June 1998, Volume 4, Number 6

Cover Art By: *Tom McKeith*

WinSock 2

Translating a Tower of Babble

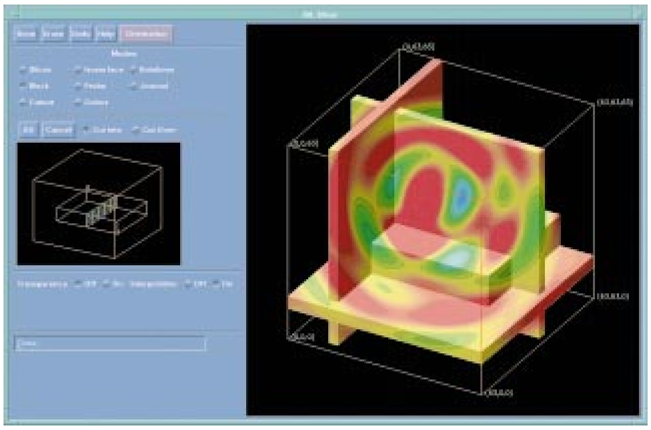
Delphi
T O O L S
New Products and Solutions

Research Systems Announces IDL 5.1

Research Systems, Inc. announced the availability of *IDL 5.1*, an updated version of the company's technical visualization and development software. This version offers support for Microsoft's ActiveX, allowing users to integrate IDL capabilities such as graphics and data analysis with COM-enabled environments (including Delphi). The ActiveX control provides Windows developers with direct, native access to IDL from within their development environment.

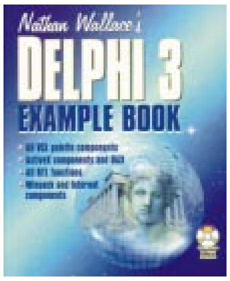
In addition, IDL 5.1 includes native Clipboard support, enhanced truetype font support, basic linear algebra subroutines, performance-tuned statistics routines with improved user interfaces, and enhancements to the LIVE_TOOLs visualization components.

IDL 5.1 is available for Windows 95, Windows NT, Mac OS, UNIX (Solaris, HP-UX, IRIX, Digital UNIX, and AIX), Linux, and Open VMS.



Research Systems, Inc.
Price: From US$1,500 for Windows, Mac OS, and Linux; from US$3,495 for UNIX and Open VMS.
Phone: (303) 786-9900
Web Site: http://www.rsinc.com

Nathan Wallace's Delphi 3 Example Book
Nathan Wallace
Wordware Publishing, Inc.



ISBN: 1-55622-490-7
Price: US$54.95
(851 pages, CD-ROM)
Phone: (972) 423-0090

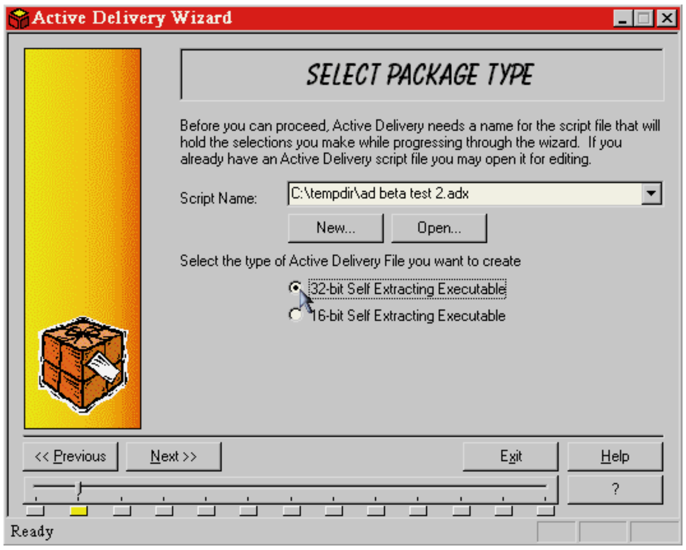
Inner Media Delivers Active Delivery

Inner Media, Inc. delivered *Active Delivery*, a self-extracting ZIP file toolkit for developers. With Active Delivery, Windows developers and non-technical content creators can package any data and/or programs into self-extracting ZIP files (Active Delivery Packages) for transmittal via Internet, intranet, e-mail, or other data-sharing architectures. These packages are then executed on the client machine, and can run programs, extract and/or decrypt files, register libraries, and display Readme files. Active Delivery also supports customization of user messages for localization and international requirements.

Active Delivery provides wizard-based and programmatic Windows-based APIs. Using the provided interfaces (DLL, VCL, OCX/ActiveX), developers can call into Active Delivery and tell it what files to add, how the Active Delivery Package is to appear, what external programs to run, etc. Developers can then ship the Active Delivery libraries royalty-free with any number of products.

Active Delivery supports many languages, including Delphi, C/C++, and Visual Basic. Active Delivery Packages may be customized to set registry variables and register libraries, as well as run specified programs. Active Delivery Packages may be created as 16- or 32-bit executables.

Inner Media, Inc.
Price: US$249; US$384 for Pro Pack; discount for registered owners of DynaZIP when purchasing Active Delivery from Inner Media.
Phone: (800) 962-2949 or (603) 465-3216
Web Site: http://www.innermedia.com



Before you can proceed, Active Delivery needs a name for the script file that will hold the selections you make while progressing through the wizard. If you already have an Active Delivery script file you may open it for editing.

Script Name: C:\tempdir\ad beta test 2.adx

New... Open...

Select the type of Active Delivery File you want to create

[✓ 32-bit Self Extracting Executable]
16-bit Self Extracting Executable

Delphi
TOOLS

New Products and Solutions

HyperAct Introduces eAuthor Help 2.0

HyperAct, Inc., a provider of Web authoring and e-commerce tools, announced *eAuthor Help 2.0*, a template-based RAD tool for large-scale Web sites and HTML Help projects.

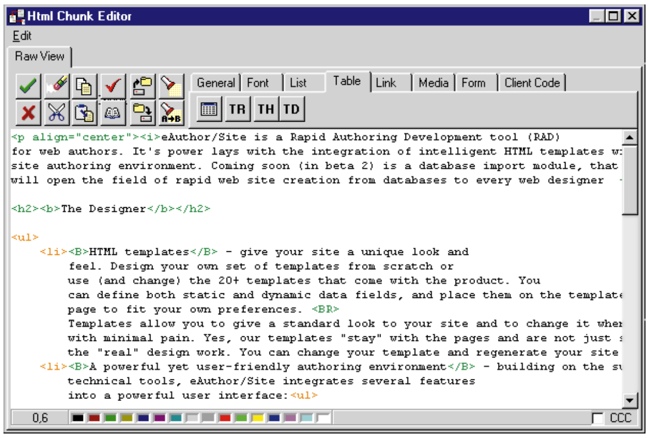
This release includes several new features, including the ability to import an entire site from an HTML page by following all its local links, and the ability to add HTML Help information-type support in the HTML Help Project template and an Information Types property to all the content templates.

eAuthor Help 2.0 also includes enhancements to existing features, such as a simplified tag insertion dialog box in the HTML Chunk Editor; HTML-based templates that can define default values to HTML Table fields display parameters; and a template engine that adds two built-in replaceable fields.

In addition, several bugs have been fixed, including the Automatic editor focus, which will be disabled if the editor is not already visible.

eAuthor Help 2.0 features Delphi and C++Builder components, plus an ActiveX control that can be added to the Visual Basic toolbox palette to embed HTML Help windows in forms.

HyperAct, Inc.
Price: US$149
Phone: (402) 891-8827
Web Site: http://www.hyperact.com



```
<p align="center"><i>eAuthor/Site is a Rapid Authoring Development tool (RAD)
for web authors. It's power lays with the integration of intelligent HTML templates w
site authoring environment. Coming soon (in beta 2) is a database import module, that
will open the field of rapid web site creation from databases to every web designer

<h2><b>The Designer</b></h2>

<ul>
    <li><B>HTML templates</B> - give your site a unique look and
        feel. Design your own set of templates from scratch or
        use (and change) the 20+ templates that come with the product. You
        can define both static and dynamic data fields, and place them on the template
        page to fit your own preferences. <BR>
        Templates allow you to give a standard look to your site and to change it when
        with minimal pain. Yes, our templates "stay" with the pages and are not just
        the "real" design work. You can change your template and regenerate your site
    <li><B>A powerful yet user-friendly authoring environment</B> - building on the su
        technical tools, eAuthor/Site integrates several features
        into a powerful user interface:<ul>
```

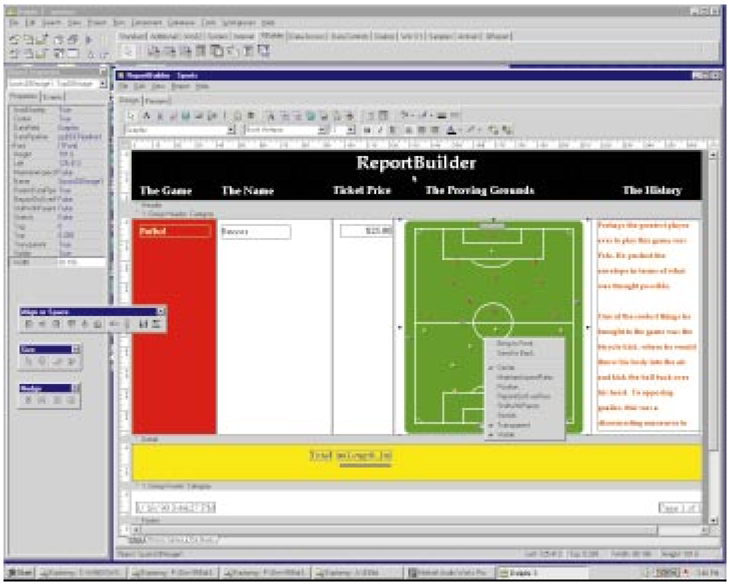
Digital Metaphors Introduces ReportBuilder 3.5

Digital Metaphors announced *ReportBuilder 3.5*, a renamed and enhanced release of Piparti 3.0, the company's reporting tool for Delphi. ReportBuilder 3.5 adds an Office97-style user interface, a bar code component, a TeeChart component, archiving capabilities, drill-down subreports, a clickable print preview component, and the RCL (Report Component Library). The RCL allows developers to create report components and install them as a native part of the product. This capability, combined with the Open Data Access and Open Data Output architectures, allows ReportBuilder 3.5 to be extended or enhanced in all areas without changing the source.

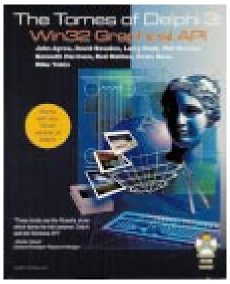
ReportBuilder and its professional version, ReportBuilder Pro, are available for Delphi 1, 2, and 3.

Technically, the Delphi implementation of the product remains the same, with no changes to class or unit names. Although the installation directories and online documentation have been changed to reflect the new name, reports created with previous versions of Piparti or Piparti Pro are fully compatible with ReportBuilder.

