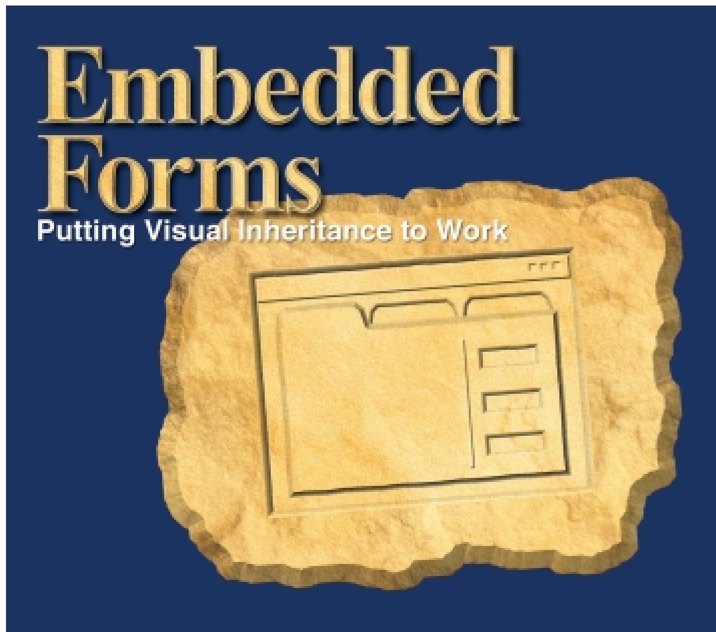
January 2000, Volume 6, Number 1



Cover Art By: *Arthur Dugoni*

ON THE COVER

6 Delphi at Work

Embedded Forms — John Simonini

Mr Simonini shares a technique for embedding forms as the pages of a tabbed notebook — a technique that facilitates discrete business logic, team development, memory management, and code reuse.

FEATURES

10 Distributed Delphi

Request Threads — Michael J. Leaver

Mr Leaver describes a resource-sharing methodology for Delphi 4/5 Client/Server Suite using DCOM, and without using Transaction Processing Monitors, such as MTS or BEA's Tuxedo.

14 Sound + Vision

A Multimedia Assembly Line: Part II — Alan C. Moore, Ph.D.

Dr Moore wraps up his two-part description of how to create a Delphi sound expert that builds sound-enabled components, including a detailed look at the code-generating engine.

18 In Development

Control Panel Applets — Peter J. Rosario

Control Panel applets are the small programs that are visible in, and run from, Windows Control Panel. Mr Rosario describes why you'd write one, and demonstrates how to do it with Delphi.

22 Dynamic Delphi

Run-time ActiveX — Ron Loewy

Sure, it's easy to incorporate an ActiveX component with your Delphi application at design time, but Mr Loewy takes it one big step further by allowing run-time integration.

REVIEWS

29 Orpheus 3

Product Review by Alan C. Moore Ph.D.

DEPARTMENTS

2 Delphi Tools

5 Delphi News

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

Delphi TOOLS

New Products and Solutions

Excel Software Announces QuickCRC 1.2

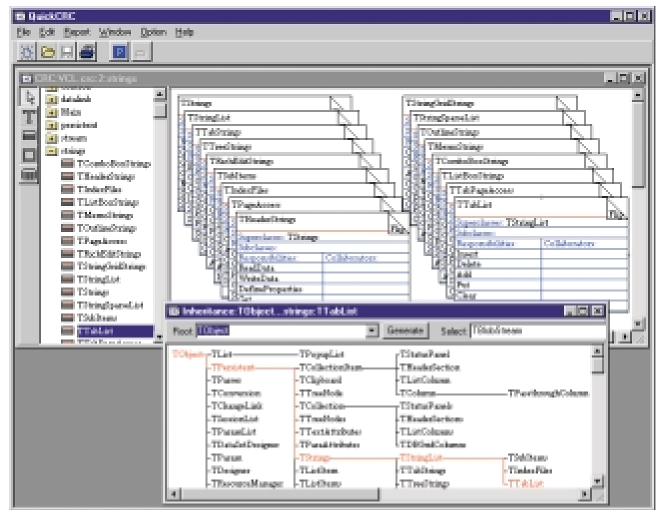
Excel Software announced ***QuickCRC 1.2*** for object-oriented software modeling on Windows or Macintosh computers. Version 1.2 increases the model capacity to support thousands of object classes, hundreds of diagrams, and long names for classes, attributes, and operations. QuickCRC 1.2 supports the Delphi, C++, and Java re-engineering features available in WinTranslator 2.0, which generates class models or CRC cards from source code.

QuickCRC uses a diagram workspace for creating card and scenario objects. A card represents the properties of a class, including its name, description, superclasses, subclasses, attributes, responsibilities, and object collaborations. A scenario represents a design mechanism defined as a series of steps involving communicating objects. Scenarios can reference cards and other scenarios. As information is entered or changed for a card or scenario object, it is instantly synchronized throughout the model. Separate diagrams partition large models into subject areas. The contents view allows a designer to navigate between diagrams. The generated inheritance graph concisely illustrates the class inheritance structure.

A text representation can be generated from information in a CRC model. This information can be used as a coding specification, transferred to other applications, or used to generate a new model. Design information can be exported to the MacA&D or WinA&D modeling tools for detailed design and code generation.

QuickCRC for Windows runs on Windows 95, 98, and NT. Models are binary-compatible between platforms.

Excel Software
Price: US$295
Phone: (515) 752-5359
Web Site: http://www.excelsoftware.com



Vista Software Releases Apollo Client/Server 5.0

Vista Software announced the release of ***Apollo Client/Server 5.0*** for Delphi developers. Based on the Apollo product (the company's native VCL replacement for the BDE) and new 32-bit SDE database technology, this client/server database engine offers features Apollo users have been requesting, including minimizing network traffic by offloading data processing onto the server. Apollo Client/Server is compatible with Delphi 4 and 5 and C++Builder 3 and 4.

Apollo Client/Server is TCP/IP-based, which allows developers to connect client applications to any computer addressable by an IP address, from a local intranet TCP/IP network, to a Web site on the Internet. Apollo Client/Server 5.0 includes support for developing non-client/server applications.

Users deploy the Apollo Server application on a networked server running Windows 95, 98, NT, or 2000. Basic Apollo server configuration requires only that developers tell the server which directories contain their database files. Users then use the *TApolloDataSet* and *TApolloConnection* components in their client applications to connect to the Apollo Server and access those registered databases.

Apollo Client/Server 5.0 features unlimited concurrent client connections per server, support for local and client/server application development, support for server-side stored procedures, user security, transaction support, triggers, complete Delphi source for all components included, and more.

Vista Software
Price: US$679; upgrade pricing is available.
Phone: (800) 653-2949
Web Site: http://www.vistasoftware.com

Tiriss Announces CB4 Tables Version 1.01

Tiriss announced version 1.01 of ***CB4 Tables***, its Delphi wrapper for using Sequiter's CodeBase instead of the BDE. CB4 Tables is a set of components that can be used anywhere you want to use dBASE IV, FoxPro, or Clipper tables, but don't want to use the BDE.

CB4 Tables' *TTable* replacement has almost all functions a *TTable* has, and is completely compatible with *TTable*. CB4 Tables' major advances include no BDE install, but a (small) DLL sold by Sequiter Software; compatibility with the BDE; compatibility with all standard Database components; and support for Delphi 3 and 4.

Tiriss
Price: US$35; US$49 with source.
E-Mail: info@tiriss.com
Web Site: http://www.tiriss.com

Delphi TOOLS

New Products and Solutions

Vista Software Launches Apollo 5.0

Vista Software announced the release of ***Apollo 5.0***, the company's native VCL replacement for the Borland Database Engine.

Some of the new Apollo 5.0 features include Delphi 5 support (and continued support for Delphi 3 and 4 and C++Builder 3 and 4); the *TApolloEnv* component, which helps isolate global/environmental settings for all *TApolloDataSet* controls; a comprehensive online help file (specific to new *TApolloDataSet*, *TApolloEnv*, and *TApolloDatabase* component architecture); updated sample programs; improved support for Woll2Woll's InfoPower components; new properties and methods for greater power and flexibility; and complete Delphi source code for *TApolloDataSet*, *TApolloEnv*, and *TApolloDatabase* components.

Vista Software
Price: US$379; upgrade pricing is available.
Phone: (800) 653-2949
Web Site: http://www.vistasoftware.com

ASTA Technology Group Announces ASTA 2.0

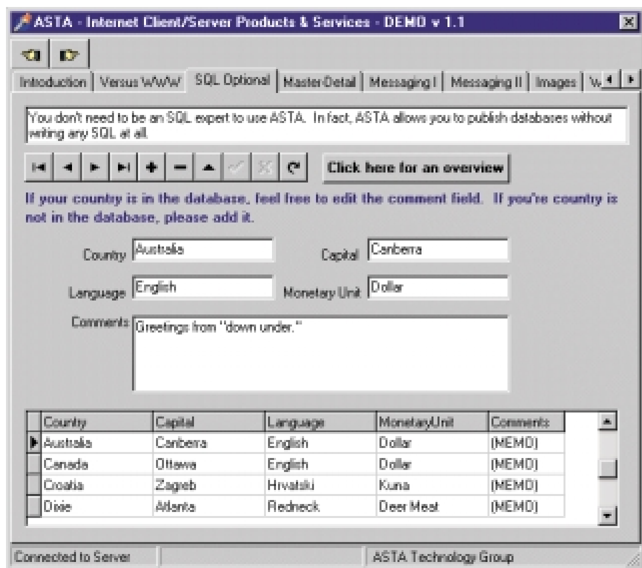
ASTA Technology Group announced the release of ***ASTA 2.0***, the company's multi-tier development tool for producing Internet applications. Delphi 5 support has also been initiated with the release.

In addition to improved performance and numerous feature enhancements, ASTA 2.0 introduces support for pure multi-tier programming, including support for business objects and Java clients.

New server-side features include an AstaProvider, which offers advanced SQL generation on the server, with events that can fire before each **update**, **insert**, or **delete** statement. The ASTA Business Objects Manager introduces a "smart man's" method of working with distributed objects. Define server-side "methods" along with params and they will become accessible to client-side work at design time. Support for Java clients has also been added with an event for communication with Java clients.

Robust ASTA servers run in three ASTA threading models — all threading is handled internally and automatically by ASTA. ASTA's client-side enhancements include Automatic Client Updates, support for Remote Directories (Internet-ready file dialog boxes), an extended suitcase model, a progress bar for visual feedback on socket reads, and the ability to clone datasets and their edits.

ASTA Technology Group
Price: Component Suite, US$249; Entry Suite, US$399 (includes Component Suite and one server license).
Phone: (800) 699-6395 or (785) 539-3731
Web Site: http://www.astatech.com



Quiksoft Announces EasyMail Objects Version 5.0

Quiksoft Corp. released version 5.0 of ***EasyMail Objects***, a set of e-mail COM objects used by software developers to e-mail-enable their applications. Applications utilizing EasyMail Objects can send, retrieve, view, compose, and print Internet e-mail. EasyMail Objects is ideal for developers using Delphi, Visual Basic, C++, ASP, and other development systems supporting COM objects.

Version 5.0 enables the developer to create and retrieve rich HTML messages, including embedded items, such as images, sounds, and video clips.

The updated IMAP4 Object enables applications to download a message's list of attachments so it can screen out messages containing large or unwanted items. The SMTP Object supports ESMTP's "Login" and encrypted "CRAM-MD5" authentication schemes. The new version also allows developers to increase performance by using callback functions instead of each object's standard COM events.

The objects are Y2K-compatible and scale to suit multi-user environments, such as Web servers. The objects support SMTP, POP3, IMAP4, ESMTP, APOP, HTML, RTF, plain text, file attachments, MIME, Base64, UUencode, Quoted Printable, customized headers, and optional NT integration.

Quiksoft Corp.
Price: From US$399 per developer.
Phone: (800) 509-8700 or (610) 544-3139
Web Site: http://www.easymailobjects.com

Delphi TOOLS

New Products and Solutions

Greg Lief Offers G.L.A.D. Components

Greg Lief is offering *G.L.A.D.* (Greg Lief's Assorted Delphi) components, a collection of over 60 Delphi components and property editors. The suite includes three user-friendly filtering components (*TGLFilterDialog*, *TGLQBE*, and *TGLQBE2*); *TGLHTMLTable*, which converts any dataset to a formatted HTML table; *TGLXLS*, which converts any dataset to Microsoft Excel format); *TGLPrintGrid*, and more.

G.L.A.D. registration includes all Delphi source code; support for Delphi 1 through 5 and C++Builder 1 through 3; a comprehensive help file; unlimited minor updates; and access to Greg Lief's Delphi Knowledge Base, an online searchable collection of Greg's favorite Delphi tips, techniques, and examples.

Greg Lief
Price: US$69
Phone: (877) REGSOFT
Web Site: http://www.greglief.com/delphi/components.shtml

Objective Software Releases ABC for Delphi Version 5

Objective Software Technology Pty Ltd. announced version 5 of its visual component library, *ABC for Delphi*.

ABC is a complete user interface toolkit, with over 200 components to enhance all aspects of user interface design, data presentation, and application support. Components include toolbars and custom menus; DB tree views; advanced DB navigation, exception handling, and help and hint editing; rich text editing; button bars; animation and transition effects; and more.

A new Dataset Adapter Framework adds support for ADO, InterBase, MIDAS, and common third-party datasets. This framework isolates ABC components from underlying dataset functionality, reducing the interface to a single point of adaptation. Developers will find it easier to use ABC in Delphi 5, with a new component browser, ABC property category, and enhanced component and property editors.

Objective Software Technology Pty Ltd.
Price: From US$149
E-Mail: sales@obsof.com
Web Site: http://www.obsof.com

TUS Announces LocoMotion

Tisfoon Ulterior Systems (TUS) announced the release of *LocoMotion*, a set of native Delphi VCL components for motion control and factory automation.

The set of 13 components control servo motors, digital and analog IO, and specialized components for linked axes, two-dimensional gantry support, and analog input filters.

Included are built-in forms for user configurable parameters, including PID filter parameters for servo motors, port number, and others. These forms have optional operator-level and maintenance-level passwords and complete online documentation. Additionally, there are built-in diagnostics forms with background status updates for each device.

Currently, the components only work with the Motion Engineering PCX/DSP card under the Windows NT operating system.

Tisfoon Ulterior Systems
Price: US$4,995 (includes source and unlimited e-mail support).
Phone: (919) 881-8322
Web Site: http://www.Tisfoon.com

Wise Introduces Wise for Windows Installer

Wise Solutions, Inc., working closely with Microsoft Corp., announced *Wise for Windows Installer*, an application installer that enables application developers to create installation programs compatible with Microsoft's new Windows Installer technology.

Microsoft's Windows Installer was originally designed for Windows 2000 (formerly Windows NT 5.0). However, both the development environment and the installations created will support Windows NT 4.0, and Windows 95/98. Designed to reduce the administrative requirements of managing Windows workstations, the technology plays a key role in Microsoft's Zero Administration initiative for Windows. The new Wise for Windows Installer enables developers to create installations that meet the new Microsoft standard.

Wise Solutions, Inc.
Price: US$795
Phone: (800) 554-8565
Web Site: http://www.wisesolutions.com

SSNet Releases NeoSecure

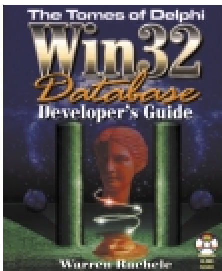
SSNet, Inc. announced the release of *NeoSecure*, which provides Windows-based software authors and proprietary content providers with a system to control and monitor product distribution and piracy.

NeoSecure is available as a plug-in (for embedding in software source code) and as a shell, providing protection to content files, such as music, video, data, and others.

Both forms of NeoSecure offer a variety of features, such as automatic e-mail notification of product installs and registrations; the ability to prevent multiple uses of a serial number; the ability to instantly disable a given serial number, thereby disabling all copies using that number; and a "backdoor" into every installed copy for news and other special announcements.

SSNet, Inc.
Price: NeoSecure Plug-in, US$200 (includes US$50 setup fee and 100 keys); NeoSecure Shell, US$300 (includes US$150 setup fee and 100 keys).
Phone: (303) 722-5240
Web Site: http://softwaresolutions.net/neosecure/index.htm

The Tomes of Delphi: Win32 Database Developer's Guide
Warren Rachele
Wordware Publishing, Inc.

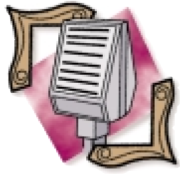


ISBN: 1-55622-663-2
Price: US$39.95
(365 pages, CD-ROM)
Web Site: http://www.wordware.com

News

L I N E

January 2000



Inprise Announces JBuilder 3 Enterprise Solaris Edition

Scotts Valley, CA — Inprise Corp. announced JBuilder 3 Enterprise Solaris Edition, making the company's enterprise development tools for the Java platform available for the Solaris operating environment. JBuilder 3 Enterprise is a comprehensive visual development tool for creating Java-platform-based applications and applets that can also include JavaServer Pages and Java Servlets technologies, JavaBeans and Enterprise JavaBeans technologies, and distributed CORBA applications for the Java 2 platform.

JBuilder 3 Enterprise Solaris Edition also includes support for the Java 2 Platform, Enterprise Edition (J2EE).

JBuilder 3 Enterprise Solaris Edition is available on the Web at http://www.borland.com. JBuilder 3 Enterprise Solaris Edition has an estimated street price (ESP) of US$2,499 for new users. Current owners of any Borland Client/Server or Enterprise product can purchase the product for an ESP of US$1,699. Current owners of any Borland Professional product can purchase the product for an ESP of US$2,199.

