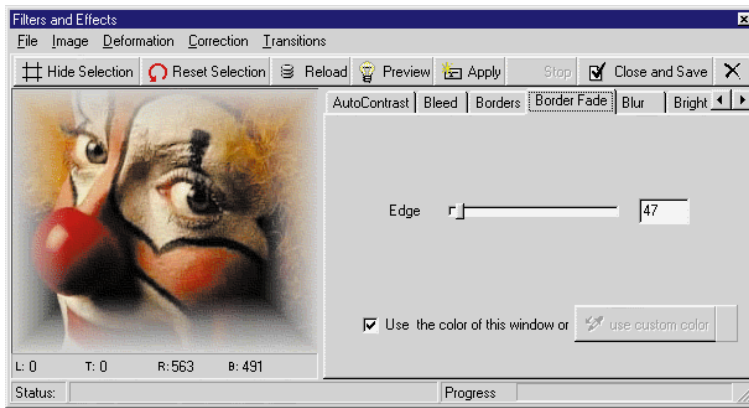


Figure 1: The ImageLib effects editor.



Figure 2: SkylineDocImage showing a multi-page document with annotations and highlighting added.

A single method call displays it, and lets you apply any of 40 special effects. The **Preview** button lets you see the selected effect. If you like what you see, click the **Apply** button to add the effect to the image. The effects from which to choose include contrast, borders, blur, brightness, border fade, number of colors, dithering method, color palette, gamma, hue and saturation, mosaic, oil paint, wave, invert, rotate, transitions, and many others.

The ImageToolbar Component

The toolbar components are also very helpful for image manipulation. For example, **MImageToolbar** contains buttons that let you:

- scan an image
- select a scanner
- load an image from a file
- preview an image
- save the image to disk
- print the image
- cut or copy the image to the Clipboard
- paste a new image from the Clipboard
- zoom in or out
- rotate the image
- flip the image
- reset the zoom to normal

- stretch the image to fill the frame with or without maintaining its aspect ratio
- add a scrolling or credits message
- display the effects editor

All you have to do is drop the **ImageToolbar** component on your form, set a single property to attach the toolbar to the image display component, and display or hide the floating toolbar by setting its *ShowToolbar* property.

Loading and Saving

The load and save dialog components are another great part of the ImageLib suite because they let you preview the image in a file before you open it. To

preview the contents of a file, double-click the file name. This gives your users the best of both worlds. They can scroll rapidly through a long list of files and still instantly see the contents of any file without opening it. Being able to see the contents of a file in the File Save dialog box is a nice feature because you can easily verify the contents of a file before you overwrite it.

The *TDBIconListBox* and *TDBIconComboBox* components let you include icons stored in **BLOB** fields in a table in a list box or combo box. The *TDBIconEditor* even lets users edit the icons in the table.

Scanning

ImageLib provides support for all TWAIN and ISIS scanners, including high-speed and duplex scanners. You also get support for automatic document feeders, so ImageLib gives you everything you need to build industrial strength document scanning and management systems. ImageLib supports the multi-page TIFF format so you can scan a multi-page document at high speed into a single file. It also uses anti-aliasing for the best possible image quality as you zoom in and out.

Another great feature for document management applications is the ability to de-skew a scanned image. If someone sets a document on the scanner crooked, a call to the *TIIDocumentImage* component's *Deskew* method will straighten it out. ImageLib also includes a video frame grabber so you can capture images from video. The Clipboard is fully supported for cutting, copying, and pasting images to or from the ImageLib components.

OCR Capabilities

If you need more than just document management, ImageLib includes an OCR component, which connects to Xerox's TextBridge OCR software. With TextBridge, you scan documents and store either the image, the converted text, or both. To showcase its document management capabilities, ImageLib comes with a complete document scanning and manipulation program, called SkylineDocImage, written in Delphi.

Figure 2 shows the SkylineDocImage application's main form with an annotation and some highlighting added to the first page of a multi-page document. You can distribute this pro-

gram as it stands, royalty-free. For an extra fee, you can get the source code. If you need to do document management, the source code is well worth the cost because you will probably have half of your application already written for you.

