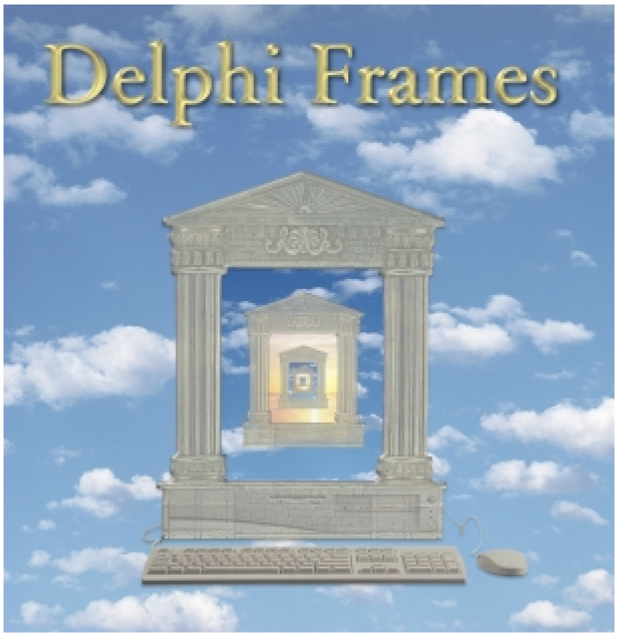
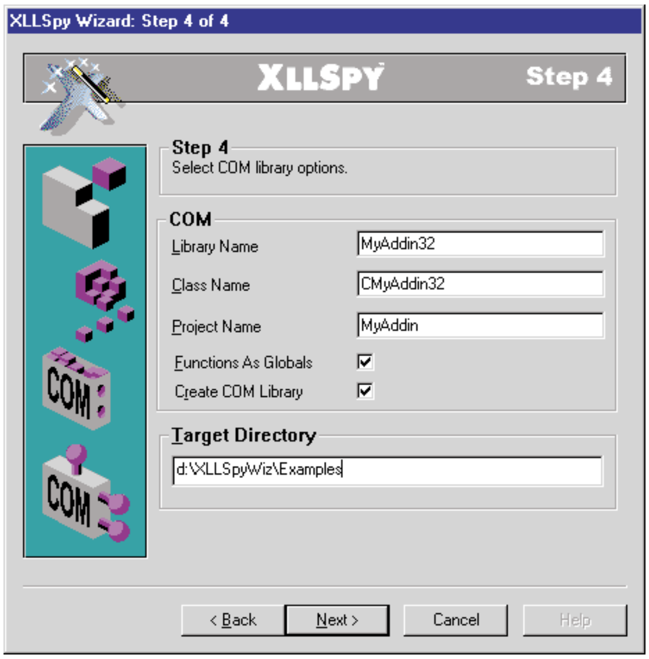
December 1999, Volume 5, Number 12



Cover Art By: *Diana Nishimura*

Xenomorph Announces XLLSpy

Xenomorph announced the release of *XLLSpy*, a product that instantly converts proprietary option analytics in a Microsoft Excel XLL add-in to a Microsoft COM library. An XLL add-in containing any number of functions can be converted into a COM library in a few seconds.



Step 4
Select COM library options.

COM
Library Name: MyAddin32
Class Name: CMyAddin32
Project Name: MyAddin
Functions As Globals ✓
Create COM Library ✓

Target Directory
d:\XLLSpyWiz\Examples

After being converted, proprietary option analytics can be used independently of Excel.

They can also be accessed directly from a variety of system architectures and development environments, such as Delphi, Visual Basic, and Visual C++.

The main advantage for traders and risk managers in having option analytics as COM components is that new financial products can be integrated into risk management systems faster.

Xenomorph
Price: Contact Xenomorph for pricing.
Phone: +44 (0)181 971 0080
Web Site: http://www.xenomorph.com

DeVries Data Systems Announces OfficePartner

DeVries Data Systems, Inc. announced *OfficePartner*, a suite of VCL components designed to integrate the Microsoft Office suite with Borland software development tools. OfficePartner enables developers using Delphi or C++Builder to exploit the power of Microsoft Office, Excel, Outlook, and PowerPoint in their own applications.

OfficePartner enables developers to build custom solutions without the in-depth knowledge of COM and Automation techniques normally necessary for programming with Microsoft Office. Design-time support allows developers to preview the results of their development efforts as they design the software.

Other OfficePartner features include point-and-click access to Microsoft Office applications; smooth handling of Office's Automation events as VCL events; royalty-free redistribution of applications built with OfficePartner; complete source code; and support for Office 97 and 2000, Delphi 3, 4, and 5, C++Builder 3 and 4, and Windows 95, 98, NT 4, and 2000.

DeVries Data Systems, Inc.
Price: US$399
Phone: (888) 866-8033
Web Site: http://www.dvdata.com

Digital Metaphors Ships ReportBuilder 4.1

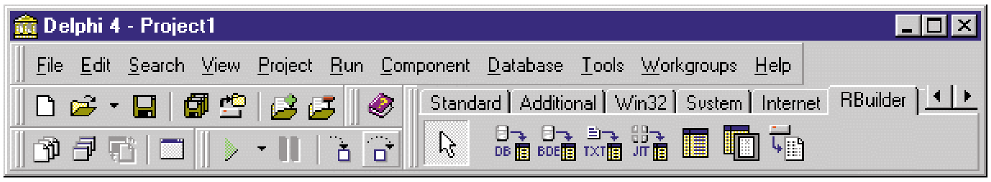
Digital Metaphors Corp. announced it is shipping *ReportBuilder 4.1*. This latest version adds a number of features, including a new Crosstab component (included in the Professional version) and support for printing JPEG, GIF, and RTF2 content.

The Crosstab component enables users to display calculated values in a multi-dimensional format. A crosstab wizard and a drag-and-drop crosstab editor can be used to quickly define any number of row, column, and value dimensions.

The ReportBuilder Image components can now be used to print BMP, WMF, JPEG, and GIF image formats. Additional formats may be supported by registering descendants of Delphi's *TGraphic*.

Users of the InfoPower RichEdit components that support the Windows RichEdit2 format can now use ReportBuilder's RichText components to print RTF2 memos.

Digital Metaphors Corp.
Price: ReportBuilder Standard, US$249; ReportBuilder Professional, US$495.
Phone: (972) 931-1941
Web Site: http://www.digital-metaphors.com


Delphi 4 - Project1
File Edit Search View Project Run Component Database Tools Workgroups Help
Standard Additional Win32 System Internet RBuilder
DB BDE TXT JIT

Object River Announces EasyUpper

Object River Information Technology, Inc. announced *EasyUpper*, a tool for transforming Delphi Client/Server units (source code, forms, and DataModules) into N-Tier/ActiveForm, and Delphi 4 N-Tier into Delphi 5 N-Tier.

Without re-creating new units, copying components, or changing properties piece by piece, Client/Server units can be transformed into N-Tier; simply give EasyUpper the source and target direction. EasyUpper divides a C/S unit into a RemoteDataModule unit, a Type Library, and a client unit. All components, properties, events, and source code can be reserved and arranged in the client or server tier.

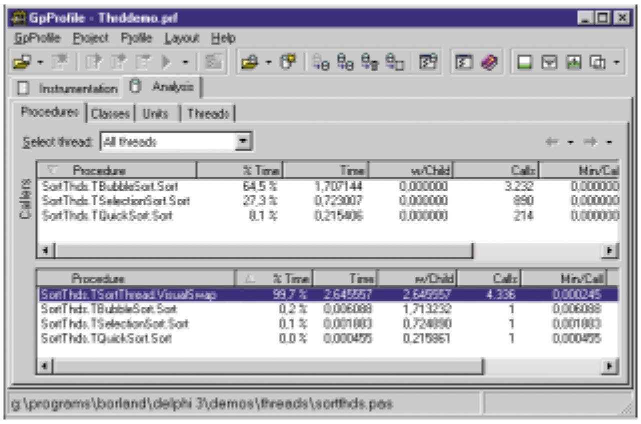
Developers can have the advantages of stateless RemoteDataModules from MIDAS 3.0, but without the pain of upgrade incompatibility. DataModules can be converted, and all reference links between forms can be reserved. Master-Detail Datasets can also be arranged at the Client and AP Server. There are various source and target versions and style options available for different situations.

Object River Information Technology, Inc.
Price: US$149
E-Mail: sophie@www.objectriver.com
Web Site: http://www.objectriver.com

Primoz Gabrijelcic Releases GpProfile 1.3.2

Primoz Gabrijelcic announced the release of *GpProfile 1.3.2*, a profiler for Delphi. The new version includes several fixes, such as the bug that caused a corrupted profile file when profiling large projects, and a bug related to output directory processing.

Other new features include additional setting to control instrumentation of pure assembler procedures, and a parser change for better compatibility with the Delphi parser (which allows {$ENDIF<} and variations to be written instead of {$ENDIF}).



Procedure	% Time	Time	w/Child	Calls	Min/Call
SortThds.TBubbleSort.Sort	64,5 %	1,707144	0,000000	3.232	0,000000
SortThds.TSelectionSort.Sort	27,3 %	0,723007	0,000000	890	0,000000
SortThds.TQuickSort.Sort	8,1 %	0,215406	0,000000	214	0,000000

Procedure	% Time	Time	w/Child	Calls	Min/Call
SortThds.TSortThread.VisualSwap	99,7 %	2,645557	2,645557	4.336	0,000245
SortThds.TBubbleSort.Sort	0,2 %	0,006088	1,713232	1	0,006088
SortThds.TSelectionSort.Sort	0,1 %	0,001883	0,724890	1	0,001883
SortThds.TQuickSort.Sort	0,0 %	0,000455	0,215861	1	0,000455

g:\programs\borland\delphi 3\demos\threads\sortthds.pas

Primoz Gabrijelcic
Price: Free
E-Mail: primoz.gabrijelcic@altavista.net
Web Site: http://www.eccentrica.org/gabr/gpprofile

Tetradyne Releases SourceView ActiveX Control Version 2.1

Tetradyne Software Inc. announced the release of *SourceView ActiveX Control Version 2.1*, a high-performance syntax-highlighting text editor component for developers.

New features in version 2.1 include drag-and-drop, column selection, bookmark support, HTML parsing support, and major performance improvements. Version 2.1 also supports Visual Basic-style automatic case conversion for reserved words. Source code for the control is available.

The SourceView ActiveX control can be customized for coloring of any language syntax. In most cases, customization can be accomplished by setting control properties. For more complex syntax requirements, COM interface hooks are provided for plugging in custom parsing implementations. Delphi, Visual Basic, Visual C++, and C++Builder examples are provided for coloring of C, Pascal, Basic, and HTML syntax.

Developers using modern development tools that support implementing COM interfaces can provide margin bitmaps and other custom display elements by implementing interfaces defined by the SourceView ActiveX Control. Using this feature, developers can display breakpoints, bookmarks, and other custom line attributes.

Tetradyne Software Inc.
Price: US$299 without source; US$499 with source.
Phone: (916) 686-8550
Web Site: http://www.tetradyne.com

MCM Design Offers TWAIN Toolkit for Delphi Version 1.8.1

MCM Design is offering the *TWAIN Toolkit for Delphi version 1.8.1*, which enables developers to implement the TWAIN standard into Delphi 3 or 4 applications by using the VCL components included in the toolkit.

The toolkit provides methods for changing the acquisition properties of a TWAIN driver. MCM has also implemented a report generator that stores all communication between an application and the TWAIN driver in a file during the test phase.

The toolkit also provides the necessary code to get you started writing your own TWAIN driver.

MCM Design
Price: US$100
Phone: +45 48146667
Web Site: http://www.mcm-design.dk

Excel Software Announces WinA&D 3.0

Excel Software announced *WinA&D 3.0* for system analysis, requirements specification, software design, and code generation.

Version 3.0 adds Java code generation, design namespaces, UML enhancements, dictionary enhancements, document templates, and productivity features. WinA&D has diagram editors for process models, data models, class models, state models, object models, structure models, and task models. Each model shows a different view of a software system that is integrated through a global data dictionary. In addition to integrated modeling, the tool provides design verification reports, scriptable HTML reports, and complete text import/export features.

WinA&D 3.0 generates Delphi, Java, or C++ code from a class model and associated dictionary information. The generated code includes ready-to-compile class declarations with empty function frames. Programmers write code for the logic of each function using either the integrated code and browse window built into WinA&D, or any standard development environment. The code generator supports features such as nested classes, template classes, overloaded functions, and mapping design elements to different code folders based on design namespaces.

WinA&D 3.0 adds interface and component objects to its extensive UML support. Interfaces are represented in the UML class model as stereotyped classes or with the abbreviated lollipop notation. Operations of an interface can be defined once and automatically incorporated into any class that implements that interface. The logical class abstraction with attributes and operations is the standard design element of a class model. Components add properties and events to represent a physical element in the software system. Typical components include Delphi components, ActiveX controls, COM objects, CORBA objects, and JavaBeans.

Excel Software
Price: Standard, US$495; Desktop, US$1,295; Educational, US$845; Developer, US$1,995.
Phone: (515) 752-5359
Web Site: http://www.excelsoftware.com

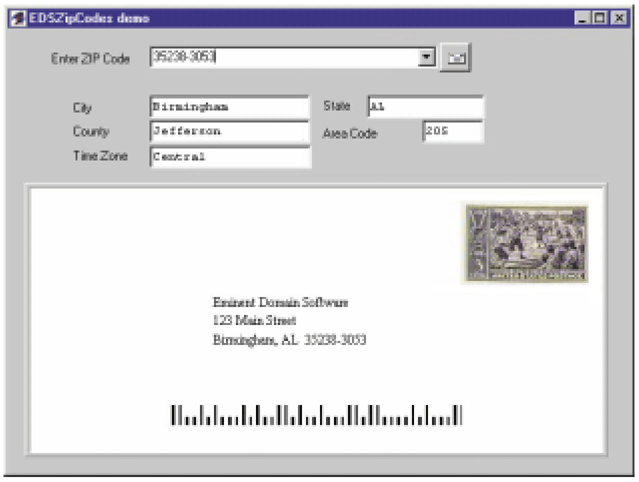
Eminent Domain Announces EDSZipCodes

Eminent Domain Software announced the release of ***EDSZipCodes 1.0*** for Borland Delphi and C++Builder. With EDSZipCodes, developers can add a complete ZIP code database without using any lines of code. EDSZipCodes takes advantage of Eminent Domain's spell-checking engine to spell-check a ZIP code. Using this technology, a 15MB ZIP code database is compressed to 1.2MB (92 percent compression).

EDSZipCodes contains two components, *TEDSZipCombo* and *TEDSZipDlg*. The *TEDSZipCombo* can be dropped on any form and behaves as a normal combobox. By setting the appropriate properties, *TEDSComboBox* will update other edit controls on your form with the corresponding city, county, state, area code, and time zone for the specified ZIP code. The *TEDSZipDlg* behaves much like an OpenDialog or PrintDialog. Calling *TEDSZipDlg.Execute* will allow the user to select the desired ZIP code.

EDSZipCode supports third-party components, including TurboPower's Orpheus, Woll2Woll Software's InfoPower, and Julian Ziersch's WPTools. Any edit component will work with EDSZipCode.

Eminent Domain Software
Price: US$89
Phone: (800) 246-5757
Web Site: http://www.onedomain.com



Enter ZIP Code: 35238-3053

City: Birmingham
State: AL
County: Jefferson
Area Code: 205
Time Zone: Central

Eminent Domain Software
123 Main Street
Birmingham, AL 35238-3053

North Winds Releases PDF Forms Toolkit Component for Delphi

North Winds, Inc. released its *PDF Forms Toolkit Component for Delphi* to be used in conjunction with Adobe's Acrobat. The component allows developers to use the Adobe PDF forms from within their Delphi program.

Developers can build their paper forms with Adobe Acrobat and populate them with data from their Delphi program. They may also electronically receive PDF forms and retrieve the field data using the PDF Forms Toolkit Component. The component also allows Delphi programmers to import and export FDF data to PDF forms with a few lines of code.

North Winds, Inc.
Price: Not available at press time.
Phone: (724) 838-8993
Web Site: http://www.nwinds.com

News LINE

December 1999

Inprise Names J.D. Hildebrand Content Director for New Community Site

Scotts Valley, CA — Inprise Corp. named J.D. Hildebrand Content Director and Editor-in-Chief of Inprise's online community for software developers. The service is currently being previewed at http://community.borland.com. The goal of the new site is to provide information and services to software developers worldwide.

As Editor-in-Chief, Hildebrand will establish and supervise systems for selecting, acquiring, writing, fact-checking, and editing content for the site. Prior to joining Inprise, Hildebrand served as editorial director of the Developer Group at PennWell Publishing Co., where he was responsible for *VB Tech Journal*, *Windows Tech Journal*, and other publications for software development professionals.

A programmer himself, Hildebrand brings over 20 years of experience in editorial management, magazine launches, trade-show management, curriculum development, editorial training, and business development to the project. He has trained thousands of magazine editors on topics ranging from fact-checking and copyediting to magazine launches and finance. Hildebrand has held editorial-management positions at Miller Freeman, Oakley Publishing, and Camden Communications. His work has received awards from the Computer Press Association, Western Publications Association, Magazine Design & Production, and *Folio:* magazine.

BDE 5.10 Available

Scotts Valley, CA — Inprise Corp. released the latest update to the Borland Database Engine (BDE) and SQL Links, version 5.10. This BDE 5.10 upgrade install will update any previous version of the US 32-bit BDE drivers with BDE 5.10.

This update will not place BDE files on a system that does not already include an older BDE version; this is an update program, not an installation program. Due to legal reasons, this will exclude the BDE from becoming freeware from Borland. Under your Borland license agreement, you still have the right to distribute the BDE, including this version, with your programs.

To download BDE 5.10 and for important install information, visit http://www.borland.com/devsupport/bde/bdeupdate.html.

Inprise and Data General to Deliver Enterprise Solutions

Scotts Valley, CA and *Westboro, MA* — Inprise Corp. announced the availability of VisiBroker for Data General Corp.'s AViiON line of DG/UX servers.

VisiBroker allows Data General to satisfy the needs of its enterprise customers seeking a scalable, flexible, and easily maintainable solution.

As part of the agreement, Inprise's Professional Service Organization and Data General are providing a wide variety of consulting and training, as well as technical support services to Data General's customers.

Data General's AViiON servers provide customers with a family of enterprise systems based on Intel architecture. AViiON provides a foundation for strategic applications in a variety of markets, including healthcare, finance, manufacturing, and others.

Data General's AV 25000 server, using the DG/UX operating system, can support up to 64 500-MHz Pentium III Xeon processors with up to 2MB of cache per processor, 64GB of memory, and over 100TB of CLARiiON Fibre Channel storage.

The AV 25000 continues Data General's high-availability tradition with dedicated diagnostics processors; AV/AlertSM "phone home" remote support; redundant, hot-swappable power and cooling; and clustering capabilities, including Disaster Recovery Clusters.

Additional information on Data General is available at http://www.dg.com.