Inprise to Support C, C++, and Delphi Development on Linux

Scotts Valley, CA — Inprise Corp. announced it is developing a Linux application development environment that will support C, C++, and Delphi development. The project, code-named "Kylix," is set for release this year.

Project Kylix is planned to be a Linux component-based development environment for two-way visual development of graphical user interface (GUI), Internet, database, and server applications. Plans are for Project Kylix to be powered by a new high-speed native C/C++/Delphi compiler for Linux and will implement a Linux verison of the Borland VCL (Visual Component Library) architecture. The Borland VCL for Linux will be designed to speed native Linux application development and simplify the porting of Delphi and C++Builder applications between Windows and Linux.

The Project Kylix design was influenced by the results from the Borland Linux Developer Survey, conducted in July 1999. The results indicate that developers are seeking RAD, database enablement, and GUI design. The Project Kylix development environment is planned to support major Linux distributions, including Red Hat Linux and the forthcoming Corel LINUX.

To learn more, visit Inprise at http://www.inprise.com.

Inprise Licenses VisiBroker CORBA Technology to HP

Inprise Corp. announced a worldwide technology licensing agreement with Hewlett-Packard Company covering Inprise's VisiBroker CORBA object request broker. HP plans to integrate VisiBroker in its Smart Internet Usage (SIU) 2.0 solution for Internet Service Providers. Financial terms of the deal were not disclosed.

SIU 2.0 gives service providers and enterprise-network managers a consolidated, Internet data-mediation platform for billing, tracking, and data mining of subscriber utilization of network resources and services.

Inprise's VisiBroker for Java and VisiBroker for C++ CORBA object request brokers are designed to facilitate the development and deployment of distributed enterprise applications that are scalable, flexible, and easily maintained.

Inprise Previews Products to Support Enterprise Applications

New York, NY — Inprise Corp. announced the beta release of the next version of its Inprise Application Server, as well as the latest releases of AppCenter and VisiBroker.

Inprise Application Server combines the benefits of EJB and CORBA. With VisiBroker, customers can create e-business applications that can handle the high volumes of transactions required for doing business on the Web. The Inprise Application Server also provides comprehensive support for the Java 2 Platform, Enterprise Edition (J2EE).

Inprise AppCenter is designed specifically to manage distributed applications, allowing customers to manage from the application level rather than from the hardware perspective. It provides a software-based centralized management and control for distributed applications, running on multiple hardware and software platforms across multiple locations. This new version of Inprise AppCenter will support the management of EJB applications and will be optimized for the Inprise Application Server.

Inprise's VisiBroker is designed to facilitate the development and deployment of distributed enterprise applications based on industry standards that are scalable, flexible, and easily maintained.

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

Inprise Centralizes European Customer Support Operation

Scotts Valley, CA — Inprise Corp. announced it will centralize its European customer support operation in Amsterdam to provide round-the-clock customer support throughout Europe. The announcement follows the expansion of Inprise's European Professional Services Organization (PSO) to provide a center of excellence in CORBA and Application Server consulting skills.

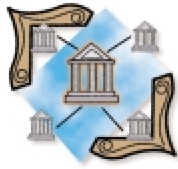
Inprise is establishing a Worldwide Customer Support Group with three major support centers around the globe: Scotts Valley, United States; Singapore; and Amsterdam, the Netherlands. It will operate as a separate entity from the PSO to ensure that customers have access to a dedicated expert resource at all times. Charles Odinot will head the European operations.

The group will provide support across the whole range of Inprise's Borland development tools and enterprise integration and database products, and will service customers in Europe and worldwide.

Customers will be able to access customer support through local call numbers in all major countries, and customer queries will be routed and handled directly through the Amsterdam center. Support services will also be provided via e-mail and through the Inprise Web site.

For more information, including customer service phone numbers worldwide, visit http://www.inprise.com or http://www.inprise-europe.com.

table of CONTENTS



By John Simonini

Embedded Forms

Putting Visual Inheritance to Work

A tabbed notebook presents information to the user in an organized fashion (see Figure 1). Unfortunately, this interface complicates the coding effort by mixing page navigation logic with business application logic. Every page presented to the user introduces more business application logic concentrated into one place — making maintenance issues a growing concern.

Embedding forms in the pages of a notebook is a technique that allows individual forms to contain discrete business logic, while another totally separate form contains the page navigation and common function logic. By dividing the logic into manageable units of page navigation and user interface, the following advantages are realized:

- Improved team development efforts.
- Improved memory management.
- Improved code reuse.

Embedding forms will give the user the perception of working with one form containing a notebook full of controls. Only the developer realizes that each page in the notebook is a separate form, allowing for improved team development efforts. Two descendant classes of *TForm* are created to accomplish the effect: *TEmbeddedNotebook* to manage page navigation, and *TEmbeddedForm* to handle user interface.

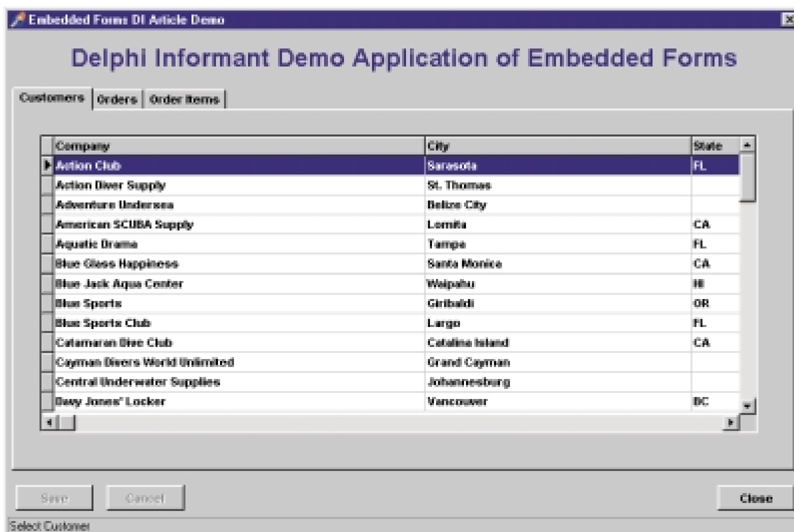


Figure 1: The example tabbed notebook application at run time.

(All source code referenced in this article is available for download; see end of article for details.)

TEmbeddedForm is a pure virtual descendant of *TForm* and is used to represent each page in the notebook. The abstract methods within *TEmbeddedForm* provide the functionality needed to support generic behaviors, such as save, cancel, and print, invoked from *TEmbeddedNotebook*. *TEmbeddedNotebook* is an abstract descendant of *TForm* that contains a *TPageControl* and several *TBitButton* objects used to manage a collection of *TEmbeddedForm* objects.

To embed a form within a *TPageControl*, follow these steps (see Figure 2):

- Eliminate the embedded form's border.
- Set the embedded form's parent to the current page.
- Fit the embedded form to the page.
- Show the embedded form modeless.

Embedding a form occurs only when the user selects a tab on the notebook. No embedded form is loaded into memory until selected by the user, resulting in improved memory management.

Divide and Conquer

Tabbed notebooks divide user interface elements into logical groups to make understanding easier for the user. Embedded forms do the same for developers, delivering on the promise of improved team development. Dividing or structuring code into logical groups is a key factor in writing maintainable code.

The *TEmbeddedForm* class has five abstract methods: *InitializeForm*, *Print*, *SaveForm*, *CancelForm*, and *CloseForm*; and one property, *CanPrint* (see Figure 3). These form the functional interface between the classes *TEmbeddedForm* and *TEmbeddedNotebook*.

InitializeForm is a procedure invoked every time the user selects a page. *InitializeForm* handles any setup logic, such as performing calculations or queries before displaying the form.

Print is a procedure that will execute whatever method of printing the developer has selected. The *CanPrint* property is used by the *TEmbeddedNotebook* to disable the user's ability to invoke the print method when no print functionality is needed.

SaveForm is a function that returns True if the save was successful. *SaveForm* is invoked when a user selects **Save**. If *SaveForm* returns False, the page is focused and the user should be prompted to fix the condition that is causing the save to fail (it could be that the embedded form is failing an edit or information is missing).

CancelForm is a function that returns True if the cancel was successful. *CancelForm* is invoked when a user selects **Cancel**. In the event *CancelForm* returns False, the page is focused and the user should be prompted to fix the condition that is causing the cancel to fail.

```
// Size, position, initialize, and show the form
// associated with the page.
procedure TFormEmbeddedNoteBook.ShowForm(
  Form: TEmbededForm);
begin
  Form.BorderStyle := bsNone;  // Eliminate border.
  // Set Parent as the active page.
  Form.Parent := PageControl1.ActivePage;
  Form.Align := alClient;       // Make form fit the page.
  Form.InititalizeForm;         // Display form.
  SetButtons(Form);             // Call to set print btns.
  Form.SetFocus;
end;
```

Figure 2: Embedding one form within another.

```
// Embedded form pure virtual class definition.
TEmbeddedForm = class(TForm)
protected
  FCanPrint: Boolean; // True if form has print capability.
public
  property CanPrint: Boolean read FCanPrint default False;
  // Housekeeping logic.
  procedure InitializeForm; virtual; abstract;
  procedure Print; virtual; abstract;  // Print form.
  // Save data changes.
  function SaveForm: Boolean; virtual; abstract;
  // Cancel data changes.
  function CancelForm: Boolean; virtual; abstract;
  // Close the form.
  function CloseForm: Boolean; virtual; abstract;
end;
```

Figure 3: The *TEmbeddedForm* class definition.

```
// Return embedded form based on the notebook page passed.
function TfrmDiDemo.FindEmbeddedForm(aPage: Integer):
  TEmbeddedForm;
begin
  case aPage of
    efCustList: Result := TfrmCustomer.Create(Self);
    efOrders:   Result := TefOrders.Create(Self);
    efOrderItm: Result := TefOrderItems.Create(Self);
  else
    Result :=  nil;
  end;
end;
```

Figure 4: Loading an embedded form.

CloseForm is a function that returns True if the user has handled all pending changes. *CloseForm*, as can be seen from the supplied source code, is best utilized by calling *CanCloseQuery*, passing in a Boolean variable. Nominally, *CloseForm* should be used to check for any pending changes. If changes exist, the user should be prompted for the disposition of these changes.

The most effective way to implement *TEmbeddedForm* is to subclass an intermediate class. The intermediate class will implement the generic rules for this project, such as behaviors for *SaveForm*, *CancelForm*, and *CloseForm*. Sub-classing all the other embedded forms to be used from the intermediate class will improve code reuse. However, for simplicity's sake, this is not the route I chose to use with the supplied demonstration project.

Implementing the class *TEmbeddedNotebook* is as simple as overriding one abstract method, *FindEmbeddedForm*, which returns a *TEmbeddedForm* (see Figure 4). Taking the time to look over the source for *TEmbeddedNotebook* reveals quite a bit of code that handles the drudgery of housekeeping.

Under the Hood

TEmbeddedNotebook is the real workhorse here. It manages a *TList* of *TEmbeddedForm* objects, as well as all the generic processing for the *TEmbeddedForm* objects. *TEmbeddedNotebook* delivers improved memory management and improved code reuse. The central focus of *TEmbeddedNotebook* is the *TList* of *TEmbeddedForm* objects named *FormList*. *TEmbeddedNotebook* handles these tasks:

- It initializes *FormList*;
- displays and hides embedded forms on page turns; and
- invokes generic processing, such as save, cancel, close, and print.

As previously mentioned, *FormList* is a *TList* of *TEmbeddedForms*. To realize improved memory management, the embedded forms won't be created until they're requested by the user clicking on their respective page. To this end, *FormList* is initialized and loaded with **nil** pointers, one for each *TTabSheet* on the *TPageControl* (see Figure 5). The reason for loading **nil** pointers is that retrieving *TEmbeddedForms* from *FormList* is dependent upon an exact correlation between the page index of the selected page and the corresponding *TEmbeddedForm*.

Showing a *TEmbeddedForm* occurs when *TEmbeddedNotebook* responds to the *TPageControl*'s *OnChange* event. *GetForm* is the function responsible for returning the correct *TEmbeddedForm* (see Figure 6).

The page index of the active form is used as the index into *FormList*. If the pointer is assigned, then *TEmbeddedForm* is returned. Otherwise, the page index of the active page is passed to

```
// Set up the PageControl to contain embedded forms.
// Create a TList to hold the TEmbeddedForm Objects.
procedure TFormEmbeddedNoteBook.InitializePageControl(
  Sender: TObject);
var
  Index: Integer;
begin
  // Turn to the first page.
  PageControl1.ActivePage := PageControl1.Pages[0];
  SetPages;  // Set Visible property for tabsheets.
  FormList := TList.Create;  // List of available forms.
  // Create placeholder for each tab.
  for Index := 0 to PageControl1.PageCount -1 do
    FormList.Add(nil);          // Empty placeholders.
  PageControl1Change(Sender);  // Show 1st form.
end;
```

Figure 5: Setup for an embedded form list.

```
// Return selected form if available;
// otherwise raise exception.
function TFormEmbeddedNoteBook.GetForm: TEmbeddedForm;
var
  PageNum: Integer;
begin
  PageNum := PageControl1.ActivePage.PageIndex;
  if not Assigned(FormList.Items[PageNum]) then
    try                          // If form doesn't exist.
      FormList.Delete(PageNum);  // Clear nil placeholder.
      // Insert new form.
      FormList.Insert(PageNum, GetEmbeddedForm(PageNum));
    except
      on E: EFormNotFound do  // New form creation failed.
        // Replace nil placeholder.
        FormList.Insert(PageNum, nil);
    end;
  // Return contents of FormList.
  Result := TEmbededForm(FormList.Items[PageNum]);
end;
```

Figure 6: Retrieving the selected embedded form.

```
// Call a routine to return the embedded form
// for a given page.
function TFormEmbeddedNotebook.GetEmbeddedForm(
  PageNum: Integer): TEmbeddedForm;
begin
  Result := FindEmbeddedForm(PageNum);
  if Result = nil then
    Raise EFormNotFound.Create('Form Not Found');
end;
```

Figure 7: Driving logic for requesting embedded forms.

```
// Process each EmbeddedForm within FormList, with the
// function provided as the parameter aFunction.
procedure TFormEmbeddedNoteBook.ProcessAllForms(
  aFunction: TFunctType);
var
  Index: Integer;
begin
  // For all the forms in the list...
  for Index := FormList.Count -1 downto 0 do
    // If form has been assigned...
    if Assigned(FormList.Items[Index]) then
      // Call the passed method with the form.
      aFunction(TEmbeddedForm(FormList.Items[Index]));
end;
```

Figure 8: Driving logic for generic processing of all loaded embedded forms.

the *GetEmbeddedForm* function, which creates and returns the correct *TEmbeddedForm* (see Figure 7).

Hiding a *TEmbeddedForm* is accomplished by responding to the *TPageControl*'s *OnChanging* event. The *OnChanging* event has a variable Boolean parameter, *AllowChange*. As a result of calling *TEmbeddedForm*'s *CloseForm* function, a Boolean will be returned that's used to set the state of *AllowChange*. The recommended practice is for *TEmbeddedForm* to perform its final editing and post all data changes here. If *TEmbeddedForm* fails an edit or a post, a message should be displayed to the user explaining how to correct the error, and a Boolean with a value of False is returned. This prevents the user from turning to another page until the error is corrected or the user cancels all pending changes.

Generic processing comes in two flavors: processing that occurs to all embedded forms currently loaded (*ProcessAllForms*) and processing that occurs only to the embedded form currently displayed (*GetForm*). Save, cancel, and close, by default behavior, fall into the "Occurs to all forms" flavor of processing. *ProcessAllForms* is the method responsible for iterating through the list of loaded forms. It's a method that accepts a procedural variable that points to a method that accepts a *TEmbeddedForm* as a parameter (see Figure 8).

```
// Print a screen dump report.
procedure TFormEmbeddedNoteBook.btnPrintClick(
  Sender: TObject);
begin
  GetForm.Print;
end;
```

Figure 9: An example of specific processing of a selected embedded form.

```
// Close all embedded forms. If all forms are closed and
// destroyed, the result is True, otherwise it's False.
function TFormEmbeddedNoteBook.CloseEmbeddedNotebook:
  Boolean;
var
  Index: Integer;
begin
  try
    Screen.Cursor := crHourglass;
    Result := True;
    Index := ClearFormList;
    if Index > -1 then
      begin
        Result := False;
        PageControl1.ActivePage.PageIndex := Index;
        Exit;
      end
    else
      FormList.Free;
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

Figure 10: Ensure all modifications have been processed before the embedded notebook is closed.

Printing falls into the other flavor of processing, which simply invokes the selected embedded form's *Print* method (see Figure 9).

Safety First

To ensure that changes the user has made are handled correctly when the *TEmbeddedNotebook* is closed, generic processing similar to "Occurs to all forms" is used. Because data integrity is of paramount importance, the user cannot be allowed to close the *TEmbeddedNotebook* until all changes have been saved or canceled by the user.

The following is the chain of events for ensuring the integrity of the changes the user has made:

- Handle *TEmbeddedNotebook*'s *OnCloseQuery* event.
- Invoke the loaded embedded form's *CloseForm* function.
- Set the value of *TEmbeddedNotebook*'s *CanClose* variable with the net result of all the embedded form's *SaveForm* function calls.

When *TEmbeddedNotebook*'s *OnCloseQuery* event is fired, it invokes *CloseEmbeddedNotebook* (see Figure 10), which invokes *ClearFormList* (see Figure 11). *ClearFormList* iterates though *FormList*, invoking the *CloseForm* function for all loaded *TEmbeddedForms*. If *CloseForm* returns True, *TEmbeddedNotebook* frees that *TEmbeddedForm*. Otherwise, the corresponding page is focused and *TEmbeddedNotebook*'s *CanClose* variable is set to False, allowing the user to remedy the situation and continue.

Great care has been taken to ensure the user cannot close the *TEmbeddedNotebook* without disposing of their changes. **Save** and **Cancel** buttons have also been provided for the user's convenience.