Digital Metaphors
Price: ReportBuilder 3.5, US$249; ReportBuilder Pro 3.5, US$495.
Phone: (214) 800-8760
Web Site: http://www.-digital-metaphors.com

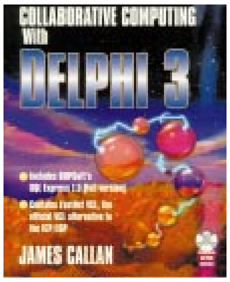


The Tomes of Delphi 3: Win32 Graphical API
John Ayres, et al.
Wordware Publishing, Inc.



ISBN: 1-55622-610-1
Price: US$54.95
(879 pages, CD-ROM)
Phone: (972) 423-0090

Collaborative Computing with Delphi 3
James Callan
Wordware Publishing, Inc.



ISBN: 1-55622-554-7
Price: US$59.95
(827 pages, CD-ROM)
Phone: (972) 423-0090

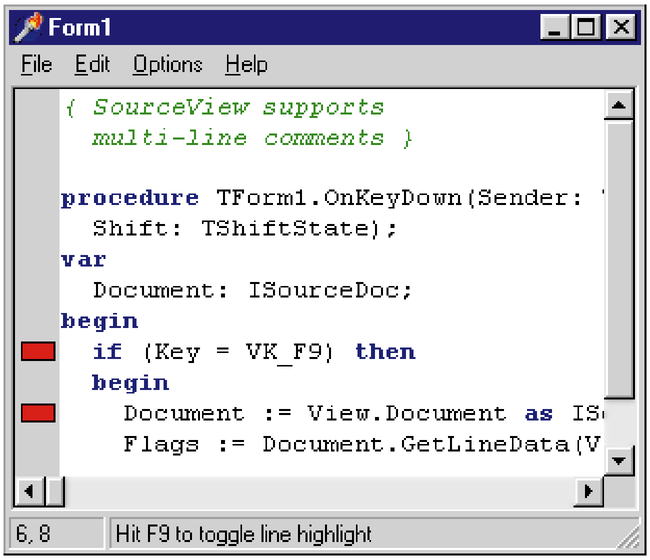
Tetradyne Releases SourceView ActiveX Control

Tetradyne Software, Inc. announced the release of the *SourceView* ActiveX control, a customizable syntax-highlighting text editor component.

SourceView can be customized for the coloring of any language syntax by setting control properties. For complex syntax requirements, OLE interface hooks are provided to plug in a parsing implementation developed in any language. Delphi, Visual Basic, and C++ examples are provided for coloring of C, Pascal, BASIC, and HTML syntax.

Developers using tools such as Borland's Delphi 3 and Microsoft's Visual Basic 5 can provide margin bitmaps and other custom display elements by implementing OLE interfaces defined by the SourceView ActiveX control. Using these features, developers can display breakpoints, bookmarks, and other custom line attributes.

SourceView also offers document-view architecture (multiple controls can be connected to a single text document), undo/redo, search-and-replace, UNICODE support, printing support, flexible event-handling options, and support for extra per-line data.



```
{ SourceView supports
  multi-line comments }

procedure TForm1.OnKeyDown(Sender: '
  Shift: TShiftState);
var
  Document: ISourceDoc;
begin
  if (Key = VK_F9) then
  begin
    Document := View.Document as IS
    Flags := Document.GetLineData(V
```

6, 8 Hit F9 to toggle line highlight

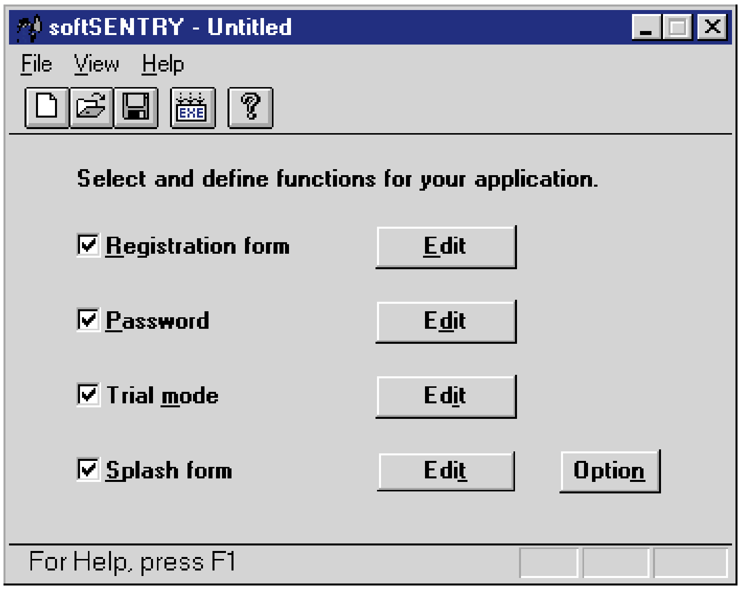
Tetradyne Software, Inc.
Price: US$299 for single developer license and unlimited redistribution with applications; multi-developer discounts are available.
Phone: (408) 377-6367
Web Site: http://www.tetradyne.com

20/20 Software Ships softSENTRY 2

20/20 Software, Inc. introduced *softSENTRY 2*, an enhanced version of its trialware and software protection tool. softSENTRY 2 works by injecting directly into executable files, or by calling a .DLL file. Added to softSENTRY are support for software subscription licensing and protection of optional modules, which give software developers more alternatives for protecting their intellectual property.

In addition, softSENTRY 2 offers increased flexibility, up to 10 passwords, flexible formula passwords, tighter integration with PC-Install, more time limitation options, and a new open architecture.

20/20 Software, Inc.
Price: US$249 for 16- or 32-bit "Lite" version; US$695 for complete version (includes 16- and 32-bit versions, plus additional features); registered owners who purchased softSENTRY 1.1 after January 1, 1998 receive a free upgrade.
Phone: (800) 735-2020
Web Site: http://www.twenty.com



Select and define functions for your application.

☑ Registration form
☑ Password
☑ Trial mode
☑ Splash form

For Help, press F1

News LINE

June 1998

Borland Completes Acquisition of Visigenic

Scotts Valley, CA — Borland announced it has completed its acquisition of Visigenic Software, Inc. Borland also announced that Roger Sippl, founder and former CEO of Visigenic, has been named the company's Chief Technology Officer, reporting to Borland Chairman and CEO, Del Yocam.

The combined operations of the companies will be conducted under the name Borland International, Inc., and will be headquartered in Scotts Valley, CA. Borland's common stock will continue to be traded on the NASDAQ National Market System under the symbol BORL. The approximate value of the transaction is US$111 million. The combined revenues of Borland and Visigenic in calendar year 1997 were approximately US$188 million.

Also, Borland anticipates a restructuring charge in its fiscal first quarter. As part of the acquisition, Borland plans to maintain a development facility at Visigenic's previous corporate headquarters in San Mateo, CA.

Founded in 1993, Visigenic has been a provider of CORBA distributed object technology for integrating heterogeneous software environments.

Borland Ships Delphi/Connect for SAP

Berlin, Germany and *Scotts Valley, CA* — Borland announced the availability of Delphi/Connect for SAP, a development environment for corporations building customized software applications that integrate with SAP R/3 enterprise systems.

Delphi/Connect for SAP leverages the SAP Business Framework and enables existing systems to evolve as newer technology is implemented, allowing developers to use Delphi to access the R/3.

Delphi/Connect for SAP prices start at US$7,500.

In a related story, a Web site promoting SAP AG's Open BAPI (Business Application Programming Interface) Network is using Delphi/Connect for SAP and InterBase Software Corp.'s SQL database to store and publish technical information. The Web site is located at http://cps.sap-ag.de/bapi/-sapwebisapi.dll.

Borland also announced it has joined SAP's Complementary Software Program as an SAP BAPI Validated Partner.

Borland Brings Java to the AS/400 with JBuilder/400 Client/Server Suite

San Francisco, CA — Borland and IBM announced a joint marketing relationship built around the JBuilder/400 Client/Server Suite, Borland's Java development tool for the AS/400 platform. The announcement was made in conjunction with IBM's release of the new version of the AS/400 operating system, OS/400 V4R2, and its Java virtual machine.

JBuilder/400 integrates the IBM AS/400 Toolbox for Java with the JBuilder environment, providing pure Java JDBC access to AS/400 data and Java access to all native AS/400 services, such as Record Level Access, SQL access, DataQueues, API Program calls, Command Calls, and IFS. JBuilder/400 Client/Server Suite also includes additional AS/400-specific features and wizards to assist corporate developers in learning Java.

Applications developed with JBuilder/400 can run on any client that supports a JDK 1.1-compliant virtual machine or on the AS/400 server (V4R2 or later), and can be partitioned between client and server.

For more information on JBuilder/400, visit http://www.borland.com/-borland400/.

Borland Japan Adds Enterprise Sales and Support Staff

Scotts Valley, CA — Borland announced it has expanded its Japanese subsidiary to focus on the enterprise development market. Borland will add approximately 15 sales, consulting, and support staff who had previously worked for OEC Japan, a joint venture between Borland and CSK Corporation (CSK is an information processing and computer services firm based in Tokyo). Borland and CSK have agreed to discontinue the joint venture.

The expansion of Borland's enterprise sales in Japan is part of an effort aimed at accelerating Borland's growth as a provider of enterprise development products and technologies. As a result of this and other moves in the past year, over 50 percent of Borland's revenue comes from corporate and enterprise products.

The new team will be responsible for selling the full suite of Borland enterprise products and technology, including the Visigenic Software product line.

Errors and Omissions

In the "File | New" column (page 41) of the April 1998 issue of *Delphi Informant*, we mistakenly printed an inaccurate URL for Eagle Software's Web site. The correct URL is http://www.-eagle-software.com.

We apologize for any confusion or inconvenience this may have caused.

table of CONTENTS

Delphi 3 / WinSock 2

By John Penman

WinSock 2

Part I: Translating a Tower of Babble

WinSock 1.1 came into being in 1993, and has become the bedrock for popular Windows-based Internet applications such as Internet Explorer, Netscape, and a host of others. One of the more popular Delphi WinSock components is dWinsock, which also uses WinSock 1.1.

So why do we need WinSock 2, if WinSock 1.1 has proven to be such an excellent API for creating TCP/IP applications? The short answer is that WinSock 1.1 wasn't designed to handle the explosive growth in multimedia Internet applications. The WinSock Group, which includes firms like Microsoft, Intel, Novell, and DEC, as well as software developers all over the world, started its work in 1995. The group designed the Windows Sockets Interface Version 2 (WinSock 2) to meet the challenge of emerging communications technologies such as real-time multimedia communications based on a protocol-independent transport interface. Windows Sockets Version 2 extends the interface considerably to include the following:

- Access to other transport protocols other than TCP/IP.
- Coexistence of multiple transport protocols using one WinSock DLL.
- Protocol-independent name resolution.
- Overlapped I/O with scatter and gather.
- Quality of Service (not yet implemented).
- Protocol-independent Multicast and Multipoint.
- Conditional Acceptance (not yet implemented).
- Connect and disconnect data (not yet implemented).
- Miscellaneous extensions, such as shared sockets.
- Layered Service Providers.