Director's Lawsuit against Inprise Dismissed

Scotts Valley, CA — Inprise Corp. announced that C. Robert Coates, who recently joined Inprise's Board of Directors, voluntarily dismissed his lawsuit in Delaware Chancery Court against the company and other directors. The lawsuit challenged the validity of certain policies and procedures recently adopted by the Board. The company stated at the time the lawsuit was filed that it believed the lawsuit was frivolous and entirely without merit.

Inprise and Corel Form Alliance to Accelerate Linux

Ottawa, Canada — Inprise Corp. and Corel Corp. announced a strategic alliance to accelerate commercial mainstreaming of Linux technology. The alliance will enable the companies to develop middle-tier solutions for distributed computing.

As part of the alliance, the companies will form a research-and-development partnership to facilitate the development of Corel's office productivity applications and Inprise's application development tools and enterprise solutions for the Linux operating system. Linux products from both companies will also be jointly marketed and distributed.

Corel Corp. is a developer of graphics and business productivity applications for the Windows, Macintosh, UNIX, Linux, and Java platforms. For more information, visit the Corel Web site at http://www.corel.com.

table of CONTENTS

By Cary Jensen, Ph.D.

Delphi Frames

Understanding Delphi 5's New Visual Container Class

Delphi 5 introduces a new visual container class that represents an important advance in rapid application development (RAD) programming. This class, *TFrame*, provides you with the ability to visually configure a set of one or more components, and then to easily reuse this configuration throughout your application. This capability is so powerful that Delphi 5's integrated development environment (IDE) was re-designed to make extensive use of frames.

This article begins with a general discussion of what frames are, and what benefits they provide. It continues with a demonstration of how to create frames, and how to modify the properties of objects that appear on frame instances. Next, you'll learn how to create event handlers for frames, and how to override or extend these event handlers in frame instances. This article concludes by showing you how to add frames to the Component palette and the Object Repository, and the benefits of doing so.

Overview of Frames

There are two primary benefits of frames. The first is that, under certain circumstances, frames can dramatically reduce the amount of resources that need to be stored in a project. The second, and generally more important benefit, is that frames permit you to visually create objects that can be duplicated and extended. These happen to be the same two benefits that you enjoy with visual form inheritance (VFI).

VFI permits you to create form objects that can be inherited from easily. The main limit to VFI is that you must use the form in an all-or-nothing fashion. Specifically, when you use VFI you always create an entirely new form. Frames, on the other hand, are more similar to panels in this respect. That is, a single form can contain two or more frames. Importantly, every frame maintains its relationship with the parent *TFrame* class, meaning that subsequent changes to the parent class are automatically inherited by the instances. Although you could achieve a similar effect using *TPanel* components, doing so would be a strictly code-based operation. That is, you would have to write the code to define the *TPanel* descendants manually. Frames, on the other hand, are designed visually, just like forms.

Frames can also be thought of as sharing some similarities with component templates (a group of one or more components that are saved to the Component palette by selecting Component | Create Component Template). However, the similarities are limited to the fact that both component templates and frames are designed visually (unlike traditional component design, which is an exclusively code-based process). The differences between component templates and frames are actually very great. As you've already learned, a frame is an instance of a defining class, and, as such, is changed when the defining class is changed. By comparison, component templates are aggregates of components. A change to a component template has no effect on objects previously created from that template.

Creating a Frame

The following steps demonstrate how to create a frame (the code for this project is available for download; see end of article for details).

1) Select File | New Application to create a new project.
2) Select File | New Frame to create a new frame. On this frame, place three labels and three DBEdits. Also place a DBNavigator and a DataSource (as shown in Figure 1). Set the captions of the labels to `ID`, `First Name`, and `Last Name`. Set the *DataSource* property of each DBEdit and the DBNavigator to `DataSource1`.
3) With this frame still selected, set its *Name* property to `NameFrame`. (More so than other objects, it's particularly important to give a frame a meaningful name.) Finally, save the frame by selecting File | Save As. In this case, save the frame using the file name NAMEFRAM.PAS.

That's all there is to creating a frame. The following section demonstrates how to put it to use.

Using a Frame

A frame is a component. However, its use typically differs from most other components that appear on the Component palette. The following steps demonstrate how to use a frame:

1) Select *Form1* of the application you created in the preceding steps.
2) Add two group boxes to the form, one above the other. Set the caption of the first frame to `Customers`, and the caption of the second to `Employees`. Your form may look something like that shown in Figure 2.
3) Now add the frames. With the Standard page of the Component palette selected, click on the Frame component and drop it in the Customers frame. Delphi responds by displaying the Select frame to insert dialog box (see Figure 3).
4) Select NameFrame. The frame will now appear in the Customers frame. Repeat this process, this time placing the frame within the Employees frame. You may have to select each frame and correct its size, depending on how you placed it originally. When you're done, your form should look similar to that shown in Figure 4.
5) Continue by placing two Table components onto the form. Set the *DatabaseName* property of both tables to `IBLocal`. Set the *TableName* property of *Table1* to `CUSTOMER` and the *TableName* property of *Table2* to `EMPLOYEE`. Make both tables active by setting their *Active* properties to True.
6) Here's where things get interesting. Select the DataSource in the Customers frame, and set its *DataSet* property to `Table1`. Normally you can't directly select objects that appear within a component, but frames are special. You can select any of the objects that appear within a frame, and work with their properties. Next, repeat this operation by selecting the DataSource in the Employees frame and setting its *DataSet* property to `Table2`.
7) Finally, hook up all the DBEdits. Assign the *DataField* property of the three DBEdits on the Customers frame to `CUST_NO`, `CONTACT_FIRST`, and `CONTACT_LAST`, respectively. For the Employees frame, set the *DataField* properties of these same DBEdits to `EMP_NO`, `FIRST_NAME`, and `LAST_NAME`.
8) Save this project and then run it. The running project will look something like that shown in Figure 5.

Frames and Inheritance

Up to this point, there may seem to be little benefit to using frames. However, it's when you use the same frame in a number of different situations, and then want to change all instances, that the power of frames becomes obvious. For example, imagine you've decided to make *NameFrame* read-only. This can be accomplished easily by simply changing the original frame; each frame instance immediately inherits all changes.

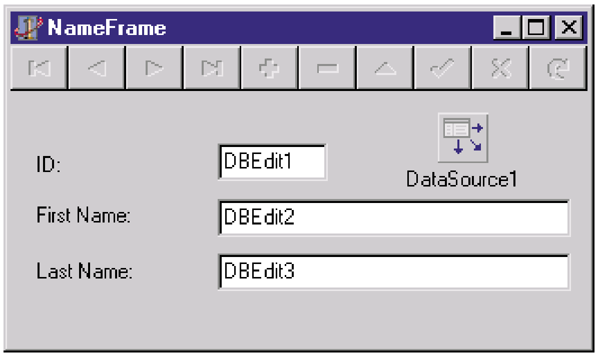


Figure 1: A simple frame for displaying an ID number, as well as a first and last name.

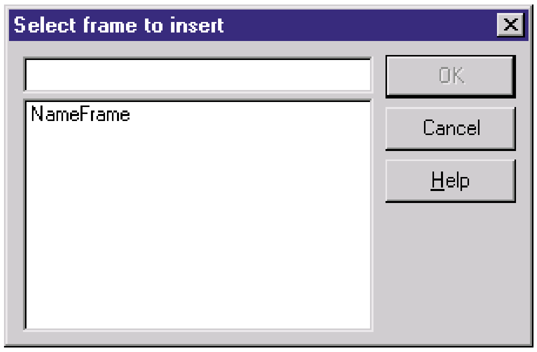


Figure 3: The Select frame to insert dialog box.

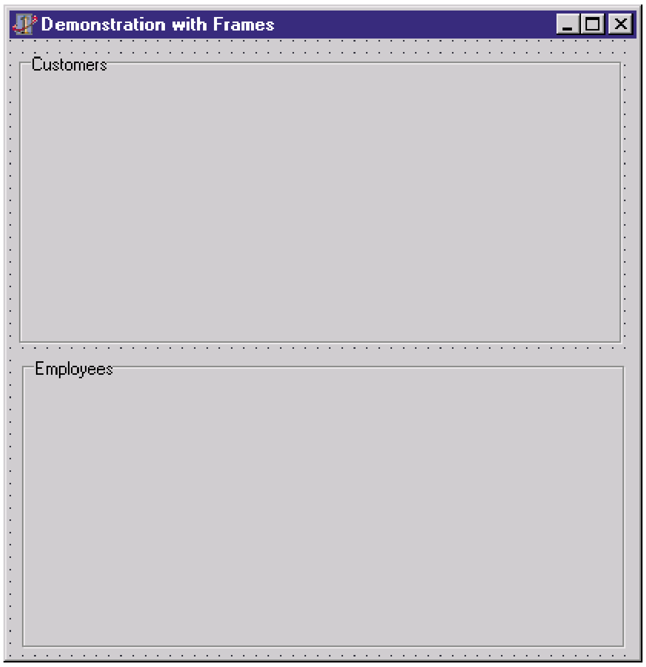


Figure 2: A form ready for the placement of frames.

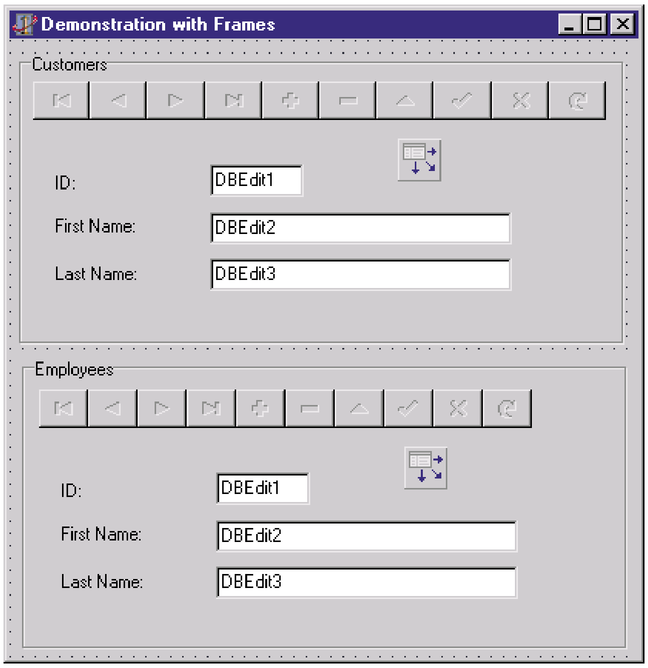
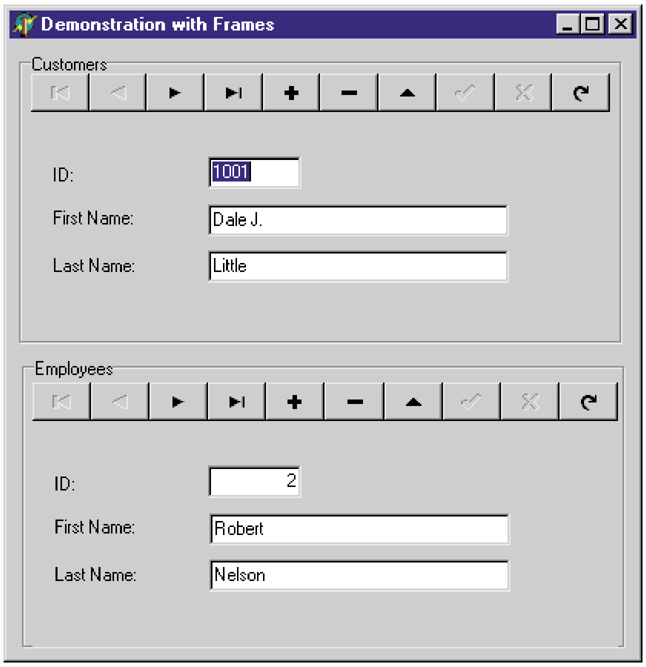


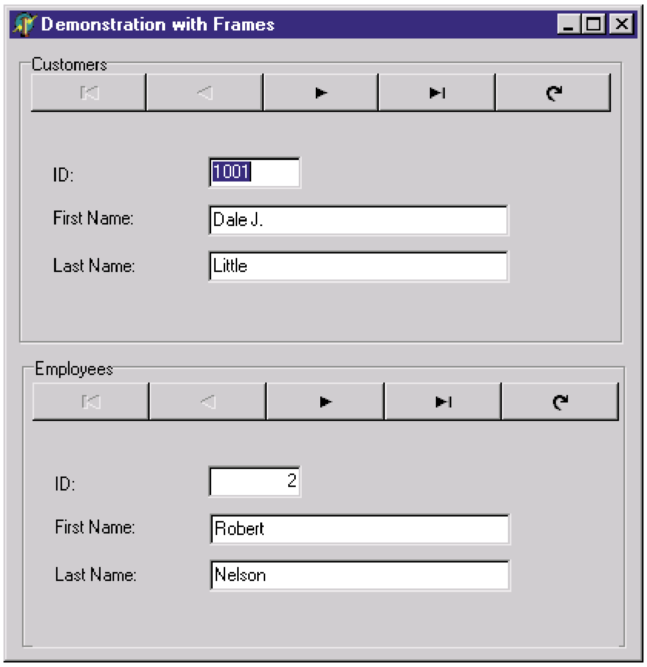
Figure 4: Two instances of *NameFrame* appear on this form.



ID: 1001
First Name: Dale J.
Last Name: Little

ID: 2
First Name: Robert
Last Name: Nelson

Figure 5: The example frame project at run time.



ID: 1001
First Name: Dale J.
Last Name: Little

ID: 2
First Name: Robert
Last Name: Nelson

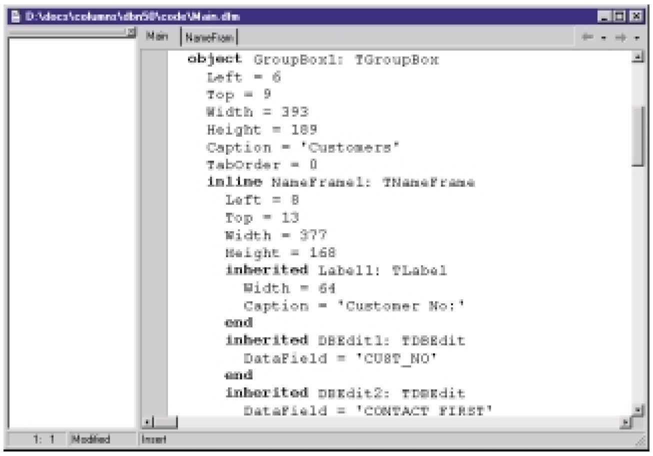
Figure 6: Updating *NameFrame* automatically causes all instances to be updated as well.

You can demonstrate this by following these steps:

1) With the project created in the preceding section, press ⇧Shift F12 and select NameFrame from the displayed list of forms.
2) Set the *AutoEdit* property of the DataSource to False.
3) Next, select the DBNavigator, expand its *VisibleButtons* property, and set the *nbInsert*, *nbDelete*, *nbEdit*, *nbPost*, and *nbCancel* flags to False.
4) Now look at your main form. Notice that both *NameFrame* descendants have inherited the changes you made to the frame (see Figure 6).

Overriding Contained Component Properties

One of the advantages of frames (one shared with VFI) is that you



```
object GroupBox1: TGroupBox
  Left = 6
  Top = 9
  Width = 393
  Height = 189
  Caption = 'Customers'
  TabOrder = 0
  inline NameFrame1: TNameFrame
    Left = 8
    Top = 13
    Width = 377
    Height = 168
    inherited Label1: TLabel
      Width = 64
      Caption = 'Customer No:'
    end
    inherited DBEdit1: TDBEdit
      DataField = 'CUST_NO'
    end
    inherited DBEdit2: TDBEdit
      DataField = 'CONTACT_FIRST'
```

Figure 7: A DFM file containing property overrides for a frame instance.

can change the properties and event handlers associated with the objects inside the inherited frame. These changes override the inherited values. Specifically, subsequent changes to the overridden property in the original frame don't affect the inherited value. The following steps demonstrate this behavior:

1) Select the label whose caption is "ID" in the Customers frame. Using the Object Inspector, change its *Caption* property to `Customer No:`. Now select the ID label for the Employees frame and change it to `Employee ID:`.
2) Press ⇧Shift F12 and select `NameFrame`. Change the caption of this ID label to `Identifier`.
3) Return to the main form. Notice that the *Caption* properties of the labels haven't changed to Identifier. They still use their overridden values.
4) This effect is accomplished through information stored in the DFM file. Figure 7 displays a relevant part of the DFM file for this project.

Notice that information about all components contained within the frame whose property values have been changed appear in the frame's inline section of the DFM file. However, this section only lists those values that have been changed. All other properties are assigned their values based either on the values set for the original frame (and which are stored in the frame's DFM file), or are designated as default values in the individual component's class declarations.

Contained Object Event Handlers

Objects contained within a frame may also have event handlers. Although events are simply properties of a method pointer type, they're treated differently than other types of properties when it comes to overriding the default behavior defined for the frame.

Let's begin by considering how an event handler is defined for a frame object. Consider the frame shown in Figure 8. (This code is found in the *Frame2* project found in the download for this article.) This frame contains two buttons, one labeled **Help** and the other **Done**. (Of course, these captions can be overridden in descendant frames.) These buttons also have *OnClick* event handlers, which are shown in Figure 9.

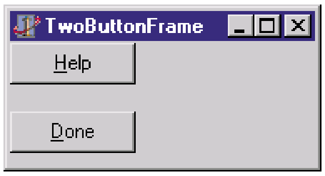


Figure 8: A frame with components that have event handlers.

Just as the event handlers for objects on a form are published methods of that form's class,

```
procedure TTwoButtonFrame.Button1Click(Sender: TObject);
begin
  if (TComponent(Sender).Tag = 0) or
     (Application.HelpFile = '') then
    MessageBox(Application.Handle,'Help not available',
               'Help',MB_OK)
  else
    Application.HelpContext(TComponent(Sender).Tag);
end;

procedure TTwoButtonFrame.Button2Click(Sender: TObject);
var
  AParent: TComponent;
begin
  AParent := TComponent(Sender).GetParentComponent;
  while not (AParent is TCustomForm) do
    AParent := AParent.GetParentComponent;
  TCustomForm(AParent).Close;
end;
```

Figure 9: The *OnClick* event handlers for the **Help** and **Done** buttons on our frame.

```
procedure TForm1.TwoButtonFrame1Button2Click(
  Sender: TObject);
begin
  with TForm2.Create(Self) do begin
    ShowModal;
    Release;
  end;
  // The following is the original, auto-generated code
  // TwoButtonFrame1.Button2Click(Sender);
end;
```

Figure 10: An overridden event handler for a *TwoButtonFrame* descendant that was placed on a form.

the event handlers of objects on a frame are published methods of that frame. (The code segment doesn't actually depict the fact that these methods are published. Rather, they're declared in the default visibility section of the frame's class declaration, and the default visibility is published.)