```
// This routine calls the ProcessAllForms with CloseForm
// method of all embedded forms. If succesful, the form is
// freed and removed from the FormList.
function  TFormEmbeddedNoteBook.ClearFormList: Integer;
var
  Index: Integer;
begin
  Result := -1;                // Set result to all deleted.
  ProcessAllForms(CloseForm); // Close all forms.
  // For all the forms in the list.
  for Index := FormList.Count -1 downto 0 do
    // If any forms are still loaded...
    if Assigned(FormList.Items[Index]) then begin
      // Form cannot be closed, return the index.
      Result := Index;
      Exit;  // Exit the for loop.
    end;
end;
```

Figure 11: Clean up of loaded embedded forms.

```
// Light data-handling function buttons.
procedure TFormEmbeddedNoteBook.SetDataButtons(
  State: Boolean);
begin
  btnSave.Enabled   := State;
  btnCancel.Enabled := State;
end;

// Handle buttons to notify user of the state of the data.
procedure TFormEmbeddedNoteBook.DSChanged(
  var Message: TMessage);
begin
  SetDataButtons(Message.lParam = wmpTrue);
end;
```

Figure 12: Data state-handling routines.

The Save and Cancel buttons' *Enabled* property is set using a user-defined Windows message WM_REMOTEDSCHANGED. Setting the *lparm* parameter of *TMessage* with a 1 or 0, depending on the state of the data or the presence of updates, communicates the state of the data (see Figure 12). The definitions needed for WM_REMOTEDSCHANGED are in the TYPES.PAS unit.

Polly Want a Cracker?

Polymorphism allows a descendant class to operate differently than its parent class for the same method call. For any framework to be usable, it must allow for its default behaviors to be changed. In nature, this is called evolution. Object-oriented design should always allow for the evolution and maturation of classes to adapt to changing business rules.

Although every attempt has been made to default to usable behaviors, "No plan ever survives contact with the enemy." Examples of default behavior are the processing for save, cancel, and close functions, which assume the user wants to operate on all pages generically.

The two most likely behaviors that will need to be changed are: Changing tabs forces the user to resolve data changes (save or cancel); and, function button clicks process all pages or a single page.

At the initial design of embedded forms, certain assumptions were made. The first assumption was that the information contained on the embedded forms was interrelated. Assuming the information was interrelated, the design stance was to force the user to either save or cancel all pending changes when they select a new tab. The second assumption was that when the user clicked Save or Cancel, the action would take effect in all loaded forms.

Altering the behavior of the embedded notebook when pages are turned can be accomplished in the following manner. In the *TPageControl*'s *OnChange* event, the *CanChangeTabs* function is invoked. *CanChangeTabs* returns the result from calling the *TEmbeddedForm*'s *Close* function. By overriding this behavior to return True, the pages of the notebook can be changed without having to resolve pending data changes.

Altering the function button behaviors of *TEmbeddedNotebook* can be accomplished in the following manner. When the Save or Cancel button is clicked, *ProcessAllForms* is invoked passing the *Save* or *Cancel* methods, respectively. This behavior can be changed by overriding the *btnSaveClick* and *btnCancelClick* event handlers. Both data state buttons, *btnSave* and *btnCancel*, invoke *ProcessAllChanges* by passing a procedural variable. If *GetForm* were called in its place, the result would be the currently focused *TEmbeddedForm*, and its *SaveForm* or *CancelForm* function could be invoked.

It's very relevant to point out that, because *TEmbeddedForm* is a pure virtual class, it can very nicely be implemented as an interface. Implementing this scheme as an interface will allow pre-existing forms to be leveraged as embedded forms. (For more information regarding interfaces, see Cary Jensen's article "Interfaces: An Introduction" in the April, 1998 *Delphi Informant Magazine*.)

Conclusion

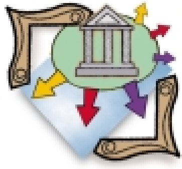
Embedded forms provide the following business advantages as stated in the introduction of this article: improved team development by allowing multiple developers to work simultaneously on what the end user perceives as one form; improved memory management by only loading those embedded forms users have selected; and, improved code reuse through the housekeeping and data integrity code supplied in the base class of *TEmbeddedNotebook*.

The embedded forms described in this article were designed as a team development technique, and as such they came to life through the joint efforts of a team. I would like to credit and thank the following people for all their efforts in making embedded forms a successful technique:

- Anil Thomas for his implementation of the procedural version of embedded forms.
- Mike Carkitto for his immeasurable help in implementing the object-oriented version of embedded forms.
- Ed Hauswirth, my friend and mentor, and the person who proved to me, much to my chagrin, that C++ is not the final word in PC development. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200001JS.

John Simonini lives in the beautiful Pocono Mountains of Pennsylvania, with his lovely wife, two adorable sons, and two goofy dogs. John is a Senior Consultant in the employ of Source Technology Corp., and can be contacted by e-mail at wizware@epix.net or jsimonini@sourcetechcorp.com.



By Michael J. Leaver

Request Threads

One Solution to Resource-sharing Problems

In an ideal world, a server system would have unlimited resources available to an unlimited number of clients. That world doesn't exist just yet. For example, a database server can only support a limited number of client connections. The database, in this case, is the limiting resource.

One way around this problem is the use of Transaction Processing Monitors, e.g. Microsoft's Transaction Server (MTS) or BEA's Tuxedo. This provides for three-tier distributed processing, with the TP Monitor taking care of resource sharing and other important things, such as transactions and data integrity. Indeed, Delphi 4/5 allows for easy creation of MTS objects, but MTS has its drawbacks and is not the answer for all situations. Other TP Monitors may be overkill for a small system, or simply too expensive in licensing, development, and support costs. There's also a steep learning curve to consider.

This article describes a resource-sharing method implemented using Delphi 4 Client/Server Suite and Microsoft's Distributed Component Object Model (DCOM), with the resource being a database (the techniques apply equally well to Delphi 5). The method could equally be applied to any modern programming language and operating system that supports threads and remote procedure calls. The resource need not be a database; it could be a file server, printer, etc. The method described in this article is based on one used within a workflow-imaging trade finance product called Prexim, developed by China Systems.

One-to-one Connections

It's a simple task to create a multi-threaded DCOM server using Delphi 4 Client/Server Suite. Such a server is capable of providing services to a potentially unlimited number of clients, with the basic limiting factors being memory and processing power.

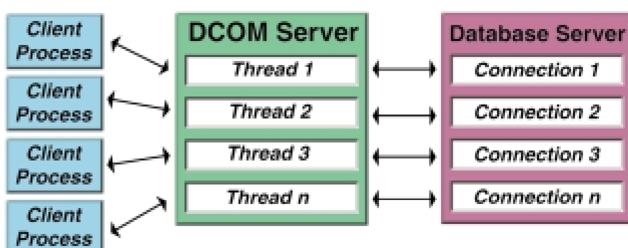


Figure 1: One-to-one connections.

Servers are never that simple, however. For example, they usually rely on services provided by other servers, such as a relational database server. Connections to these underlying servers are generally expensive in terms of resource usage, and must, therefore, be used sparingly. It isn't practical for each thread of a server to own a dedicated connection to another database server. For example, if there are 100 clients connected to a server, there will be 100 corresponding connections to the underlying database server (see Figure 1).

This is clearly impractical, and in many cases impossible. It's a huge waste of resources, as those database connections are unlikely to be used all the time by all the clients. A better solution would be for clients to share these precious resources.

Threads

How do you share a connection to a database server? A simple DCOM server would create a dedicated connection to the database server for each client connection. What's needed is a pool of connections that all clients can share. This sharing must be transparent to the clients who need not be aware of such complexities of implementation.

One solution is for the DCOM server to open a number of connections to the database server when it starts. The number of connections may be configurable and fixed once the DCOM server starts. Preferably, it should be dynamic and increase or decrease on the fly depending upon the load, i.e. load balancing.

Now when a client connects to the DCOM server, it doesn't create a new connection to the database server. Instead it asks one of the threads created earlier to do the processing and return the results, which in turn will be passed back to the client (see Figure 2).

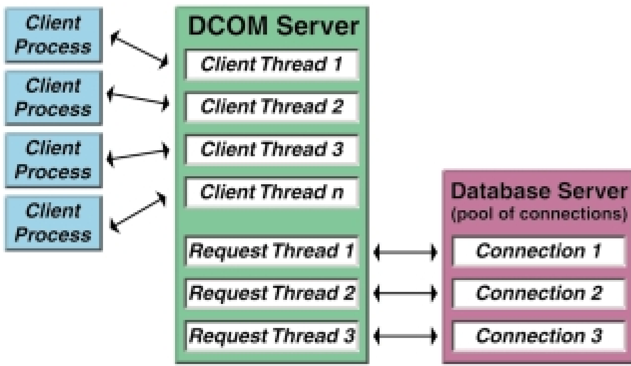


Figure 2: Request thread connections.

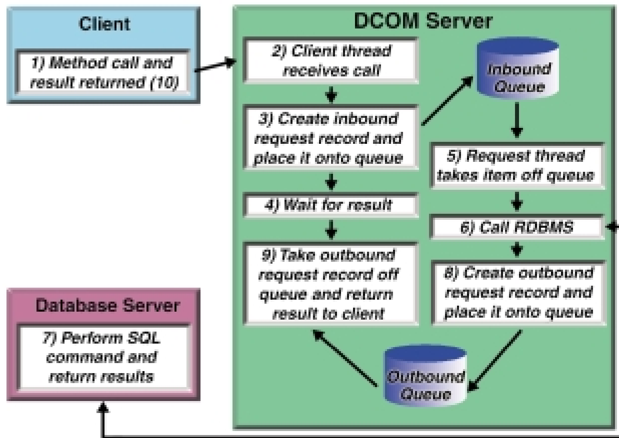


Figure 3: A diagram of the communication between the client and request threads.

There are two types of threads being discussed here. When a client connects to a multi-threaded DCOM server, the server spawns a thread. This thread, which we'll call a "client thread," is responsible for servicing that client. Therefore, each client connection to a DCOM server has its own client thread. We'll call the threads that are spawned when the server initializes and created before any clients connect to the server "request threads." It's these limited request threads that are shared by the potentially unlimited client threads.

The problem now is how to communicate with those request threads. We need an efficient method of inter-thread communication.

Communication

To get straight to the point, one answer is to use good-old-fashioned First-In-First-Out (FIFO) queues. This avoids starvation, because no priorities are involved. Because we're dealing with threads, synchronization of access to these queues is critical to avoid difficult-to-find data corruption bugs. Using mutex objects, semaphores, or many of the other objects available in modern operating systems can solve this. Luckily for us, Delphi provides a simple interface to these potentially difficult-to-use operating system objects.

Communication between these threads works in the following order. The DCOM server starts and spawns one or more request threads. Each request thread has its own connection to the database server, and when a client connects to the DCOM server, a client thread is spawned. The client thread can't talk directly to the database server because it has no connection to it. For the client to talk to the database server, it must ask a request thread to perform that task on its behalf. The client thread does this by creating an inbound request record, and placing it onto the inbound request queue.

A request record describes what processing the client thread wants the request thread to perform. The client now waits for the result (as we will discuss later, it need not block on the wait). One of the request threads takes the item out of the inbound queue and performs the task, which involves talking to the database server via the connection the request thread owns, e.g. the BDE.

The results are placed into an outbound request record and put onto the outbound request queue. An outbound request record contains the results of the processing, and any other relevant information, such as error messages. The client thread, which has been waiting for the result, takes the outbound request record off the outbound queue. After any further processing, the results are returned to the calling client (see Figure 3).

Spawning Request Threads

When the DCOM server starts, it must spawn one or more request thread for the connections to the database server. More threads give more throughput, but increase memory use. Also, at some point the throughput from having more threads decreases. Ideally, some method of load balancing should be used, but that's beyond the scope of this article.

The reqint.pas unit contains a function named *RequestThreadsStart* that must be called by the DCOM server when it starts. This function initializes synchronization objects and spawns the required number of request threads. It allows for initialization failure of one or more request threads, i.e. five requests may be asked for, but, perhaps, only two request threads can actually be created. The minimum number required is specified as an argument to the function.

The first thing *RequestThreadsStart* does is create the inbound and outbound queues, as well as the semaphores used to coordinate reading from and writing to those queues. It would be very inefficient to poll the queues waiting for new records to arrive, so a more efficient method using semaphores is used instead.

Making Requests

The client threads must create a request record detailing what is required of the request threads. This record must then be placed onto the inbound queue. *Method1* and *Method2* demonstrate how simple this is (see Figure 4). An inbound request record is filled with the request details. Then *g_RequestInterface.RequestSync* is called to send the request to the request threads and wait for the result.

The *RequestSync* function is an interface for *RequestIn* and *RequestOut*, where *RequestIn* passes the request onto the request threads, and *RequestOut* checks to see if the results are available for collection. If *Method1* or *Method2* wanted to make an asynchronous request, they would call *RequestIn* and *RequestOut* separately without calling upon *RequestSync*.

How does a client thread know which results to collect from the outbound queue? By taking results with the correct result handle. When a request is made, it's given a unique result handle that is simply a unique number (unique for each server process), the current date and time, and a random string of characters. The date and time are stored in the result handle, so the server knows if a result has timed out. A random string of characters is placed on the end to reduce the chance that another client

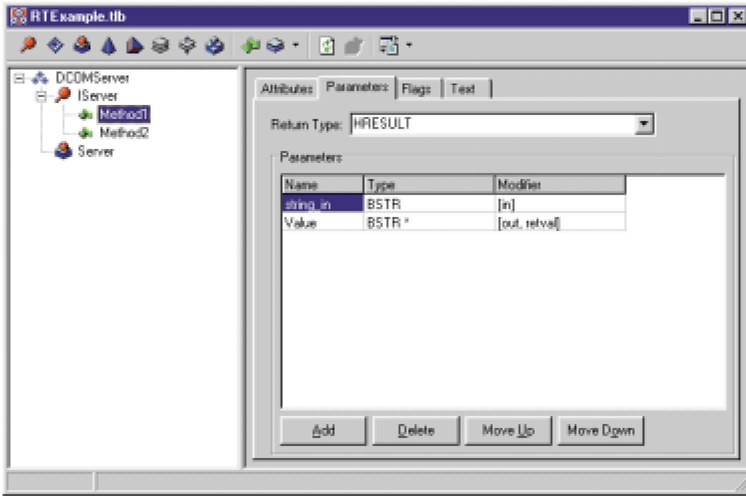


Figure 4: The example DCOM server displayed in the Type Library editor.

will try to steal another's results. Remember that the result handle may be used asynchronously at a later point by the client, so it makes sense to make it a single string instead of multiple strings.

As explained earlier, semaphores are used to signal both client threads that results are available for collection, and request threads that a request has been made. *g_SemInQ* is the inbound semaphore, and *g_semOutQ* is the outbound semaphore.

Processing Requests

TRequestThread.Execute contains a simple loop that shows what each request thread does. First it takes the first request record it can get from the inbound queue (via the function *TakeFromQueue*). Next it processes the item, places the results onto the outbound queue, and frees the memory used by the inbound request, as it's no longer required (see Figure 5).

Collecting Results

The *TRequestThread.RequestOut* member function is used to collect the results from the request threads. Supplied with a request handle (previously created when the client thread made a request via *RequestIn*) and a place to store the results, it first checks to see if the request has timed out. If a request is over a certain age, that request's results can't be collected because they will have been garbage collected and removed from the outbound queue. If the results weren't purged, the outbound queue would eventually grow too large if the client threads didn't collect their results. Remember that if an asynchronous request had been made, the client may be on another machine. The client program may exit before collecting the results, or the network connection may go down. There are many reasons why results may never get collected. Deciding if a result may not get collected isn't possible.

Having resolved that the request isn't too old, the outbound queue semaphore (*g_SemOutQ*) is checked. If it's signaled, then there are results in the queue, else the queue is empty and hence the results aren't available yet. The queue is searched for a matching request that hasn't been handled, i.e. another thread hasn't already taken the results. Because the list is locked, there is no need for mutexes or other thread synchronization methods when it's searched. Only one request thread can search the outbound queue at a time. A result can't be taken from the queue during the search, so its handled state is flagged as being True.

If the result was found, it can then be removed from the queue and returned to the client thread. When no result is found, the outbound queue semaphore must be re-signaled to indicate that there are still pending results. Without re-signaling, other request threads would assume the queue is empty.

Asynchronous

Imagine an extension to Delphi that allowed asynchronous function calls without the programmer needing to worry about how it's actually implemented. Figure 6 shows that the asynchronous function can be called as per any other function with no special considerations.

Control returns to the procedure before the function has completed and returned its result. The procedure would only block waiting for the result if it tried to use one of the values returned, or set by the asynchronous function, e.g. *AsyncReturn* or *out_arg*. Wouldn't that be a very useful extension to any language?

With request threads, it's possible to do this now within a process and outside of the process. For example, asynchronous COM calls are possible with request threads. The client would call the COM method as usual, but instead of getting its result, the client would get a unique handle. At a later point, the client would call the COM method again with the handle supplied earlier and would now get the results back. The client who originally made the call need not supply the handle. Another client process on another machine could collect the results as long as it used the correct handle.

```
// Take out of g_InQ and place items in g_OutQ.
while not Self.Terminated do begin
  inq := TakeFromQueue;
  if Self.Terminated then
    Break;
  // Process it.
  outq := ProcessItem(inq);
  if outq = nil then
    Continue;
  // Add it to the out Q.
  AddToQueue(inq^.request_in.rhandle, outq);
  Dispose(inq);
end;
```

Figure 5: *TRequestThread.Execute* showing what each request thread does.