Although WinSock 2 packs more functionality, it is backwards-compatible with WinSock 1.1, so all existing WinSock 1.1 applications — including those developed in all incarnations of Delphi — can run as before. However, additional features make the WinSock 2 interface trickier to use — at least with the new features. This short series of articles will bring you up to speed with developing WinSock 2 applications using 32-bit Delphi. To this end, we'll develop two sample applications to explore the new features of WinSock 2.

This article concentrates on the issues of transport protocols and protocol-independent name resolution, with a demonstration application named TowerOfBabel. In the next article, we'll use threads to create a fast, simple file-transfer application using overlapped I/O with scatter and gather.

Before using the sample code, you need to upgrade to WinSock 2 if you're running Windows 95. If you aren't sure which version of WinSock you have, just run the TowerOfBabel application; it will inform you if WinSock 2 isn't on your system. If the program can't find WinSock 2, download the upgrade from the Internet (see the "Resources" section at the end of this article). Users of Windows NT 4.0 don't need to perform this check; the operating system has WinSock 2 embedded.

WinSock 2 Speaks Esperanto

With WinSock 1.1 the WinSock DLL comes with a proprietary transport protocol stack. Since WinSock 2 must handle transport protocols other than TCP/IP (e.g. DecNet and Novell's NDS), the WinSock Group changed the WinSock architecture to support multiple transport protocols. The new architecture follows the Windows Open System Architecture (WOSA) model shown in Figure 1.

So, instead of having a different WinSock DLL for each transport protocol stack, one WinSock 2 DLL handles different transport protocols from different vendors transparently. Of course, you could have more than one implementation of a WinSock 1.1 DLL to handle different transport protocols simultaneously, but rarely do the different implementations peacefully coexist. WinSock 2 uses a transport Service Provider Interface (SPI) to simultaneously manage different transport protocols. A transport SPI provides a gateway between WinSock 2 and the protocol stacks.

The architecture also provides a name space SPI for WinSock 2 to manage different name-resolution schemes that are protocol independent. A Name Space Provider provides a name resolution scheme, such as DNS.

A WinSock 2 application can use the *WSAEnumProtocols* API function to determine the availability of transport protocols. For example, a server advertises the protocols that are available, and listens on all transport protocols for any clients. A Novell-based client using IPX/SPX protocol can connect to any server that can use IPX/SPX. The benefit is that we don't need to modify the client application to make use of a service that is based on a different transport protocol from the client's. This protocol independence is one of WinSock 2's strengths.

A WinSock 2 program can also seamlessly use different host-name resolution systems, such as DNS, NIS, X.500, SAP, etc. This is known as Protocol Independent Name Resolution. We can determine what name resolution systems are available to a WinSock 2 application by executing the *WSAEnumNameSpaceProviders* API function.

Using *WSAEnumProtocols*

Our sample application, named TowerOfBabel, demonstrates the *WSAEnumProtocols* API function. We use this API function to detect any transport protocols present on the machine. (The entire program is available for download; see end of article for details.) We define the *WSAEnumProtocols* API in WINSOCK2.PAS, the WinSock 2 interface unit, like this:

```
function WSAEnumProtocols(lpiProtocols: PInt;
  lpProtocolBuffer: PWSAPROTOCOL_INFOA;
  lpdwBufferLength: PDWORD): u_int; stdcall;
```

Note that *WSAEnumProtocols*, like its siblings in the WinSock 2 interface unit, has the **stdcall** directive at its tail. This is a calling convention we must use for APIs written in a different language. In this case, the WS2_32 DLL is written in C.

The *lpiProtocols* parameter is a null-terminated array of protocol values. When we want *WSAEnumProtocols* to find all available transport protocols on the machine, we set *lpiProtocols* to **nil**. Otherwise, *WSAEnumProtocols* returns information on those protocols listed in the *lpiProtocols* array.

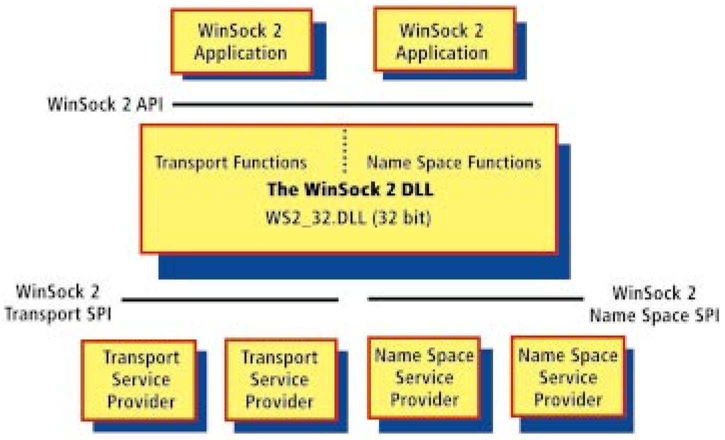


Figure 1: The Windows Open System Architecture (WOSA).

```
PWSAPROTOCOL_INFOA = ^TWSAPROTOCOL_INFOA;
TWSAPROTOCOL_INFOA = packed record
  dwServiceFlags1     : DWORD;
  dwServiceFlags2     : DWORD;
  dwServiceFlags3     : DWORD;
  dwServiceFlags4     : DWORD;
  dwProviderFlags     : DWORD;
  ProviderId          : TGUID;
  dwCatalogEntryId    : DWORD;
  ProtocolChain       : TWSAPROTOCOLCHAIN;
  iVersion            : u_int;
  iAddressFamily      : u_int;
  iMaxSockAddr        : u_int;
  iMinSockAddr        : u_int;
  iSocketType         : u_int;
  iProtocol           : u_int;
  iProtocolMaxOffset  : u_int;
  iNetworkByteOrder   : u_int;
  iSecurityScheme     : u_int;
  dwMessageSize       : DWORD;
  dwProviderReserved  : DWORD;
  szProtocol: array [0..WSAPROTOCOL_LEN+1-1] of u_char;
end;
```

Figure 2: Type declaration of the PWSAPROTOCOL_INFOA record.

When the call to *WSAEnumProtocols* succeeds, the function fills the second parameter, *lpProtocolBuffer*, with PWSAPROTOCOL_INFOA records. Figure 2 shows the definition of the PWSAPROTOCOL_INFOA packed record. The last parameter, *lpdwBufferLength*, defines the size of the *lpProtocolBuffer*. In Delphi, we call *WSAEnumProtocols* like this:

```
NoProtocols := WSAEnumProtocols(nil, @Buffer, @BufferSize);
```

where *NoProtocols* is the number of all transport protocols found on the machine.

If *WSAEnumProtocols* fails, it returns a value of SOCKET_ERROR, a WinSock constant with a value of -1. The cause of failure is usually that *BufferSize* is too small, which we can remedy by simply increasing its size. In the TowerOfBabel application, we define *BufferSize* to be 8192 bytes, which should be enough for most networked PCs.

The *WSAEnumProtocols* function returns a wealth of information for each transport protocol in the *Buffer* parameter. Figure 3 shows some of the more interesting details. To

Field	Description
dwServiceFlags1	bitmask describing the services provided by the protocol
dwServiceFlags2	reserved
dwServiceFlags3	reserved
dwServiceFlags4	reserved
dwProviderFlags	information regarding how this protocol should be presented in the protocol catalog
ProviderId	unique global identifier assigned for the service provider of the protocol
dwCatalogEntryId	unique identifier assigned by WinSock 2 for each PWSAPROTOCOL_INFOA record
iVersion	protocol version identifier
iAddressFamily	address family type, e.g. AF_INET
iSocketType	socket type, e.g. SOCK_STREAM
iProtocol	protocol type, e.g. TCP/IP
iNetworkByteOrder	specifies the number as "big-endian" or "little-endian"
iSecurityScheme	indicates type of security scheme in place, if any
dwMessageSize	maximum message size supported by the protocol
szProtocol	string identifying the protocol

Figure 3: This table highlights some of the information returned by *WSAEnumProtocols*.

```
ProtocolCount := 0;
lpProtocol  := PWSAProtocol_Info(@Buffer[ProtocolCount]);
Size  := SizeOf(lpProtocol^);
while ProtocolCount <= NoProtocols - 1 do begin
  with lpProtocol^ do begin
    { Rest of code. }
  end;
  Inc(ProtocolCount);
  Offset  := ProtocolCount * Size;
  lpProtocol  := PWSAProtocol_Info(@Buffer[Offset]);
end;
```

Figure 4: A portion of the *GetProtocols* procedure in which transport protocol data is extracted.

extract the data, we need to typecast *Buffer* as a PWSAPROTOCOL_INFOA record. As *Buffer* contains a zero-based array of PWSAPROTOCOL_INFOA records. We start with the first one like this:

```
ProtocolCount  := 0;
lpProtocol  := PWSAProtocol_Info(@Buffer[ProtocolCount]);
```

where *lpProtocol* is a local variable of type PWSAPROTOCOL_INFOA. We use the *ProtocolCount* variable to iterate through the list of PWSAPROTOCOL_INFOA records in the *Buffer* array. For each transport protocol in *Buffer*, we point *lpProtocol* to the PWSAPROTOCOL_INFOA record.

Figure 4 shows a code fragment from the *GetProtocols* procedure that uses an offset to access each protocol in the *Buffer* array. After extracting the information on a transport protocol from *lpProtocol* we use the new value of *Offset* to get the next transport protocol in *Buffer*.

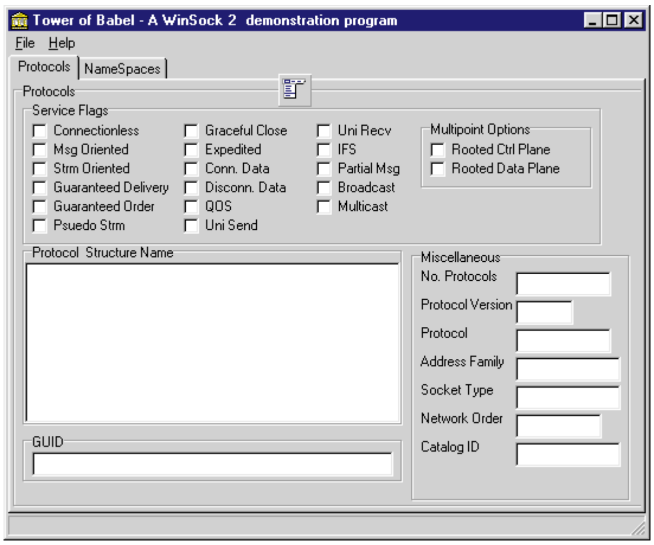


Figure 5: The TowerOfBabel application in the IDE.

Putting *WSAEnumProtocols* to Work

Now that we have a basic understanding of the *WSAEnumProtocols* API, let's examine the TowerOfBabel application in detail. Figure 5 shows the application at design time. We use *TPageControl* to set up two pages, Protocols and NameSpaces. The Protocols page displays the information for each transport protocol. The NameSpaces page displays information on the available Name Space Providers on a machine, which we discuss later in this article.