If you inspect the code associated with the *Button2Click* event handler, which is associated with the Done button, you'll notice that the event handlers associated with the frame introduces an interesting artifact. Specifically, *Self* is the frame, not the form in which the frame is contained. Consequently, it isn't possible to simply invoke the *Close* method from within this event handler to close the form. When an unqualified method invocation appears in code, the compiler assumes you want it to apply to *Self*. Because a *TFrame* object doesn't have a *Close* method, the compiler generates an error if you simply use an unqualified call to *Close*.

Because the frame in this example is designed to be embedded within a form, the event handler uses the *GetParentComponent* method of the frame to climb the containership hierarchy within which the frame is nested. Once a *TCustomForm* instance is found (which will either be a *TForm* descendant or a custom form based upon *TCustomForm*), that reference is used to invoke the form's *Close* method.

Overriding Contained Object Event Handlers

If you're familiar with event overriding in VFI, you'll recall that Delphi embeds a call to **inherited** from within an overridden event handler on a descendant form. You can then alter the generated code to either add additional behavior before, or following, the call to **inherited**, or conditionally invoke **inherited**, or you can omit the call altogether.

Frame descendants don't use **inherited** when invoking the event handler for an object embedded on the parent frame. Instead, the ancestor frame's method is called directly. For example, if you place the *TwoButtonFrame* frame (shown in Figure 8) onto a form and then double-click it, Delphi will generate the following code:

```
procedure TForm1.TwoButtonFrame1Button2Click(
  Sender: Object);
begin
  TwoButtonFrame1.Button2Click(Sender);
end;
```

In this generated code, *TwoButtonFrame1* is the frame descendant of *TTwoButtonFrame* (the original frame's class). *Button2Click*, as you saw in the earlier code segment, is the event handler for the Done button on that frame. As a result, this code invokes the original event handler, passing it the *Sender* that was passed to the button on the frame instance.

This means that event handling introduces another interesting feature. Specifically, in these situations, *Sender* is generally not a member of the *Self* object. Indeed, *Sender* is usually a member of the form object, and *Self* is the frame object.

Figure 10 shows an overridden event handler for a *TwoButtonFrame* descendant that was placed on a form. In this case, the original behavior is "commented out," so the new behavior completely replaces the originally defined behavior for the Done button.

The caption of this button was also overridden, so it displays the text, `Start`. Figure 11 shows the form on which this *TwoButtonFrame* descendant appears.

Frames that Save Resources

The form shown in Figure 11 actually contains two frames. We've already discussed the *TwoButtonFrame* frame. The second frame displays the company logo, and is named *LogoFrame*.

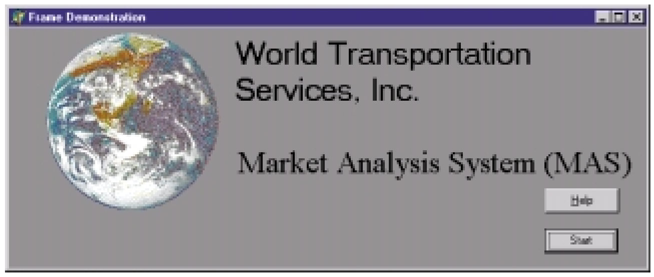
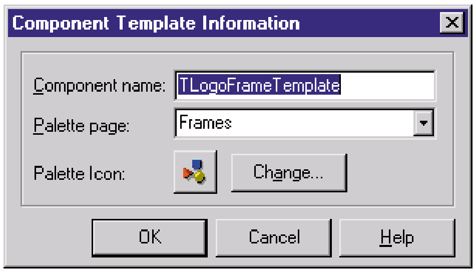

Frame Demonstration
World Transportation Services, Inc.
Market Analysis System (MAS)
Help
Start

Figure 11: This *TwoButtonFrame* instance overrides both the caption and the *OnClick* event handler.

```
object LogoFrame: TLogoFrame
  Left = 0
  Top = 0
  Width = 239
  Height = 178
  TabOrder = 0
  object Image1: TImage
    Left = 0
    Top = 0
    Width = 239
    Height = 178
    Align = alClient
    Picture.Data = {
      07544269746D6170D6540000424DD654000000000000760000...
```

Figure 12: A segment of *LogoFrame*'s DFM file.


Component Template Information

Component name: TLogoFrameTemplate

Palette page: Frames

Palette Icon:

Figure 13: The Component Template Information dialog box.

LogoFrame appears on more than one form in the *FramDemo* project. The alternative to using a frame to display the logo is to place an *Image* object on each form upon which you want the logo to appear. However, the use of a frame for this purpose significantly reduces the amount of resources that must be compiled into the .EXE, and, therefore, results in a smaller executable.

The reason for this can be seen if you consider the following segment of the DFM file for the form shown in Figure 11:

```
inline LogoFrame1: TLogoFrame
  Left = 6
  Top = 6
  Width = 211
  Height = 182
  inherited Image1: TImage
    Width = 211
    Height = 182
  end
end
```

If, instead, a *TImage* instance had been placed onto the form, the DFM file for the form would have had to contain the entire binary representation of the logo. Figure 12 shows a segment of *LogoFrame*'s DFM file. (Note that it shows only a tiny portion of the entire hexadecimal representation of the binary resource.) Furthermore, every form containing one of these images would have repeated this resource. When a frame is used, however, that resource is defined only once.

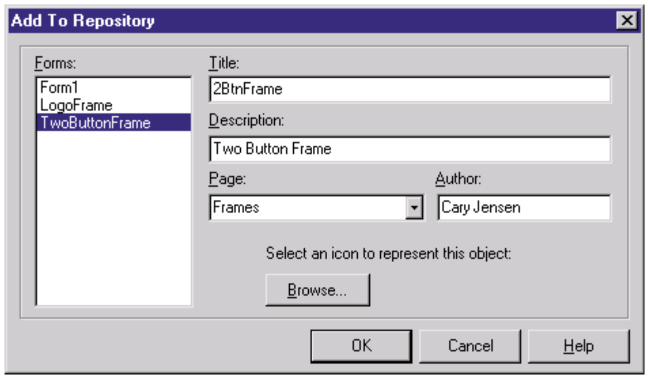
Simplifying Frame Use

Within a single, small project, it's fairly easy to use the Frame component on the Standard page of the Component palette. For larger projects, however, or for situations where you want to use the same frame in multiple applications, you need something easier. Fortunately, Delphi permits you to place individual frames onto the Component palette, permitting these frames to be used easily and repeatedly without the extra steps required by the Frame component. A frame can also be placed into the Object Repository, permitting it to be copied easily. Both of these techniques are described in the following sections.

Adding a Frame to the Component Palette

By placing a particular frame onto the Component palette, you make its placement as simple as any other. By comparison, using the Frame component on the Standard page of the Component palette requires four steps and limits you to placing frames already defined within your project. To place a particular frame onto the Component palette, follow these steps:

1) Save your frame to disk. If you want to use this frame in multiple applications, it's highly recommended that you save the frame to a directory that won't be deleted when you update Delphi. For example, create a folder named C:\Program Files\Borland\DelphiFrames and store your frames there.
2) Select the frame and right-click on it. Select **Add to Palette**.


Add To Repository

Forms: Form1, LogoFrame, TwoButtonFrame (selected)

Title: 2BtnFrame

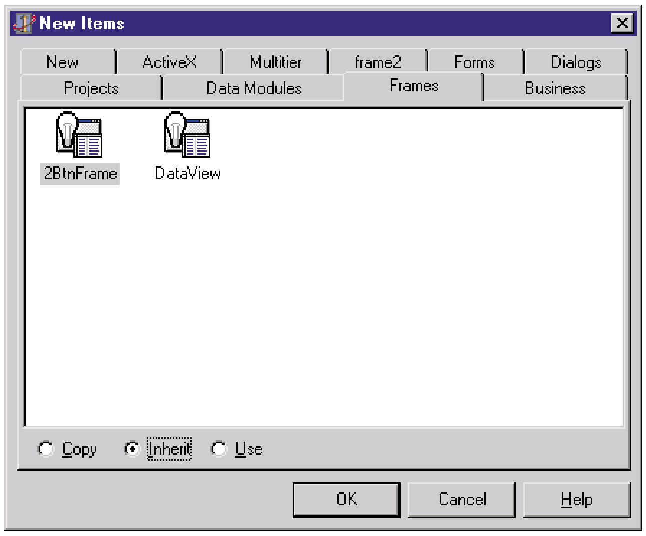
Description: Two Button Frame

Page: Frames

Author: Cary Jensen

Select an icon to represent this object:

Figure 14: The Add To Repository dialog box.


New Items

Selected: Inherit

Figure 15: The location of your frame template.

Delphi displays the Component Template Information dialog box (see Figure 13).

3) Define the name of the frame component in the **Component name** field, the page of the Component palette on which you want the frame to appear in the **Palette page** field, and, if you've created a custom 24 x 24 pixel, 16-color icon for the frame, click the **Change** button to select this .BMP file. Click **OK** when you're done.

Using a Frame from the Component Palette

To use a frame previously placed on the Component palette, select the page of the Component palette onto which you saved the frame, select the frame's icon, and drop it onto the form on which you want a descendant of that frame to appear. This process requires only two steps.

Adding a Frame to the Object Repository

By adding a frame to the Object Repository, you make it easy to copy it into a new project. Especially important is the ability to use the inheritance offered by the Object Repository to place an inherited frame into a new project, thereby maintaining the relationship between the frame and its ancestor. To add a frame to the Object Repository, follow these steps:

1) Save your frame to disk. In addition to saving this frame to Delphi's OBJREPOS directory or to a shared directory, you can also save it to the same one to which you save frames that you add to the Component palette. Saving the frame to a shared directory is especially nice if you are using a shared

object repository. This permits multiple developers to share frames.

2) Right-click the frame and select **Add To Repository**. Delphi responds by displaying the Add To Repository dialog box (see Figure 14).
3) Fill out the Add To Repository dialog box just as you would for any template you're adding to the Object Repository. Click **OK** when done.

Using a Frame from the Object Repository

To use a frame from the Object Repository, use the following steps:

1) Select **File | New**.
2) Select the page of the Object Repository to which you saved your frame template (see Figure 15).
3) Select the icon for the frame; then select the **Inherit** radio button.
4) Click **OK** to add an inherited version of the frame to your project.

If you select the **Copy** radio button instead of the **Inherit** radio button, the newly added frame will be a copy of the original frame. This is useful when you want to create a new frame, but don't want to maintain a relationship between it and the original.

Conclusion

Does it make a difference whether you place a frame you want to reuse on the Component palette or the Object Repository? The answer is a strong "Yes!" In most cases, you'll want to place frames you use frequently onto the Component palette. When you place a frame from the Component palette, you're always placing an instance of the frame class. You can then easily change the properties and event handlers of this instance as described earlier in this article. By comparison, placing a frame from the Object Repository creates a new class, not an instance. This new class is either a copy of the original or a descendant, depending on which radio button you select in the Object Repository dialog box. If you want to use a frame in a project, it makes a great deal of sense to place an instance, rather than define a new class for your frame. For this purpose, saving the frame to the Component palette is the best approach.

The one situation where you might want to use the Object Repository is when you're specifically creating hierarchies of frames, where each frame descendant introduces additional objects, methods, or event handlers. Here, the inheritance offered by the Object Repository makes it easier for you to create each new descendant. However, once you've defined the frame descendants you want to use regularly, I would again suggest that you add these to the Component palette to simplify their use. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\DEC\DI9912CJ.

Cary Jensen is president of Jensen Data Systems, Inc., a Houston-based database development company. He is co-author of 17 books, including *Oracle JDeveloper* [Oracle Press, 1998], *JBuilder Essentials* [Osborne/McGraw-Hill, 1998], and *Delphi in Depth* [Osborne/McGraw-Hill, 1996]. He is a Contributing Editor of *Delphi Informant*, and is an internationally respected trainer of Delphi and Java. For information about Jensen Data Systems consulting or training services, visit http://idt.net/~jdsi, or e-mail Cary at cjensen@compuserve.com.

UNDOCUMENTED

PIDLs / Shell IDs / Delphi 2-5 / 32-bit Windows

By Kevin J. Bluck and James Holderness

The Secret World of PIDLs

Working with Windows' Shell Item Identifiers

One of the most visible changes introduced with Windows 95 was the new shell interface. Love it or hate it, the shell fundamentally changed how applications integrate with the Windows operating system. Unfortunately, Microsoft has been parsimonious when distributing information about how to integrate your applications with the shell. Frankly, it's shocking how little information is available about shell programming, and almost all of what is available is in the form of unbelievably dry technical details with strong soporific qualities. Moreover, virtually no information is available for Delphi developers, because almost all Windows API documentation is for C/C++ programmers.

This article is a start at changing that sad state of affairs. We're sorry to say there will be no nifty components in this piece. Instead, we'll examine the unglamorous plumbing at the heart of shell programming. The center of everything, when programming the shell, is a little-understood construct known as the PIDL. That's right — PIDL.

The Shell Namespace

One of the core concepts of the new shell is the *namespace*, a fancy word for whatever system is used to navigate the computer's data. In DOS, the namespace consisted entirely of the file system. This was a hierarchical or tree-like namespace, with its root in the aptly named "root directory." Technically, there were several namespaces available on each computer, one for each logical disk drive. Even as late as version 3.1, the advent of Windows didn't significantly change this namespace convention. With the release of Windows 95, however, the namespace paradigm changed dramatically.

The namespace of Windows 95 (and that of its "descendants," such as Windows NT and Windows 98) is still hierarchical. However, it no longer corresponds directly to the file system. Instead, the file system is integrated into a larger namespace. This new namespace developed the generic concept of folders and items. A folder is analogous to the old DOS directory, its purpose being to contain other namespace elements, such as other folders and items. It doesn't contain any other useful data. An item is the exact opposite. Similar to a DOS file, it contains useful data, but no other namespace elements. It's important to draw distinctions between the file system elements and the new namespace elements. All file system directories are folders, but not all folders are directories. All files are items, but not all items are files.

This new namespace was rooted at the Desktop folder. The Desktop is the basic screen the user sees after starting Windows. Conceptually, everything in the system exists within the Desktop. This includes the My Computer folder, which contains the old DOS namespaces — the logical disk drives. The Desktop and My Computer folders obviously aren't part of the file system. Neither are special folders, such as Control Panel, Printers, Recycle Bin, and Network Neighborhood. These namespace folders are distinct from the file system.

A Rose by Any Other Name

For something to be useful, it must be identifiable. This truism naturally applies to folders and items. Every distinct folder and item must have a name that is unique within its namespace. This leads to an important concept: relative and absolute names. A *relative name* is unique only relative to a given parent. An *absolute name* is unique within the entire namespace. If you prefer, you may think of the absolute name as a name that is relative to the namespace root.

Under the old DOS file system, every element of the system was uniquely identified by its fully qualified path. The fully qualified path was the absolute name. A certain 8.3-style file name or single directory name might be repeated many times on a given disk drive, but the same name couldn't exist more than once inside a given directory. The 8.3-style file name alone is a relative name, unique only to the directory containing the file. Tracing and pre-pending the complete list of relative directory names from the root to the destination item always results in a completely unique absolute name for every element on the disk.

Most developers are familiar with the DOS-style notation for forming an absolute name in the namespace. A fully qualified path begins with the drive letter, delimited by a colon. The "root" is assumed, more or less equivalent with the drive itself. Each directory along the path to the ultimate goal is listed by its relative name. Each directory name is delimited by a single backslash, so that you can tell where one name ends and the next begins. The last entry after the last backslash is the relative name of the item being identified, whether it's a directory or a file. The entire string together constitutes the directory or file's absolute name.

This naming system worked adequately for the DOS file system. However, with the advent of Windows 95, not all folders and items could be described in terms of the file system. A new naming system was needed to uniquely identify all the elements of the namespace.

Redmond's answer was two new data structures. Each element's relative name was specified with an item identifier, defined as a SHITEMID record, and known in Delphi as *TShItemID*. These records could be concatenated as needed, conceptually the same operation as directory names being strung together with backslashes. Such a string of records was known as an item identifier list, or IDL, defined by the ITEMIDLIST record type, and known in Delphi as *TItemIDList*. Because IDLs are almost always manipulated by pointers, the entire structure became commonly known as a PIDL (pronounced and sometimes written as "piddle"). This pointer type is known in Delphi as *PItemIDList*. PIDLs, or pointers to item identifier lists, are the universal means of identifying elements within the namespace shell. All of these Delphi data types are defined in the standard VCL ShlObj unit.

The precise structure of these records often causes confusion. The most important thing to understand about the internal structure of PIDLs is that you're not supposed to understand most of it. DOS-style paths, as you know, are string types, and are, therefore, human-readable. PIDLs are binary types, and most of their content isn't intended for your direct manipulation. Adding to the confusion is the fact that *TShItemID* and *TItemIDList* are variable-length types. Their declarations are, therefore, less than clear.

The following code snippet shows the declaration of *TShItemID*:

```
TSHItemID = packed record
  cb: Word;  // Size of the record (including cb).
  abID: array[0..0] of Byte;  // Item ID data.
end;
```

First is a Word known as *cb*. Translated from that lovely Hungarian-notation brevity C programmers seem to prefer, *cb* means count of bytes. It's the total size, in bytes, of the entire *TShItemID* record. You need this information because the next data member can be of almost any size up to about 65,533 bytes. At first glance, the *abID* (array of bytes ID) data member seems pretty weird. An array with a single element of Byte? Why not just Byte?

Don't take this literally. The Windows programmers took advantage of the C language's ability to de-reference any desired index of an array, even if that index exceeds the original boundaries defined for the array. This isn't really an array of one byte. They just say that in the definition to provide a placeholder for later use. It's actually an array of (*cb* - SizeOf(*cb*)) bytes. Exactly what data goes in this array? Quite literally, anything. Unless you're writing a shell namespace extension, which is well beyond the scope of this article, you have no business poking around in that data. For most purposes, just resign yourself to not knowing what's in there, and simply accept that it exists.

The declaration of *TItemIDList* is, if possible, even more unintuitive:

```
TItemIDList = packed record
  mkid: TShItemID;
end;
```

It consists of a single data member, *mkid*, of type *TShItemID*. Again, why even bother? What's the point of declaring a record that only has one element? Again, the Windows designers are exploiting a feature of the C language. Just as the *abID* array above wasn't necessarily an array of just one byte, this isn't necessarily just one *TShItemID*. It's actually a list of *TShItemID* structures, one packed right after another. The end of the list is marked by a *TShItemID* record, which has the *cb* data member set to `0`. Obvious, right? Maybe a picture will help. Figure 1 shows a graphical representation of a *TItemIDList*, which contains three *TShItemID* records. Notice that the value of *cb* is always two greater than the number of bytes contained in *abID*, except for the end-of-list marker. This is because the value in *cb* also includes the size of the Word-sized *cb* data member itself, which just happens to be 2. Notice also how a fourth "dummy" *TShItemID* record is provided with the *cb* data member set to `0` to mark the end of the list.