```
function AsyncExample(const in_arg: Integer;
  var out_arg: string): Boolean; Async;
begin
  // Do some processing. Set the value of out_arg.
  // Return some result.
  Result := True;
end;

procedure UsingTheExample;
var
  AsyncReturn: Boolean;
  SomeString: string;
begin
  // Call our async function.
  AsyncReturn := AsyncExample(123, SomeString);

  // At this point we can continue doing something
  // else while that function does whatever it does.

  // Let's get the result.
  if AsyncReturn then
    AnotherString:= 'The string result is:' + SomeString
  else
    AnotherString:= 'Error';
end;
```

Figure 6: Calling an asynchronous function with no change in method.

Unfortunately, the client must be aware that the call is asynchronous and must store the handle. There is also the possibility that the client will never collect the result, so some garbage collection must be performed periodically within the server. Security must also be taken into consideration so that clients don't try to steal results meant for others.

Improvements

There are many ways to improve upon the simple method shown in this article, and there are still many more things that haven't been described.

First, it must be said that the example code isn't what it should be and can clearly be improved upon. Delphi is an object-oriented language, but the example code given doesn't make much use of its object-orientation. A generic request threads class would be a very powerful class indeed.

Request threads can span across multiple calls. For example, a client thread may need to make multiple calls to the same request thread because that request thread has state, e.g. a cursor connection to the database server, or perhaps even an open database transaction. This is possible within request threads simply by locking a request thread to one client thread via a unique handle. This handle can be passed each time the client thread needs to use a specific thread. Locking a thread doesn't necessarily mean other client threads can't use it while it's idle. For example, if the request thread is locked because it has a cursor open, it doesn't mean other client threads can't use it to read other tables as long as the cursor isn't affected. It all gets complicated and other issues arise, such as unlocking threads, orphaned request threads, etc. For that reason, it's not described in this article.

Because requests are defined within records and are used within queues, it allows for easy recording of requests for later playback. For example, a database transaction could be performed by recording requests and storing them in a list instead of sending them to request threads one by one. Later, the entire list could be sent to a request thread for immediate processing as a single block of requests. The block of requests could be processed within one database transaction within the request thread.

The example code also doesn't show how to ensure that the DCOM server has completed initialization before client calls are accepted. If a client call is received before the queues have been set up, for example, then problems may occur. Request records may need to contain a large amount of data, as they need to support all types of requests. To reduce memory usage, variant records should be used, or perhaps even variant variables.

Request threads beg for tuning parameters, e.g. TTL (time-to-live) values for request in the queues, time-out values when waiting for results, optimal number of threads to use, etc. This hasn't been given in the example code, but it's not difficult to add. What's more difficult is automatically creating the best values for a specific implementation.

Conclusion

Request threads allow a simple way to make asynchronous calls to shared resources. Delphi allows easy creation of such a method because of its strong support for COM, threading, and access to the Win32 API.

The source code, which is available for download (see end of article for details) includes a demonstration COM out-of-process server and a client test program. To enable threading on the server side, and so allowing the server to process multiple client calls simultaneously, the Borland-supplied source code thrddcf.pas has been employed. Without this, the client's requests would be queued by the server and processed serially, one by one. This file may be found in the directory Demos\Midas\Pooler. Because of the multi-threaded nature of the server, it can't be used with Windows 95. It should be used on Windows NT only, whereas the client program can be used on Windows 95 or NT. Before using the client program, the server must be registered. This can be done by executing the server with the command line argument /regserver (or more easily from the Delphi IDE). Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200001ML.

Michael J. Leaver is the Product Manager for the China Systems Prexim system. China Systems is a world leader in trade finance technology. Prexim is a workflow imaging solution for trade finance processing written using Delphi 4 C/S. Michael can be reached via e-mail at MJLeaver@csi.com.

By Alan C. Moore, Ph.D.

A Multimedia Assembly Line

Part II: Generating a Component

Last month we began building a Delphi sound expert (Wizard) that produces multimedia components. Based on a simpler expert, SndExprt 1.0, it adds sound-playing (.WAV file) capabilities to any component. We also dealt with the expert structure and the user interface. Before we take a detailed look at the code-generating engine, let's review that structure.

The sound-generating property, *SoundPlayEvents*, and its subproperties are arranged in a hierarchy. The main property added to each component, *SoundPlayEvents*, contains a series of properties, *XSoundPlayEvent*, where X is the name of one of the sound events supported. These properties, such as *ClickSoundPlayEvent*, *EnterSoundPlayEvent*, and *ExitSoundPlayEvent*, contain their own subproperties. Those that manage the playing of sounds are shown in Figure 1.

With *SoundSource*, the flags in *fdwSound* (SND_ALIAS, SND_FILENAME, and SND_RESOURCE) determine how the *Filename* property will be interpreted as a file name, a resource identifier, or an alias for a system event. *Filename* is a string holding the name of the file, resource, or other location. All of these subproperties are included in the *?SoundPlayEvent* property, where "?" is the name of one of the sound events.

Property	Description
SoundSource	Type of sound source to which the *FileName* parameter points; enumerated type: *TSoundSource* (*ssAlias*, *ssAlias_ID*, *ssFilename*, *ssMemory*, *ssResource*, *ssNone*). The 16-bit version with *sndPlaySound*: (*ssFilename*, *ssMemory*). It's associated with *PlaySound*'s *fdwSound* flags in *fdwSound* (SND_ALIAS, SND_FILENAME, and SND_RESOURCE).
Filename	A string associated with *PlaySound*'s first parameter, *lpszSound*, a pointer to a null-terminated string that specifies the sound to play.
Yield	A Boolean property that determines whether the requested sound can interrupt a sound that's playing.
hmodValue	Of type HMODULE. Associated with the second parameter (*hmod*) of the *PlaySound* function. It's the handle of the executable file (DLL or EXE) that contains the resource to be loaded. Unless SND_RESOURCE is specified, this parameter must be set to **nil**.
?SoundPlayOption	Of type *T?SoundPlayOption*, where the "?" represents one of the event classes, e.g. *Click*, *Enter*, or *Exit*. All ways of playing a .WAV file (synchronously, asynchronously, in a loop) are supported.

Figure 1: Subproperties of *XSoundPlayEvent*.

How do we program our expert to generate a component with this new composite property? You'll notice we use several types and data structures from the unit, SndTypes.pas, introduced in last month's installment. Now let's explore some of the details.

Programming the Expert Engine

In writing an expert, its engine is where the real programming generally takes place. It's also where I had to write most of the new code for the updated expert. In both the original and revised experts, I found it helpful to have a prototype of the type of component the expert would generate to which we could refer. That prototype included all the new functionality and properties we've been discussing.

After you've made all your choices and clicked Finish, the code for your new component is generated (CompGen1.pas) and opened in the Delphi IDE editor. There you can look at the code, edit it, or save it. Let's concentrate on the component-building code. Here is the most important routine in the engine:

```
procedure WriteALine(StringIn: string);
begin
  TempMemoStr.Add(StringIn);
end;
```

TempMemoStr is a StringList that collects all of the code to be saved in a file. Because we call this method for every line of generated code, the

CompGen1.pas unit is now too large to include in its entirety. Instead, we'll highlight some of the more interesting routines. Compared to the earlier version, the *WriteTypes* method is expanded considerably. It writes the support types and a specific Sound Option Type for each enabled event. In the *WriteSoundOptions* method, lines such as the following read the data collected from running the expert, and produce the proper types by using strings associated with the event to name them:

```
if spPlayReturn in
     SoundFactorsArray[j].SoundPlayOptions then
  AddSoundOption(EventPrefixes[j] + 'spPlayReturn');
```

The *EventPrefixes* array stores prefixes for each event with an appropriate name: "ck," "dd," and "do" are prefixes for the *Click*, *DragDrop*, and *DragOver* events, respectively.

There are other significant changes in this new version that are necessary to create the new property structure in the target components created. In the previous version, we simply added two properties to each generated component. Here we add just one, but it's a monster with multiple class properties, each of which has a number of subproperties. Figure 2 shows the new property from a newly produced component in the Object Inspector with a few of these subproperties shown.

The new Sound Option Types we just discussed provide the basis for the most important subproperties used under each event subproperty. The work of writing the main component class and the host of event subclasses is done in the *WriteNewClasses* procedure. There are several nested procedures within this method, including *WriteMainClass*, which writes the new component class, and *WriteSoundClasses*, which writes a new class for each Sound Event. Let's examine that procedure, as it demonstrates several interesting issues. Pay particular attention to the nested procedure within *WriteSoundClasses*, *WriteSoundClass*. This latter procedure, shown in Listing One (on page 17), writes each of the individual sound event classes.

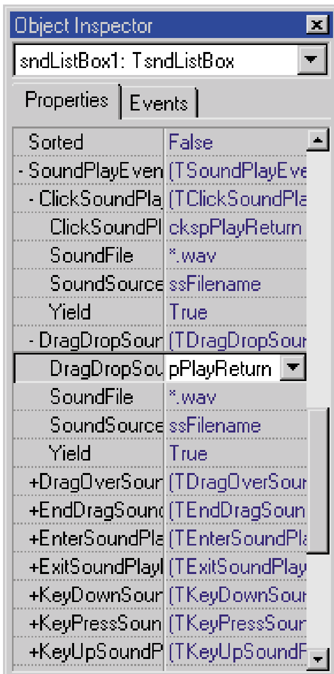


Figure 2: Main property with a few of these subproperties in a new component shown in the Object Inspector.

As you can probably guess, this one procedure writes the bulk of the new property with all of its subproperties. The variable or changing portions of this code depend upon a number of arrays of strings: *BasicEventNames*, which includes the same event names used with the check boxes (*Click*, *DragDrop*, etc.); *EventNames*, which is the name of the actual method used to fire the event (*DoEnter* instead of *Enter*, for example); and a smaller array, *ParametersArray*, which contains the strings used for parameters when the inherited methods are called in the overridden event methods.

In this code, and in code we examined last month, *EventMax* is used a great deal. This constant represents the number of events supported minus one, because all of the iterations are zero-based. It's used in most of these arrays, as well as in most iterations. By using this constant, it will be easy to change this code by adding or deleting supported events. For example, take a look at *WriteConstructors*. A procedure nested within it, *WriteSoundClassesConstructors*, iterates through all possible events and writes a constructor for each one the user has selected. Similarly, the *WriteEventHandlers* procedure writes event handlers for those events.

The code in Listing One may seem like a lot of code for producing a series of rather simple classes, but it does more than just write all of these classes. Let's start with the individual classes. First, we need to determine which events will be sound-enabled by iterating through the check-box flags with:

```
for j := 0 to EventMax do
  if EventCheckBoxArray[j]= ecbsChecked then
    ...
```

Then we build an array that contains information on which events are sound enabled. We use the *NewClasses* variable to monitor how many are actually used. As we iterate through all of the possible events, we build that array:

```
NewClassArray[NewClasses] := BasicEventNames[j];
```

BasicEventNames contains the basic names of the enabled events. We use this array later to write the implementation code. The code that follows simply writes the remainder of the class declaration. When we reach the last line, we call a nested procedure with:

```
WriteSoundPlayEventsClass;
```

This procedure writes another class — the main class that contains all of the individual sound-enabled events and their subproperties. With this procedure, we make good use of the *NewClassArray* to create a property for each sound-enabled event. Figure 3 shows the generated code for this class.

You'll note that we have just one event — the *Click* event. We could just as easily have three or 12. We must create one more class — the component class. Some of the information for this class comes from the first page of our expert (see last month's article), its class name, and its ancestor class. Other than that, the code is identical for any component created with this expert. Figure 4 shows a main component's class declaration.

Upon casual examination, this code is deceptively simple. Keep in mind, however, the many subproperties within the *SoundPlayEvents* property we just discussed. If you enable many

```
TSoundPlayEvents = class(TPersistent)
  private
    FClickSoundPlayEvent : TClickSoundPlayEvent;
  public
    constructor Create(AOwner: TComponent);
  published
    property ClickSoundPlayEvent : TClickSoundPlayEvent
      read FClickSoundPlayEvent write FClickSoundPlayEvent;
end;
```

Figure 3: The *TSoundPlayEvents* class in a generated component.

events and then view all of the details, it could take up most of the space in the Object Inspector (again, refer to Figure 2).

Next, we need to create constructors for these classes. That chore is rather straightforward, so we'll skip a detailed discussion of it. One issue worth mentioning is that we need a way to handle the options on our pop-up dialog box concerning defaults (.WAV files or Yield). In the individual event classes, we handle that situation with the following code, using data collected in our expert:

```
if SoundFactorsArray[j].WavFileDefault then
  WriteALine('  FSoundFile := ''' +
    SoundFactorsArray[j].DefaultWavFile + ''';')
else
  WriteALine('  FSoundFile := ''*.wav'';');
WriteALine('  FSoundSource := ssFilename;');
if SoundFactorsArray[j].Yield then
  WriteALine('  FYield := True;')
else
  WriteALine('  FYield := False;');
```

```
TSoundButton = class(TBitBtn)
  private
    FSoundPlayEvents : TSoundPlayEvents;
  public
    constructor Create(AOwner: TComponent); override;
    procedure Click; override;
  published
    property SoundPlayEvents : TSoundPlayEvents
      read FSoundPlayEvents write FSoundPlayEvents;
end;
```

Figure 4: A new component's main class declaration.

```
procedure TSoundButton.Click;
var
  fdwSoundValue : DWORD;
  AhmodValue    : HMODULE;
  AFileName : array [0..255] of Char;

  function GetfdwSoundValue(
    ASoundPlayEvent: TSoundPlayEvent): DWORD;
  begin
    case  ASoundPlayEvent.SoundSource of
      ssFilename: Result := SND_FILENAME;
      ssMemory: Result := SND_MEMORY;
      ssAlias: Result := SND_ALIAS;
      ssAliasID: Result := SND_ALIAS_ID;
      ssResouce: Result := SND_RESOURCE;
    end;
  end; { GetfdwSoundValue }

begin
  with SoundPlayEvents do begin
    fdwSoundValue := GetfdwSoundValue(ClickSoundPlayEvent);
    if (fdwSoundValue = SND_RESOURCE) then
      AhmodValue := ClickSoundPlayEvent.hmodValue
    else
      AhmodValue := 0;
    StrPCopy(AFileName, ClickSoundPlayEvent.SoundFile);
    if ClickSoundPlayEvent.yield then
      fdwSoundValue := (fdwSoundValue OR SND_NOSTOP);
    case ClickSoundPlayEvent.ClickSoundPlayOption of
      ckspPlayReturn:
        PlaySound(AFileName, AhmodValue, fdwSoundValue or
                  snd_Async or snd_NoDefault);
      ckspPlayNoReturn:
        PlaySound(AFileName, AhmodValue, fdwSoundValue or
                  snd_Sync or snd_NoDefault);
      ckspPlayNoSound:   ;
      ckspPlayContinuous:
        PlaySound(AFileName, AhmodValue, fdwSoundValue or
                  snd_Async or snd_Loop or snd_NoDefault);
      ckspEndSoundPlay:
        PlaySound(nil, AhmodValue, fdwSoundValue or
                  snd_Async or snd_NoDefault);
    end; { case }
  end;
  inherited Click;
end;
```

Figure 5: An event handler override to play a sound.

Even more interesting to examine is the method that actually does the work, in this case the *Click* method (see Figure 5).

There are various steps involved here. First, we use the local function, *GetfdwSoundValue*, to determine the sound source type (*SoundSource*), and return the corresponding flag. We then plug that flag into the last parameter of the *PlaySound* function along with other appropriate flags. Unless we use the SND_RESOURCE flag, the *hmodValue* is set to zero. One option (that we don't enable here) concerns default sounds: *snd_NoDefault* is always set. Of course, we could add an additional subproperty. That will have to wait for the next version.

Conclusion

We haven't investigated every detail of the code-generating engine. However, the other procedures use similar approaches and these same string arrays to accomplish their purpose. This version represents a major enhancement of the previous one. We've extended the expert to handle other standard events besides the *Click* event. Unfortunately, we haven't provided a way to handle events that might apply to just a few components, let alone new events we write ourselves. This would be difficult to accomplish.

It would also be nice to have a property editor to use with our new *SoundsEvents* property. Again, this would be tricky, because each property will have its own character with a different list of subclasses, as well as a different list of properties within each subclass. Still, there might be an appropriate and elegant means of doing this. I plan to continue to update this expert, so be sure to let me know what you'd like to see. In the meantime, enjoy using the many sound-enabled components you'll create using this tool. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200001AM.

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 — *WriteSoundClasses* routine