Going back to the Protocols page: We have several CheckBox and Edit controls, and a single ListBox control. The ListBox control, *lbProtocols*, displays the list of names of transport protocols found by *WSAEnumProtocols*. As we click on a transport protocol, the contents of the other controls change. A *TList* object is used to synchronize the contents of the controls.

In the *GetProtocols* procedure, *WSProtoList* (a *TList* object), stores the details of each transport protocol in the *WSProtoInfo* object when we extract the data from the *Buffer* array (see Figure 6). The data of the PWSAPROTOCOL_INFOA record is stored in the fields of the *WSProtoInfo* object at the same time.

When we click the name of the transport protocol in the *lbProtocols* ListBox, the *lbProtocolsClick* event handler calls

```
with lpProtocol^ do begin
  // Create a new entry for a protocol;
  // then add the data to the list.
  New(WSProtoInfo);
  with WSProtoInfo^ do begin
    // Copy data into WSProtoInfo's fields
    // from the Buffer array.
    WSProtoList.Add(WSProtoInfo);
  end;
end;
```

Figure 6: A partial listing of the *GetProtocols* procedure; setting up a list of *WSProtoInfo* objects.

UpdateProtoFields. In *UpdateProtoFields*, the following assignment does the synchronization trick:

```
WSProtoInfo := WSProtoList.Items[lbProtocols.ItemIndex];
```

The *lbProtocols.ItemIndex* parameter locates the correct *WSProtoInfo* object in the *WSProtoList* list. The rest of *UpdateProtoFields* seeds the controls with fresh data from *WSProtoInfo* (see Figure 7).

On the Protocols page, the *edProtocolGuid* Edit control displays the GUID string for each transport protocol. We use the *GUIDToString* function to convert the protocol's GUID to a string. When I started working on this project, I came across an interesting problem that took me a couple of hours to solve. I had trouble compiling the TowerOfBabel application because the Delphi 3 compiler refused to compile the TGUID type as defined in the copy of the WINSOCK2.PAS I had. Commenting out the TGUID type declaration in WINSOCK2.PAS forced the compiler to use the correct TGUID definition in the System unit, and the application compiled successfully. The modified WINSOCK2.PAS is included in this month's files.

Starting and Ending TowerOfBabel

When we start TowerOfBabel, the application checks for the existence of WS2_32.DLL. If it isn't present, the operating system (Windows 95 or Windows NT 4.0) displays a dialog box explaining the problem, and the application dies gracefully.

After locating the DLL, the application's *FormCreate* procedure calls the *Start* function to interrogate the DLL for its version number. The version number is 2.2, which is hard-coded as a constant in TowerOfBabel to which the WS2_32.DLL's version number must match. If the DLL's version number is different, the *Start* function returns *False*, which causes TowerOfBabel to display an error message, and call *Application.Terminate*.

If the interrogation of WS2_32.DLL is successful, the *FormCreate* procedure creates *WSProtoList*:

```
WSProtoList := TList.Create;
```

Following this list creation, we initialize a special Boolean flag, *WSProtoListFreed*, to *False*. This is a useful device to ensure that the *WSProtoList* and its *WSProtoInfo* objects are freed when TowerOfBabel terminates to prevent resource leakage. In the *Exit1Click* procedure (in the **File | Exit** option in the menu), the application checks the state of *WSProtoListFreed*. If the flag is *False*, it calls *CleanUpLists* to free the list and its objects before calling the *Close* procedure.

Using the *WSAEnumNameSpaceProviders* API

Now we look at another new WinSock 2 feature: protocol-independent name resolution. When a WinSock client attempts to connect with a server on the Internet, it must first resolve the server's name to an IP address. The Domain Name System (DNS) is usually the service that a client accesses to perform host-name resolution on the Internet. However, there are similar systems, such as Novell's NDS and X.500 services, that behave differently from DNS (the details of this are outside the scope of this article; see "Resources" for more information).

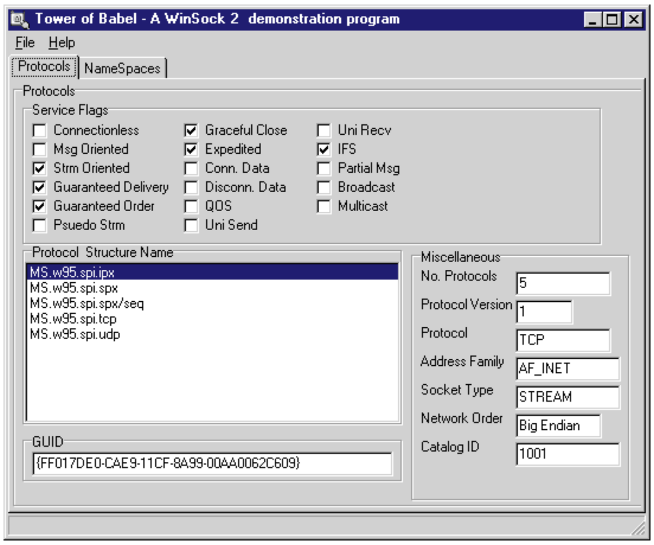
We use *WSAEnumNameSpaceProviders* to determine the available name space providers (e.g. DNS) on a networked computer. WinSock 2 provides an interface for each name space provider — a piece of code that sits behind the WinSock 2 DLL. WinSock 2 defines *WSAEnumNameSpaceProviders* as:

```
function WSAEnumNameSpaceProviders(
  lpdwBufferLength: PDWORD;
  lpnspBuffer: PWSANAMESPACE_INFO): u_int; stdcall;
```

The *WSAEnumNameSpaceProviders* API returns the number of name space providers on the system. From Delphi, we call the function like this:

```
NoNameProviders :=
  WSAEnumNameSpaceProviders(@BufferSize,@Buffer);
```

The first parameter, *BufferSize*, which is a double word pointer, sets the size of the second parameter, *Buffer*. *Buffer* contains the data required to populate the PWSANAMESPACE_INFOA


Tower of Babel - A WinSock 2 demonstration program

File Help

Protocols | NameSpaces

Protocols

Service Flags

- [ ] Connectionless
- [ ] Msg Oriented
- [x] Strm Oriented
- [x] Guaranteed Delivery
- [x] Guaranteed Order
- [ ] Psuedo Strm
- [x] Graceful Close
- [x] Expedited
- [ ] Conn. Data
- [ ] Disconn. Data
- [ ] QOS
- [ ] Uni Send
- [ ] Uni Recv
- [x] IFS
- [ ] Partial Msg
- [ ] Broadcast
- [ ] Multicast

Protocol Structure Name

MS.w95.spi.ipx
MS.w95.spi.spx
MS.w95.spi.spx/seq
MS.w95.spi.tcp
MS.w95.spi.udp

GUID: {FF017DE0-CAE9-11CF-8A99-00AA0062C609}

Miscellaneous

No. Protocols	5
Protocol Version	1
Protocol	TCP
Address Family	AF_INET
Socket Type	STREAM
Network Order	Big Endian
Catalog ID	1001

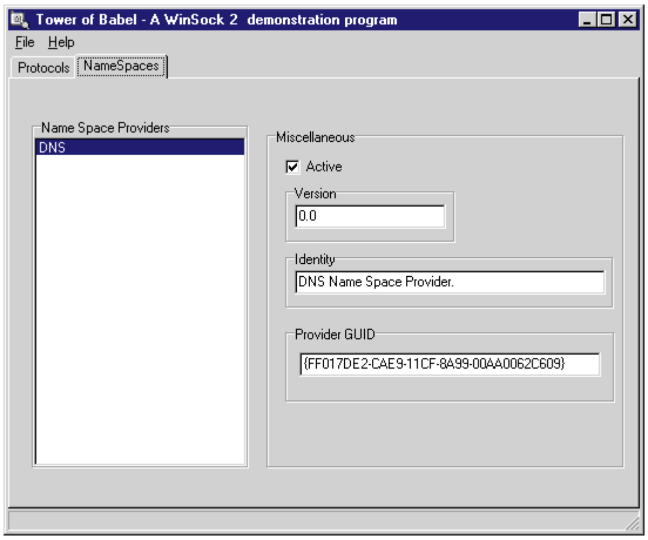
Figure 7: TowerOfBabel in action; the Protocols page.

```
BufferSize := PDWORD(ArraySize);
NoNameProviders :=
  WSAEnumNameSpaceProviders(@BufferSize, @Buffer);
if NoNameProviders = SOCKET_ERROR then
  begin
    sbStatusMsg.Panels[0].Text := 'Error : ' + WSAErrorMsg;
    ShowMessage(
      'Call to WSAEnumNameSpaceProviders failed! ' +
      'Try increasing size of buffer.');
    Exit;
  end;

lpNameSpaceProvider := PWSANAMESPACE_INFO(@Buffer[0]);
Size := SizeOf(lpNameSpaceProvider^);

for NPCount := 0 to NoNameProviders - 1 do begin
  with lpNameSpaceProvider^ do begin
    { Rest of code. }
  end;
end;
```

Figure 8: A partial listing of *GetNSProviders* that extracts data for a name space provider.



DNS

☑ Active

Version: 0.0

Identity: DNS Name Space Provider.

Provider GUID: {FF017DE2-CAE9-11CF-8A99-00AA0062C609}

Figure 9: TowerOfBabel's NameSpaces page.

record. Figure 8 shows how the *GetNSProviders* procedure extracts the data for a name space provider.

As with all WinSock functions, we must check the value returned by *WSAEnumNameSpaceProviders* for errors. When an error occurs, we must determine the cause of the error by calling *WSAGetLastError*. On success, *WSAEnumNameSpaceProviders* returns the number of name space providers found on the machine.

To enumerate the name space providers, we use the same approach as we did for the protocols. As we loop through the *Buffer* data, we typecast it as a PWSANAMESPACE_INFOA record before extracting the data. We use the same technique in *GetProtocols* to get the next name space provider, i.e. we use an offset. As we extract the details of the name space provider, we copy the data to the *WSNSInfo* object in the *WSNSList* — another *TList* object. We use this object to synchronize the contents in the Edit controls with the *lbNameSpaceProviders* ListBox control on the NameSpaces page. When we click on a name space provider in *lbNameSpaceProviders*, the *lbNameSpaceProvidersClick* event handler calls *UpdateNSPFields* to update the contents in the controls on the NameSpaces page (see Figure 9).

Looking Ahead

We've explored two WinSock 2 functions, *WSAEnumProtocols* and *WSAEnumNameSpaceProviders*. Next time, we'll take a look at creating a speedy file-transfer application using more of WinSock 2-specific APIs, including those for overlapped I/O.

Resources

- For up-to-date information on WinSock 2, including the upgrade kit and links to other WinSock 2 sites, point your browser to http://www.sockets.com/winsock2.htm.
- WinSock 2 documentation is available from Intel at http://www.intel.com/ial/winsock2/specs.htm. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806JP.