Hopefully, the purpose of the *cb* data member is becoming clear. This is the only reliable guidepost you have for navigating a *TItemIDList*. The *PItemIDList* pointer type points to the first byte of the *TItemIDList*. Unless the *PItemIDList* is **nil**, you can expect there will be at least one *TShItemID* in the list. You can consider the *PItemIDList* pointer to also be of type *PShItemID*, pointing at the first *TShItemID* record in the list. The *cb* data member of that *TShItemID* tells you how far forward you need to increment your *PItemIDList* pointer to be pointing at the start of the next *TShItemID* in the list. If *cb* is `0`, then you know you're at the end of the list. Because you know nothing about the format of the data in *abID*, you would have no idea where to find the end of the *abID* data without *cb*.

Figure 2 shows a code sample that accepts a *PItemIDList* as a parameter and iterates through the entire *TItemIDList*, returning the total size of the list. This information would be useful in determining the bytes to allocate for a buffer to hold a copy of the list. Notice in particular how the local pointer to the item identifier is incremented with the value of the current item identifier's *cb* data member to make it point to the start of the next item identifier.

Just as there are relative and absolute file paths, there are also relative and absolute PIDLs. An absolute PIDL is one that is fully qualified with a string of *TShItemID* records that walk down all the way from the root of the namespace, the Desktop folder. A relative PIDL is usually taken to mean a PIDL with a single *TShItemID* contained in the list, naming a folder or item relative to its immediate parent folder. Obviously, this single ID is only useful if you know which is its parent folder.

Folders in the shell are represented by a COM interface called *IShellFolder*. We won't get into a lot of detail about *IShellFolder*, as it's a rather broad topic that deserves its own article. Suffice it to say that a given reference to *IShellFolder* represents a given shell folder. The various methods provided by *IShellFolder* nearly always deal with relative PIDLs, because the *IShellFolder* object itself serves to identify the PIDL's parent folder. The family of SH... shell API functions, however, doesn't typically include a reference to a specific *IShellFolder* object. Therefore, these standalone shell functions generally take and return absolute PIDLs. Consequently, it's important to keep track of what sort of PIDL your pointer is referencing, as a PIDL returned by an *IShellFolder* method isn't likely to be immediately useful in an SH... function.

PIDL Memory Allocation

One tricky aspect of PIDLs is that they're frequently allocated in one module and then freed in a module written by a different party. Very often, shell functions internally allocate and return a PIDL, which you are responsible for freeing later. This means that memory can be allocated by code written by one development tool, and freed by code written in a totally different tool.

This can be problematic, as different development environments often use different memory allocation schemes. For example, using Delphi's *FreeMem* procedure to free memory originally allocated by some C compiler's *malloc* RTL function would most likely end up corrupting the heap. As a result, the memory buffers to contain PIDLs must always be allocated and freed by the *shell task allocator*. You must never use other means available in Delphi, such as *GetMem* or *FreeMem*, or other Windows API calls such as *GlobalAlloc*. Using only the shell task allocator ensures that the PIDL's memory will always be allocated and freed using the same scheme, regardless of the development environment used for the module.

The shell task allocator is implemented via the *IMalloc* COM interface. *IMalloc* provides a fairly complete engine for memory allocation. *IMalloc* is defined in the ActiveX standard unit in Delphi 3 or later, or the OLE2 unit in Delphi 2 (see Figure 3). The simplest way to get a reference to an *IMalloc* interface is to use the *SHGetMalloc* API function. This documented function is defined in the standard ShlObj unit. Its declaration is also shown in Figure 3.

A basic example of using this allocation engine is shown in Figure 4. Notice that in Delphi 3 and later, it isn't necessary to explicitly release the COM interface. This is done for you automatically when the variable referencing the interface goes out of scope. However, in Delphi 2, you need to explicitly call the interface's *Release* method.

If you don't need the full services provided by *IMalloc*, but instead merely want to allocate or free buffers, there is a streamlined way to go about it. There are two simple undocumented functions, *SHAlloc* and *SHFree*, which can allocate and release memory using the shell allocator without the effort of obtaining an *IMalloc* interface. Like all undocumented functions discussed in this article, they're exported from SHELL32.DLL, and they're exported only by ordinal. *SHAlloc* uses ordinal 196, and *SHFree* is ordinal 195. Note that these functions are intended for generic buffers. When freeing a PIDL, it's preferable to use the *ILFree* function, exported by ordinal 155, even if it's not much more than a wrapper around *SHFree*. It also makes sure the PIDL isn't **nil** before calling *SHFree*. The declarations for these three functions are shown here:

```
function  SHAlloc(BufferSize: ULONG): Pointer; stdcall;
procedure SHFree(Buffer: Pointer); stdcall;
procedure ILFree(Buffer: PItemIDList); stdcall;
```

cb	abID	cb	abID	cb	abID	cb	abID
24	22 bytes	17	15 bytes	20	18 bytes	0	0 bytes

Figure 1: A graphic example of *TItemIDList*.

```
function GetPIDLSize(PIDL: PItemIDList): Integer;
var
  CurrentID: PShItemID;
begin
  // Check PIDL is not nil.
  if (PIDL <> nil) then
    begin
      // There will always be at least two bytes for the
      // terminating cb.
      Result := SizeOf(CurrentID.cb);
      // Initialize the local item id pointer and walk
      // through the list until the terminating cb = 0 is
      // encountered. Add the value of each cb along the
      // way to the result.
      CurrentID := PShItemID(PIDL);
      while (CurrentID.cb <> 0) do begin
        Inc(Result, CurrentID.cb);
        Inc(PChar(CurrentID), CurrentID.cb);
      end
    end
  else
    // If PIDL is nil, return 0 size.
    Result := 0;
end;
```

Figure 2: *TItemIDList* navigation.

```
IMalloc = interface(IUnknown)
  ['{ 00000002-0000-0000-C000-000000000046 }']
  function Alloc(cb: Longint): Pointer; stdcall;
  function Realloc(pv: Pointer; cb: Longint):
    Pointer; stdcall;
  procedure Free(pv: Pointer); stdcall;
  function GetSize(pv: Pointer): Longint; stdcall;
  function DidAlloc(pv: Pointer): Integer; stdcall;
  procedure HeapMinimize; stdcall;
end;

function SHGetMalloc(var ppMalloc: IMalloc):
  HResult; stdcall;
```

Figure 3: The *IMalloc* COM interface.

```
var
  Allocator: IMalloc;
  Buffer:    Pointer;
begin
  // Get an IMalloc interface.
  SHGetMalloc(Allocator);
  // Allocate 50-byte buffer.
  Buffer := Allocator.Alloc(50);
  // Expand buffer to 100 bytes.
  Buffer := Allocator.Realloc(Buffer,100);
  Allocator.Free(Buffer);  // Free buffer.
end;
```

Figure 4: Using the *IMalloc* COM interface.

From Paths to PIDLs and Back Again

Typically, PIDLs are returned to you by certain functions and passed around to other functions, without you ever really knowing anything about the internal structure of the IDL and without you ever needing to create a PIDL. Obviously, if you're implementing a namespace extension, you'll need to create item identifiers for the objects in your own virtual folders, but the format of those item identifiers is specific to your folder, and you can do whatever you want with them. But what happens if you need to create a PIDL for somebody else's namespace, such as a path in the file system?

The "documented" way would be to get an *IShellFolder* interface for the desktop, convert the path string in question into a *PWideChar* null-terminated UNICODE string, and then call the desktop *IShellFolder*'s *ParseDisplayName* method with the *PWideChar*. What a drag! If you're feeling lazy, you might find it easier to use one of these three functions:

```
function SHILCreateFromPath(Path: Pointer;
  PIDL: PItemIDList; var Attributes: ULONG):
  HResult; stdcall;

function ILCreateFromPath(Path: Pointer):
  PItemIDList; stdcall;

function  SHSimpleIDListFromPath(Path: Pointer):
  PItemIDList; stdcall;
```

SHILCreateFromPath is basically just a wrapper around the desktop folder's *ParseDisplayName*, and *ILCreateFromPath* is just a simplified wrapper around *SHILCreateFromPath*. However, *SHSimpleIDListFromPath* implements the whole process itself. The ordinal values are 28, 157, and 162, respectively.

We suspect that *SHSimpleIDListFromPath* is somewhat faster because it doesn't validate the path you provided to it. Both *SHILCreateFromPath* and *ILCreateFromPath* validate the path before conversion. If you pass an invalid path to either of these functions, you'll get a **nil** result. If you specify an unavailable drive, such as the A: floppy drive when there is no disk, you can get an error dialog box. If you specify an unavailable network resource, the function may block for quite a while until the network request times out and you finally get your **nil**.

SHSimpleIDListFromPath, on the other hand, doesn't validate the path, and so you can get a PIDL for any correctly formed path you can dream up without error. However, this function may not always generate completely correct PIDLs. We've noticed that when PIDLs created by *SHSimpleIDListFromPath* are passed to the *SHBrowseForFolder* function, occasionally the resulting display name and icon are incorrect. Accordingly, we recommend the use of *SHSimpleIDListFromPath* only when you're likely to pass invalid paths and don't want an error to result.

Rather, if you need to convert an absolute PIDL into a file system path, wonder of wonders, there is an actual documented function to do that fairly simply. It's the *SHGetPathFromIDList* function, shown here, available from the standard ShlObj unit (there are *AnsiChar* and *WideChar* variants):

```
function SHGetPathFromIDList(PIDL: PItemIDList;
  Path: PAnsiChar): BOOL; stdcall;

function SHGetPathFromIDListW(PIDL: PItemIDList;
  Path: PWideChar): BOOL; stdcall;
```

Notice that the *Path* parameter takes a pointer to a null-terminated string. You must provide a pointer to a buffer capable of accepting MAX_PATH chars plus a null terminator, or risk an access violation, because there is no parameter for specifying the size of your buffer.

Display Name

If you need to determine the display name of a PIDL, the documented method is to use the method *IShellFolder::GetDisplayNameOf*. After you figure out which of the three possible ways the string was returned to you is the correct one, you'll have the "user-friendly" name for the shell object the PIDL represents.

It's also possible to get the display name from the documented *SHGetFileInfo* API function, if you don't mind figuring out which flags to use and fishing around in the returned record for the data you want. Or, if you're in a hurry, you can use the *ILGetDisplayName* function. *ILGetDisplayName* basically calls the desktop's *IShellFolder::GetDisplayNameOf* function with the flag SHGDN_FORPARSING. The ordinal value for *ILGetDisplayName* is 15. As you can see from the code snippet below, this function doesn't return the "normal" short display name for file system objects, but rather the fully qualified path. If you want the short display name, you will be better off using *SHGetFileInfo*:

```
function ILGetDisplayName(PIDL: PItemIDList;
  Name: Pointer): LongBool; stdcall;
```

Windows NT and *PWideChar*

You may have noticed that the string-type parameters for the undocumented functions previously shown are declared as type *Pointer* instead of *PChar*. This is because of a small "catch" that is common for undocumented functions. All of these string-type parameters take *PAnsiChar* on Windows 95, and *PWideChar* on Windows NT.

There's no choice of ANSI or UNICODE as would be expected for a documented function. Windows 95 uses the ANSI version only, and Windows NT uses the UNICODE version only. Take it or leave it. If you want your applications to function correctly on both platforms, you're going to have to check what operating system is in use at run time. The *Win32Platform* global variable provided in the SysUtils unit is handy for this. If your application is running on NT, you'll need to convert any string parameters to *PWideChar* before calling the function. When the function returns, you'll also obviously need to convert any returned strings back to *PAnsiChar*. It may be annoying, but that's the price you pay for using undocumented functions.

Equality and Parenthood

If you need to determine if two PIDLs are equal, the approved method is to use the *IShellFolder::CompareIDs* method. Relative PIDLs can be compared using their common parent's *IShellFolder*, and absolute PIDLs must be compared using the Desktop folder's *IShellFolder*. This method also allows you to determine which PIDL should come first in a sorted list if they aren't equal. However, as usual, there are undocumented shortcut methods. To test whether two PIDLs are equal, you may use the *ILIsEqual* function. If you need to determine whether a particular PIDL is a child of another PIDL, you would call the *ILIsParent* function passing it the suspected parent and child. If you require that the child be an immediate descendant (e.g. it is a child of the parent folder itself, not one of the parent folder's subfolders), then you would set the *ImmediateParent* parameter of the function to True. The following code shows the function declarations:

```
function ILIsEqual(PIDL1: PItemIDList; PIDL2: PItemIDList):
  LongBool; stdcall;
function  ILIsParent(PIDL1: PItemIDList;
  PIDL2: PItemIDList; ImmediateParent: LongBool):
  LongBool; stdcall;
```

The ordinals for these functions are 21 and 23, respectively. Note that the equality of two ID lists can't necessarily be determined with a binary comparison if you're tempted to try that. Equivalent PIDLs can conceivably have different internal binary structures. Both of the functions previously shown use the Desktop folder's *IShellFolder::CompareIDs* method to perform equality tests. The *ILIsParent* function, of course, only tests whether the base PIDL of the child is equal to the parent PIDL.

Parsing a PIDL

Sometimes you have a need to parse a PIDL, identifying individual IDs contained in the list. There seem to be no documented functions for these tasks. Apparently, Microsoft expects you to implement functions to slice and dice PIDLs yourself. Luckily, we've got the inside scoop for you.

If you need to determine the total size in bytes of all the identifiers in a PIDL, you can use the *ILGetSize* function. If you need to iterate forward through each item identifier in a PIDL, you'll probably find *ILGetNext* very useful. When given a PIDL (or a pointer to any ID in the list, for that matter), the function will return a pointer to the next item identifier in the list. If the PIDL is **nil** or is already pointing to the last item in the list, the function will return **nil**. For the specific case of finding the last item identifier in the list, you can just call *ILFindLastID*.

An even more specific form of search is the *ILFindChild* function. Given a parent PIDL and a child PIDL, it will return a pointer to the unique portion of the child. For example, if you passed it PIDLs for the folder 'C:\DIR' as the parent and the item C:\DIR\FILE.TXT as the child, it would return a pointer to that portion of the child PIDL that represents FILE.TXT. If the given child isn't a child of the parent, the function will return **nil**. The ordinals for these functions are 152, 153, 16, and 24, respectively (see Figure 5).

Copying and Combining

Something even more useful when dealing with PIDLs is the ability to make a copy of a PIDL passed to you by the shell. When passed an existing PIDL, the *ILClone* function will allocate and return a new identical copy of that PIDL. The *ILCloneFirst* function, on the other hand, will return a new PIDL containing only the first item identifier from the source PIDL. If you need a copy of the last item identifier, you could use a combination of *ILFindLastID* and *ILCloneFirst*. For other portions of the IDL, you would have to use *ILGetNext* and *ILCloneFirst*. The ordinals for these two functions are 18 and 19, respectively, and the definitions are shown here:

```
function ILClone(PIDL: PItemIDList): PItemIDList; stdcall;

function ILCloneFirst(PIDL: PItemIDList):
  PItemIDList; stdcall;
```

If you want to combine two PIDLs, you would use the *ILCombine* function. Given two PIDLs, it will create a new PIDL containing the two source lists joined consecutively. If you want to combine a single item identifier with a PIDL, you would use the *ILAppendID* function. It can be used to append a *TItemID* record to the beginning or end of an existing IDL. However, unlike *ILCombine*, the original PIDL is destroyed by this operation. The *ILAppendID* function can also be used to create a PIDL from an item identifier alone by passing a **nil** for the PIDL. The ordinals for these functions are 25 and 154, respectively. The function declarations are shown here:

```
function ILCombine(PIDL1: PItemIDList; PIDL2: PItemIDList):
  PItemIDList; stdcall;

function  ILAppendID(PIDL: PItemIDList; ItemID: PShItemID;
  AddToEnd: LongBool): PItemIDList; stdcall;
```

Global Memory Cloning

As we mentioned earlier, the memory for an IDL should almost always be allocated using the shell's memory allocator. However, there are two functions that use a different method of allocating and freeing memory, *ILGlobalClone* and *ILGlobalFree* (ordinals 20 and 156). The function declarations are:

```
function ILGlobalClone(PIDL: PItemIDList):
  PItemIDList; stdcall;

procedure ILGlobalFree(PIDL: PItemIDList); stdcall;
```

On Windows NT, these global functions just use the default process heap (as returned by *GetProcessHeap*). This led us to believe that heap allocations were in some way more efficient than the shell allocator, and that global functions were only used internally by the shell for reasons of efficiency.

However, on Windows 95 most of the internal structures in the shell need to be shared between all instances of the DLL. In the case of PIDLs, the memory used when allocating them obviously has to be shareable as well. *ILGlobalClone* solves this problem by using an undocumented shared heap for the allocations, conveniently making the pointers accessible from anywhere. In general, this is a specialized technique only applicable to "magic" programming within the Explorer itself. We're sure Microsoft would frown on those who dare to use this facility for their own purposes.

Truncation

If you need to delete an entire PIDL, just use the *ILFree* function. However, if you only need to remove the last item identifier from the end of a list, you can use the *ILRemoveLastID* function:

```
function ILRemoveLastID(PIDL: PItemIDList):
  LongBool; stdcall;
```

```
function ILGetSize(PIDL: PItemIDList): UINT; stdcall;

function ILGetNext(PIDL: PItemIDList):
  PItemIDList; stdcall;

function ILFindLastID(PIDL: PItemIDList):
  PItemIDList; stdcall;

function ILFindChild(ParentPIDL: PItemIDList;
  ChildPIDL: PItemIDList): PItemIDList; stdcall;
```

Figure 5: PIDL parsing functions.

The ordinal value is 17. The return value is True if the operation was successful. However, note that it doesn't actually free any memory; it just resets the end of list marker. Unfortunately, this is the only deletion function that exists. If you want to remove an item identifier from the beginning of an IDL, the best you can do is use a combination of *ILGetNext* and *ILClone* to make a copy of the original list starting with the second ID in the list, and then delete the original PIDL with *ILFree*. Trying to delete IDs from the middle of the list would be even more complicated, but we can't imagine that such needs are very common.

Conclusion

PIDLs aren't an exciting topic by any stretch of the imagination, but a good understanding of their nature is essential to virtually any task in programming with the Windows 95 shell. It's unfortunate that Microsoft has done such a poor job of publicizing the information, which developers need to properly integrate their applications with the shell. We hope this article has provided the Delphi developer community with the basic foundation necessary to undertake any project that will interface with the shell. The mechanics of manipulating PIDLs may be dull, but we're sure your users won't think the well-integrated applications you can produce using this knowledge are boring. Get ready to write something amazing! ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\DEC\DI9912KB.

Kevin Bluck is an independent contractor specializing in Delphi development. He lives in Sacramento, CA with his lovely wife Natasha. He spends his spare time chasing weather balloons and rockets as a member of JP Aerospace (http://www.jpaerospace.com), a group striving to be the first amateur organization to send a rocket into space. Kevin can be reached via e-mail at kbluck@ix.netcom.com.

James Holderness is a software developer specializing in C/C++ Windows applications. He also runs a Web site on undocumented functions in Windows 95 at http://www.geocities.com/SiliconValley/4942. He is currently working for FerretSoft LLC, where he helps create the Ferret line of Internet search tools at http://www.ferretsoft.com. James can be reached via e-mail at james@ferretsoft.com or jholderness@geocities.com.

By Alan C. Moore, Ph.D.