```
procedure WriteSoundClasses;
var
  j, NewClasses : Integer;
  NewClassArray : array [0..EventMax] of string;

  procedure WriteSoundPlayEventsClass;
  var
    i: Integer;
  begin
    WriteALine('  TSoundPlayEvents = class(TPersistent)');
    WriteALine('    private');
    for i := 0 to (NewClasses-1) do
      WriteALine('  F' + NewClassArray[i] +
                 'SoundPlayEvent : T'  + NewClassArray[i] +
                 'SoundPlayEvent;');
    WriteALine('public');
    WriteALine(' constructor Create(AOwner: TComponent);');
    WriteALine('    published');
    for i := 0 to (NewClasses-1) do begin
      WriteALine('  property ' + NewClassArray[i] +
                 'SoundPlayEvent : T' + NewClassArray[i] +
                 'SoundPlayEvent read F' +
                  NewClassArray[i] + 'SoundPlayEvent');
      WriteALine('             write F' + NewClassArray[i] +
                 'SoundPlayEvent;');
    end;
    WriteALine('end;');
    WriteALine('');
  end;  { WriteSoundPlayEventsClass }

begin  { WriteSoundClasses }
  NewClasses := 0;
  for j := 0 to EventMax do
    if EventCheckBoxArray[j]= ecbsChecked then
      begin
        NewClassArray[NewClasses] := BasicEventNames[j];
        inc(NewClasses);
        WriteALine('  T' + BasicEventNames[j] +
          'SoundPlayEvent = class(TSoundPlayEvent)');
        WriteALine('    private');
        WriteALine('  FSoundFile : TFileName;');
        WriteALine('  FSoundSource : TSoundSource;');
        WriteALine('  FYield : Boolean;');
        WriteALine('  F' + BasicEventNames[j] +
                   'SoundPlayOption : T' +
                    BasicEventNames[j] +'SoundPlayOption;');
        WriteALine('    public');
        WriteALine(
          '   constructor Create(AOwner: TComponent);');
        WriteALine('    published');
        WriteALine('  property SoundFile: TFileName ' +
                   'read FSoundFile write FSoundFile;');
        WriteALine('  property SoundSource: TSoundSource ' +
          'read FSoundSource write FSoundSource ' +
          'default ssFilename;');
        WriteALine('  property Yield : Boolean ' +
          'read FYield write FYield default False;');
        WriteALine('  property ' + BasicEventNames[j] +
          'SoundPlayOption : T' BasicEventNames[j] +
          'SoundPlayOption read F' + BasicEventNames[j] +
          'SoundPlayOption write F' + BasicEventNames[j] +
          'SoundPlayOption;');
        WriteALine('end;');
        WriteALine('');
      end;
  WriteSoundPlayEventsClass;
end;
```

End Listing One

By Peter J. Rosario

Control Panel Applets

Integrating Configuration Programs with Windows

Control Panel applets are the small programs that are visible in, and run from, Windows Control Panel. They are typically used to configure hardware, the operating system, utility programs, and application software. This article shows you how to create and install your own Control Panel applet.

Why create your own custom Control Panel applet? Many programs you develop require configuration. You probably store the configuration parameters in an .INI file, the registry, or a database. Some programmers add code to their main programs to allow users to display, change, and save configuration parameters, perhaps making it accessible through an Options menu choice.

However, there are many reasons why you should consider placing this code in a separate Delphi project. Placing the configuration code in a separate Delphi project not only makes it more modular, and thus easier to debug, it also makes it more amenable to parallel development within a team of programmers. The separate Delphi project will also exhibit high cohesion and low coupling.

By placing the configuration code in a separate Delphi project, you can more easily prevent end-user access to the code. You may want just an administrator to be able to change the configuration parameters. If this is the case, you could install the compiled project on just the administrator's machine or, if it is on a file server, you could modify the file's execute permission so that only administrators can execute it.

Placing the configuration code in a separate Delphi project allows you to convert it to a Control Panel applet, making it appear more professional and integrated with Windows. Users and administrators are used to looking in Control Panel when they want to configure something. Why should it be any different when it comes to your program?

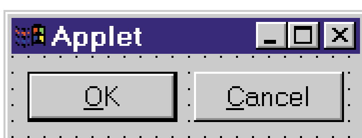


Figure 1: The Control Panel applet's main dialog box.

A Control Panel applet is a special .DLL that resides in the Windows system directory. In this article, we'll discuss implementing a simple dialog box run from an .EXE, converting the dialog box to a .DLL, and converting the .DLL into a special .DLL with the file extension .CPL. (All three projects are available for download; see end of article for details.)

Simple Dialog Box Executable

The first Delphi project builds an executable and uses only one unit/form: AlMnDlg.pas/AlMnDlg.dfm (see Figure 1). The project uses an icon that is different from the default Delphi torch-and-flame icon. The form's name is *MainDlg*, its *BorderStyle* property is set to *bsDialog*, and it contains two buttons. Each button's *ModalResult* property is set to something other than *mrNone*. When we're done, this dialog box will be what appears when you activate the applet from the Control Panel.

Dynamic Link Library

The second Delphi project builds a .DLL and uses the same form as the first project. It also adds a second unit, AlMain.pas, that implements the *Execute* procedure. The unit's **interface** section contains the *Execute* procedure's header so it can be used outside the unit. The procedure uses the **stdcall** calling convention, because it will be exported from the .DLL. *Execute* simply creates the dialog box, shows it modally, and destroys it. Its definition is shown here:

```
procedure Execute; stdcall;
begin
  AlMnDlg.MainDlg :=
  AlMnDlg.TMainDlg.Create(nil);
  try
    AlMnDlg.MainDlg.ShowModal;
  finally
    AlMnDlg.MainDlg.Free;
    AlMnDlg.MainDlg := nil;
  end;
end;
```

It was easy enough to test the first Delphi project's compiled executable, Applet.exe. All we had to do was run it. To test the second Delphi project's compiled executable, Applet.DLL, we'll have to build a program whose sole purpose is to exercise the .DLL. This is done in AlDriver.dpr. It contains a single form named *MainDlg* that resides in ADMain.pas/ADMain.dfm, and has a single **Execute** button (see Figure 2).

The button's *OnClick* event handler calls the *Execute* procedure exported by Applet.DLL:

```
procedure TMainDlg.ExecuteButtonClick(
  Sender: System.TObject);
begin
  ADMain.Execute;
end;
```

For the Applet DLL's *Execute* procedure to be visible in the ADMain unit, it must be imported from the .DLL. Once it's imported, it appears to the rest of the code to actually exist in the unit at the location of the **import** statement. (This explains why the fully qualified procedure name, *ADMain.Execute*, works.) The procedure is statically imported using the **external** directive:

```
procedure Execute; external 'Applet.DLL';
```

Because no path is specified, Windows uses an algorithm to search for the .DLL. One of the directories searched is the Windows system directory. Another is the directory from which the application loaded. In fact, that directory is searched first, and is the preferred directory to use. The search algorithm is documented in win32.hlp (on the **Index** tab, search on "LoadLibrary").

It's simple to go into AlDriver.dpr's **Project | Options** menu choice, change to the Directories/Conditionals page, and type the directory path to where Applet.DLL is located in the **Output directory** combo box. If you do so in your driver project, it will automatically be saved in the same directory as the .DLL whenever your driver executable is built.

To test Applet.DLL, place AlDriver.exe into Applet.DLL's directory if it's not already there. Run AlDriver.exe and click the **Execute** button. You should see the original dialog box on screen. Because it's shown modally, and both buttons have modal results set to something other than *mrNone*, clicking either one of them closes (and frees) the dialog box. When you're done testing the .DLL, close AlDriver.exe.

Congratulations! You now know how to place a form in a .DLL. Only a few more steps are required to convert the .DLL into a Control Panel applet.

Control Panel Applet

To test the second Delphi project, we had to build a driver program to load the .DLL, import a subroutine, and execute the subroutine. To test the final Delphi project (the one that builds the Control Panel applet), we won't have to build a driver program. Why? Windows itself will be the driver program. This controlling application is usually Control Panel itself, or the Control Panel folder in Windows Explorer. Like our driver program, the controlling application will have to load the applet, import a subroutine, and execute the subroutine.

Figure 2: The driver executable's main dialog box.

To load the Control Panel applet, the controlling application must first find it. The simplest way to allow the controlling application to find the applet is to copy it to the Windows system directory. How does it distinguish between Control Panel applet .DLLs and regular .DLLs? Control Panel applet .DLLs have the extension .CPL — not .DLL. You can make your project automatically use the .CPL extension instead of the .DLL extension. To do so, select **Project | Options**. On the Application page, type `cpl` in the **Target file extension** edit box.

To import a subroutine from a .DLL, the controlling application must use the subroutine's name or index. Windows uses the name, and looks for a function named *CPlApplet* with the following signature:

```
LONG APIENTRY CPlApplet(
  HWND hwndCPl,   // Handle to Control Panel window.
  UINT uMsg,      // Message.
  LONG lParam1,   // First message parameter.
  LONG lParam2    // Second message parameter.
);
```

The code is shown in C because it's from the Microsoft Help file win32.hlp (refer to the "References" section at the end of this article for more information). The Object Pascal equivalent is:

```
function CPlApplet(hwndCPl: Windows.THandle;
  uMsg: Windows.DWORD; lParam1, lParam2: System.Longint):
  System.Longint; stdcall;
```

This function header is declared in cpl.pas. If you have the Professional or Client/Server versions of Delphi, you have access to the source for the cpl.pas unit. You don't need it to create Control Panel applets, but it's heavily commented, and therefore provides good documentation.

Unlike our *Execute* procedure, the *CPlApplet* function is called many times and performs multiple functions, depending on what parameter values are passed. The table in Figure 3 shows the possible values for the *uMsg* parameter. (The information found in this table comes mostly from win32.hlp.)

Each CPL_XXX constant is defined in the CPL unit. The example project's *CPlApplet* function uses these constants (see Figure 4).

As the description for CPL_GETCOUNT indicates, it's possible to implement multiple Control Panel applets (i.e. dialog boxes) per .CPL. The example project, however, implements only one.

After you tell Windows how many dialog boxes your .CPL implements, it calls the *CPlApplet* function again with *uMsg* equal to CPL_INQUIRE once for each dialog box. The *lParam1* parameter tells you which dialog box the function call is for. It will be numbered from 0 to *NumberOfDialogBoxes* - 1. Because the example project only implements one applet, the *CPlApplet* function will only be called once so it doesn't handle the cases where *lParam1* is other than 0.

What	When	Why
CPL.CPL_INIT	Called immediately after the .CPL containing the applet is loaded.	The *CPlApplet* function should perform initialization procedures, e.g. memory allocation if necessary. If it can't complete the initialization, it should return zero, directing the controlling application to terminate communication, and release the .CPL. If it can complete the initialization, it should return any non-zero value.
CPL.CPL_GETCOUNT	Called after the CPL_INIT function call returns any non-zero value.	The *CPlApplet* function should return the number of dialog boxes it implements.
CPL.CPL_INQUIRE	Called after the CPL_GETCOUNT function call returns a count greater than, or equal to, 1. The *CPlApplet* function will be called once for each dialog box, indicating which dialog box with its 0-based index placed in *lParam1*.	The *CPlApplet* function should provide information about a specified dialog box. The *lParam2* parameter points to a CPLINFO record. The *CPlApplet* function uses this record to tell the controlling application the applet's name, description, and icon.
CPL.CPL_DBLCLK	Called after the user has chosen the icon associated with a given dialog box.	The *CPlApplet* function should display the corresponding dialog box, and carry out any user-specified tasks.
CPL.CPL_STOP	Called once for each dialog box before the controlling application closes, indicating which dialog box with its 0-based index placed in *lParam1*.	The *CPlApplet* function should free any resources associated with the given dialog box.
CPL.CPL_EXIT	Called after the last CPL_STOP function call and immediately before the controlling application uses the *FreeLibrary* function to free the .CPL containing the applet.	The *CPlApplet* function should free any remaining resources, and prepare to close.

Figure 3: Possible values for the *CPlApplet* parameter *uMsg*.

```
function CPlApplet(hwndCPl: Windows.THandle;
  uMsg: Windows.DWORD; lParam1, lParam2: System.Longint):
  System.Longint; stdcall;
const
  NonZeroValue = 1;
begin
  case uMsg of
    CPL.CPL_INIT: Result := NonZeroValue;
    CPL.CPL_GETCOUNT: Result := 1;
    CPL.CPL_INQUIRE:
      case lParam1 of
        0:
          begin
            Result := NonZeroValue;
            CPL.PCPLInfo(lParam2)^.idIcon := AlConst.IIcon;
            CPL.PCPLInfo(lParam2)^.idName := AlConst.SName;
            CPL.PCPLInfo(lParam2)^.idInfo := AlConst.SInfo;
            Result := 0;
          end;
        else
      end;
    CPL.CPL_DBLCLK:
      begin
        Result := NonZeroValue;
        AlMnDlg.MainDlg := AlMnDlg.TMainDlg.Create(nil);
        try
          AlMnDlg.MainDlg.ShowModal;
        finally
          AlMnDlg.MainDlg.Free;
          AlMnDlg.MainDlg := nil;
        end;
        Result := 0;
      end;
    CPL.CPL_STOP: Result := 0;
    CPL.CPL_EXIT: Result := 0;
  end;
end;
```

Figure 4: An implementation of the applet's exported function *CPlApplet*.

The CPLINFO record is defined in win32.hlp as:

```
typedef struct tagCPLINFO { // cpli
  int idIcon;
  int idName;
  int idInfo;
  LONG lData;
} CPLINFO;
```

and in cpl.pas as:

```
PCPLInfo = ^TCPLInfo;
tagCPLINFO = packed record
  idIcon : System.Integer;   // Icon resource id.
  idName : System.Integer;   // Name string res. id.
  idInfo : System.Integer;   // Info string res. id.
  lData  : System.Longint;   // User defined data.
end;
CPLINFO = tagCPLINFO;
TCPLInfo = tagCPLINFO;
```

The controlling application allocates memory for this record, and passes your *CPlApplet* function a pointer to it in the *lParam2* parameter. All your function has to do is dereference the pointer, fill in its fields, and return zero. But what should the function fill the record with?

The controlling application needs three things from your applet in order to display it inside the Control Panel properly: an icon, a name, and a description. These three things must be resources linked into your executable with unique identifiers. The record is filled with the resource identifiers. How do you link resources into and use them from your executable? There are five things you must do:

1) find a suitable icon,
2) create a text resource file,
3) compile the text resource file into a binary resource file,
4) link the binary resource file into your executable, and
5) use the resources in your Object Pascal code.

The example project uses one of the icons that comes with Delphi, but renames it to Applet.ico. The text resource file, Applet.rc, is shown here:

```
#include "AlConst.pas"

STRINGTABLE
{
  SName, "Applet",
  SInfo, "Test applet"
}

IIcon ICON ..\Applet.ico
```

There are two kinds of resources in this resource file: a string resource (STRINGTABLE), and an icon (ICON) resource. Each string resource has a pair of values: its identifier and its value. The value is shown in double quotes. The identifier is a constant that represents an integer. The constants are defined in the unit AlConst.pas (see Figure 5), which is included within Applet.rc by using the **#include** directive.

The icon resource also has a pair of values: its identifier and the file that contains the icon. The identifier comes from the AlConst unit, just like the string resource identifiers. The file name shown (..\Applet.ico) includes path information because Applet.ico isn't in the same directory as Applet.rc. Now, two of the five tasks required to link in and use resources are finished: finding a suitable icon, and creating a text resource file. What remains is to compile the text resource file into a binary resource file, link the binary resource file into the executable, and use the resources in Object Pascal code.

To compile the text resource file into a binary resource file, use brcc32.exe. This command-line utility comes with Delphi and can be found in the Delphi \Bin directory. Change to the directory that contains Applet.rc and use the following command:

```
brcc32.exe Applet.rc
```

This creates an output file in the same directory, and with the same name as the input file Applet.rc, but with the extension .RES. Applet.RES is the binary resource file. You can inspect the file by opening it with the Delphi Image Editor (from the Tools menu).

Linking the binary resource file into the executable is a simple matter of adding a compiler directive to Applet.dpr:

```
{$R ..\Applet.RES}
```

```
unit AlConst;

interface

const
  SName = 1;
  SInfo = 2;
  IIcon = 3;

implementation

end.
```

Figure 5: The AlConst.pas file.

In the sample project, the Applet.RES file generated from Applet.rc is in the directory immediately above Applet.dpr, hence the ..\ path information in front of the file name. It's a good thing, too, because Delphi automatically generates another Applet.res file in the same directory as the .dpr. This explains the directive you always see in Delphi project files:

```
{$R *.RES}
```

The asterisk here means "the same file name as the .dpr," not "any file name."

Now that the binary resource file will be linked into your executable the next time it's recompiled, how do you go about using the resources in Object Pascal? All you have to do now is include the AlConst unit in the **uses** clause of the unit that needs access to the resource identifiers. In the example project, this is the AlMain unit.

The only other *uMsg* parameter values that need explanation are CPL_STOP and CPL_EXIT. Because the sample project allocates and deallocates needed memory from within the CPL_DLBCLK **case** statement block, the CPL_STOP and CPL_EXIT **case** statements don't have to do anything except indicate success by returning 0.

Conclusion

Windows' open architecture, and Delphi's combination of ease and power, allow you to locate configuration code in custom Control Panel applets. Using custom Control Panel applets makes your applications look more professional, polished, and integrated with Windows.

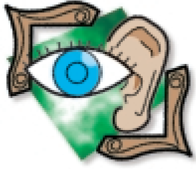
References

The win32.hlp file is part of Microsoft's Windows software development kit. It comes with Delphi, and if you accepted the default locations when you installed Delphi, it can be located at either C:\Program Files\Common Files\Borland Shared\MSHelp if you have Delphi 4 or 5, or at C:\Program Files\Borland\Delphi 3\HELP if you have Delphi 3. Open the file, make sure the Contents tab is selected, and scroll down until you see Control Panel Applications.

The CPL unit found in cpl.pas is a port of cpl.h. It comes with Delphi, and if you accepted the default locations when you installed Delphi, it can be found at either C:\Program Files\Borland\Delphi5\Source\Rtl\Win\cpl.pas if you have Delphi 5 (substitute 4 for 5 if you're using Delphi 4), or at C:\Program Files\Borland\Delphi 3\Source\Rtl\Win\cpl.pas if you have Delphi 3. Another reference from Inprise can be found at http://www.borland.com/devsupport/delphi/faq/FAQ1043D.html, although it seems to be old code (Delphi 2), because it isn't aware of the CPL unit added in Delphi 3. A reference from Microsoft can be found at http://support.microsoft.com/support/kb/articles/q149/6/48.asp. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200001PR.

Peter J. Rosario is a consultant with System Innovations Group (http://www.sysinnov.com) based in the metropolitan Washington, D.C. area. He and his lovely wife Shelley have three boys named Caleb, Nathan, and Matthew. You can contact him at PRosario@sysinnov.com with comments or questions or if he and his company can provide you with a Delphi, Microsoft SQL Server, or Web-based solution.



By Ron Loewy

Run-time ActiveX

Embedding ActiveX Controls at Run Time

The release of Delphi 3 introduced the ability to use ActiveX controls in Delphi applications. A developer who wants to use an ActiveX control imports it into the Delphi development environment using the Component | Import ActiveX Control menu option in Delphi 4/5. The Delphi import utility creates a Pascal wrapper around the ActiveX control, and adds it to the Delphi component library.

The developer can then drag the control onto a Delphi form and use it in the application. So, it appears that Delphi supports everything one could want to do with an ActiveX control. However, a closer inspection shows that a Delphi application can use an ActiveX control only if it's known to Delphi at compile time. This is different from other forms of COM objects, such as Automation objects that can be used by Delphi with the *CreateOleObject* procedure in the ComObj.pas system unit. The truth is that an ActiveX control can be instantiated like an Automation object and used in code, but the most important part of the ActiveX control, the visual representation of the functionality it encapsulates, cannot be used. In this article, I'll introduce a relatively simple method to use "unknown" ActiveX controls at run time.

Why Bother?

The December, 1999 issue of *Delphi Informant* contains an article I wrote about an application extension framework via COM interfaces. The article discusses the subjects of creating an application object model, creating a framework for COM-based plug-ins, and integrating the COM plug-ins with the application's menu structure. The plug-ins described in this article perform logic functions by accessing the application's object model. If the plug-in needs to collect information from the user, it displays a modal dialog box that is separate from the forms of the application.

Consider the idea of an application extension framework that needs to include embedded views that are part of the plug-ins created by the application users or third-party vendors: If you want to take advantage of the COM benefits, you need a method to embed these views into your application. A natural fit for COM-based embedded "views" is ActiveX controls.

Let's consider an application that allows the user to manage and edit rich media. Assume a database of rich media elements at a newspaper where media items (pictures, videos, audio clips, etc.) are indexed for easy retrieval by the reporters and researchers that work for the publication. In this article, a researcher might enter a set of keywords and get a collection of hits, a set of articles related to the media element. The researcher then needs to click the different hits, and view/listen to them to determine if they meet his or her needs.

Assume that we wrote this application as a standard Delphi application. We support standard Windows .bmp files. We also wrote the support for other popular file formats, such as .eps, .gif, .jpeg, and .avi files. Unfortunately, if we want to support new formats, such as .mgeg video, RealAudio, or .png images, we'd need to recompile the application and distribute it to our users. And we'd need to perform the same task every time a new media format needs to be added. In addition to the hassle of recompiling and distributing, we'd also have to contend with increasing application source code size, the need to maintain all the code when a new version of the compiler arrives, and the possibility of introducing bugs when making these changes.

An alternative solution is to define a standard way to register media types with the system, and store the metadata about the media element with the media element. When the element needs to be viewed, an ActiveX control will be embedded within the application and will display the element. When a new media type needs to be added, a new ActiveX control will be written to display this media type, and the ActiveX control will be registered with the system. With this solution, the code pieces for every element are separate from the rest of the system. Every project is smaller in scope, and

is therefore easier to write and maintain. Different developers can write the different modules without the need to retain similar coding styles, make changes in code written by other people, or cause unintentional harm to code. If we sell our news media indexer application to other users, the users can add support to new media types, or other elements specific to their organization without access to our code. (Consider a publication that uses an internally developed XML DTD to store information about news items, and wants to display this information using a graphical, hierarchical view specific to their needs.)

ActiveX Control = Automation Object + Visual Stuff

Before we start investigating the ins and outs of the visual parts of an ActiveX control, it's a good idea to quickly review the COM object's pecking order. Every COM object is an object that implements the *IUnknown* interface. This interface provides the ability to retrieve other interfaces supported by the object (via the *QueryInterface* method), and the object's lifetime handling via reference counting.

An Automation object is an object that implements the *IDispatch* interface. This interface allows non-compiled languages, such as VBScript, JScript, and VBA, to access the object's properties and methods by packing parameters into standard memory structures that are passed to the object. A Delphi-created Automation object also supports a type library, a binary representation (metadata) of the object's properties and methods.

To keep it simple: An ActiveX control is an Automation object that implements a set of interfaces that allow an ActiveX host application to embed the control visually into one of its windows. The control will display itself within the area the application provides for the control, and will control the focus and mouse within this area.

An ActiveX Control in Delphi Clothes

Our quest to understand ActiveX controls, and learn how to embed one at run time in an application, starts by inspecting the code generated by Delphi when an ActiveX control is imported. For the purposes of this article, I imported the Microsoft Internet Explorer (*WebBrowserOC*) control and inspected the code created by Delphi in the ShDocVw_TLB.pas file that Delphi placed in the \Imports sub-directory of the Delphi installation directory. Every ActiveX control you import and inspect will be fine for the purpose of understanding what makes an ActiveX control useable by Delphi.

Looking for the definition of *TWebBrowser* in the generated file, we realize that the class descends from the *TOleControl* class. The Delphi 3 and 4 Help files describe *TOleControl* as follows: "TOleControl is derived from TWinControl and handles the interactions with OLE necessary for using OCX controls in the Delphi environment. It is unlikely that you will ever need to derive a component from this class."

This description is accurate; the code Delphi generates when it imports an ActiveX control creates a VCL wrapper around the control (derived from *TWinControl*), and allows your Delphi application to interact with the correct OLE embedding interfaces. We must, however, disagree with the claim that it's unlikely that we'll derive a component from this class, as we'll do just that to create our run-time embedding code.

If we continue to inspect the code generated by Delphi for the *TWebBrowser* class, we can see that it contains two main parts: code specific to the interface implemented by the specific ActiveX control, and maintenance code not defined for this interface. I specifically chose the Microsoft IE control, because the main interface of the control, *IWebBrowser2*, is well documented in the Microsoft INetSDK, and it was easy to separate the control-specific functionality (methods such as *GoBack*, *GoHome*, and *Navigate* in this specific control) from code generated by Delphi to manage the OLE interaction.

Usually, when we inspect the code of a specific control we imported, we would only be interested in the code specific to this control and its functionality, and we are happy that Delphi automatically creates all the boring maintenance code for us. Not so in this case; the interesting code, for our purposes, is the code that creates and manages the communication with the OLE interfaces.

In the **protected** section of the generated code, we see the functions *CreateControl* and *InitControlData*, which should be inspected. We should also note the *ControlInterface* property, defined in the **public** part of the class definition. These functions include the secret to the way a Delphi ActiveX wrapper interacts with the OLE interfaces.

Inspecting the code further reveals that *CreateControl* returns the control-specific interface from an internal field named *OleObject*. *InitControlData* initializes a *ControlData* structure and returns a pointer to it. One of the interesting values defined in the control data structure is the class ID of the ActiveX control. This leads us to investigate the code for *TOleControl* to better understand what's happening.

TOleControl is defined in the file OleCtrls.pas, which is installed in the \Source\VCL directory of the standard Delphi installation. If we inspect this file, we find that *TOleControl* is a *TWinControl* descendant that implements the following interfaces:

- *IUnknown* — it is, after all, a COM object.
- *IDispatch* — and an Automation object.
- *IOleClientSite*, *IOleControlSite*, *IOleInPlaceSite*, etc. — the interfaces required to host an ActiveX control.

What we can learn from this is that the *TOleControl* descendant is an ActiveX host (or Site in OLE speak) that talks to the ActiveX control and tells it where it can display itself. Looking further into the code, we can see that *OleObject* (mentioned previously) is an *IOleObject* interface reference.

Let's now inspect the *Create* constructor of a *TOleControl* class. Like every other *TComponent* descendant, it receives an *Owner* component, but looking at the code reveals that it also calls the *InitControlData* method, which we've noticed in the generated *TWebBrowser* code. The internal *CreateInstance* method of *TOleControl* is later called from the constructor, and this function uses some of the fields of control data structure filled by *InitControlData* to instantiate the control interface and set the *OleObject* interface by using the COM library's *CoCreateInstance* function.

By now, it's clear that a *TOleControl*-derived class starts a specific ActiveX control by filling the control data with the specific information of this ActiveX control. It provides access to the control-specific functionality (after it was created) via the *CreateInterface* property.

One would assume that the Delphi IDE knows how to fill the control data memory structure when it creates the control's wrapper code by reading the control's type library and setting all the information as required. In theory, we could implement the code that reads and parses a type library, and reproduce this functionality at run time to create a run-time embeddable control host. My needs for run-time embedding of ActiveX controls are actually rather simple, so instead of going to the trouble of implementing this code, I chose to take a simpler approach.

Different Class IDs, Same Control

I decided that I wanted all the controls to be treated as if they were the same control. They would all have the same functionality as far as my applications were concerned. The only difference between them would be the implementation details, which aren't of any interest to the application.

Therefore, I decided to create a *TOleControl* descendant that will have the same *ControlInterface* (the same functionality as far as the application is concerned), and will only differ in the *ClassID* between the different ActiveX controls that will be embedded by the component. This doesn't make the class I will discuss here a true Jack-of-all-trades class that can host any ActiveX control, but a more limited control host that can load at run time all the ActiveX controls that implement a specific interface.

If you remember our sample application, the media indexing application we discussed previously, you can see that every media browser object needs to support the same functions: receive a pointer to the media element data, and display this data. For most applications that want a run-time ActiveX hosting facility, a common interface can be easily found. Obviously, a development tool, such as Delphi, that needs to be able to host every kind of ActiveX regardless of interfaces that it supports, will need to be able to read and parse the control's type library, but our application is much simpler.

Defining the Control's Functionality Interface

To use the embedded control from the application, we need to define the interface that the control must implement. This is really an application-specific task, but some guidelines can be useful:

- Assuming that the application is an Automation server, and provides an object model that can be used by external applications, scripts, or plug-ins, it makes sense to provide the entry point to the application's object model to the control. Thus, I like to have a *SetApplication(MyAppRef: ImyApplication)* method in the interface that needs to be implemented by the control.
- In most cases, there's a reason to pass data between the application and the control. The way this is done is specific to your application, but it's usually initiated with the *GetInformation* and *SetInformation* methods.
- Because the application activates and deactivates the control based on something the user does in the user interface, it's usually a good idea to have the *Activate* and *Deactivate* methods in the interface. Basically, you can think of the control's life cycle as follows:
  1) The user performs an operation that requires the activation of control X.
  2) The application creates the control using the GUID and embeds it in its window.
  3) The application uses the *SetApplication* method of the interface to tell the control how it can call it (the application) back.
  4) The application uses the *SetInformation* method to transfer some information to the control so it will be able to initialize itself.
  5) The application calls the control's *Activate* method. The control now needs to activate its user interface and be ready to accept user input.
  6) The user works with the control until it's done with the object that required the use of the control, and does something that requires the removal of the control (maybe choosing another object, or choosing a close function).
  7) The application calls the control's *Deactivate* method.
  8) The application calls the control's *GetInformation* to get information that the user edited in the control (if editing is part of the control's functionality).

In addition to these functions, which are usually part of a control interface, every application will define more methods that need to be implemented by embedded controls. It usually makes sense to provide methods in the application's object model that facilitate some of the back-and-forth communication carried between the control and the application. For example, assume that the control reads information from a stream managed by the application; the application's object model will need to provide methods that allow the control to perform operations, such as *ReadInteger*, *ReadString*, etc.

The Sample Application

The sample application will be a simple form of the media search application we just discussed. We'll create an application that allows the user to create folders of hierarchical media information. For every media item — a media item will be brought from a standard file (I didn't want to implement a database to store the data for this sample) — the user can provide different attributes, keywords, and other pieces of information.

Our application will provide the visual tree that will allow the user to choose different media items. It will also allow the user to add new items and to save/load a collection of these items. We won't provide any built-in functionality to edit/view any of the media items in the application; instead, the application will define an architecture that will allow us to add media types at run time and implement a viewer/editor for them as ActiveX controls. Figure 1 shows our application displaying a bitmap item using the RTBMP.Viewer ActiveX control.

Designing the Item Data Structure

The application stores the project's information in the *Objects* array of the treeview nodes. The root node and folders have no object associated with them. Nodes that represent a media item have a *TMediaItem* object associated with them.

Our application saves the information about a project in one file. Therefore, it needs to have a way to read and store all the information about the media items, even when it doesn't know what kind of information a specific media item allows to edit/store about itself.

The *TMediaItem* goals are as follows:

- Store the information about the media file.
- Store media item attributes (keywords, notes, etc.).
- Store and load the item attributes.
- Search for hits on a specific keyword.

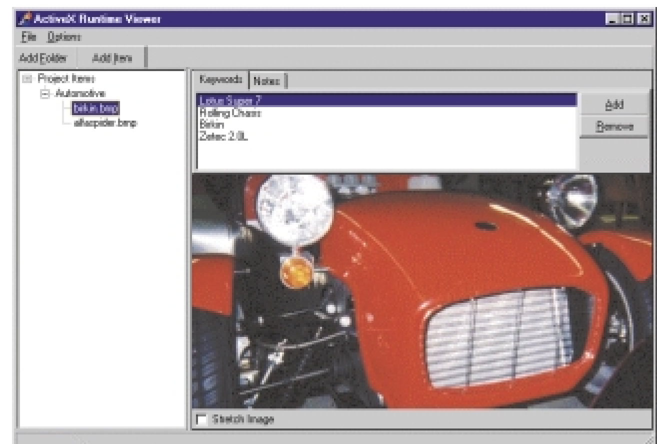


Figure 1: The application displaying a bitmap media item via the RTBMP.Viewer ActiveX control.

```
TMediaItem = class(TComponent)
  private
    // Pointer to the media file.
    FMediaFile : string;
    // The item "editable" attributes.
    FMediaAttr : TMemoryStream;
    // The keywords associated with the item.
    FKeywords : TStringList;
  protected
    function GetAutomationObject: IDispatch; virtual;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure WriteToStream(AStream: TStream);
    procedure ReadFromStream(AStream: TStream);
    function MatchKeyword(AKeyword: string): Boolean;
    property MediaFile: string
      read FMediaFile write FMediaFile;
    property MediaAttr: TMemoryStream read FMediaAttr;
    property Keywords: TStringList read FKeywords;
    property AutomationObject: IDispatch
      read GetAutomationObject;
end;
```

Figure 2: The *TMediaItem* class definition.

The design of the class is rather simple: It stores a string field that points to the media item file, stores a string list that holds the keywords associated with the media item, and holds a memory stream that includes all the attribute information. The class definition is shown in Figure 2.

Let's discuss the **public** section methods and properties briefly:

- *Create* is a constructor that creates the internal keywords string list and media attributes memory stream.
- *Destroy* cleans after the object.
- *WriteToStream* writes the item's information to a stream; used by the save function of the application.
- *ReadFromStream* reads the item's information from a stream; used by the load function of the application.
- *MatchKeyword* determines if the item matches a specified keyword; used by the application's search function.
- *MediaFile* is the name of the media file associated with the item.
- *Keywords* is the list of search words associated with the item.
- *MediaAttr* is a stream of item-specific attributes managed by the ActiveX editor for the media file type.
- *AutomationObject* is the Automation object that represents the item in communications between the editor/viewer ActiveX and the application.

The Application Object Model

The application exposes its functionality to the ActiveX viewer/editor via an object model. For the purpose of this sample, only two Automation objects are defined. The first, named *Application*, is simply the wrapper around the application and exposes the *Selected* property that points to the selected media item. The other is the *RTViewMediaItem* Automation object that is a wrapper around a *TMediaItem* instance, which we discussed previously.
In addition to access to the media item's *MediaFile* and *Keywords* properties, the Automation object provides a set of methods to read/write information from/to the *MediaAttr* stream. These include methods such as the read/write functions *ReadIntFromStream* and *WriteStrToStream*, and maintenance functions, such as *ClearStream*, *ResetStream*, and *StreamSize*.

When the user clicks on a node in the tree that represents a media item, the ActiveX control that is the viewer/editor of this item will get a reference to *RTViewMediaItem* that represents this item.

The ActiveX Viewer/Editor Common Interface

The application will host ActiveX media item viewers/editors that share a common interface. Every ActiveX control that will be created to view/edit a media type needs to implement this interface. We define this interface as *IRTViewObject* in the application's type library. Its definition follows:

```
IRTViewObject = interface(IDispatch)
  ['{ 9EE4FE00-2A1A-11D3-A5EA-0040053BA735 }']
  procedure ActivateEditor; safecall;
  procedure DeactivateEditor; safecall;
  procedure SetEditedObject(const AnObj: IRTViewMediaItem);
    safecall;
  function Get_TypesCount: Integer; safecall;
  function Get_TypeItem(i: Integer): WideString; safecall;
  function Get_ViewerGuid: WideString; safecall;
  property TypesCount: Integer read Get_TypesCount;
  property TypeItem[i: Integer]: WideString
    read Get_TypeItem;
  property ViewerGuid: WideString read Get_ViewerGuid;
end;
```

Again, let's discuss some of the procedures and properties:

- *SetEditedObject* is called by the application after the ActiveX control has been created for a specific media item. A reference to the media item *RTViewMediaItem* object (discussed in the previous section) is passed.
- *ActivateEditor* is called when the viewer is displayed. It's used to read the media file information, display it, and retrieve the keywords information and other attributes associated with the item.
- *DeactivateEditor* is called when the viewer is discarded, and is used to store the information edited by the user (keywords, additional attributes) back with the application.
- Additionally, the properties *TypesCount*, *TypeItem*, and *ViewerGuid* are used when registering the ActiveX with the system. (We will discuss viewer registration later.)

Creating a Run-time Control Host

So far, we've figured out that ActiveX control hosts in Delphi descend from *TOleControl*, and that during the time Delphi imports a control, it parses its type library and builds the knowledge about the control interface into code. We, on the other hand, decided that we will pre-define the control's functionality by defining an interface that the control will have to implement.

Now is the time to implement our run-time ActiveX control host. The rtAXHost.pas unit, available for download with the rest of this article's code, implements the *TRuntimeActiveXHost* class, a descendant of *TOleControl*. The definition of the class is shown in Figure 3.