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

Delphi 1, 2, and 3 / Borland Database Engine

By Bill Todd

Delphi Import/Export

Part II: Streams and Bit Fiddling

In the first article of this two-part series, we looked at various ways of accessing data, including using the BDE ASCII driver and Delphi's I/O routines. In this article, we'll continue to explore techniques for accessing data more effectively, including using *FileStream* objects and bitwise operators.

Working with Objects that Use Files

In Part I, we saw the rich set of features that Pascal provides to work with files. It's time to point out that we frequently don't need to write file-access code in Delphi. Why? All of the following Delphi objects have built-in methods (*LoadFromFile* and *SaveToFile*) for reading/writing data from/to files:

- *TBitmap*
- *TGraphic*
- *TIcon*
- *TMemoryStream*
- *TMetaFile*
- *TPicture*
- *TStringList*
- *TString*

```
procedure TForm1.OpenBtnClick(Sender: TObject);
begin
  { Get the directory path from the file list. }
  filePath := FileListBox1.Directory;
  { If the path does not end with \ add one. }
  if filePath[Length(filePath)] <> '\' then
    filePath := filePath + '\';
  { Add the file name to the path. }
  filePath := filePath + FileListBox1.FileName;
  { Empty the memo and read the file. }
  Memo1.Lines.Clear;
  Memo1.Lines.LoadFromFile(filePath);
end;
```

Figure 1: The **Load Memo** button's click procedure loads text into the Memo component.

The following components also have their own methods for file I/O:

- *TOLEContainer*
- *TOutline*

However, this list tells only part of the story, because all of the components that include any of the previously listed objects also include their file I/O routines. For example, the lines in a *Memo* component are actually a *TStringList* object, so you can load the memo by calling *LoadFromFile*, and save the memo's contents by calling *SaveToFile*.

The code in Figure 1 is from the TSTRING.DPR sample project **Load Memo** button's click procedure, and shows how the text from the file is loaded into the Memo component on the form. This is the same code you saw last month in the discussion of text-file processing, except for the last line, which calls the *LoadFromFile* method of the memo's *Lines* property to read the contents of the text file into the memo. The following code is from the **Save Memo** button's click procedure:

```
procedure TForm1.SaveBtnClick(Sender:
                              TObject);
begin
  Memo1.Lines.SaveToFile(filePath);
end;
```

```
procedure TForm1.UpperBtnClick(Sender: TObject);
var
  UpList: TStringList;
  i: Word;
begin
  UpList := TStringList.Create;
  UpList.LoadFromFile('upper.txt');
  for i := 0 to UpList.Count - 1 do
    UpList[i] := UpperCase(UpList[i]);
  UpList.Sorted := True;
  UpList.SaveToFile('upper.txt');
  UpList.Free;
end;
```

Figure 2: The *OnClick* event handler for a button in UPPER.DPR.

Here, a call to the *SaveToFile* method saves the contents of the memo. You can see these methods in action by running the program (see end of article for download details), opening a text file, making some changes, and saving the file. The same technique works with *TList* and *TComboBox* components. The following code is from the **Load List** button's *OnClick* event handler:

```
procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  ListBox1.Items.LoadFromFile('months.txt');
end;
```

You can also perform this same type of string manipulation in your code behind the scenes, using *TStringList* objects. The code in Figure 2 is from the *OnClick* event handler for the button in the project UPPER.DPR. This code declares an instance of *TStringList* called *UpList*, and begins by calling its constructor in the statement:

```
UpList := TStringList.Create;
```

Next, the code loads the contents of the text file UPPER.TXT into the string list, then uses a **for** loop to iterate through the list and convert each string in the list to upper case. The statement:

```
UpList.Sorted := True;
```

sets the string list's *Sorted* property to *True*, which causes the string list to sort itself. The call to *SaveToFile* saves the upper-case text back to UPPER.TXT. The last statement in this code:

```
UpList.Free;
```

is perhaps the most important. Remember that when you're done with the object, you must always call an object's destructor to release the memory it uses. Notice that the **for** loop uses the string list's *Count* property to determine the number of strings in the list and access each string like an element in an array. Because the first string in a string list always has an index of zero, the last string's index is always *Count* - 1.

Using *FileStream* Objects

FileStream objects let you read and write binary data. The *FileStream* methods don't know or care what the data is; they simply let you transfer a stream of bytes between a buffer area in memory and a disk file. You can specify the number of bytes to transfer and the location in the file where the transfer will begin. The STREAM.DPR project, available for download, demonstrates the use of *FileStreams*.

```
procedure TStreamForm.ReadBtnClick(Sender: TObject);
var
  Stream: TFileStream;
  Buff: array[0..31] of Char;
  Count: Longint;
begin
  { Create a file stream. }
  Stream := TFileStream.Create('stream.bin',
  fmOpenReadWrite);
  try
    { Read some bytes from the stream. }
    Count := Stream.Read(Buff, 26);
    { Put a null at the end of the buffer so you can treat
      it as a null-terminated string. }
    Buff[Count] := #0;
    { Display what you have read. }
    ShowBuff(Buff, Count, Stream.Position);
  finally
    Stream.Free;
  end;
end;
```

Figure 3: This *OnClick* event handler creates and reads from a *FileStream* object.

The code in Figure 3, from the **Read Stream** button's *OnClick* event handler, shows how to create and read from a *FileStream* object. This routine begins by declaring an instance variable, *Stream*, of type *TFileStream*. A character array, *Buff*, serves as the memory buffer into which the data read from the stream will be placed; the *Count* variable is used to hold the number of bytes actually read by the stream. The statement:

```
Stream := TFileStream.Create('stream.bin',fmOpenReadWrite);
```

calls the stream's constructor method, *Create*, to create an instance of the *FileStream* object. The *Create* method takes two parameters; the first is the name of the file and the second is the file mode. For a list of the file modes, search for *TFileStream* in Delphi's online Help, click **Methods** to display the list of methods, then select **Create**.

The call to the *Read* method takes two parameters. The first is the buffer into which the data read from the stream will be placed; the second is the number of bytes to read. The *Read* method returns the number of bytes actually read from the stream. If the number of bytes read is less than the number of bytes requested, you've reached the end of the file.

In this example, the file you are reading, STREAM.BIN, contains the lower-case letters "a" through "z." The statement:

```
Buff[Count] := #0;
```

puts a null character into the buffer after the last byte read. This lets the *ShowBuff* method that displays the contents of the buffer in the Memo component on the form

```
procedure TStreamForm.ShowBuff(const Buff: array of Char;
  Count, CurrentPos: Longint);
var
  I: Integer;
begin
  with Memo1 do begin
    { Convert the buffer from a null-terminated string to
      a Pascal string and assign it to the first line of
      the memo. }
    Lines[0] := StrPas(Buff);
    { Show the number of bytes read and the current
      position of the stream. }
    Lines.Add('Bytes Read = ' + IntToStr(Count));
    Lines.Add('Stream Position = ' + IntToStr(CurrentPos));
  end;
end;
```

Figure 4: The *ShowBuff* method.

```
procedure TStreamForm.WriteBtnClick(Sender: TObject);
var
  InStream,
  OutStream: TFileStream;
  Buff: array[0..31] of Char;
  Count: Longint;
begin
  { Create the input file stream. }
  InStream := TFileStream.Create('stream.bin', fmOpenRead);
  { Create the output file stream. }
  OutStream := TFileStream.Create('stream.out', fmCreate);
  try
    { Move the stream pointer to the location where you
      want to begin reading. }
    InStream.Seek(10, 0);
    { Read some bytes from the stream. }
    Count := InStream.Read(Buff, 5);
    { Put a null at the end of the buffer so you can treat
      it as a null-terminated string. }
    Buff[Count] := #0;
    { Display what you have read. }
    ShowBuff(Buff, Count, InStream.Position);
    { Write contents of buffer to the output stream. }
    OutStream.Write(Buff, Count);
  finally
    InStream.Free;
    OutStream.Free;
  end;
end;
```

Figure 5: The *OnClick* event handler for the **Write Stream** button.

treat the buffer as a null-terminated string. The *ShowBuff* method, shown in Figure 4, copies the contents of the buffer array to the first line of the Memo, then displays the number of bytes read, and the position of the stream after the read.

The code from the **Read From Middle** button's *OnClick* event handler is identical to the code from the **Read Stream** button, except for the call to the *FileStream*'s *Seek* method to set the position of the *Stream* to the eleventh byte before reading five bytes into the buffer:

```
Stream.Seek(10, 0);
```

In the call to *Seek*, the first parameter is the position you want to set the *FileStream* to; the second parameter identifies the starting point (or origin). There are three possibilities: If the origin is zero, the position is relative to the beginning of the file; if it's 1, the position is relative to the current position; if it's 2, the position is relative to the end of the file. Note that the position can be a negative number, if you wish to move backward.

The code from the **Write Stream** button's *OnClick* event handler copies five bytes from one *FileStream* to another and is shown in Figure 5. Again, this code is similar to that you've seen. This time, however, the method opens a stream for reading, and creates a new file for output. It reads five bytes into the buffer, and writes those five bytes to the output stream with the call to the *Write* method:

```
OutStream.Write(Buff, Count);
```

The *Write* method takes the buffer and the number of characters to write as parameters. At the end of each of the *OnClick* event handlers, the *i* objects are destroyed by calling the destructor method, *Free*. This closes the file and releases the memory used by the *FileStream* object.

Fiddling with Bits

So far, this series has dealt mainly with ASCII files as examples. If you need to work with other types of files, the untyped file routines (*BlockRead* and *BlockWrite*) and file streams will let you read and write any file, regardless of what it contains. However, once you have read the byte stream from a file, you may need to perform some manipulation of the data before you can use it. For example, you may encounter a mainframe file that contains numbers in sign, overpunched format, or COBOL Comp format. Before you can use these numbers, you will have to convert them to a format that Delphi understands; this means accessing the values from the file at the bit level.

Fortunately, Pascal includes a complete set of bitwise operators for working with integers; you can access values on a bit-by-bit basis, or work with groups of bits. The bitwise operators are **and**, **or**, **xor**, **not**, **shl**, and **shr**. The two you're likely to find most useful for converting data from a foreign format to something Delphi can understand are **and** and **shr**. The **and** operator lets you determine the value of a bit, which effectively lets you extract the value of any bit, or combination of bits.

Working with the bitwise operators involves two values: the value you wish to operate on and a mask value. The truth table in Figure 6 shows the effect of the **and** operator. The most important rows in this table are the second and third. Note that if the value of the mask is 1, the result is always equal to the value you **and** it with. This means you can extract the value of any bit or combination of bits using the **and** operator and a mask value that has those bits set to 1.

The truth table in Figure 7 shows how the bitwise **or** operator works. Again, the significant rows are the second and third. If the bit is on in the mask value, it will be on in the result.

Value	Mask	Result
0	0	0
0	1	0
1	1	1
1	0	0

Figure 6: Truth table for the bitwise **and** operator.