A Multimedia Assembly Line

Part I: Enhancing a Delphi Expert

In early 1997, I wrote a Delphi Wizard (expert) to produce multimedia components with WAV-file-playing capabilities. That original expert was limited in a number of respects. Here we'll extend that expert, adding greater flexibility and functionality. We'll also provide a model for other complex experts that generate component code.

In this first installment of a two-part series, we'll begin with an overview of the expert and the sound-playing capabilities it supports. Then we'll see how that functionality is implemented in the expert and the components it produces. We'll discuss the expert interface and the code to implement the user interface. Next month, we'll explore the code-generating engine.

Basic Functionality of the Expert

Although I've made some cosmetic changes to the first page of the expert, it is functionally identical to the first version (see Figure 1). The sound-playing functionality of the generated components in the earlier version was implemented through the component's two properties: the *TSoundPlayOptions* enumerated type, which indicated how the sound would be played; and the *TSoundFile* string, which indicated the .WAV file to be played (which could be the default if none were selected).

TSoundPlayOptions provided five options for sound component creators/users (see Figure 2). All of these options are realized using various Windows API flags with the *sndPlaySound* function. A flag not shown in the figure, *snd_NoDefault*, is used with each sound-playing option, so if a particular .WAV file isn't present, the default sound won't be played. In this new version, we'll add a Boolean property, *PlayDefault*, which will determine if the default sound is played.

These flags are used in a **case** statement within the *OnClick* method (in the old version), or the particular event method (in the new version). In the earlier version, I used the older Windows 3.x function, *sndPlaySound*. It's declared in the mmsystem.pas file:

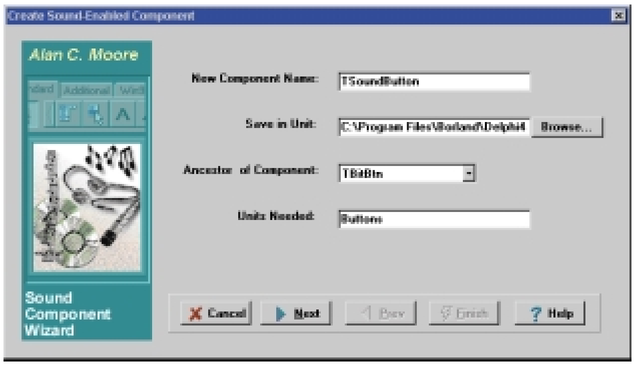
```
function sndPlaySound(lpszSoundName: PChar;
  uFlags: UINT): BOOL; stdcall;
```

where *lpszSoundName* is the name of a .WAV file and *uFlags* are the Windows API constants needed to produce the desired result. In addition to supporting this function for the Delphi 1 version of this expert, we'll also use the new *PlaySound* function in the 32-bit version:

```
function PlaySound(pszSound: PChar; hmod:
  HMODULE;fdwSound: DWORD): BOOL; stdcall;
```

The additional parameter, *hmod*, is a resource file containing sound resource(s). These two functions are at the core of this expert and the components it creates. As in the previous version, this expert allows you to easily add sound-playing properties and functionality to any component, or a series of components. The previous version was a standard expert. Here we'll use an add-in expert for the 32-bit version, retaining the standard version for the Delphi 1 version.

First, we'll make it possible to sound-enable events in addition to *OnClick* (the only one we could



New Component Name: TSoundButton
Save in Unit: C:\Program Files\Borland\Delphi4
Ancestor of Component: TBitBtn
Units Needed: Buttons

Figure 1: The first page of the expert.

Flag	Description
snd_Sync	Sound played synchronously; function doesn't return until after the sound is finished playing.
snd_Async	Sound played asynchronously; function returns immediately and doesn't wait until the sound is finished playing.
snd_Memory	Plays a sound file already in memory.
snd_Loop	Used with *snd_Async*, plays sound continuously until a call to *sndPlaySound* with NIL filename.
snd_NoStop	Don't interrupt a sound currently playing to play a new sound.

Figure 2: Flags used with the *sndPlaySound* function.

enable in the older version). Then we'll add functionality to allow you to choose whether the default sound should play when no file is chosen; we'll also add the choice of whether a new sound will terminate a sound already playing. Finally, we'll expand the set of *SoundPlayOptions* to include not playing a sound at all. In sum, we'll demonstrate how to build a fairly complex expert with added features and options. Let's begin by examining the new events supported.

More Events and Options

Which events to enable? I decided to include all those derived from *TWinControl* (*OnEnter*, *OnExit*, *OnKeyDown*, *OnKeyPress*, and *OnKeyUp),* and most of those derived from *TControl* through Delphi 3 (*OnClick*, *OnDblClick*, *OnDragDrop*, *OnDragOver*, *OnEndDrag*, *OnMouseDown*, *OnMouseMove*, *OnMouseUp*, and *OnStartDrag* [in the 32-bit version]). We'll see how this is implemented when we examine the user interface.

So what options are we to add? A new property, *SoundSource*, indicates the type of sound source to which the *FileName* parameter points. It's of the enumerated type, *TSoundSource* (*ssAlias*, *ssAlias_ID*, *ssFilename*, *ssMemory*, *ssResource*, *ssNone*). The 16-bit version uses a subset of this type, which works with *sndPlaySound*: *ssFilename*, *ssMemory*. Another new Boolean property, *Yield*, determines if the requested sound will interrupt a sound that is already playing. This property is connected with the *snd_NoStop* flag. All of the flags associated with the *PlaySound* function (introduced in Windows 95) are shown in Figure 3.

First, we'll examine some of the main variables and structures used to communicate between the various units. Then we'll explore the process of creating the new expert.

Basic Types and Data Structures

To implement this functionality, I needed to add several new types and data structures to the expert. These are defined in SndTypes.pas, shown in Listing One (beginning on page 21). We've already examined the new enumerated type, *TSoundSource*. As explained already, you can now sound-enable any of the standard events for each new component you generate. Each time you click on one of the event check boxes, the expert needs to know which one it is so that it can store the data selected in the proper place. We use the *EventSelected* enumerated type to accomplish this:

```
EventSelected = (esClick, esDragDrop, esDragOver,
  esEndDrag, esEnter, esExit, esKeyDown, esKeyPress,
  esKeyUp, esMouseDown, esMouseMove, esMouseUp,
  esStartDrag);
```

Flag	Description
snd_Application	Sound played using an application-specific association.
snd_Alias	*pszSound* parameter is a system-event alias in the registry or WIN.INI.
snd_Alias_Id	*pszSound* parameter is a predefined sound identifier.
snd_Async	Sound is played asynchronously with *PlaySound* returning immediately.
snd_FileName	*pszSound* parameter is a filename.
snd_Loop	Sound is played repeatedly until *PlaySound* is called with *pszSound* at nil.
snd_Memory	Sound event's file is loaded in RAM, memory pointed to by *pszSound*.
snd_NoDefault	If the sound cannot be found, don't use default sound (returns silently).
snd_NoStop	Sound event will yield to (not stop) a sound event that is already playing.
snd_NoWait	If the sound driver is busy, return immediately without playing the sound.
snd_Purge	Sounds for the calling task are stopped.
snd_Resource	*pszSound* parameter is a resource identifier; *hmod* identifies its instance.
snd_Sync	Sound is played synchronously; *PlaySound* returns after sound completes.

Figure 3: Flags used in the *PlaySound* function.

The expert also needs to keep track of the status of each check box and the possible sound event class it may eventually produce. We use another enumerated type to accomplish this:

```
EventCheckBoxStatus =
  (ecbsUndefined, ecbsDefined, ecbsChecked);
```

The first possibility, *ecbsUndefined*, indicates the check box has never been checked. The second, *ecbsDefined*, indicates that data has been entered for the particular check box. Then, if you enter data, uncheck the box, and then decide to check it again, the original data entered in the popup dialog box will appear and can be edited. The final possibility, *ecbsChecked*, simply keeps track of the checked status of the check box and is used in the component-producing engine, which we'll be discussing toward the end of this article. A value for each event/check box is stored in the following array:

```
EventCheckBoxArray:
  array [0..EventMax] of EventCheckBoxStatus;
```

Because event names are used in constructing the new classes we'll be discussing below, we need to keep track of them in a constant array whose members correspond to the event/check boxes:

```
EventNames: array [0..MaxEvents] of string =
  ('Click', 'DragDrop', 'DragOver', 'EndDrag', 'Enter',
   'Exit', 'KeyDown', 'KeyPress', 'KeyUp', 'MouseDown',
   'MouseMove', 'MouseUp', 'StartDrag');
```

The information from the dialog box applying to each sound event (see Figure 4) is stored in the following structure:

```
SoundFactors = packed record
  SoundSource : TSoundSource;
  Yield : Boolean;
  WavFileDefault : Boolean;
  DefaultWavFile : TFileName;
  SoundPlayOptions : TSoundPlayOptions;
end;
```

Because there are 13 possible sound events (12 in Delphi 1), the data is stored in a public array of the expert class:

```
SoundFactorsArray: array [0..MaxEvents] of SoundFactors;
```

Now that you've been introduced to the main variables and data structures used in this expert, you'll be able to study the code with greater comprehension. Now we need to look at the steps involved in writing the code. Consider three basic steps in expert creation:

1) Creating the Delphi expert interface
2) Creating the user interface
3) Writing the expert engine

We'll have quite a bit to say about each of these major steps. We'll begin by discussing the Delphi expert interface. In this case, we'll create a library project that will work in either 16-bit or 32-bit Delphi environments.

Updating the Delphi Expert Interface

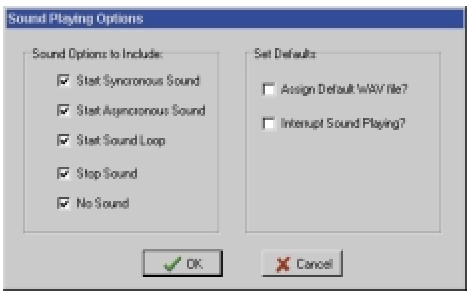
Every Delphi expert must have an interface so Delphi can load it, execute it, and unload it. This interface varies, depending on the type of expert you're creating. There were three types of experts in Delphi 1 with a new type, the add-in expert, starting with Delphi 2. Add-in experts are similar in many ways to standard ones, but are installable anywhere in Delphi's menu structure. As mentioned already, we've added this new type of expert for all of the 32-bit versions. To support both 16- and 32-bit versions, we've had to include many conditional compilation statements in the project file shown in Listing Two (beginning on page 22) and unit files.

Certain methods are required in each of the three original expert types: *Execute*, *GetIDString*, *GetName*, and *GetStyle*. Standard and add-in experts require additional methods. Standard experts include two unique methods, *GetMenuText* and *GetState*. Add-in experts use all the basic methods except *Execute* (we keep it for convenience here, however), and add these methods:

```
function GetAuthor: string; override;
procedure RunSoundExpert(Sender: TIMenuItemIntf);
constructor Create;
destructor Destroy; override;
```

RunSoundExpert is triggered when our menu option is clicked and calls the *Execute* method. You immediately see the dialog box we saw in Figure 1. How do we add our menu item? The *Create* method is where we set up our new menu item by using tool services to access Delphi's menu structure. Because we create a new menu item here (*NewMenuItem*), we must also free it in the *Destroy* method.

To use tool services, you must first initialize them. You accomplish this with the *InitExpert* function, which isn't part of our expert class definition. It has a slightly different format in each Delphi version (again, see Listing Two). The functions in the 32-bit versions end with **stdcall** and **export**; in Delphi 1 they end with only **export**. The function definitions are indicated in the following conditional define:


Sound Playing Options

Sound Options to Include:
- [✓] Start Syncronous Sound
- [✓] Start Asyncronous Sound
- [✓] Start Sound Loop
- [✓] Stop Sound
- [✓] No Sound

Set Defaults:
- [ ] Assign Default WAV file?
- [ ] Interrupt Sound Playing?

Figure 4: The Sound Playing Options dialog box.

```
{ $IFDEF WIN32 }
function InitExpert(ToolServices: TIToolServices;
  RegisterProc: TExpertRegisterProc; var Terminate:
  TExpertTerminateProc): Boolean; stdcall; export;
{ $ELSE }
function InitExpert(ToolServices: TIToolServices;
  RegisterProc: TExpertRegisterProc; var Terminate:
  TExpertTerminateProc): Boolean; export;
{ $ENDIF }
```

This function is also used to register the expert with Delphi using the *RegisterProc* method. There are many differences between the 16-bit and 32-bit versions of this expert. These are clearly indicated with conditional statements.

If you need to free any memory or perform other cleanup (which we didn't in this instance), you'll need a *DoneExpert* procedure, which is included for your reference. For Delphi 1 only, a *FaultHandler* function must also be included.

Now that we've fully examined the expert interface, let's look at the user interface implemented in SndExp1.pas (not shown here due to space constraints). The entire project is available on disk or for download; see end of article for details. Note that global variables used in this and other units are declared in SndTypes.pas (again, see Listing One).

Creating a User Interface

Stage two, creating the expert's user interface, is a more familiar task for most of us. As in the previous version, the interface is a modal dialog box with two pages. The first page is similar to that of the first version; the second page is completely new. In both pages, the choices you make drive the expert's engine, which generates the code for the sound component in the third stage. Let's discuss the details of that interface.

The main form consists of a *TNotebook* at the top and a *TPanel* at the bottom. The *TNotebook* has two pages and the *TPanel* contains five *TBitBtns*: *CancelBtn*, *PrevBtn*, *HelpBtn*, *NextBtn*, and *FinishBtn*. *PrevBtn* moves to page one of the notebook, *NextBtn* moves to page two, and *FinishBtn* writes and displays the new sound component. The form contains two standard dialog boxes: SaveDialog and OpenDialog. The SaveDialog dialog box is used to select a file in which to save the final results (sound component); OpenDialog allows us to select a default .WAV file. So far, all of this applies to both versions.

The first notebook page (seen in Figure 1) contained the same controls in each version: four *TLabels*, three *TEdits*, and a *TComboBox* (third control). The controls on this page allow the user to set the name for the new component, the unit in which to save it, its ancestor, and the additional units to include in the **uses** clause (usually the ancestor's unit at a minimum).

The second *TEdit* and the *TCombo* behave in unusual ways. By writing an *OnClick* (and *OnEnter*) method, clicking on or moving to the second *TEdit* automatically brings up *TSaveDialog* so we can choose a file name and directory for our new component. The *PopulateComboBox* method automatically fills the check box with the names of all components installed in the VCL. It uses some of Delphi's tool services to accomplish this. Let's see how the magic works.

Method	Use of Method
GetComponentCount(X)	Where X is the index number of a module, this method returns the number of components registered in that module.
GetComponentName(X,Y)	Where X is the index number of a module and Y is the index number of a component registered in that module, this method returns the name of that component.
GetModuleCount	Returns the number of registration modules (not component units) in the VCL (Complib.dcl).
GetModuleName(X)	Where X is the index number of a module, this method returns the name of that module.

Figure 5: The tool services methods used in the expert to get information on installed components.

Using Tool Services to Access Installed Components

Delphi's tool services, defined in the ToolIntf.pas unit, can be very useful in writing experts. They provide background information and access to Delphi's inner workings. For example, we can find out the number of forms or units used in a project with *GetFormCount* and *GetUnitCount*, respectively. In addition to retrieving information, we can perform useful actions like opening a project (OpenProject), opening a file (OpenFile), saving a project (SaveProject), or saving a file (SaveFile), all from within an expert we write.

There are four special methods in the tool services related specifically to components installed in the VCL (Complib.dcl). The methods used in this expert are shown in Figure 5.

In both versions of this expert, I used all but the last method. Take a look at the *TSoundCompForm.PopulateComboBox* method in SndExp1.pas to see how they're used. We used two loops within a **try..finally** block. The outer loop iterated through all the installed modules:

```
for CurrentMod := 0 to NumMods 1 do begin
```

and the inner loop iterated through the components registered in each module if there were any:

```
if (NumComps > 0) then
  for CurrentComp := 0 to NumComps 1 do begin
```

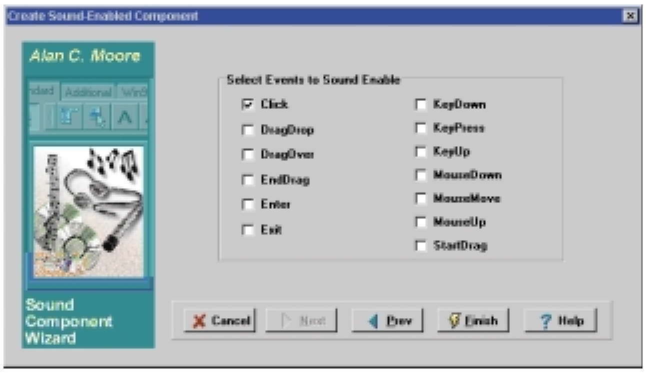
We'll have a bit more to say about the tool services when we reach the end of this section and discuss the **Finish** button. First, let's take a look at the second page.

On the second page of the expert, you can check the events you want to implement in the new component (see Figure 6). Whenever a new event is checked, another dialog box prompts for the parameters for that sound event (refer to Figure 4). This dialog box always appears with either the default values checked (in this case none), or the previously selected values (if the event had been selected previously).

You must choose at least one event before the **Finish** button becomes enabled. When you click on **Finish** the expert closes, and the generated code comes up in the Delphi IDE editor. This is accomplished with another ToolServices function:

```
iTools.OpenFile(UnitName);
```

where *iTools* identifies a local copy of tool services.



Select Events to Sound Enable

☑ Click
☐ DragDrop
☐ DragOver
☐ EndDrag
☐ Enter
☐ Exit
☐ KeyDown
☐ KeyPress
☐ KeyUp
☐ MouseDown
☐ MouseMove
☐ MouseUp
☐ StartDrag

Figure 6: On the second page of the expert, choose the events you want to implement.

Conclusion

We'll have to leave it there for now. We've discussed most of the basic issues involved with the expert interface and user interface. Next month we'll discuss the expert's component-creating engine. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\DEC\DI9912AM.