Conclusion

One problem with adding imaging capabilities to an application is that users frequently want more features added. With ImageLib, your users will be hard pressed to define a feature that you can't provide quickly and easily.

ImageLib's document management support makes it stand out from the crowd. With the low cost of hard disk space, zip and Jazz drives, and CD recorders, the market for applications that can archive and manipulate documents is poised for growth. With ImageLib, you can easily meet the document management needs of your clients. ImageLib is a very powerful, complete, mature, and well-documented product. Δ

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books and is a member of Team Borland, providing technical support on the Borland Internet newsgroups. He is a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a nationally-known trainer, and has taught Paradox and Delphi programming classes across the country and overseas. He was an instructor on the 1995, 1996, and 1997 Borland/Softbite Delphi World Tours, and is a Contributing Editor for *Delphi Informant*. He can be reached at Bill.Todd@compuserve.com or (602) 802-0178.

INFORMANT
FACT FILE

ImageLib is a powerful, complete, mature, and well-documented image manipulation tool that supports all common static and multimedia file formats. With a complete set of imaging functionality including panning, zooming, an effects editor with 40 image effects, rotating, inverting, Clipboard support, and full scanning and faxing support, your users will be hard pressed to request a feature you cannot provide quickly and easily.

Skyline Tools
A division of Creative Development, Ltd.
20537 Dumont St., Suite A
Woodland Hills, CA 91364

Phone: (818) 346-4200
Fax: (818) 888-5314
E-Mail: sales@imagelib.com
Web Site: <http://www.imagelib.com>
Price: ImageLib Corporate Suite, US\$599;
DocImage Source Code, US\$399;
ImageLib Combo (includes @theEdge),
US\$199



Reuse, Recycle

Code reuse is a subject near and dear to many a programmer's heart. It seems the subject means even more to managers, because increased reuse should mean less development time and thus, a lower overall cost to produce the software. Why then, is it so difficult to find successful stories of code reuse? To answer this question, let's look at a typical software company.

How many string utility functions do you think exist at your company? Put another way, how many string utility "wheels" have been reinvented at your company alone? How many have been reinvented throughout the world? Do you think every one of those are identical? You might even argue that one particular implementation is the fastest, while another is more flexible, and yet another addresses needs specific to your company. Why shouldn't there be one set of reusable code to address these issues?

In the Delphi market, VCL components contribute extensively to code reuse. Look at any third-party Delphi Web page for confirmation. ActiveX controls, COM and CORBA objects, JavaBeans, and Design Patterns all contribute to the potential success of code reuse as well. However, some purists may argue that this isn't code reuse because you're not *really* reusing source code, but rather, reusing compiled code. Fine, let them debate that semantic point. Meanwhile, I'll be developing systems using methods, techniques, and libraries that have been written and debugged over time. Perhaps I'll coin a new buzz-word by using the phrase "component reuse" to more accurately portray what all of us Delphi developers already know. Component reuse works, and it shows in our schedules.

The whole purpose of reusing code is to allow you to not worry about the implementation details of how to write the code from scratch. Therefore, by definition, there is some higher level of abstraction shielding you from these implementation details. This is what affords increased productivity when employing component reuse strategies.

It lets you focus on the big picture of getting one component (i.e. library) talking to another, as opposed to fretting over the details associated with creating this component. Why then, would anyone care whether we are actually linking an OBJ file into our EXE, or that we create a DLL or OCX to encapsulate our business logic? As long as the component can be reused, it's a success.

Finally, let's address what it will take to make component reuse a reality in a programming shop. There are two groups that are affected: programmers and management.

Programmers need to change their attitudes, behaviors, and years of habits to focus on how they can help the company through component reuse. A programmer:

- must first ask this question before coding a new library: "Has this already been written, and, if so, can I integrate the existing library more easily than writing it from scratch?"
- can no longer think: "I'll just code it this way and clean up the code after the release."
- must constantly think in global terms. After every set of functions, a programmer should think: "Can someone else use this library, and, if so, what can I do to make it easier to reuse?"
- should no longer look at their code as *their* code. It's "public domain" to the company.