Value	Mask	Result
0	0	0
0	1	1
1	1	1
1	0	1

Figure 7: Truth table for the bitwise **or** operator.

Value	Mask	Result
0	0	0
1	0	1
1	1	0
0	1	1

Figure 8: Truth table for the bitwise **xor** operator.

The **xor** bitwise operator toggles the state of the bits set in the mask in the third and fourth rows (see Figure 8).

The last two bitwise operators, **shl** and **shr**, shift the bits in the value to the left and right, respectively. Using them, you can modify the values you are working with on a bit-by-bit basis as necessary to convert the data from one format to another.

ODBC Drivers

If you need to import data from, or export data to, complex formats that aren't directly supported either by the BDE or Pascal's file I/O features, look for an ODBC driver. ODBC drivers are available for most common file formats and provide an easy alternative to learning how to read or write a complex file structure.

Conclusion

The BDE and Object Pascal have a variety of tools you can use to get data into, or out of, your programs in the format you need. For complex file formats, ODBC drivers offer an easy alternative. In addition, third-party DLLs are available for importing and exporting many formats. As we've seen in this two-part series, there are many options to help us meet our import/export needs. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806BT.

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

Delphi 1, 2, 3 / Algorithms

By Rod Stephens

Tough Decisions

Building Delphi Decision Trees

When you strip away the glitz and flash, the multi-media and virtual reality, computers are decision-making tools. We use them to visualize situations, analyze the expected results of different decisions, and from that, pick a course of action.

Many decisions have so many consequences, however, that analyzing them all can be hard, even for a computer. For example, suppose you have US$100 million to spend on one or more of 100 different investments. You want to pick a selection of investments that maximizes your expected profit but costs no more than the money you have to spend. Any given investment will either be in the solution you chose, or it won't. The two possibilities for each investment combine to make the total number of possible combinations of investments 2^{100}, or more than 1^{30}. You must select the most profitable of all these options. Even if your computer could examine one million of these combinations per second, it would take you more than 4 x 10^{16} years to explore them all — more than two million times the age of the universe.

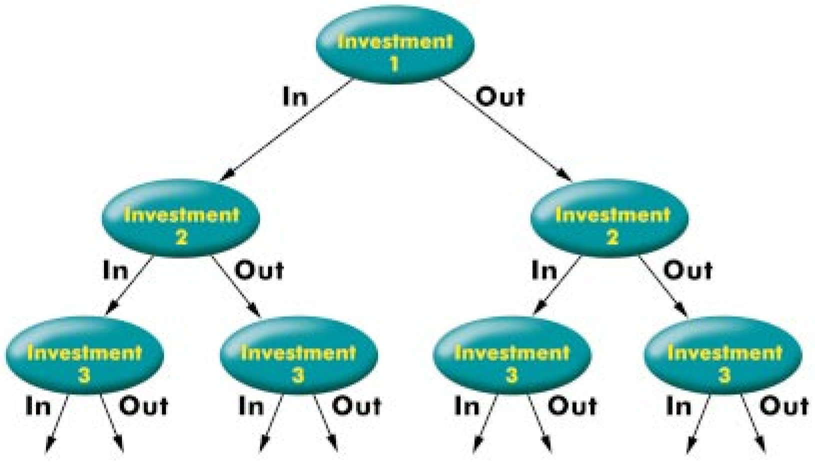


Figure 1: An investment decision tree.

This article explains *decision trees*, data structures you can use to model these types of difficult decisions, and how to implement them in Delphi. It also explains several ways you can search even the largest decision trees.

Decision Trees

You can model tough decisions, like the investment example, using decision trees. Each node in the tree represents a partial decision. In this example, it represents the decision to spend money on an investment or not. The branches coming out of the node represent the different possible choices for the partial decision. For example, the left branch extending out of the top node might correspond to spending money on the first investment option. The right branch would correspond to not investing in that option. Figure 1 shows a small investment decision tree. This tree involves only three investment options, so there are only 2^3 (or eight) possible combinations at the bottom of the tree.

Finding the best solution to the original problem corresponds to finding the best path through the decision tree — from its root at the top, to a leaf at the bottom. The definition of "best path" depends on the particular problem. For this investment example, the path must not include more nodes than can fit within the $100 million spending limit. The "best path" is the one that reaches a leaf with the greatest possible profit.

Exhaustive Search

One way to search for the best path is to visit every node in the tree. Each time the program reaches a leaf node at the bottom of the tree, it evaluates the cost and profit of the investment package represented by the node. If the cost is no greater than $100 million, and the profit is greater than the best profit found so far, the program records this package as the best solution at that point. After examining the entire tree, the program knows which combination is best. This method is called *exhaustive search* because the program exhaustively examines every node in the tree.

The demonstration program, Decide, stores investment costs and profits in the *Cost* and *Profit* arrays. (Decide is available for download; see end of article for details.) It uses the Boolean array *TestUse* to indicate whether an investment opportunity is included in a test solution. For example, *TestUse[10]* is *True* if investment 10 is part of the test solution. The value *BestProfit* records the profit of the best solution found so far, and the Boolean array *BestUse* indicates the investment options included in the best solution.

The *SearchExhaustively* procedure (shown in Figure 2), takes as a parameter the index of the test investment it should consider. It assumes the earlier investments have already been fixed. For example, when the argument value is 12, investments 1 through 11 have already been added or excluded from the test solution.

```
procedure TDecideForm.SearchExhaustively(
  test_item: Integer);
var
  i : Integer;
begin
  // If we have assigned every item, see if we have
  // found an improved solution.
  if (test_item > NumChoices) then
    begin
      if ((TestCost <= CostAvailable) and
          (TestProfit > BestProfit)) then
        begin
          // Save the improved solution.
          for i := 1 to NumChoices do
            BestUse[i] := TestUse[i];
          BestProfit := TestProfit;
        end;
      Exit;
    end;

  // Add test_item to the test solution.
  TestUse[test_item] := True;
  TestCost := TestCost + Cost[test_item];
  TestProfit := TestProfit + Profit[test_item];

  // Recursively see what we can find.
  SearchExhaustively(test_item + 1);

  // Take test_item back out of the test solution.
  TestUse[test_item] := False;
  TestCost := TestCost - Cost[test_item];
  TestProfit := TestProfit - Profit[test_item];

  // Recursively see what solution we can find
  // without test_item in the solution.
  SearchExhaustively(test_item + 1);
end;
```

Figure 2: Searching a decision tree exhaustively.

The procedure first adds the test investment to the test solution and recursively calls itself to examine the other investment options. It then removes the test investment from the test solution and again recursively calls itself to evaluate other options. As it makes these calls, it updates the test solution's cost and profit accordingly.

At some point during the recursive process, the procedure is called with a test investment index greater than the available number of options. At that point, all the investments have been assigned either in or out of the test solution, and the procedure is at a leaf node in the decision tree. The procedure examines the test solution's cost and profit. If the cost is within the spending limit, and the profit is better than the best profit found so far, the routine updates the *BestProfit* and *BestUse* values to save this solution.

Branch and Bound

For simple problems, exhaustive search is adequate. Unfortunately, many useful decision trees are extremely large. If you really had $100 million to invest, all sorts of people would probably appear to give you dozens, if not hundreds, of investment suggestions. The corresponding decision tree would be enormous.

The problem with exhaustive search is that it visits every node in the tree, even though many can't possibly represent good solutions. For example, the left-most node represents a solution that includes every investment. This would almost surely cost more than the spending allowance. The right-most node corresponds to a solution including no investments. Adding any investment to this solution will increase total profit. These are extreme examples, but decision trees generally contain many nodes that represent solutions that are obviously bad.

A technique called *Branch and Bound* can help you trim many of these obviously bad choices from the tree. The algorithm searches the tree recursively, much as exhaustive search does. As it progresses, it keeps track of the current cost of the test solution. If it ever reaches a point where the cost of the test solution exceeds the spending allowance, the algorithm stops examining that solution.

For example, suppose in a 20-investment problem, the program has assigned the first 10 options in and out of the test solution. If the total cost of the items already placed in the test solution is greater than the allowance, the algorithm stops examining this solution. There's no reason to consider the remaining 10 items because the cost of the solution can only increase.

This test allows the algorithm to prune branches that are too expensive. Branch and Bound uses another test to exclude branches that don't produce enough profit. As the algorithm progresses, it keeps track of the total profit not yet committed by the algorithm. If the profit given by the test solution so far, plus this uncommitted profit, is no greater than the profit of the best solution found so far, the algorithm discards the solution.

```
procedure TDecideForm.BranchAndBound(test_item : Integer);
var
  i : Integer;
begin
  // If we've reached a leaf node, we know
  // this solution is an improvement.
  if (test_item > NumChoices) then
    begin
      for i := 1 to NumChoices do
        BestUse[i] := TestUse[i];
      BestProfit := TestProfit;
      Exit;
    end;

  // Try to add test_item to the test solution.
  if ((TestCost + Cost[test_item] <= CostAvailable) and
      (TestProfit + UnusedProfit > BestProfit)) then
    begin
      // Add it.
      TestUse[test_item] := True;
      TestCost := TestCost + Cost[test_item];
      TestProfit := TestProfit + Profit[test_item];
      UnusedProfit := UnusedProfit - Profit[test_item];

      // Recursively test this solution.
      BranchAndBound(test_item + 1);

      // Take the item back out of the solution.
      TestUse[test_item] := False;
      TestCost := TestCost - Cost[test_item];
      TestProfit := TestProfit - Profit[test_item];
      UnusedProfit := UnusedProfit + Profit[test_item];
    end;

  // Try a solution without test_item.
  if (TestProfit + UnusedProfit -
      Profit[test_item] > BestProfit) then
    begin
      UnusedProfit := UnusedProfit - Profit[test_item];
      BranchAndBound(test_item + 1);
      UnusedProfit := UnusedProfit + Profit[test_item];
    end;
end;
```

Figure 3: Searching a decision tree with Branch and Bound.

For example, suppose in a 20-investment problem that the program has assigned the first 10 options, and the profit of the test solution so far is $5 million. Suppose the remaining 10 items have a combined profit of $10 million. If the program has already found a solution worth $16 million, it can stop working on this test solution. Even if it added all the remaining investments to the package, it could increase the total profit to only $15 million — not enough to beat the current best.

These tests have the added benefit that any time the algorithm reaches a leaf node, the test solution is certain to be better than the best solution found so far. Otherwise, the leaf node would have been trimmed off in the previous step when the last investment was added to the test solution and the remaining profit was reduced to zero. The fact that leaf nodes are always improvements simplifies the algorithm. Figure 3 shows the Delphi source code for Branch and Bound.