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

Begin Listing One — SndTypes.pas

```
unit Sndtypes;  { Global Types for SndExprt Project. }

interface

uses SysUtils;

type
  String80 = string[80];
  String30 = string[30];
  String15 = string[15];
  String2  = string[2];
  TSoundPlayOption = (spPlayReturn, spPlayNoReturn,
    spPlayContinuous, spEndSoundPlay, spPlayNoSound);

  TSoundPlayOptions = set of TSoundPlayOption;

  SoundFactors = packed record
    Yield : Boolean;
    WavFileDefault : Boolean;
    DefaultWavFile : TFileName;
    SoundPlayOptions : TSoundPlayOptions;
  end;

  TSoundSource =
    { $IFDEF VER80 }
      (ssFilename, ssMemory);
    { $ELSE }
      (ssAlias,ssAliasID,ssFilename,ssMemory,ssResouce);
    { $ENDIF }

  { Current event selected in the Expert. }
  EventSelected = (esClick, esDragDrop, esDragOver,
    esEndDrag, esEnter, esExit, esKeyDown, esKeyPress,
    esKeyUp, esMouseDown, esMouseMove, esMouseUp
    { $IFNDEF VER80 }
      , esStartDrag
    { $ENDIF }
    );

  EventCheckBoxStatus =
    (ecbsUndefined, ecbsDefined, ecbsChecked);

const
  { Number of events user can sound-enable. 0-based;
    actual number of events is one more. }
  { $IFNDEF VER80 }
    EventMax = 12;
  { $ELSE }
    EventMax = 11;
  { $ENDIF }

  BasicEventNames: array [0..EventMax] of String15 =
    ('Click', 'DragDrop', 'DragOver', 'EndDrag', 'Enter',
     'Exit', 'KeyDown', 'KeyPress', 'KeyUp', 'MouseDown',
     'MouseMove', 'MouseUp'
    { $IFNDEF VER80 }
      , 'StartDrag'
    { $ENDIF }
    );

  EventNames: array [0..EventMax] of String15 =
    ('Click', 'DragDrop', 'DragOver', 'DoEndDrag',
     'DoEnter', 'DoExit', 'KeyDown', 'KeyPress', 'KeyUp',
     'MouseDown', 'MouseMove', 'MouseUp'
    { $IFNDEF VER80 }
      , 'DoStartDrag'
    { $ENDIF }
    );

  ParametersArray: array [0..8] of String80 =
    ('',                        { For Click, DoEnter, DoExit  }
     'Source: TObject; X, Y: Integer',      { For DragDrop    }
     'Source: TObject; X, Y: Integer; State: TDragState;' +
     ' var Accept: Boolean',               { For DragOver    }
     'Target: TObject; X, Y: Integer',      { For EndDrag     }
     'var Key: Word; Shift: TShiftState', { KeyDown,KeyUp     }
     'var Key: Char',                       { For KeyPress    }
     'Button: TMouseButton; Shift: TShiftState; ' +
     'X, Y: Integer',           { For MouseDown and MouseUp   }
     'Shift: TShiftState; X, Y: Integer', { For MouseMove     }
     'var DragObject: TDragObject');       { For StartDrag    }

  ParamsArray: array [0..8] of String80 =
    ('',                       { For Click, DoEnter, DoExit   }
     '(Source, X, Y)',                      { For DragDrop    }
     '(Source, X, Y, State, Accept)',      { For DragOver     }
     '(Target, X, Y)',                       { For EndDrag    }
     '(Key, Shift)',               { For KeyDown and KeyUp    }
     '(Key)',                                 { For KeyPress }
     '(Button, Shift, X, Y)', { For MouseDown and MouseUp    }
     '(Shift, X, Y)',                        { For MouseMove  }
     '(DragObject)');  { For StartDrag in 32-bit versions     }

  EventPrefixes: array [0..EventMax] of String2 =
    ('ck', 'dd', 'do', 'ed', 'en', 'ex', 'kd', 'kp', 'ku',
     'md', 'mm', 'mu'
    { $IFNDEF VER80 }
      , 'sd'
    { $ENDIF }
    );

  fdwSoundFlags: array [0..3] of String15 =
    ('SND_ALIAS', 'SND_FILENAME', 'SND_RESOURCE', 'NIL');

var
  { Storage for properties. }
  SoundFactorsArray: array [0..EventMax] of SoundFactors;
  { Used for formatting new types in CompGen1. }
  SoundOptions: array[0..EventMax] of Integer;
  { Holds the string value for the 3rd PlaySound
    parameter from the fdwSoundFlags array. }
  fdwSoundValue: String15;

  AnEventSelected: EventSelected;
  EventIsEnabled: array [0..12] of Boolean;
  EventCheckBoxArray: array [0..12] of EventCheckBoxStatus;

implementation

end.
```

End Listing One

Begin Listing Two — SndExprt.dpr

```
{ TSoundExpert, Sound Component Delphi Expert
  (c) 1996, 1999 Alan C. Moore. }
library SNDEXPRT;

uses
  { $IFNDEF VER80 }
    ShareMem,
  { $ENDIF }
  Forms,
  WinTypes,
  ExptIntf,
  ToolIntf,
  VirtIntf,
  SysUtils,
  WinProcs,
  Sndtypes in 'SNDTYPES.PAS',
  OptDlg in 'OptDlg.pas' { SoundPlayingOptions },
  compgen in 'compgen.pas',
  SndExp1 in 'sndexp1.pas' { SoundCompForm };

type  { Expert Class for a Standard Expert. }
  TSoundExpert = class( TIExpert )
    function GetName: string; override;
    function GetComment: string; override;
    function GetStyle: TExpertStyle; override;
    function GetState: TExpertState; override;
    function GetMenuText: string; override;
    function GetIDString: string; override;
    { $ifndef Ver80 }
    function GetAuthor: string; override;
    procedure RunSoundExpert(Sender: TIMenuItemIntf);
    constructor Create;
    destructor Destroy; override;
    { $endif }
    procedure Execute; override;
  private
    { $IFNDEF VER80 }
    NewMenuItem: TIMenuItemIntf;
    { $ENDIF }
  end;

{ Expert Methods }
{ $ifndef Ver80 }
constructor TSoundExpert.Create;
var
  MainMenu : TIMainMenuIntf;
  MainMenuItems, ComponentMenu : TIMenuItemIntf;
begin
  inherited Create;
  MainMenu := nil;
  MainMenuItems := nil;
  ComponentMenu := nil;
  NewMenuItem := nil;
  try
    try
      MainMenu := ToolServices.GetMainMenu;
      MainMenuItems := MainMenu.GetMenuItems;
      ComponentMenu := MainMenuItems.GetItem(6);
      NewMenuItem := ComponentMenu.InsertItem(0, 'S&ound',
        'TSoundComponentExpert', '', 0, 0, 0,
        [mfVisible, mfEnabled], RunSoundExpert);
    finally
      ComponentMenu.Free;
      MainMenuItems.Free;
      MainMenu.Free;
    end;
  except
    ToolServices.RaiseException(ReleaseException);
  end;
end;
{ $ENDIF }

{ Exceptions must be handled this way. }
procedure HandleException;
begin
  ToolServices.RaiseException(ReleaseException);
end;

{ $ifndef Ver80 }
procedure TSoundExpert.RunSoundExpert;
begin
  try
    ExecuteExpert(ToolServices);
  except
    ToolServices.RaiseException(ReleaseException);
  end;
end;

destructor TSoundExpert.Destroy;
begin
  NewMenuItem.Free;
  inherited Destroy;
end;

function TSoundExpert.GetAuthor: string;
begin
  try
    Result := 'Alan C. Moore';
  except
    HandleException;
  end;
end;
{ $endif }

function TSoundExpert.GetName: string;
begin
  try  { try/except blocks needed for each expert method. }
    Result := 'Sound Component Delphi Expert';
  except
    HandleException;
  end;
end;

function TSoundExpert.GetComment: string;
begin
  try
    Result := 'Delphi Expert to add Sound to a Component';
  except
    HandleException;
  end;
end;

function TSoundExpert.GetStyle: TExpertStyle;
begin  { All experts have this method. }
  try
    { $ifndef Ver80 }
      Result := esAddIn;
    { $else }
      Result := esStandard;
    { $endif }
  except
    HandleException;
  end;
end;

function TSoundExpert.GetState: TExpertState;
begin  { Possible states of menu item. }
  try
    Result := [esEnabled];
  except
    HandleException;
  end;
end;

function TSoundExpert.GetMenuText: string;
begin
  try
    Result := 'Sound Component';  { Help menu Text. }
  except
    HandleException;
  end;
end;

function TSoundExpert.GetIdString: string;
begin
  try
    Result := 'TSoundExpert';  { Unique ID for expert. }
  except
    HandleException;
  end;
end;

procedure TsoundExpert.Execute;
begin  { Executes our DLL in main form unit. }
  try
    ExecuteExpert(ToolServices);  { In sndexp1.pas. }
  except
    HandleException;
  end;
end;

procedure DoneExpert; export; { Clean up code, if needed. }
begin
  { Exit code here if needed. }
end;

{ Delphi 2/3/4 and Delphi 1-specific ways
  of accessing TIToolServices. }
{ $IFDEF WIN32 }
function InitExpert(ToolServices: TIToolServices;
  RegisterProc: TExpertRegisterProc; var Terminate:
  TExpertTerminateProc ): Boolean; stdcall; export;
{ $ELSE }
function InitExpert(ToolServices: TIToolServices;
  RegisterProc: TExpertRegisterProc; var Terminate:
  TExpertTerminateProc ): Boolean; export;
{ $ENDIF }
begin
  { Make sure this is the first and only
    instance of these services. }
  Result := ExptIntf.ToolServices = nil;
  if not Result then
    Exit;
  ExptIntf.ToolServices := ToolServices;
  if ToolServices <> nil then
    Application.Handle := ToolServices.GetParentHandle;

  Terminate := DoneExpert;  { We know where to exit. }

  RegisterProc(TSoundExpert.Create);
end;
{ $IFDEF WIN32 }
{ $ELSE }

function FaultHandler(FaultID: Word; FaultAddr: Pointer):
  TFaultResponse; export;
begin
  DefaultExceptHandler(FaultID, FaultAddr);
end;
{ $ENDIF }

exports
  { $IFNDEF WIN32 }
    FaultHandler name FaultHandlerSignature resident,
  { $ELSE }
  { $ENDIF }
  InitExpert  name ExpertEntryPoint resident;
begin
end.
```

End Listing Two

By Ron Loewy

An Automation Server

Object Model Design and Implementation

This article describes how to design an Automation server, and build it with Delphi. It also describes how to test the server by implementing it as a plug-in, and using script to put it though its paces. Let's begin with a review of the technology involved, and terminology used to describe it.

An application is an Automation server when it allows an external application (or scripting tool) to control it programmatically, via a COM interface. The Microsoft Office applications (Word, Excel, etc.) are good examples of Automation servers. For example, many applications use Word's mail merge capabilities from external applications to create letters, reports, and forms populated with data from the managing (client) application. Excel is similarly used by many enterprise applications as a calculation engine, e.g. a pre-defined spreadsheet is created and data is transferred to Excel by the client application (known as the Automation client) to perform complex calculations. Other examples include network charting applications that employ general-purpose tools such as Visio or Micrografx Charter to diagram a network.

Scripting is a feature that allows "power users" to automate tasks in the application by writing a script using a language integrated with the application. Microsoft offers VBA (Visual Basic for Applications) as the scripting language for its Office suite. VBA is also now widely available in non-Microsoft products, such as AutoCAD and Visio. Microsoft Internet Explorer allows you to use VBScript or JavaScript, Netscape Navigator offers JavaScript, and other tools offer more obscure scripting languages.

Plug-ins are externally compiled objects that "plug in" the application and add functionality via new menu commands, objects that can be manipulated, etc. Internet Explorer uses plug-ins to add new functionality. The Alexa Internet Explorer plug-in is a good example of a third-party tool that enhances the use of a mainstream application via a plug-in. Although Delphi doesn't use COM to implement plug-ins, the Open Tools API is a framework that allows you to add new objects (project and form experts) and plug-ins (add-ins and Help menu experts) to Delphi.

Why COM?

There are some advantages to using COM as a way to create an application extension framework. COM defines a standard method to offer Automation. Any application that knows how to talk to an Automation server will be able to automate your application via COM interfaces.

Excellent scripting engines are available free from Microsoft in the form of JavaScript (JScript) and VBScript. These scripting engines require an interface into your application via COM interfaces and standard Automation objects. Many users are familiar with Visual Basic, and VBScript is an acceptable substitute. Many Internet warriors are familiar with JavaScript, and Microsoft's JScript is a well-debugged version of this Java look-alike scripting engine. Using ActiveScript, you can provide VBScript, JScript, or both (see "Delphi 3 ActiveX" by Dan Miser in the February, 1998 issue of *Delphi Informant Magazine* for more about ActiveScript usage from Delphi). In fact, if you write the code to support one of these languages, you'll need to make minimal changes to support the other. Other scripting engines (e.g. Perl) are available that will also plug into an ActiveScript-enabled application.

Unlike DLLs that require extensive documentation, COM-based Automation objects are "self documenting" via type libraries. Though you should always provide documentation for an application extension framework, you won't need to create interface modules for every language you want to support, and you won't be limited to tools that can call DLLs. (Note: Delphi's Open Tools API was created with Delphi 1 and has been enhanced since that time.)

Although we often hear that Delphi uses Delphi to extend itself, Delphi allows experts and add-ins created with other tools to be installed. The Open Tools API, however, must provide special functions to allow C/C++ functions to call and manipulate Delphi functions, objects, and data structures. Even though COM plug-ins have more overhead than Delphi plug-ins, I have the feeling that Delphi would benefit from the cleaner approach to application extension via COM. (The Delphi debugger is implemented as a COM object of some sort, so Inprise developers are also aware of the benefits of COM for application extensibility.)

Planning the Extension Framework

The most important issue to understand about providing an application extension framework is the functionality of your application you want to expose — not the things the plug-ins, scripts, or external applications will have to do. COM is an object-based architecture. Your application should expose a set of objects that can be manipulated by the "extending" agents. The term "object model" seems to be the standard way to refer to an application-specific extension API. Examples include COM (Component Object Model) and Internet Explorer's (and the W3C) DOM (Document Object Model), which is used to describe the objects that make an HTML page (and with MSIE 5 XML documents). Even IBM used SOM (System Object Model) in the past. (When the sample application that provides template-based authoring had to have an extension framework, I named it Authoring Templates Object Model [ATOM] with the purpose of writing a white paper in the future titled "Splitting the ATOM.")

The root object of the Word object model is named *Application*, an object that provides access to Word and provides sets of collections (arrays) for sub-objects used in the application. In Word's case, the *Application* object provides access to the documents edited by the application, and a *Document* object that can be obtained from the application provides access to paragraphs, images, links, and other collections. Similarly, Excel's *Application* object provides access to worksheet objects that in turn provide access to cells.

When you design your object model, try to understand the data structures and objects with which your application is built. In the sample application, the code is nothing more than a visual way to represent a hierarchical database of objects. It made sense to provide access to this database via a single object obtained from the *Application* object. This object is named *Project* in the sample application; it provides access to all the nodes in the hierarchy. If your application manages relational databases, a good idea would be to expose common tables as objects via the root of the hierarchy. If your application is used for the creation of some documents, it would make sense to expose "document objects" (these documents don't need to be called documents; they can be worksheets, charts, images, etc.) as objects.

Once you understand what objects and data structures you want to expose (these provide the "guts" of your application), it's time to think of the way you want extensions to be able to manipulate the user interface side of the application. This usually means ways to expose the menubar, commandbars, and other elements of the application. I will provide sample code that exposes and manipulates the menubar, a common and popular way to offer scripting and extensibility.

Implementing an Object Model

If you've created your application using object-oriented techniques, it's easy to encapsulate your application's objects as Automation objects exposed in an object model. Assuming you use the convention of naming the object model root *Application*, it would be a good idea to hold a field in your application's main form that points to an instance of the *Application* Automation object's interface. This object can be instantiated in the *OnCreate* event of the main form. In the sample application, I created an Automation object named *eAuthorApp* (using Delphi's Automation object expert on the ActiveX page of the New Items dialog box). The main form holds a field using the following declaration in its **private** section:

```
FeAuthorApplication : IeAuthorApp;
```

and exposes it in the **public** section:

```
property eAuthorApplication: IeAuthorApp
  read FeAuthorApplication;
```

Finally, the *OnFormCreate* event handler includes the following code to instantiate the object:

```
try
  FeAuthorApplication := TeAuthorApplication.Create;
except
  ShowMessage(
    'eAuthor.eAuthorApplication could not be created');
end;
```

Providing access to the other objects in the object model is done via properties and methods of the Automation object. Because the sample application's database objects descend from a common ancestor, I created a virtual *CreateAutomationObject* method in the ancestor and created a default Automation object that applies to every object in the database. Some of the objects that descend from this common ancestor override *CreateAutomationObject* and return a pointer to an Automation object that provides more functionality than the default object. Your design will have to be based on the actual design of your objects and data structures.

My application's *Application* object provides an entry into the database object model by exposing a *Project* property defined of type *IDispatch*. (Remember that properties and methods defined in Automation objects must be defined using the Type Library editor and implemented in the implementation unit). Because the sample application has only one entry point into the database object model and because this entry point is commonly used, I want to cache it to see that my implementation of *TeAuthorApplication* includes a private pointer to *IDispatch*:

```
ProjectObject : IDispatch;
```

The *Get_Project* method that implements the *Project* property access (it's a read-only property) is implemented as follows:

```
function TeAuthorApplication.Get_Project: IDispatch;
begin
  if (not Assigned(ProjectObject)) then begin
    ProjectObject :=
      eAuthorSite.HyperTextProject.CreateAutomationObject;
    IUnknown(ProjectObject)._AddRef;
  end;
  Result := ProjectObject;
end;
```

As you can see, the main form holds a global *HyperTextProject* Delphi object and calls the *CreateAutomationObject* method to retrieve a pointer to its *IDispatch* interface. Notice the need to call *_AddRef* to ensure the object's reference count will keep it cached in memory. Obviously, I need to free it in the destructor:

```
destructor TeAuthorApplication.Destroy;
begin
  if (Assigned(ProjectObject)) then
    IUnknown(ProjectObject)._Release;
end;
```

Now let's examine the *CreateAutomationObject* for my object's ancestor:

```
function TObjectWithFields.CreateAutomationObject;
var
  NewObj: TEditableObject;
begin
  NewObj := TEditableObject.Create;
  NewObj.WrappedObject := Self;
  Result := NewObj;
end;
```

Remember, *TObjectWithFields* is the ancestor object to all the objects in the hierarchical database managed by the sample application. *EditableObject* is the Automation object created to represent this object from the COM side of things.

The interesting thing to notice is that *TEditableObject's* implementation has a *WrappedObject* property that holds a *TObjectWithFields*. Using this technique, a COM object can always be created from a Delphi object calling its *CreateAutomationObject*, and the Delphi object (which is always in memory) can be referred from the COM object by referring to its *WrappedObject* property.

Notice that *CreateAutomationObject* doesn't cache the object in memory using the *AddRef/Release* method of reference counting. Because the Delphi objects are always "alive," we can always use them to create the COM objects. The COM objects are created and stay alive only for the duration they are needed by the external application/script/plug-in.

Because every object in the sample application descends from *TObjectWithFields*, I wanted every COM object that represents a descendant of this object to retain *EditableObject's* methods and properties. This is surprisingly easy to do when you create a new Automation object (e.g. *AutoProject* represents the *THyperTextProject* object referred to previously). I inherited the interface defined for this object in the type library from *IEditableObject* and inherited the implementation object (*TAutoProject* in this example) from *TEditableObject* (see Figure 1).

Debugging Your Object Model

Before you continue with the design of a plug-in framework, you must ensure the object model you designed works as expected and provides access to functionality needed by extension "clients." When I was ready to test and debug my COM object model, I considered writing an Automation controller application with Delphi to test the server. Although this approach would work, I decided to test my object model by developing the scripting code and testing the object model from within the sample application. This made the concept of setting debug breakpoints and writing scripting code to test new objects easy and compile-free. (Again, see "Delphi 3 ActiveX" in the February, 1998 issue of *Delphi Informant Magazine* for more about ActiveScript and its inclusion in Delphi applications.)