```
TRuntimeActiveXHost = class(TOleControl)
  protected
    FIntf: IRTViewObject;
    FClassID: TGUID;
    CControlData: TControlData;
    procedure InitControlData; override;
    procedure InitControlInterface(const Obj: IUnknown);
      override;
  public
    constructor CreatePlugin(AOwner: TComponent;
      AParent: TWinControl; AClassID: TGUID);
    procedure Activate;
    procedure Deactivate;
    procedure SetEditedObject(EditedObj: IRTViewMediaItem);
    property ControlInterface: IRTViewObject read FIntf;
  end;
```

Figure 3: The class definition for *TRuntimeActiveXHost*.

The *CreatePlugin* constructor is used to instantiate the control. It receives a parent *TWinControl* to display itself on, and a GUID for the ActiveX class. The control then creates the ActiveX control and displays it in the area of the parent control.

The *InitControlData* method is used to set the ActiveX type library information. It initializes the *CControlData* structure that defines the number of events, the class ID, and other ActiveX options. The *Activate*, *Deactivate*, and *SetEditedObject* methods are wrappers around the calls to the ActiveX control's methods that are part of the *IRTViewObject* we discussed previously.

For most projects where you need to embed ActiveX controls, you can use RTAXHost.pas as your skeleton, and replace the interface-specific methods for your application. The application uses the *TRuntimeActiveXHost* control when the user clicks on a node in the tree that represents a media item.

The TreeView's *OnChange* event is defined as follows:

```
procedure TRTViewer.ItemsViewerChange(Sender: TObject;
  Node: TTreeNode);
begin
  FreeCurrentViewer;
  SwitchToNode(Node);
end;
```

First, we release the currently active viewer (if one exists), which then switches to the new node. Switching to a new node uses the following code fragment in the case of a media item:

```
FCurrentItem := TMediaItem(ANode.Data);
CreateActiveXHost;
if (Assigned(RTHost)) then
  RTHost.Activate;
```

where *RTHost* is an instance of *TRuntimeActiveXHost*, which was created by calling the *CreatePlugin* constructor from *CreateActiveXHost*. In the next section, we'll discuss ActiveX control registration with the application and show how the creation is accomplished. Notice that after the control host has been created, its *Activate* method is called (which in turn calls the actual ActiveX's *Activate* method).

ActiveX Control Registration and Activation

The application needs to know which ActiveX control to activate for a specific media type. The application keeps an internal string list that contains lists of names and values in the format, as follows:

```
TYPENAME=GUID
```

TYPENAME is the extension of the file's media type (for example, BMP for bitmaps, HTML for html, etc.). The GUID is a string representation of the class ID of the ActiveX control that implements the viewer for this media type.

When the *CreateActiveXHost* method is called to create the viewer, the media item's file extension determines the media type, and the internal list is used to get the GUID of the ActiveX. This GUID is passed to the *CreatePlugin* method of the ActiveX control host, and the viewer is created.

The internal list is read from the registry upon application start-up, but the values have to be added to this list somehow. We can't expect users to type 40-character-long GUID numbers to associate them with specific media types, but we can expect them to type the control name.

Registration of new ActiveX viewers is done using the application's **Options | Add ActiveX** menu item. The user is asked for the ActiveX name (e.g. RTBMP.Viewer, which we'll discuss later).

The application then starts the ActiveX as if it was a simple Automation object (using *CreateOleObject*), and uses the properties *TypesCount*, *TypeItem*, and *ViewerGuid*, which the ActiveX needs to implement. An ActiveX viewer can register itself with the system for more than just one media type; the RTBMP.Viewer ActiveX control supports .bmp and .wmf files, for example.

Application Roundup

Believe it or not, this is all we need to do to host ActiveX controls at run time in the application. We will now continue to discuss a sample viewer ActiveX that can display graphic files (.bmp, .wmf), and edit keywords and notes associated with this file.

Writing a Bitmap Viewer ActiveX Control

We're now ready to start the real fun: writing ActiveX controls that can be registered with the application and provide viewers/attribute editors for the different media types. The sample we'll introduce is a .bmp and .wmf file viewer.

We start by creating a new ActiveX library. I named this library RTBMP. Next, we need to create an ActiveX control in this library. The easiest way to create such a control in Delphi is to use the **File | New | ActiveForm** wizard.

I gave the name Viewer to the ActiveForm (the class name is *TViewer*) and saved the ActiveForm file under the name BMPVIEW.pas (and BMPVIEW.dfm).

To create an ActiveX that can be used by our application, we must add the application's typelib unit (RTVIEW_TLB.PAS) to the **uses** clause of the ActiveForm unit, and add the *IRTViewObject* interface to the list of interfaces supported by the ActiveForm unit. The definition of the *TViewer* class now reads:

```
TViewer = class(TActiveForm, IViewer, IRTViewObject)
```

We must also add the interface methods to the class definition as follows:

```
procedure ActivateEditor; safecall;
procedure DeactivateEditor; safecall;
procedure SetEditedObject(const AnObj: IRTViewMediaItem);
  safecall;
function Get_TypesCount: Integer; safecall;
function Get_TypeItem(i: Integer): WideString; safecall;
function Get_ViewerGuid: WideString; safecall;
```

Before we continue to inspect the implementation, let's discuss the graphical UI that will be used for the control. We will use a *TImage* control to display the media file. The top of the window will have a tabbed interface with one tab that includes a keyword list and buttons to add/remove keywords, and the other tab will include a memo where the user can write notes about the picture.

In addition to these, we'll have a checkbox at the bottom of the control that, when clicked, stretches the image to the size of the avail-

able space and, when unchecked, will display the image in its original size. We'll also define the property *EditedObject* of *IRTViewMediaItem* type that will be tracked by the control to provide access to the application's media item.

Let's start looking at the code for the control registration. The *Get_ViewerGuid* method returns the GUID created by Delphi for our class in the typelib unit file. (In RTBMP_TLB.pas, we copy the value of *Class_Viewer* and return it in our method):

```
function TViewer.Get_ViewerGuid;
begin
  Result := '{ 01B4C0E3-2A36-11D3-A5EA-0040053BA735 }';
end;
```

We now continue to the task of defining the types that will be associated with the viewer. Delphi's *TImage* supports .bmp and .wmf files, so we'll register ourselves with these two types. *Get_TypesCount* defines the number of types our viewer supports, and *Get_TypeItem* returns the different types:

```
function TViewer.Get_TypesCount;
begin
  Result := 2;  // BMP and WMF are supported.
end;

function TViewer.Get_TypeItem;
begin
  case i of
    0 : Result := 'BMP';
    1 : Result := 'WMF';
  end;
end;
```

Let's discuss the other functions. *SetEditedObject* sets the internal *EditedObject* property to the *IRTViewMediaItem*, which is passed to us from the application:

```
procedure TViewer.SetEditedObject;
begin
  FEditedObject := AnObj;
end;
```

ActivateEditor is called when the viewer is activated. We access the *EditedObject* to get the name of the media file and display it in the *TImage* control, read the keywords information from the control, and, if the attribute stream associated with the item is not empty, we read the note and the value of the stretch checkbox using the stream functions that the application exposes to us:

```
procedure TViewer.ActivateEditor;
var
  FileName : string;
begin
  FileName := EditedObject.MediaFileName;
  Image1.Picture.LoadFromFile(FileName);

  KeywordList.Items.Text := EditedObject.Keywords;
  if (EditedObject.StreamSize > 0) then begin
    EditedObject.ResetStream;
    Memo1.Lines.Text := EditedObject.ReadStrFromStream;
    StretchBox.Checked := EditedObject.ReadBoolFromStream;
    StretchBoxClick(Self);
  end;
end;
```

Finally, *DeactivateEditor* saves the note, keywords, and value of the stretch checkbox to the memory stream in the application:

```
procedure TViewer.DeactivateEditor;
begin
  EditedObject.Keywords := KeywordList.Items.Text;
  EditedObject.ClearStream;
  EditedObject.WriteStrToStream(Memo1.Lines.Text);
  EditedObject.WriteBoolToStream(StretchBox.Checked);
end;
```

The rest of the code in the control deals with the trivial issues of managing and editing the keywords.

It's remarkable: Every media item viewer that we'll implement will be as simple to create as this because of the simple architecture we created.

Using the Application

To use the application, you need to perform the following steps:

1) Compile the project RTView.dpr.
2) Compile the ActiveX project RTBMP.dpr, and use the **Run | Register ActiveX Server** option to register the control with the system.
3) Start the application, and use the **Add Folder** button to create a folder.
4) Use the **Add Item** button and add a *.bmp file (you should have some in Delphi's images subdirectory or in the Windows directory).
5) Click on the new **Bitmap** node, and notice that nothing is displayed.
6) Register the .bmp viewer with the application. First, click on the root node in the tree. Select the **Options | Add ActiveX** menu option and enter the ActiveX name RTBMP.Viewer (ProjectName.ClassName). Click **OK**.
7) Choose the media file item in the tree again. You should now be able to see the file and the keyword/notes editor in the pane to the right of the image.

Although not discussed in this article, the sample code includes the source of another media view project, named RTHTML.Viewer, which can be compiled, registered with the application, and used to view *.html and *.htm files using the MSIE WebBrowser control. Figure 4 shows the RTHTML.Viewer.

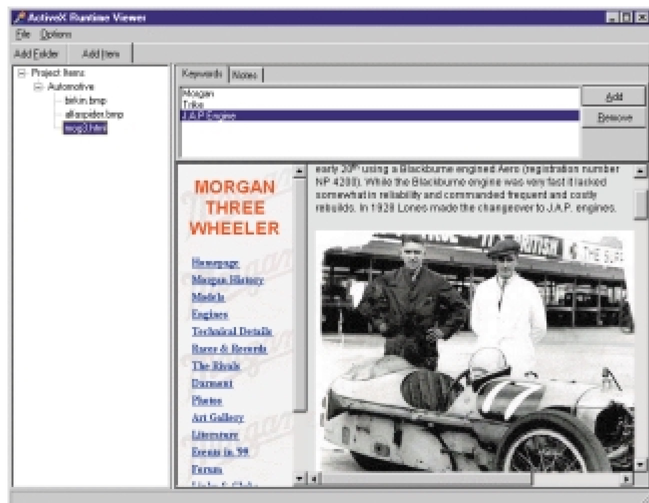
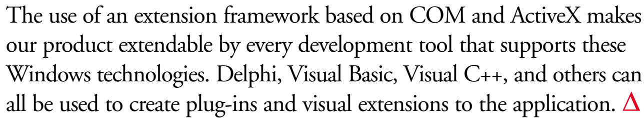


Figure 4: The application displaying an HTML page using the RTHTML.Viewer ActiveX control.

Conclusion

ActiveX controls can be embedded in Delphi applications at run time with surprisingly little work. Combining the techniques described in this article, and the plug-in article from the December, 1999 issue,

allows us to take another step into the design of extendable applications that can be updated and enhanced without recompilation.



The use of an extension framework based on COM and ActiveX makes our product extendable by every development tool that supports these Windows technologies. Delphi, Visual Basic, Visual C++, and others can all be used to create plug-ins and visual extensions to the application. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\JAN\DI200001RL.

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 Alan C. Moore, Ph.D.

Orpheus 3

An Award-winning Product Gets Better

The first component library I bought when I began using Delphi in 1995 was TurboPower's Orpheus. I immediately became a fan. The second version of Orpheus came out in 1996 with support for Delphi 2, as well as a number of important enhancements. Orpheus 3 has finally arrived, and it was definitely worth the wait.

A Strong Data Entry Foundation

While Orpheus was the first major component library for Delphi, it was by no means the first product created by TurboPower. When Delphi appeared, TurboPower was well established as a leader among Borland third-party tool producers. The roots of Orpheus include DOS tools and the Data-Entry Workshop for Turbo Pascal for Windows. Data entry remains at the center of this library, and is one of its great strengths.

Even before it was fashionable, TurboPower included validated entry capabilities in its libraries, including its DOS libraries. Orpheus supports both hard and soft validation. It includes two abstract base classes, *TOvcBaseEntryField* and *TOvcBasePictureField*, which provide the basis for the other classes. The basic derivatives are the components, *TOvcSimpleField*, *TOvcPictureField*, and *TOvcNumericField*. There's also a *TOvcEFColors* class. Of course, the components have data-aware versions: *TOvcDbSimpleField*, *TOvcDbPictureField*, and *TOvcDbNumericField*. Orpheus also includes a supporting cast of routines and special field types to handle a variety of programming situations. If you need fields that are required, read-only, or uninitialized, you've covered. If you're concerned about international issues, you'll be pleased with Orpheus' support. Its EntryFields work closely with Windows' international settings and its display strings, maintained in a single resource file, can easily be edited for various languages. If your application requires custom validation, the excellent documentation shows you how to accomplish this with a clear example.

Closely related to the single entry field components are those used in the Orpheus tables. I must admit that when I first used Orpheus, it took a while to get used to the complexity of its tables, i.e. having to use several field components along with a table component. However, the learning curve was not all that steep, and once I became proficient I was impressed with the level of control. The power that this approach gives you is immense: You can define the characteristics of individual rows and columns, as well as the characteristics of individual cells.

Orpheus tables are supported by various classes that give you a great deal of control, such as *TOvcTableColors*, *TOvcGridPenSet*, *TOvcTableRows*, and *TOvcTableColumns*. These last two classes have powerful editors that allow you to set many properties. Of course, there are a plethora of field components, including: *TOvcTCBitmap*, *TOvcTCGlyph*, *TOvcTCIcon*, *TOvcTCCheckBox*, *TOvcTCString*, *TOvcTCMemo*, *TOvcTCColHead*, and *TOvcTCRowHead*. These are based on custom components from which you can derive your own specialized field components.

In addition to data entry and table components, Orpheus also provides enhancements to standard components. As we'll discover, these enhancements have been expanded greatly in version 3.

Buttons, Labels, List Boxes, and Power Combo Boxes

Similar to Delphi's BitBtn, Orpheus' AttachedButton can be physically "attached" to another Windows control so that it moves when the other control is moved. Its functionality will usually be related to the control to which it is attached, such as a "browse button" attached to an edit field in which a filename is entered. Spinners — which provide a mouse interface for changing an entry field's value — can be viewed as specialized types of buttons. While their values are generally numeric, they can be used to cycle through a series of any valid values, such as Yes, No or Male, Female. Available in various styles, they can be "docked" to an associated entry field. They can also repeat their cycling when a spinner button is held down.

A new control, the Button Header, is actually a hybrid that combines the capabilities of Delphi's *THeader* and *TSpeedButton* controls. An obvious use would be to place the Button Header at the top of a column and then perform some action on that column (such as sorting) when the button is clicked. Also new, a URL label allows you to add a Web-like interface to your applications.

An Orpheus veteran, the Virtual ListBox, can handle a nearly unlimited number of entries. How? It's designed so that it doesn't store anything on its own. Instead, you must provide data when it needs to paint itself. It supports advanced features such as up to 128 tab settings, horizontal scrolling, and the ability to display rows of data in any combination of colors.

A Data-Aware Column ListBox provides the functionality of a list box with data-awareness. It displays the values of a specific field in a database with each row representing a different record in the database table. Among other new controls is the Checklist Box, a scrolling list of check boxes, each of which can be in any of three states: checked, unchecked, or disabled (grayed). Such Checklist Boxes are useful in installation programs or similar applications where several items must be chosen at a given time.

The History Combo Box works a lot like the standard VCL *TComboBox* but includes a history capability in which the most recently used items are copied to the top of the list for easy access. Other combo boxes we'll be examining shortly have the ability to display a selection history.

Specialized combo boxes, referred to as "Power Combo Boxes," are a welcome addition to Orpheus 3. These Power Combo Boxes provide the functionality of standard Windows combo boxes but with specialized data pre-loaded, history lists, and auto-search capability. The latter feature allows the user to incrementally search for a specific item in the list: as the user types, the list of items is narrowed to just those items that match the typed input. Some of these combo boxes are appropriate for any Delphi application, while others are geared especially to database programming. Of the latter group, some can optionally be filtered to include only specific fields from the underlying data source.

No general library would be complete without a means of selecting files and directories. Directory and File Combo Boxes are two of Orpheus' Power Combo Boxes. You can use them together or separately to provide a means for users to navigate lists of available directories and/or files on a disk drive. The File Association Combo Box contains the list of registered file types found in the Win32 registry, allowing users to choose the application with which to open a given data file. The Font Combo Box comes with a list of fonts installed on the current machine; text can even be displayed using the actual font. Symbols next to each font name indicate whether it's a TrueType or printer font.

The Database Alias Combo Box contains a list of the aliases defined by the Borland Database Engine. Whenever you select an alias with this control, it can automatically update the value of the Database Table Combo Box to reflect the selection. A Database Alias Dialog offers the same functionality as the Database Alias Combo Box, but exposes it in an easy to use dialog box. The Data-Aware Field Combo Box displays a list of database fields in a combo box format with optional history lists and auto-search capabilities. An Index Selector control displays a combo box that is pre-loaded with the names of the available indices in the underlying dataset. You can program it to substitute designated text in place of the actual index names in the combo box list.

Finally, the Data-Aware Table Name Combo Box displays a list of tables from an underlying database in a combo box format, with optional history lists and auto-search capabilities. You can synchronize this control with the new Data-Aware Alias Combo Box to automatically update this control's contents whenever the active alias is changed.

Calendars, Clocks, and Time Calculation

Earlier versions of Orpheus included date/time picture fields, calendars, and strong support for working with dates and times. These capabilities have been expanded considerably in the new version, especially with the addition of the new clock components. The Calendar and Data.Aware Calendar components provide a month-at-a-glance display with support for navigating months, years, or going to specific dates.

An important addition, the Clock controls (standard and data-aware) display a real-time analog clock with a customizable clock-face display. Of course, you can create and use additional clock faces if you wish. Clock controls can be used to display the actual time or elapsed time (stopwatch mode), making them useful in a variety of situations.

The new Date Edit Entry and Data-Aware Date Edit Entry Fields accept date entries in a variety of formats, including English text ("today," "tomorrow," and "next week") or a variety of numeric formats. They can convert these formats to a standardized date display and even include a drop-down calendar for interactive date selection. The Data-Aware Date Edit and Data-Aware Time Edit work like the corresponding standard controls but add the ability to tie the display and edit capabilities to a database field.

A Plethora of Data-aware Controls

Virtually every control that should have a data-aware version, does have a data-aware version. We've already seen several. There are also some rather unexpected ones with useful features. For example, the Data-Aware Display Label control works just like the Rotated Label control, but includes the ability to connect to the DisplayName of a database field while the Data-Aware Picture Label Control displays a field's value. Using these two controls together, you can gather and display data from a database without knowing all of its characteristics beforehand. Since both controls descend from the Rotated Label control, either can work with any TrueType font or can be rotated to any angle.

Editing large blocks of text is sometimes needed in a database application. Orpheus' Data-Aware Text Editor works like the standard Editor control but includes the ability to connect to a specific field in a database table. The Data-Aware Memo Dialog provides an easy way to display and edit the contents of a Memo Field in a database table with just a single line of code.

Orpheus 3 includes three Data-Aware Array Editors. They are similar in look and feel to the data-aware list boxes, but include the ability to edit their data. Each one corresponds to one of the three common types of data in Orpheus edit controls: numeric data, simple string data, and string data based on a picture mask.

The Data-Aware Entry Fields work just like the standard Entry Fields, but with the added capability of getting and saving their data to the fields of an underlying data source. Orpheus' Entry Fields allow for fully validated, highly customizable data entry. Again, these Entry Field controls correspond to the numeric, simple string, and picture string data types.

The Data-Aware ReportView is a data-aware version of the new ReportView control discussed later. Orpheus includes three

ReportViews: the standard edition, the Data edition, and this Data-Aware edition. Of course, there is a Data-Aware Table (Grid), as complex, flexible, and powerful as the *TOvcTable*, but with the ability to edit data from an underlying data source.

Other User-interface Controls

Some user-interface controls help us organize the desktop. Orpheus' popular Notebook provides a tabbed interface to multiple pages, each of which can serve as parent to any other control. You can set the drawing style for a Notebook's tabs or even display those tabs on any edge of the Notebook. An unusual feature is the Notebook's *ConserveResources* property, which can effectively postpone the construction of a Notebook's individual pages and controls until they are needed.

Notebooks are one means of organizing a desktop; splitters provide a means of creating multi-paned windows and are a common Windows convention. With the enhanced Splitter control, you can arrange multiple Splitter controls inside themselves, creating as many "panes" as you need.

Among the important user interface controls in Orpheus are the editors and the viewers. The *TOvcEditor* control is identical to Delphi's *TEdit*, with the addition of the capability to have an attached label and to provide access to an Orpheus Controller for its descendants. Of course, there is a data-aware version, *TOvcDbEditor*, which adds the capability of connecting to a data source and editing the text of a memo field. The File Viewer and Text File View controls can display large amounts of data in a browsable display. The standard File Viewer can display files of any kind (such as binary files), while the Text File Viewer can display text files stored in disk files.

We often want to return to an application and have it "remember" the state in which we left it. Orpheus 3's State Controls can be used to save the characteristics of forms and the controls placed on them between program runs.

Miscellaneous Controls and New Stars

While meters are not new in Orpheus 3, they are enhanced considerably with new visual effects possible through the use of bitmaps. The new Peak Meter shows a graphical representation of the current value of an application-defined measurement and the highest value during the current session. Another important addition, the MRU control, automatically handles a list of the most recently used documents, on either the File menu or elsewhere in your application. The use of this control isn't limited to filenames, either. Developers can use it to manage any list of strings. Other miscellaneous components include a customizable Calculator control, a Calculator Dialog, and a Calendar Dialog. One of the least glamorous of these controls might rank among the most useful: the Timer Pool can manage multiple timer events using just one Windows Timer resource. Now let's look at the new stars.

A major concern for many Delphi developers is having the ability to incorporate the look and feel of Microsoft applications in their own applications — without having to build the new controls from scratch. I recently saw a query on an Internet discussion list looking for a component that emulated the left pane of Microsoft Outlook, a particularly popular application. Some time ago Orpheus users made the same request of TurboPower, and the company delivered.

INFORMANT FACT FILE

Orpheus is an outstanding library of data-entry, user-interface, and general-purpose Delphi components. At the core of the library are powerful data-entry components, with simple, picture, and numeric entry-field components. The cadre of table components provide the basis for building powerful and flexible grids. With over 100 classes and components, including everything from specialized timers to flexible notebooks, enhanced basic components to leading edge controls, this is truly a comprehensive library. Example programs, full source code, extensive online help, and superb newsgroups provide all the help you could possibly need.

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

Phone: Within the US, (800) 333-4160; outside the US, (719) 260-9136
Fax: (719) 260-7151
E-Mail: info@turbopower.com
Web Site: http://www.turbopower.com
Price: List price, US$279; upgrades from Orpheus 1.x or Essentials Vol. 1, US$119; upgrades from Orpheus 2.x, US$139; upgrades from any other TurboPower product, US$223.

Among the stars of Orpheus 3 are the new LookOut Bars (see Figure 1) and the ReportViews (see Figure 2), which provide just this functionality. The LookOut Bars consist of zero or more sliding folders. Each of these folders can contain icons representing different program operations. Users can easily navigate to the different folders by clicking on a folder's group tile. They can also click icons to select specific operations within a folder.

The *TOvcLookOutBar* component includes properties and methods that give you a great deal of power and flexibility. For example, the *InsertFolder* and *RemoveFolder* methods allow you to add or remove

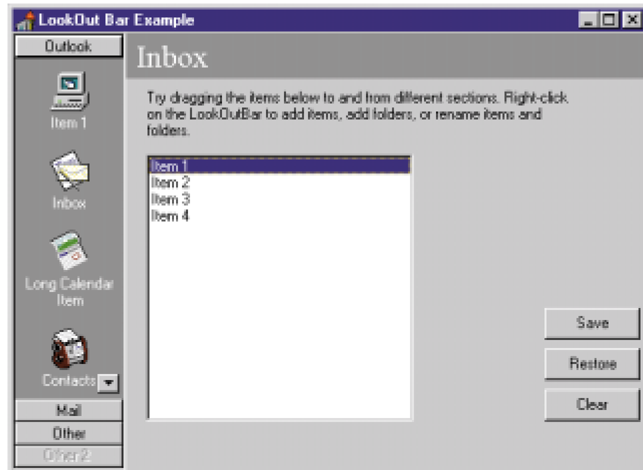


Figure 1: An example Orpheus program demonstrating the new LookOut Bars.

Figure 2: An example Orpheus program showcasing one of the new ReportView components.

folders at run time. Similarly, the *InsertItem* and *RemoveItem* methods allow you to add or remove items to and from folders at run time. You can also rename folders or items as well as take advantage of a number of other useful capabilities.

The other significant new component, *TOvcReportView*, implements a sophisticated columnar list box that allows you to sort and group the columns in a variety of ways. Again, an increasing number of popular applications include this kind of functionality, i.e. to display multiple views of the data in the columns. If the data shown were a list of files, you could sort that data by filename, size, or date just by clicking on the header tile. You also have the ability to sort in ascending or descending order; to create groups or groups within groups; and to customize the visual appearance of the control.

There are other ReportView components. *TOvcDbReportView* is a data-aware version of the *TOvcReportView* component. As with other data-aware controls, it allows you to connect to the fields of a database. The third ReportView component, *TOvcDataReportView*, has the same visual appearance and many of the same properties of the *TOvcReportView* component, but implements the *Items* property. This largely eliminates the data management chores you need to handle in the lower-level *TOvcReportView* component. On the other hand, you don't have quite the same control over data formatting and sorting that you do with the lower-level control. This illustrates one of the major strengths of this library, and indeed all of the TurboPower libraries: Orpheus makes few assumptions about how you might need to use a particular control or class of controls; rather it gives you many choices and options.

Documentation and Support

Although I haven't mentioned every component and class, I have tried to provide a good idea of the richness of this library and some of its major strengths. Documentation and support are usually important considerations in shopping for software. The massive expansion in components in Orpheus has led to an equally impressive expansion in its documentation. The excellent manual, now in two volumes comprising over 1,000 pages, describes each component, its uses, properties, methods, and events. It also provides excellent background information on many related topics. As with previous versions, Orpheus 3 includes full source code and many example programs. To provide easy and rapid support, TurboPower offers an online newsgroup, where you can generally get questions answered within 24 hours.

Conclusion

If someone were to ask me what third-party library they should buy if they could buy only one, I would recommend Orpheus. I don't know of any other library that is so rich with useful and powerful components. The documentation and the level of support are unsurpassed. But if you have doubts, download the trial version and find out for yourself. I don't think you'll be disappointed. ∆

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.

The Case for Delphi

Just when you thought you could relax and enjoy programming with the best Windows development tool, the unthinkable has happened. Your boss, who possesses about five percent of your programming knowledge, is raising questions about the continued use of Delphi.

In the past year, I saw two threads on two different Internet lists dealing with this very scenario. One such thread began like this: "I have recently started developing with Delphi and now our IT department is questioning our choice of Delphi instead of the more mainstream Microsoft products. They are concerned that by using Delphi we will not be able to integrate into new technologies like Office 2000, etc. unless we are using Visual Basic/VBA. I have tried to convey to them that everything that can be done in VB/VBA can also be done in Delphi faster with better end results. What they are looking for are reasons why Delphi is indeed better. If I can't provide a solid list of reasons, I may be forced to switch to Visual Basic, at which point I would just as soon use C++, because the learning curve is not that much greater than the mess in VB. It amazes me to have to write hundreds of lines of VB code to do something that could be done [with] drag and drop in Delphi."

Immediately jumping into the discussion, I pointed out that with this last statement, the gentleman had already begun to develop his rationale for continuing to use Delphi. The many answers that followed over the next week or so provided a wealth of additional arguments. It also helped me better understand some of the important differences between Delphi and VB. I must confess that I've never tried VB. I did try Basic for a few months in the early 1980s, but once I tried TurboPascal 2 I knew I had found the answer to all of my programming needs. I never looked back. Let's examine some of the pro-Delphi arguments that developers contributed to this thread.

Practical advice. One contributor suggested this very practical and powerful approach: "Ask them just what they think Delphi can't do, and show them they're wrong! Borland's Web site does a good job of pushing the various capabilities Delphi has, including dealing with each of MS's new fads." But, how can you really get that point across? A developer who programs in both Delphi and Visual Basic made this practical suggestion: "My advice to you would be to let a VB programmer in your IT department have an honest go at trying Delphi for a couple of weeks and let him or her provide you and your management with their opinion. That is what I did when Delphi 1 came out — I tried it, I liked it, and I have been working on it ever since."

If your manager thinks of himself or herself as a "hands-on engineer" and would prefer to see both tools in action, you might try this approach: "Just make two programs, one in VB and one in Delphi. Program an intentional error in both, and look what happens. The Delphi app will stay after the error message appears. The VB app will disappear after the 'Run time error number XXXX' message." Now, exactly how do we define RAD?

Some would argue that there's no better way to demonstrate the power of a programming language than by showcasing applications written with the language. One individual provided some examples of programs he'd written in Delphi. These included: 1) a program to remap and unmap network drives in Windows while eliminating their letters; 2) a template extender for the Windows New command; 3) an anti-virus program for a network that can be automatically updated with every login; and 4) a replacement for Netscape's setup program that lets you enter all your configuration information, such as e-mail, homepage, and bookmark locations as part of the setup. I'm tempted to devote an entire column just to a survey of some of the great applications written with Delphi.

Some one-liners. If you're looking for a quick list of Delphi's strengths, here are some of the one-liners I encountered in these messages. Some deal with Delphi's underlying language, Object Pascal; some deal with Delphi's ability to work at a low level; others deal with the kinds of applications you can produce in Delphi; some relate to Delphi's component capabilities; and some of them highlight Delphi's powerful database and Web application capabilities. With minimal editing, I present them in the order encountered:

- Many of the functions packaged with Delphi are only available as expensive add-ons in VB.
- Delphi provides much easier hardware access than VB.
- Delphi provides easier access to the Windows API than VB, i.e. you need not declare functions from the Windows API, just use them.
- Unlike VB, Delphi gives you the ability to use assembly language right in your Object Pascal code.
- Delphi creates very small executables that can do some powerful Windows "tweaking."
- Delphi creates true stand-alone applications.
- While Delphi gives you the ability to create DLLs and call them from an application, you aren't forced to redistribute a DLL (VBRun*.dll) with your application.
- Since VB components are OCX-based, you must install these OCXes with your application; Delphi components are fully linked into the final executable.
- Enterprise-level communication/object-brokering technology servers are easier to build in Delphi.
- Delphi provides the ability to create powerful components; you can't create components in VB unless you use an outside compiler such as Visual C++.
- There's no inheritance in VB as there is in Delphi.
- A database app written in Delphi is considerably faster than a database app written in VB.
- Native database drivers for Sybase ASE and MS SQL Server come with Delphi/BDE; this makes database access with SQL servers faster than having to go through ODBC as you must with VB.
- With Delphi, you can easily create or modify objects to include business rules and customization.
- With some Delphi versions you get the VCL source code.
- Delphi is based on Object Pascal, a true OO language; while VB may use objects, it can hardly be considered an OO language.
- Debugging support in Delphi is far superior to that of VB, helping greatly to accelerate the development cycle.
- Delphi is more robust, providing better error handling.
- With Delphi, you can use pointers in your code if you need to.

- Delphi is built with Delphi; VB is built with C++. This in itself should tell you a lot about the VB language and its limitations.
- Real-time statistical systems (and other real-time data-processing systems) are extremely sluggish in VB compared to their Delphi/C++ equivalents (a quote attributed to a very good VB programmer).
- You can't write IIS Web server applications in VB as you can with Delphi.

Arguments from the experts. Different arguments will work with different managers. Some managers will be moved by pure logical arguments; others by emotion and threats ("If you take away my favorite language, I'll leave, gosh darn it!"); still others by the words of established experts, particularly one who has switched from VB to Delphi. If you've been a regular *Delphi Informant Magazine* reader, you'll recall a comparison of VB to Delphi by Bruce McKinney in the April, 1998 issue. As I recall, it was very thorough, fair, and balanced. If you'd like to find out in more detail why this recognized VB guru and author of *Hardcore Visual Basic* [Microsoft Press, 1997] decided to leave his "favorite language," surf on over to http://www.devx.com/upload/free/features/vbpj/1999/mckinney/mckinney1.asp. He spares none of the gory details. In the process, you'll find a plethora of strong, well-reasoned arguments for sticking with Delphi.

However, McKinney is hardly the only respected writer in the field who is a strong advocate for Delphi. The individual who contributed the URL above to the discussion also provided two quotes from Bill Machrone, which appeared in *PC Week*: "Of the 12 major large-scale projects being developed by my company, four are Delphi, six are VB, one is C++, and one is a complete Internet project. Of the 12 projects, only the Delphi projects are on budget, on schedule, are meeting the performance requirements, and have the highest customer satisfaction." (Bill Machrone, *PC Week*, March 31, 1997.) This sounds almost like a Borland advertisement for Delphi. And yet, it's an independent assessment by a respected writer. Here's another one: "I think it's the finest visual development environment available today. It's easy enough for a neophyte to produce simple, robust applications, and deep enough for an expert to create exciting, highly customized programs. The resulting code is faster and smaller than Visual Basic's. Delphi's Language Pack makes it the choice product for developing internationalized products." (Bill Machrone, *PC Week*, March 10, 1997.)

In addition to the Web site I just mentioned, there is another one that includes a link to McKinney's article and many others in a Web article entitled "Delphi vs. Others" at http://delphi.miningco.com/library/weekly/aa042799.htm. Most of the articles are comprehensive comparisons of various RAD environments, such as Delphi, Visual Basic, and PowerBuilder.

Conclusion. As with many threads, this one eventually expanded to related topics. One had to do with why we might face such a problem at all, and raised the issue of Borland/Inprise's marketing of Delphi. One individual put it in very strong terms: "Wake up Inprise ... You have the best tools under your eyes! I love Delphi, and I'd never give it up in all my life. This means that I want to see it survive well into the future. Get yourselves some new radical advertisers and shamelessly promote your wonderful [tools]. With proper marketing, Delphi can once again become one of Inprise's most powerful and versatile languages used by a majority of the IT industry."

Many of us would certainly qualify as Delphi zealots. If we're honest, we must admit that there's a place for VB and other development languages. Another contributor to the thread summed it up better than I could hope to with the following words: "Not a language point, but an argument point would be that an excellent VB programmer may or may not know something about the OS, the API, memory management, and the hardware architecture of the PC and how to interact with it. An excellent Delphi programmer will likely know a good bit about all of these. The fact is, with Delphi, there's always an option, always a way to get it done. With VB, sometimes there's no way to do what you want to do."

Until next time, enjoy working with Delphi, the most powerful Windows development tool ever! Δ

— Alan C. Moore, Ph.D.

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

table of CONTENTS