Decide reports on the number of decision tree nodes visited by each of its algorithms. While exhaustive search visits all 2^{N+1} nodes in an *N* investment decision tree, Branch and Bound visits far fewer. In one test with 25 investments, exhaustive search visited more than 67 million nodes; Branch and Bound visited only 1,453. The exact number of nodes visited depends on the specific costs, profits, and spending allowance.

```
procedure TDecideForm.HillClimbing;
var
  unspent, best_i, best_profit, i : Integer;
begin
  unspent := CostAvailable;

  // Repeatedly look for an item to add to the solution.
  while (True) do begin
    // Find the item with largest profit that fits
    // the solution.
    best_profit := -1;
    best_i := -1;
    for i := 1 to NumChoices do begin
      // If the item has not yet been used, and if
      // we have the funds to afford it, and if it
      // gives better profit, take it.
      if ((not BestUse[i]) and
          (Cost[i] <= unspent) and
          (Profit[i] > best_profit)) then
        begin
          best_i := i;
          best_profit := Profit[i];
        end;
    end;

    // If best_i < 0, no more items will fit
    // in the solution so we're done.
    if (best_i < 0) then
      Exit;

    // Add the item found to the solution.
    BestUse[best_i] := True;
    BestProfit := BestProfit + best_profit;
    unspent := unspent - Cost[best_i];
  end;
end;
```

Figure 4: Searching a decision tree with a hill-climbing heuristic.

Hill Climbing

For some problems, however, even Branch and Bound isn't fast enough. In that case, you can still use a heuristic to find an approximate solution to the problem. A heuristic is an algorithm that should produce a good solution, but is not guaranteed to find the best solution.

Different heuristics have differing amounts of success for different problems. One common type of heuristic is a *hill-climbing* algorithm. The idea is for the algorithm to make choices that always move the program closer to its goal. It's called hill climbing because it's similar to one way a lost hiker can try to find the top of a mountain at night. Even in the dark, the hiker can always move uphill, closer to the goal. Naturally, the hiker might come to the top of a smaller hill and be unable to find the top of the mountain. That can be a problem with hill climbing heuristics as well. The algorithm may come to a local maximum that isn't the best possible solution.

In the investment example, the hill-climbing heuristic considers all the remaining investment options, and picks one that fits within the remaining cost allowance and has the largest profit. If all the options have roughly the same cost, the algorithm will simply pick those that have the largest profits. If the costs are identical, this will give the best possible solution. If the costs are different, the algorithm may sometimes pick an

```
procedure TDecideForm.SearchRandomly;
const
  NUM_TRIALS = 1000;
var
  trial, unspent, num_items, num, i : Integer;
begin
  // Attempt NUM_TRAILS * NumChoices trials.
  for trial := 1 to NUM_TRIALS * NumChoices do begin
    unspent := CostAvailable;

    // Add randomly selected items to the solution
    // until no more will fit.
    while (True) do begin
      // See how many items can fit in the solution.
      num_items := 0;
      for i := 1 to NumChoices do
        if ((not TestUse[i]) and
           (Cost[i] <= unspent)) then
          num_items := num_items + 1;
      // See if no more fit.
      if (num_items < 1) then
        Break;
      // Pick an item randomly.
      num := Trunc(Random(num_items)) + 1;
      // Find the item.
      for i := 1 to NumChoices do
        if ((not TestUse[i]) and
           (Cost[i] <= unspent)) then
          begin
            num := num - 1;
            if (num < 1) then
              Break;
          end;
      // Select it.
      TestUse[i] := True;
      unspent := unspent - Cost[i];
      TestProfit := TestProfit + Profit[i];
    end;

    // We have finished creating the random solution.
    // See if it is an improvement.
    if (TestProfit > BestProfit) then
      begin
        for i := 1 to NumChoices do begin
          for i := 1 to NumChoices do
            BestUse[i] := TestUse[i];
          BestProfit := TestProfit;
        end;
      end;  // End saving improved solution.

    // Reset the test solution for the next trial.
    TestProfit := 0;
    TestCost := 0;
    for i := 1 to NumChoices do
      TestUse[i] := False;
  end;  // End of this trial.
end;
```

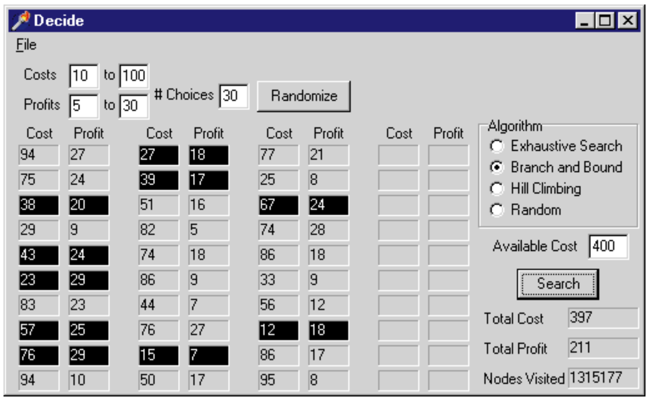
Figure 5: Searching a decision tree using a random heuristic.

expensive option with a high profit, when it might be better to pick two cheaper options with a greater combined profit.

Figure 4 shows the hill-climbing heuristic used by Decide. The algorithm repeatedly looks through the unused investment choices to find the one with the highest profit that costs no more than the remaining spending allowance. When no more items fit into the solution, the heuristic is done.

Random Search

Exhaustive search is slow because it examines every node in the decision tree. Branch and Bound trims the number of nodes visited, but it still visits a substantial fraction of the nodes. For an *N*-item investment problem, hill climbing must make, at most, *N* selections. That makes it extremely fast for even the largest problems. Unfortunately, that also means the heuristic examines only one of the huge number of possible solutions; the chances of it finding the best solution possible are extremely small.



Costs 10 to 100
Profits 5 to 30
Choices 30

Cost	Profit	Cost	Profit	Cost	Profit	Cost	Profit
94	27	27	18	77	21		
75	24	39	17	25	8		
38	20	51	16	67	24		
29	9	82	5	74	28		
43	24	74	18	86	18		
23	29	86	9	33	9		
83	23	44	7	56	12		
57	25	76	27	12	18		
76	29	15	7	86	17		
94	10	50	17	95	8		

Algorithm
- Exhaustive Search
- [✓ Branch and Bound]
- Hill Climbing
- Random

Available Cost 400
Total Cost 397
Total Profit 211
Nodes Visited 1315177

Figure 6: The Decide program after finding a solution using Branch and Bound.

Another heuristic that sometimes works better than hill climbing is a *random* heuristic. The program simply picks several solutions at random, recording the best it finds. Each trial has only a small chance of finding a good solution. In fact, chances are, the hill-climbing heuristic will produce a better solution than a random one. However, hill climbing produces a single solution and then stops. A random heuristic can select many random solutions until it obtains a reasonable result. Figure 5 shows the source code for the random heuristic used by Decide.

Caveat implementor: The functions described in this article are recursive in nature, and there's no message processing occurring during their execution. This maximizes the CPU's time, but also means Windows cannot preempt them, i.e. on large problems, the machine may appear to lock up, and you may be unable to halt processing — even with Ctrl Alt Delete. Real-world application of these algorithms should either include Windows messaging, limit the size of the calculations, or both.

Conclusion

The Decide program demonstrates exhaustive, Branch and Bound, hill climbing, and random search (see Figure 6). Experiment with it to see how performance changes with different input values. You will notice, for example, that Branch and Bound may examine more nodes if you increase the spending allowance.

Many other heuristics are possible for the investment problem. Some select items with small costs instead of large profits. Others make random selections, then swap items in and out of the solution, trying to make improvements. Using the code in Decide as a starting point, try making a few heuristics of your own. With experience, you can build algorithms that produce reasonable solutions for even the largest problems. Perhaps some day you'll win the lottery and have a chance to put your heuristics to practical use. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806RS.

Rod Stephens has written several books, including *Custom Controls Library* [John Wiley & Sons, Inc., 1998] and *Visual Basic Graphics Programming* [John Wiley & Sons, Inc., 1996]. His book *Visual Basic Algorithms* [John Wiley & Sons, Inc., 1998] explains dozens of algorithms like these. Visit his Web site at http://www.vb-helper.com, or contact him at RodStephens@vb-helper.com.

By Cary Jensen, Ph.D.

Run-time Type Information

An Introduction to Delphi's Undocumented RTTI

This article provides an introduction to run-time type information (RTTI), which is information the Delphi compiler stores in the code segment of your compiled project. This information is associated with published properties of a class, and it provides a mechanism for treating the symbolic information associated with your types as strings. One example of how RTTI impacts your everyday use of Delphi is the Object Inspector. The Object Inspector displays the names of published properties. This information is retrieved using RTTI.

Delphi ships with a unit, named typinfo.pas, that contains the RTTI functions and procedures, as well as type declarations used by these functions. By adding this unit to your unit's **uses** clause, you can call these functions to access RTTI. At a minimum, you should consider taking a look at the typinfo.pas file in Delphi's\Source\VCL directory.

This article demonstrates several uses of RTTI. One note of caution is in order, however. Borland has specifically not documented the typinfo unit, and reserves the right to change it in any new release of Delphi. It is essential that Borland maintain this right, since the RTTI features are critical to the operation of Delphi itself. As a result, it's possible that if you use RTTI in your application, subsequent changes to the typinfo unit in a new version of Delphi will require you to make modifications to your programs if you want to recompile them under the new version.

Getting RTTI for Enumerated Types

Enumerated types are used throughout Delphi. An example of this is the *TCommonAvi* enumerated type, which declares the valid values for the *CommonAVI* property of an Animate control. The following is the *TCommonAvi* declaration from Delphi 3's comctrls unit:

```
type TCommonAVI = (aviNone, aviFindFolder,
                   aviFindFile,
  aviFindComputer, aviCopyFiles, aviCopyFile,
  aviRecycleFile, aviEmptyRecycle,
  aviDeleteFile);
```

RTTI permits you to do two things with enumerated types. You can retrieve strings that contain the name of each value in the enumerated type, and you can identify the ordinal position within the enumerated type of one of its valid values using a string representation of the value.

You obtain a string equivalent of an enumerated type value using the *GetEnumName* function:

```
function GetEnumName(TypeInfo: PTypeInfo;
                     Value: Integer): string;
```

The first argument is a pointer to the enumerated type's RTTI information, and the second argument is the ordinal position of the value within the enumerated type. *GetEnumName* returns a string representing the corresponding enumerated type value.

You get the ordinal position of an enumerated type value based on a string using the *GetEnumValue* function:

```
function GetEnumValue(TypeInfo: PTypeInfo;
  const Name: string): Integer;
```

Like *GetEnumName*, the first argument is a pointer to the RTTI information. The second argument is a string that represents the enumerated type value. This function returns the ordinal position of the corresponding value.

The relationship between these two function calls is demonstrated with the following code segment:

```
var
  s: string;
  i: Integer;
begin
  s := GetEnumName(TypeInfo(TCommonAvi),3);
  // Displays aviFindComputer.
  ShowMessage(s);
  i := GetEnumValue(TypeInfo(TCommonAvi),s);
  // Displays 3.
  ShowMessage(IntToStr(i));
```

As mentioned, these functions require a pointer to the RTTI for an enumerated type. Since the pointer for a particular type may change from one version of Delphi to another, you must use the *TypeInfo* function to retrieve this information. *TypeInfo* takes a single argument, the type of the enumerated type whose RTTI pointer you want to return:

```
function TypeInfo(TypeIdent): Pointer;
```

The use of these functions is demonstrated in the ANIMATE project, which also demonstrates the use of the Animate component from the Win32 page of the Component palette. (This project, and the other demonstration projects discussed in this article, are available on disk and for download; see end of article for details.) Figure 1 shows the main form of this project as it might appear while running. There's a ComboBox that lists the various valid values for the *TAnimate.CommonAVI* property. In most applications, you would have populated this ComboBox using its *Items* property at design time.