For my purposes, I added a **Scripts** menu option to the main menu of the main form. This option opens a script dialog box that includes two *TMemo* components: one used as an editor, and the other as a console to test the results of the script.

In the *TActiveScriptSite* implementation of my *ActiveScript* unit, I exposed the *Application* and *Project* objects using the code shown in Figure 2.

Finally, I connected the console to the *Application* Automation object by adding *Write*, *WriteLine*, and *ClearConsole* methods. The *Write* implementation follows:

```
procedure TeAuthorApplication.Write(
  const AText: WideString);
begin
  if (Assigned(eAuthorSite.Console)) then
    eAuthorSite.Console[eAuthorSite.Console.Count - 1] :=
      eAuthorSite.Console[eAuthorSite.Console.Count - 1] +
      AText;
end;
```

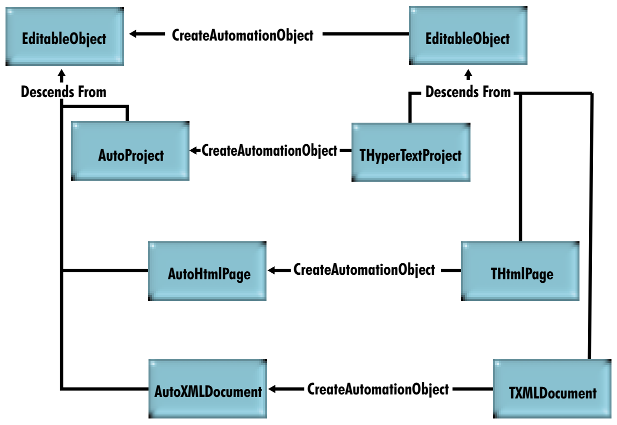


Figure 1: A COM-based object model as it is implemented from a hierarchy of Delphi objects.

```
function TActiveScriptSite.GetItemInfo(
  ItemName: WideString; dwReturnMask: DWord
  out UnkItem: IUnknown; out TypeInfo: ITypeInfo): HResult;
var
  ObjDispatch : IDispatch;
begin
  { Does the engine want the Automation object's
    IUnknown pointer? }
  if (dwReturnMask = SCRIPTINFO_IUNKNOWN) and
     (ItemName = 'Application') then
    UnkItem := eAuthorSite.eAuthorApplication;

  if (dwReturnMask = SCRIPTINFO_IUNKNOWN) and
     (ItemName = 'Project') then
    UnkItem := eAuthorSite.eAuthorApplication.Project;

  { Does the engine want the Automation object's
    type information? }
  if (dwReturnMask = SCRIPTINFO_ITYPEINFO) and
     (ItemName = 'Application') then begin
    ObjDispatch := eAuthorSite.eAuthorApplication;
    { Get a handle to our Automation object's
      type library. }
    ObjDispatch.GetTypeInfo(0,0,TypeInfo);
  end;

  if (dwReturnMask = SCRIPTINFO_ITYPEINFO) and
     (ItemName = 'Project') then begin
    ObjDispatch := eAuthorSite.eAuthorApplication.Project;
    ObjDispatch.GetTypeInfo(0,0,TypeInfo);
  end;

  Result := S_OK;
end;
```

Figure 2: Exposing the *Application* and *Project* objects.

Console is a global *TStrings* property of the main form. When the *Application* object needs to write a line to the console, it simply adds it to the console.

Providing Menubar Access via COM Interfaces

Now that we have an object model in place, it's time to think of ways to provide access to the user interface of the application. A common feature to expose is the menubar. The *Application* (COM) object of the example program exposes two functions: *MenuExists*, which determines if a menu path (e.g. **File | Save**) exists; and *Menu*, which returns an Automation object that represents a specific menu item based on a menu path provided.

I wrote the *FindMenu* function (see Figure 3) to find a menu item based on its path. The only interesting part of this function is stripping ampersand characters (used to determine menu hot-keys) from menu item captions.

I defined an Automation object named *AutoMenuItem* that has a *WrappedMenu* property (like the *WrappedObject* property of *EditableObject* objects discussed earlier). This COM object wraps *TMenuItem* and exposes its properties and methods (like *Checked*, *Enabled*, *Caption*, *Hint*, *ShortCut*, *Execute*, *SubItem*, etc.). In addition, *AutoMenuItem* has a *SetHandler* method required for plug-ins. We'll discuss this function later.

The sample application includes code that demonstrates how to expose the menu from an *Application* object and includes an implementation of a *Menu* Automation object. Once this mechanism is in place, it's easy to activate menu commands from a script or an automating client using code such as this JavaScript example:

```
Application.Menu("File|Save").Execute();
```

Menu access becomes more important once we need to discuss the issue of plug-ins.

```
function TeAuthorApplication.FindMenu;

  function FixCaption(ACaption: string): string;
  var
    g : Integer;
  begin
    Result := ACaption;
    g := pos('&', Result);
    while (g > 0) do begin
      Result := copy(Result, 1, g - 1) +
                copy(Result, g + 1, length(Result) - g);
      g := pos('&', Result);
    end;
  end;

  function FindByLevel(PartialPath: string;
    MenuRoot: TMenuItem): TMenuItem;
  var
    FirstLevel, RestOfPath : string;
    i, p : Integer;
  begin
    Result := nil;
    p := pos('|', PartialPath);
    if (p > 0) then
      begin
        FirstLevel := Copy(PartialPath, 1, p - 1);
        RestOfPath := Copy(PartialPath, p + 1,
                           length(PartialPath) - p);
       end
    else
      begin
        FirstLevel := PartialPath;
        RestOfPath := '';
      end;

    i := 0;
    while (i < MenuRoot.Count) do begin
      if (FixCaption(
            MenuRoot.Items[i].Caption) = FirstLevel) then
        Result := MenuRoot.Items[i];
        inc(i);
    end;
    if (Assigned(Result) and (RestOfPath <> '')) then
      Result := FindByLevel(RestOfPath, Result);
  end; { FindByLevel }

begin
   Result := FindByLevel(MenuPath, eAuthorSite.Menu.Items);
end; { TeAuthorApplication.FindMenu }
```

Figure 3: The *FindMenu* function to find a menu item based on its path.

Setting Plug-in Rules

A plug-in is a separately compiled code module that can be "plugged" into the application at run time. Typically, the plug-in adds entries to the application's menubar. When the user chooses a menu item that activates the plug-in, the plug-in needs to take over execution. It will then usually converse with the application via the application's object model.

In the sample application, I defined two interfaces that must be implemented by a plug-in:
- *IeAuthorApplicationAddIn*, which defines two methods: one that receives a reference to the application's object model root (*SetApplication*); and one that registers the plug-in's menus with the application (*RegisterMenus*).
- *IeAuthorMenuHandler*, which defines the *Execute* function that is called when a menu item associated with the plug-in is selected by the user.

Figure 4 shows a sample plug-in that adds an entry to the end of the **Help** menu. When this menu item is selected, it displays a dialog box. *TAddInObject* is defined as an Automation object and it implements the two interfaces mentioned previously.

```
type
  TAddInObject = class(TAutoObject, IAddInObject,
    IeAuthorApplicationAddIn, IeAuthorMenuHandler)
  private
    eAuthorApp : IeAuthorApp;
  public
    // IeAuthorApplicationAddIn
    procedure RegisterMenus; safecall;
    procedure SetApplication(
      const App: IeAuthorApp); safecall;
    // IeAuthorMenuHandler
    procedure Execute(Tag: Integer); safecall;
  end;
```

Figure 4: *TAddInObject* as an Automation object.

SetApplication stores the object model root in the *eAuthorApp* variable:

```
procedure TAddInObject.SetApplication;
begin
  eAuthorApp := App;
end;
```

RegisterMenus uses the menu facilities of the *Application* object to add a new menu entry. Notice the use of the *AutoMenuItem*'s *SetHandler* method (see Figure 5). We'll discuss this method in the next section.

```
procedure TAddInObject.RegisterMenus;
var
  NewMenu : OleVariant;
  TheMenu : IAutoMenuItem;
begin
  TheMenu := eAuthorApp.Menu('Help') as IAutoMenuItem;
  NewMenu := TheMenu.AddSubItem;
  NewMenu.Tag := 3;
  NewMenu.Caption := 'Add In!';
  NewMenu.SetHandler(Self as IeAuthorMenuHandler);
end;
```

Figure 5: Adding a new menu entry.

Finally, *Execute* is simple:

```
procedure TAddInObject.Execute;
begin
  ShowMessage('Add In Tag ' + intToStr(Tag));
end;
```

Notice the use of the *Tag* property to store information about a menu item. This allows a single plug-in to register multiple menu items, and differentiate between them in the *Execute* method.

Adding Plug-in Support

To have the application recognize plug-ins, two issues need to be resolved: a method to register plug-ins with the application, and a way for the application to know what menu items are associated with which plug-ins and how to call them.

The first issue is easy to resolve. I prefer to use the Windows registry to store a list of references to all the plug-ins recognized by the application. The application also provides a plug-in manager dialog box, which offers a way for users to specify the names of all the COM Automation objects that the application should load as plug-ins when it starts.

Handling menu-to-plug-in association is a bit more complicated, and includes the use of a new menu item class (I named it *TAutomatedMenu*) derived from *TMenuItem*. The definition of this new class is remarkably simple:

```
type
  TAutomatedMenu = class(TMenuItem)
  private
    FHandler : IeAuthorMenuHandler;
    procedure HandleClick(Sender: TObject);
  public
    constructor Create(AOwner: TComponent); override;
    property Handler: IeAuthorMenuHandler
      read FHandler write FHandler;
  end;
```

As you can see, an automated menu includes a reference to an *IeAuthorMenuHandler* interface, and a pre-defined click event handler named *HandleClick*. As you may recall from the previous section, the *AutoMenuItem* wrapper around the menu item object has a *SetHandler* method that's used by a plug-in that creates a new menu item. As you can imagine, *SetHandler* sets the *Handler* property of an automated menu item to point to itself.

With this business handled, the rest of the class implementation is trivial. The *Create* constructor sets the *OnClick* event to the *HandleClick* event handler, which in turn uses the handler's *Execute* method:

```
constructor TAutomatedMenu.Create;
begin
  inherited Create(AOwner);
  OnClick := HandleClick;
end;

procedure TAutomatedMenu.HandleClick;
begin
  if (Assigned(FHandler)) then
    FHandler.Execute(Tag);
end;
```

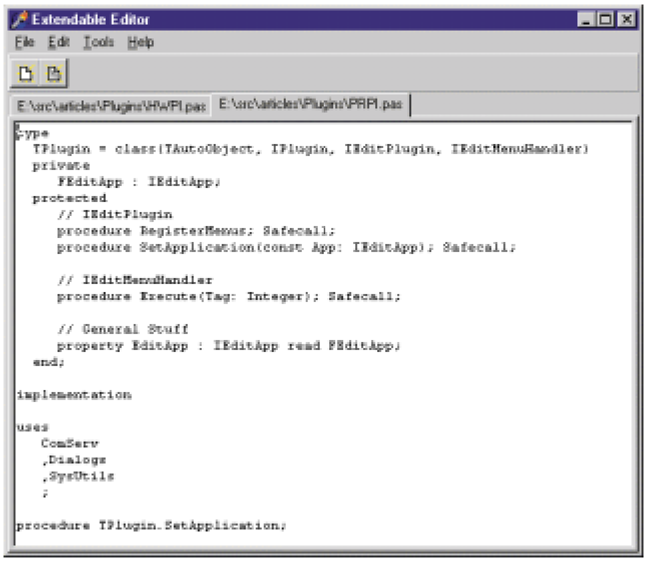
Sample Application

To demonstrate application extension via COM plug-ins, I wrote a simple text editor application (see Figure 6). The application uses a simple tabbed interface that allows you to edit multiple text documents. Think of it as a poor man's Notepad with extensibility. I decided to implement only a subset of the functionality one would normally implement in such an application, which is the minimum I needed to make the example useful.

The code that is available as part of this article includes the project editproj.dpr, the Delphi source for the sample application (available for download; see end of article for details). MainForm.pas is the definition and the code of the main form. PIMgr.pas is the code for the plug-in manager dialog box, which is used to "register" COM plug-ins with the sample application.

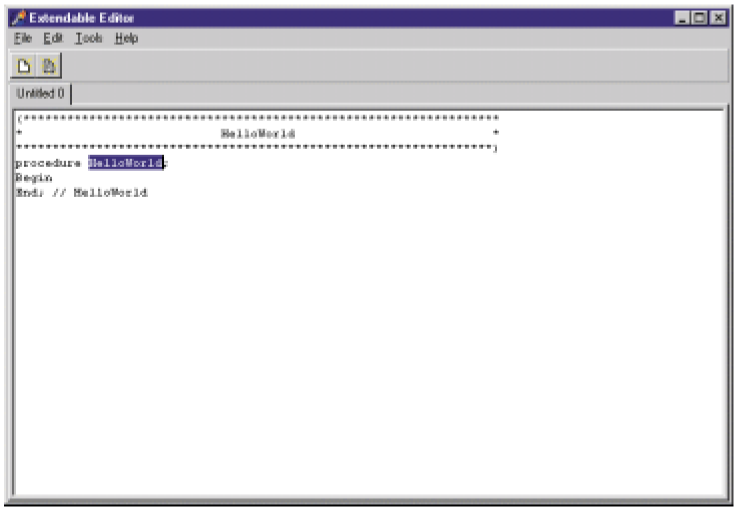
The application's object model consists of an *Application* object that provides access to the menu bar and to the document being edited. A more complete example will allow access to documents not currently edited by the user and the ability to switch between documents, but for the purpose of this example, this is all I needed.

The COM *Application* object (implemented in AutoApp.pas) provides access to COM objects implemented in the AutoMenu.pas file, and to the COM document object implemented in the AutoDoc.pas. The *Menu* object allows you to set menu attributes (caption, short-cut, visibility, and — for newly created menus — code handler), and allows you to create new menu entries that will be inserted into the application's menu bar. The document object provides access to the selected text and cursor location, and allows us to insert new text lines into the document.


Extendable Editor
File
Edit
Tools
Help
E:\src\articles\Plugins\HW\PI.pas
E:\src\articles\Plugins\PRPI.pas

```
type
  TPlugin = class(TAutoObject, IPlugin, IEditPlugin, IEditMenuHandler)
  private
     FEditApp : IEditApp;
  protected
     // IEditPlugin
     procedure RegisterMenus; Safecall;
     procedure SetApplication(const App: IEditApp); Safecall;

     // IEditMenuHandler
     procedure Execute(Tag: Integer); Safecall;

     // General Stuff
     property EditApp : IEditApp read FEditApp;
  end;

implementation

uses
   ComServ
   ,Dialogs
   ,SysUtils
   ;

procedure TPlugin.SetApplication;
```

Figure 6: The sample application at run time.



```
(******************************************************************
*                         HelloWorld                              *
******************************************************************)
procedure HelloWorld;
Begin
End; // HelloWorld
```

Figure 7: The results of using the ProcRemark plug-in to create a procedure header remark and body for the *HelloWorld* procedure.

In the application's *OnShow* event, the list of "registered" plug-ins is read from the registry, and these plug-ins are activated. The plug-ins in turn create new menu items that are added to the application's menubar. The user can now select the new menu entries and activate them.

The type library of the sample project also defines two interfaces, *IEditMenuHandler* and *IEditPlugin*, that must be implemented by plug-in COM objects. The application source also includes XtdMenu.pas, which implements a *TAutomatedMenu* menu item descendant that can activate a plug-in via the *IEditMenuHandler* interface from a menu.

I wrote two simple plug-ins to show the capabilities of what we can do with COM objects compiled separately from our application. The first, defined in HelloWorld.dpr, defines an Automation object named HelloWorld.Plugin that implements the required interfaces (see HWPI.pas), and adds a Hello World menu item to the Help menu. As can be expected, the plug-in displays a dialog box with the universal message displayed for programmer education worldwide.

A more sophisticated plug-in is implemented in ProcRemark.dpr, which takes advantage of the application's object model and adds two menu items to the Edit menu. The *Remark* procedure adds a three-line remark with the selected text between lines of asterisks, which are located above the line of the currently selected text (see Figure 7). The Procedure Body menu item adds a **begin..end** procedure body beneath the selected text in the editor. A long time ago, I wrote two functions like these in Brief's extension language (Brief being the text editor created by Underware Software and being sold for a while by Borland). It's amazing how much more productive I became when my Pascal sources were always prefixed and suffixed with the same style. The code for the function's implementation is provided in PRPI.pas. It's trivial, and, as such, left unexplained in this article.

Conclusion

Today's applications require extensibility and programmability. The Microsoft COM foundation is an excellent way to provide scripting, Automation, and plug-in capabilities from your application. Creating an extensible application requires a well-thought-out object model that maps the application's objects and concepts as COM objects that can be accessed from extension modules. Plug-ins can be created by adding support for minimal extension interfaces within the applications. COM plug-ins can be created in different development tools, and doesn't require recompiling the application. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\DEC\DI9912RL.

Installing the Sample Application

The code provided with the application includes the source and the compiled versions of the application and the plug-ins. You'll need to perform the following steps to use the plug-ins with the application:

1) Register the plug-in DLLs with the COM system (use regsvr32.exe in Window's \System directory. Or, if you're rebuilding the DLLs with Delphi, use the Run | Register | ActiveX Server menu option. (When you're deploying an application that uses COM plug-ins, you'll register the ActiveX servers from within your set-up program. Most set-up development tools allow you to set a file attribute and declare it as a self-registering ActiveX/OCX server.)
2) RegSvr32 HelloWorld.DLL
3) RegSvr32 ProcRemark.DLL
4) You need to register the plug-ins with the sample application. Start EditProj.exe and display the Plugin Manager dialog box using the Tools | Plug-ins menu option (see Figure A). Use the Add button to add HelloWorld.Plugin and ProcRemark.Plugin to the plug-in list. When you close the Plugin Manager dialog box, the new menu items will be added to the application.

— *Ron Loewy*

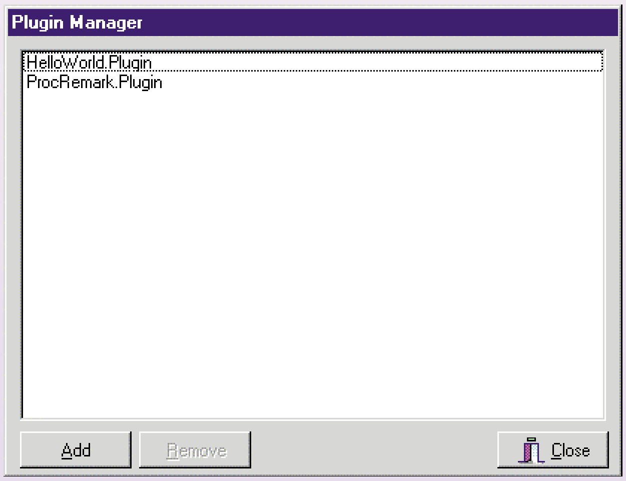


Figure A: The Plugin Manager dialog box.