Managers typically want to see progress yesterday. Instant gratification is fine, but just as in life, seldom does that translate into long-term happiness. Managers:

- must accept design as necessary to software development. This is true for all projects, but it is even more important if you want to try and leverage your code through component reuse.
- should plan for reuse. This can be as simple as encouraging developers to take on the mind-set of component reuse.
- should establish in-house peer review of code to help identify candidates for reuse.
- should have a "toolsmith" to ensure the consistency of the code being checked into the common library.

Software development companies should do everything they can to promote component reuse. Companies who take full advantage of component reuse could find themselves with a team of programmers whose job is to make the toolset easy enough for any new programmer to be productive immediately. Another benefit is that the group of component developers can bridge many departments, helping each become more effective with their software development efforts.

Employing component reuse strategies at your company is a major change, but the efforts and eventual successes that follow will pay for the increased planning. Δ

— 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>.

Developer Ethics: Respecting Intellectual Property

As developers, we are all aware of the amount of work we put into creating Delphi applications, components, and tools. When we develop something new and powerful, we have a sense of satisfaction and excitement. We also expect to receive certain rewards, usually in the form of monetary compensation. But what if someone else takes the results of our work and proclaims it as their own?

How many readers remember the legal dispute between Borland and Lotus some 10 years ago in which the latter sued the former over similarities between Lotus 1-2-3 and Borland's Quattro Pro (since then sold to Corel Corporation)? At the time, I thought it was a bit silly. However, Lotus felt that, in the name of "compatibility," Borland had gone too far in copying elements of the user interface. I think Borland won, pointing out an important exception to what specific intellectual property can, or should, be protected: basic user interface elements (like menus and dialog boxes) can not and should not be protected. What then can and should be protected?

Let's begin with the example that inspired this column. I write many product reviews for *Delphi Informant*. As a result, I often correspond with developers and companies. A representative of one company wrote to me recently, describing what appears to be a serious infringement. Another company developed a similar library of components, with similar functionality. The programmers at the first company found it strange that: "Many of the structures, objects, and variable names of [our library] can be found easily in their DCU files." The clincher was the discovery that one of the developers at the second company was a registered subscriber to the original library of components. While the source code for the original set of components was (and is) available to developers who purchase the product, the second company did not make their source code available.

On behalf of the first company (which was asking my advice on what to do), I

contacted developers I knew and learned they had faced similar problems. One step the aggrieved company was considering was changing their policy on making source code available. Despite the apparent intellectual theft, my initial reaction was that it would be a serious mistake to change this policy. The other developers I contacted agreed. Further, they stated that the key to success in this industry is in continuing to develop the set of components so their robustness and functionality was always on the cutting edge.

My developer friends also advised that the aggrieved company should proceed very carefully before pursuing legal action. The cost could be more than it was worth. There was an interesting development during the course of these discussions. On the Internet, users complained about a lack of support and a high level of bugs in the "new" product. As you can imagine, they were the same bugs that had already been removed from the "older" product. It seems there is some justice in the developer world.

Often, we can solve problems like this very straightforwardly. One of the developers I contacted told me of a different case of "borrowing" intellectual property. His company had worked hard to come up with an impressive Web site on which to market their Delphi products. It turned out to be so impressive that another Delphi developer used much of the visual appearance and even one of the snappy slogans. When my contact discovered this, he simply wrote to the perpetrator, pointed out the surprising "similarities," and asked that they come up with their own design and language.

They complied. Here, at least, is a case where simply confronting the other party was sufficient to solve the problem. If you're faced with a similar problem, I think this is the best place to start.

One problem with dishonest people is they believe they'll never be caught. Because they tend to repeat this sort of behavior, I think it's more likely they will be caught, later if not immediately. Internet discussion groups, list servers, and newsgroups make this even more likely. Fortunately, most Delphi developers are honest and would not stoop to such schemes. Most take a great deal of pride in their abilities and would not think of stealing from another developer. As we've seen, however, there are exceptions, and it's the responsibility of all of us to do what we can to stop them. That's what I hope to contribute by writing this column. If you've experienced, or know of, similar instances, please let me know. While I would refrain from naming specific individuals or companies in these pages, I would not hesitate to write further on this topic to promote respect for the intellectual property of developers. Δ

— 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.