Ron Loewy is a software developer for HyperAct, Inc. He is the lead developer of eAuthor Help, HyperAct's HTML Help authoring tool. For more information about HyperAct and eAuthor Help, contact HyperAct at (515) 987-2910 or visit http://www.hyperact.com.

By Bill Todd

1stClass

A Set of First-class Delphi Components

If you're looking for a set of controls to give your applications a distinctive look and feel, look no further than 1stClass from Woll2Woll Software. 1stClass includes Shape Button, Image Button, Button Group, Color Combo, Color List, Tree View, DB Tree View, Tree Combo, Font Combo, Image Form, Imager, Label, OutlookBar, and Status Bar components that offer a host of features not found in their Delphi counterparts.

If you like the look of Microsoft Outlook, the *TfcOutlookBar* component will let you duplicate it, and more. Working with the OutlookBar is surprisingly easy. The first step is to drop the OutlookBar on your form and set its *Align* property to position the bar either vertically at the left or right side of the form, or horizontally at the top or bottom. Next, select the *OutlookItems* property and click the ellipsis button to open the Collection Editor. Using the context menu or toolbar in the Collection Editor, add one or more pages to the OutlookBar. The page buttons are actually *TfcShapeBtn* components, so you can assign a graphic to the button's *Image* property and the button assumes the shape of the graphic. Click the active page in the OutlookBar and select its *Images* property to assign an ImageList component that contains the images you want to use for the items on this page. Next, select the *Items* property. Open the Collection Editor for the *Items* property and add the number of items you need for the current page. Now select each item, set its *ImageIndex* property to the correct image in the ImageList component, then switch to the Events page of the Object Inspector and create an *OnClick* event handler. That's all there is to it. Another remarkable feature of the OutlookBar is the ability to embed other components in the bar.

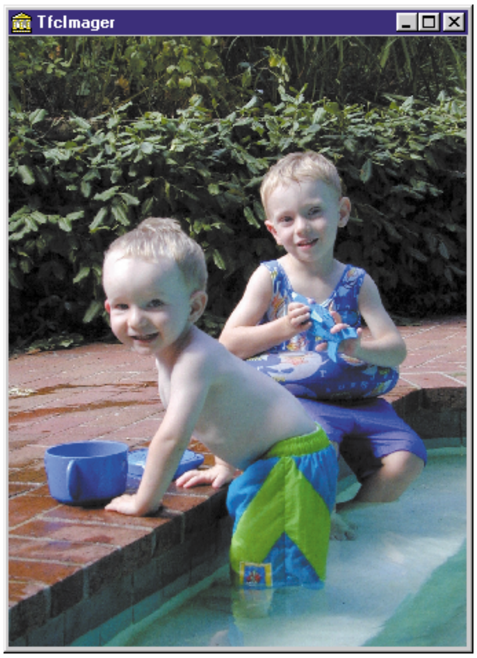


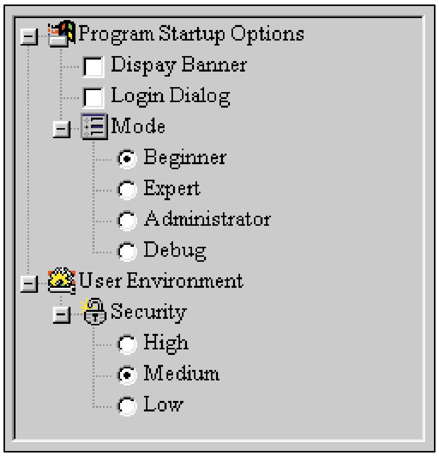
Figure 1: *TfcImager* with a JPEG photo loaded.

The *TfcImager* control is similar to Delphi's *TImage* component, but with much more power. The *Picture* property of *TfcImager* lets you load a JPEG, bitmap, icon, enhanced metafile, or Windows metafile image. Using other properties, you can change the appearance of the image in various ways. For example, you can adjust the contrast, brightness, and saturation, or give the image an embossed, blurred, or sponged look. You can also flip or rotate the image in any direction and apply a tint. Figure 1 shows an Imager with a digital camera photograph loaded. You can also set the *Imager* property of the *TfcDBTreeView* or *TfcOutlookBar* controls so they'll use the Imager to supply a background bitmap.

1stClass also includes a Tree View component and a Tree Combo component. Each node in the *TfcTreeView* consists of a label, one or more images, and a list of optional checkboxes or radio buttons. Each node can also have a list of subnodes, which in turn can have subnodes, allowing you to build a tree of any depth. The *TfcTreeCombo* shares most of the capabilities of the Tree View in a combo-box format. The Tree Combo component can also be embedded in the *TwwDBGrid* component that is part of Woll2Woll's InfoPower Suite. Figure 2 shows a Tree View that contains both checkboxes and radio buttons.

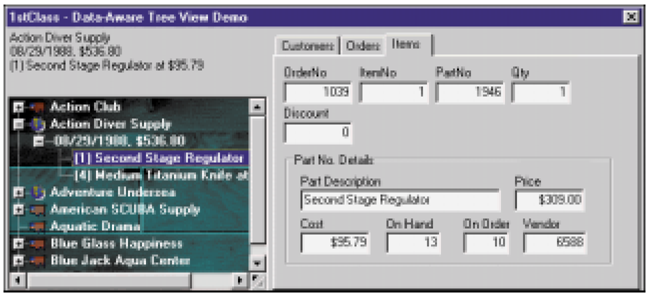
TfcDBTreeView is a data-aware Tree View component that can automatically model any master/detail relationship regardless of the complexity or the number of datasets involved. The datasets can consist of any combination of *TTable* and *TQuery* components. To use the DB Tree View component, you must set the *DataSourceFirst* property to the DataSource component for the dataset that is the root node of the tree. Next, you must set the *DataSourceLast* property or the *DataSource* property. Setting *DataSourceLast* will cause all DataSources between the root node and the DataSource specified in *DataSourceLast* to appear in the tree. Using the *DataSources* property lets you specify a list of DataSources to display. The *DataFields* property provides control over which fields are displayed at each level in the tree. Figure 3 shows a data entry form that incorporates a DB Tree View. The DB Tree View provides a live view of the data, so navigating from one record to another in the DB Tree View will change the current record in any form that uses the same dataset, and vice versa. (Note also the use of an Imager component to provide a graphical background for the DB Tree View in Figure 5).

TfcImageBtn is a class with all the features of the standard Delphi *TButton*, *TBitBtn*, and *TSpeedButton* classes, but it gets its shape and appearance from an image. To give the button a different look when it's depressed, assign different images to the button's *Image* and *ImageDown* properties. The up and down images can be completely different, so the button will have a different appearance and shape when it's down. You can even create clickable maps, as shown in Figure 4, by using an ImageBtn for each area of the map.



- Program Startup Options
  - ☐ Dispay Banner
  - ☐ Login Dialog
  - Mode
    - ● Beginner
    - ○ Expert
    - ○ Administrator
    - ○ Debug
- User Environment
  - Security
    - ○ High
    - ● Medium
    - ○ Low

Figure 2: The *TfcTreeView*.



1stClass - Data-Aware Tree View Demo

Action Diver Supply
08/29/1988, $536.80
(1) Second Stage Regulator at $95.79

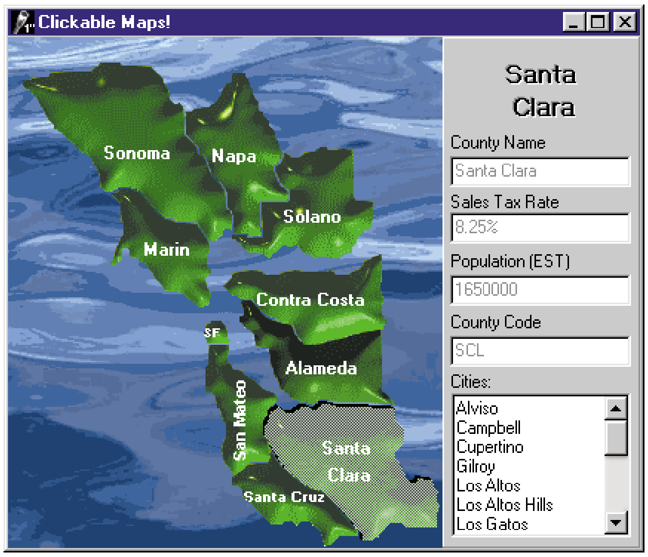
- Action Club
- Action Diver Supply
  - 08/29/1988, $536.80
    - [1] Second Stage Regulator
    - [4] Medium Titanium Knife at
- Adventure Undersea
- American SCUBA Supply
- Aquatic Drama
- Blue Glass Happiness
- Blue Jack Aqua Center

Customers | Orders | Items

OrderNo: 1039
ItemNo: 1
PartNo: 1946
Qty: 1
Discount: 0

Part No. Details
Part Description: Second Stage Regulator
Price: $309.00
Cost: $95.79
On Hand: 13
On Order: 10
Vendor: 6588

Figure 3: The sample Customers, Orders, and Items tables in a *TfcDBTreeView*.



Santa Clara
County Name: Santa Clara
Sales Tax Rate: 8.25%
Population (EST): 1650000
County Code: SCL
Cities:
Alviso
Campbell
Cupertino
Gilroy
Los Altos
Los Altos Hills
Los Gatos

Figure 4: A clickable map using *TfcImage* buttons for the counties.

TfcShapeBtn combines the functionality of *TSpeedButton* and *TBitBtn* with the ability to create buttons of virtually any shape. Figure 5 shows the standard shapes for the ShapeBtn. If the standard shapes don't meet your needs, you can create custom shapes by defining a list of points that describe the shape. You can control the button's color as well as the highlight colors that are used for three-dimensional effects. The image and shape buttons provide *OnMouseEnter* and *OnMouseLeave* events, which you can use to make the button change its appearance when the mouse cursor passes over the button.

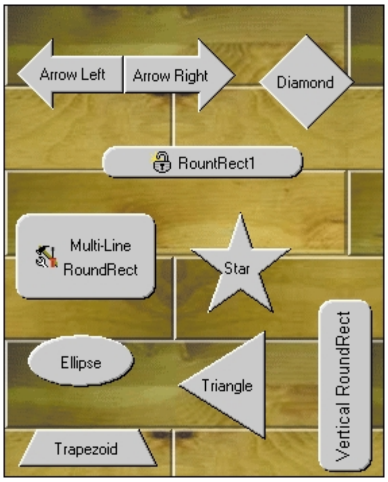


Figure 5: The standard shapes for *TfcShapeBtn*.

TfcButtonGroup lets you easily create a collection of shape buttons or image buttons. You can arrange the buttons in any number of rows and columns, as well as adjust the spacing between adjacent buttons. Three behaviors are available when the buttons are clicked: The buttons can behave as a group of radio buttons, as toggle buttons, or as click buttons. If you set the *ClickStyle* property to *bcsRadioGroup*, only one button can be depressed at a time. Optionally, you can allow all the buttons to be up, or you can require that one always be depressed. The *bcsCheckList* option allows any combination of buttons to be depressed at one time, simulating a group of checkboxes. Choosing *bcsClick* makes the buttons behave as regular buttons so they remain depressed only as long as you hold the mouse button down. The *TfcButtonGroup* class also provides a *Transparent* property. When set to True, all areas of the button group not covered by a button will be transparent. Figure 6 shows a button group with the *Transparent* prop-

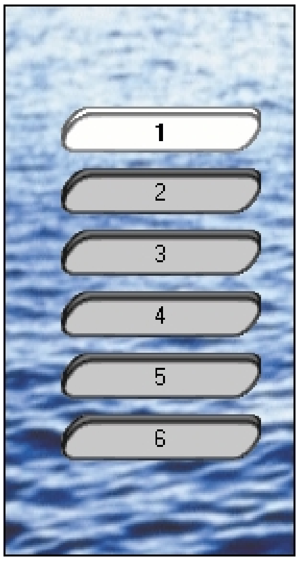


Figure 6: A *TfcButtonGroup* with transparent background.

erty set to True so the background graphic fills the area around the buttons.

When you drop the *TfcImageForm* control on a Delphi form, you create a form whose shape is defined by the non-transparent areas of the image specified by the *Picture* property. The *TfcImageForm* also sets the host form's *BorderStyle* to *bsNone* to remove the form's border and title bar. The result is a form that consists only of the graphic assigned to the *Picture* property. Although the form has no title bar, users can still drag it by depressing the left mouse button over any part of the form. You can also assign any control on the form to the ImageForm's *CaptionBarControl* property. Now dragging the assigned control will drag the form, and attempting to drag any other area of the form will not.

INFORMANT FACT FILE

If you're looking for a set of controls to give your applications a distinctive look and feel, look no further than 1stClass from Woll2Woll Software. 1stClass provides an excellent set of components for adding sophistication to the user interface of your Delphi programs.

Woll2Woll Software
2217 Rhone Drive
Livermore, CA 94550

Phone: (800) 965-2965
Fax: (925) 371-1664
E-Mail: sales@woll2woll.com
Web Site: http://www.woll2woll.com
Price: Standard version, US$199; Professional version, US$299.

The *TfcColorCombo* and *TfcColorList* components provide users with tools to easily select a color. The Color Combo component is data-aware and can be used to store colors in a database table by storing either its name or integer value. You can choose from system or standard color lists, or construct your own custom colors. If you're an InfoPower user, you can embed the ColorCombo and all the other 1stClass combo components into the InfoPower Grid and RecordView components.

TfcStatusBar resembles the Delphi *TStatusBar* but with much more flexibility. You can add any number of panels to the status bar and automatically display the hint properties from controls on the form; the current date, time, or both; the state of CapsLock, NumLock, ScrollLock, and Insert; and the computer name or the name of the current user. You can also embed your own components in the StatusBar's panels. You can also size the panels in a status bar proportionally so the panels will occupy the same percentage of the status bar's width when the form is resized.

TfcFontCombo is a combobox for selecting fonts with an optional most recently used list at the top. Setting the *MaxMRU* property to a value other than zero will cause the most recently used fonts to appear at the top of the drop-down list. *TfcLabel* is a label component that supports shadow, extrusion, engraved, embossed, and outline effects.

Conclusion

1stClass is an excellent set of components for adding sophistication to the user interface of your Delphi programs. Woll2Woll still believes in providing user manuals, and 1stClass comes with a well-indexed, 178-page, wire-bound manual that provides a complete reference to all the components and their custom properties, methods, and events. 1stClass comes in a Standard version, which supports Delphi 3 through 5, and a Professional version, which supports Delphi 3 through 5 and C++Builder 3 and 4. You can give 1stClass a try by downloading the demonstration version from Woll2Woll's Web site. Δ

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is a Contributing Editor of *Delphi Informant*, a co-author of four database programming books and over 60 articles, and 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. He can be reached at bill@dbginc.com.

In Review: Delphi 4 Books

Now that Delphi 5 has shipped, let's put the Delphi 4 books into perspective. I'll also include some classics — books that have been around (in some cases since Delphi 2) that are still worth adding to your collection. Many of these continue to be relevant, although, of course, none will cover the new features specific to Delphi 5.

The classics. We can usually expect much from a book that has been published in several editions. It generally indicates a high level of quality and strong confidence on the part of the publisher and the audience. I'll examine three such enduring classics, one of which I reviewed in these pages. Of course, all three emphasize Delphi 4's new features; all three are comprehensive, covering the major aspects of Delphi and its underlying language, Object Pascal; and all three deal with the major Delphi disciplines, including component writing, database programming, and the IDE. However, each has its particular emphasis and strength.

Among them, *Mastering Delphi 4* by Marco Cantù [SYBEX, 1998] continues to be the best entry-level work. In fact, I still refer to it from time to time when I decide to explore an area of Delphi that's new to me. Always an innovator, Cantù has come up with what I consider to be an effective new approach to publishing: Rather than include all the book's code on a CD-ROM, he has made it available at his Web site. I wonder if this will become a trend in Delphi publishing; it's certainly worth imitating.

Charlie Calvert's Delphi 4 Unleashed [SAMS, 1998] is an update of a Delphi classic. It didn't appear in a Delphi 3 version, but is back for version 4. A great deal has changed since the Delphi 2 version, so much so that calling this an update is really not fair. Gone is the introduction to Object Pascal that began the earlier edition. In its place are several chapters designed to introduce the reader to Delphi programming — topics such as program design, Delphi's IDE, and enhancements in version 4. However, the excellent discussions of component writing and multimedia remain in the new edition, along with updated database topics. In particular, this book really shines in its treatment of the new technologies MIDAS, CORBA, and COM/DCOM.

I reviewed *Borland Delphi 4 Developer's Guide* by Steve Teixeira and Xavier Pacheco [SAMS, 1998]. Although I pointed out that one of the great strengths of this work is its collection of tips, the value of this book goes much further. It begins with an excellent chapter on Object Pascal. As with the other two classics mentioned, it provides an overview of the extensions to the language introduced in Delphi 4. I found the chapters on dynamic link libraries and multithreading particularly excellent. It also covers component-writing topics, including component editors and packages. A large part of this work deals with database programming, going well beyond the usual topics, and continues generally to be the most "advanced" of the three titles.

New works. As with Delphi 3, there have been some new specialized and general works that have appeared in the last year. One specialized reference I found particularly useful was Clay Shannon's *Developer's Guide to Delphi Troubleshooting* [Wordware Publishing, 1999], which is essentially an encyclopedia of Delphi error messages. If you've been programming for years in Delphi (or Turbo Pascal, for that matter), you'll encounter some "old friends" here. At first I wondered if there would be anything new for me. I didn't have to wait long. Within a few days of receiving this valuable reference, one of the error explanations saved me hours of needless effort in getting an API call to work.

I would be remiss not to mention the return of a venerable programming author, Tom Swan, with his *Delphi 4 Bible* [IDG Books Worldwide, 1998]. As Warren Rachele correctly points out in his February, 1999 *Delphi Informant Magazine* review, this work is particularly appropriate for the intermediate-level developer who wants to advance. I'm not as convinced, however, that this book would not also benefit less-experienced Delphi developers, particularly if they had mastered one of the three classics previously discussed.

Library essentials. To conclude, we'll take a walk down memory lane and review some of the earlier works of some Delphi gurus. The essential Delphi library would be incomplete without Ray Lischner's *Secrets of Delphi 2* [Waite Group Press, 1996] and *Hidden Paths of Delphi 3* [Informant Communications Group, 1997]. While these provide a wealth of valuable information for component writers, Ray Konopka's *Developing Custom Delphi 3 Components* [Coriolis Group Books, 1996] remains the essential work for component writers. Finally, *Delphi Developer's Handbook* [SYBEX, 1998] by Marco Cantù and Tim Gooch remains one of my favorites for advanced topics in application development. You can find out more about many of these books by reading the full reviews on the *Delphi Informant Magazine* Web site at http://www.DelphiMag.com.

Next month we'll explore a "what if" that none of us looks forward to dealing with. Until then ...

— *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 on the Internet at acmdoc@aol.com.