In this project however, the ComboBox is loaded dynamically from RTTI using the *GetEnumName* function from within the form's *OnCreate* event handler (see Figure 2). A **for** loop iterates through the *TCommonAvi* enumerated type. For each ordinal position, the *GetEnumName* function is called, and the returned value is added to the ComboBox's *Items* property.

RTTI is used again to assign the value selected in the ComboBox to the Animate's *CommonAVI* property. This is performed from the ComboBox's *OnChange* event handler, shown in Figure 3. This code includes an additional step for the purpose of clarity. Specifically, the value returned by *GetEnumValue* is assigned to an intermediate variable named *ValueOrd*. This variable is then cast as a *TCommonAVI* type. Instead of using the variable *ValueOrd*, the value returned by the *GetEnumValue* could have been cast directly, permitting these two steps to be represented by a single statement.

Getting Object Property Listings

As you learned earlier, it's possible to get the names of published properties using RTTI. This is done by populating a *PPropList* using a call to *GetPropList*:

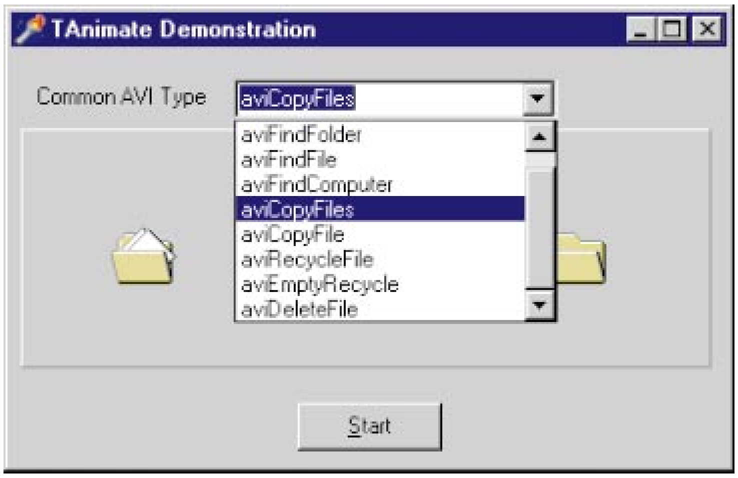


Figure 1: The *TCommonAVI* values displayed in the ComboBox are discovered at run time using RTTI.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  ca: TCommonAvi;
begin
  // For each value of the TCommonAVI enumerated type.
  for ca := Low(TCommonAvi) to High(TCommonAvi) do
    // Get string equivalent of the enumerated type value.
    ComboBox1.Items.Add(GetEnumName(TypeInfo(TCommonAvi),
                        Ord(ca)));
  // Initialize ComboBox to the first item in the list.
  ComboBox1.ItemIndex := 0;
end;
```

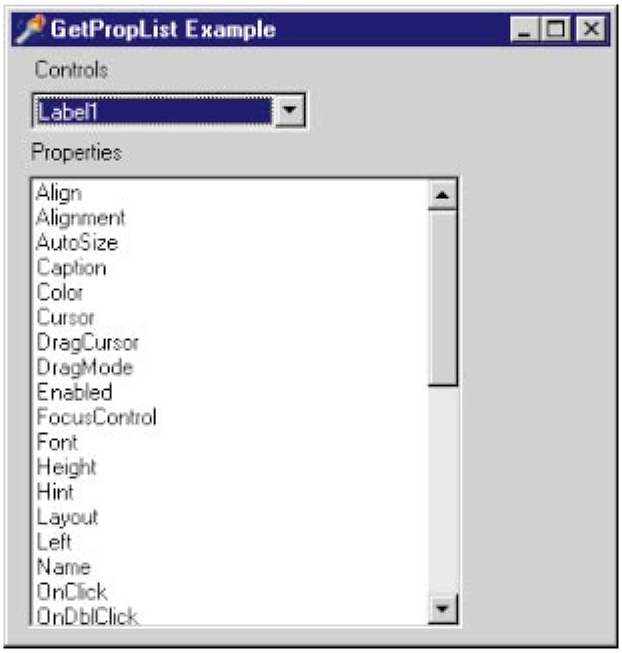
Figure 2: The ComboBox is loaded dynamically from RTTI using the *GetEnumName* function from within the form's *OnCreate* event handler.

```
procedure TForm1.ComboBox1Change(Sender: TObject);
var
  ValueOrd: Integer;
begin
  if Animate1.Active then
    begin
      Button1.Caption := '&Start';
      Animate1.Stop;
    end;
  // Get the ordinal position of the value associated
  // with the selected string in the ComboBox.
  ValueOrd := GetEnumValue(TypeInfo(TCommonAvi),
                 ComboBox1.Items[ComboBox1.ItemIndex]);
  // Cast this ordinal value to the TCommonAVI type.
  Animate1.CommonAVI := TCommonAVI(ValueOrd);
end;
```

Figure 3: Assigning a value to the Animate component's *CommonAVI* property using the ComboBox's *OnChange* event handler.

```
function GetPropList(TypeInfo: PTypeInfo;
  TypeKinds: TTypeKinds; PropList: PPropList): Integer;
```

A *PPropList* is an array of *TPropInfo* records, and each record holds information about a particular property. This record includes fields such as *Name* and *PropType*. The first argument is the *TypeInfo*. Unlike *GetEnumName*, for which you must use *TypeInfo*, there is a second, alternative way to get the *PTypeInfo* argument. Since this function is used on objects, you can use the *ClassInfo* property as this first argument. The second argument is a set of the property types. Following is



Align
Alignment
AutoSize
Caption
Color
Cursor
DragCursor
DragMode
Enabled
FocusControl
Font
Height
Hint
Layout
Left
Name
OnClick
OnDblClick

Figure 4: The names of the published properties of a class selected in the **Controls** ComboBox are displayed in a ListBox. These values are discovered using *GetPropList*.

the declaration of the *TTypeKind* enumerated type, which defines the valid values for this set:

```
TTypeKind = (tkUnknown, tkInteger, tkChar, tkEnumeration,
  tkFloat, tkString, tkSet, tkClass, tkMethod, tkWChar,
  tkLString, tkWString, tkVariant, tkArray, tkRecord,
  tkInterface);
```

This is the *TTypeKinds* declaration:

```
TTypeKinds = set of TTypeKind;
```

Other useful *TTypeKind*-related declarations in this unit include the following:

```
const
  tkAny = [Low(TTypeKind)..High(TTypeKind)];
  tkMethods = [tkMethod];
  tkProperties = tkAny - tkMethods - [tkUnknown];
```

The third argument of *GetPropList* is the *PPropList* that is populated with the property information. Finally, *GetPropList* returns an integer representing the number of properties returned in the *PPropList*.

The use of *GetPropList* is demonstrated in the PROPLIST project. The main form for this project is shown in Figure 4. This form includes a ComboBox, whose contents are populated at run time with the form's *OnCreate* event handler with the names of the components appearing on the form:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  ComboBox1.Items.Clear;
  for i := 0 to Form1.ComponentCount -1 do
    ComboBox1.Items.Add(Form1.Components[i].Name);
  ComboBox1.Text := '';
end;
```

```
procedure TForm1.ComboBox1Change(Sender: TObject);
var
  PropList: PPropList;
  i: Integer;
  CompName: string;
begin
  PropList := AllocMem(SizeOf(PropList^));
  i := 0;
  CompName := ComboBox1.Items[ComboBox1.ItemIndex];
  ListBox1.Items.Clear;
  try
    GetPropList(FindComponent(CompName).ClassInfo,
                tkProperties + [tkMethod], PropList);
    while (PropList^[i] <> nil) and
          (i < High(PropList^)) do begin
      ListBox1.Items.Add(PropList^[i].Name);
      Inc(i);
    end;
  finally
    FreeMem(PropList);
  end;
end;
```

Figure 5: The ListBox is populated with the property names of the object selected in the ComboBox. This operation is performed from the ComboBox component's *OnChange* event handler.

The ListBox is populated with the property names of the object selected in the ComboBox. This operation is performed from the ComboBox's *OnChange* event handler, as shown in Figure 5.

This code is generic, in that the *PTypeInfo* is extracted using the *ClassInfo* property of a component, a pointer to which is returned using the *FindComponent* method. *FindComponent* returns a reference to an instance of an object based on a string, which, in this case, is the selected component name in the ComboBox. Since *FindComponent* returns a *TComponent* reference, and *Name* is a property of *TComponent*, it's unnecessary in this example to cast the reference returned by *FindComponent* to another class.

Using RTTI with Properties

Polymorphism permits you to treat objects that descend from different classes similarly. For example, you can access the *Name* property of any component that descends from *TComponent* using a *TComponent* reference. This is exactly what's being done in the following code, which comes from the PROPLIST example described earlier. It's used to populate the ComboBox with a list of the form's component names:

```
for i := 0 to Form1.ComponentCount -1 do
  ComboBox1.Items.Add(Form1.Components[i].Name);
```

However, the ability to treat these components polymorphically in this way is possible only when the property (or method) being accessed is visible in the shared ancestor class. *Name*, in this example, is declared **published** in *TComponent*, and therefore satisfies this requirement.

When two or more objects have the same property, but that property is not declared as **public** or **published** in a common ancestor, you cannot access it polymorphically using a reference to the ancestor. An example of a property such as this is *Color*. The *Color* property of both the *TEdit* and *TMemo* classes is inherited from *TControl*. However, this property is declared as **protected** in *TControl*. The *Color* property is re-declared as **published** in the *TEdit* and *TMemo* class definitions, respectively. Since *TEdit* and *TMemo* do not share this property in an ancestor class with sufficient visibility to be accessed at run time, it isn't possible to treat these two classes polymorphically with respect to the *Color* property. For example, the following code generates a compiler error:

```
for i := 0 to Self.ControlCount - 1 do
  Self.Controls[i].Color := clTeal;
```

By comparison, the *Hint* property, which is declared as **published** in *TControl*, can be treated polymorphically for any *TControl* descendant. For example, the following code compiles properly:

```
for i := 0 to Self.ControlCount - 1 do
  Self.Controls[i].Hint := 'hi' + IntToStr(i);
```

Fortunately, RTTI provides a mechanism that permits you to access **published** properties in a generic fashion across classes, even when those classes do not share a visible inherited version of the property. This mechanism is provided through a series of procedures with names like *SetOrdProp*, *SetStrProp*, *SetMethodProp*, and so forth.

These set methods require a *PPropInfo* reference to the property. This reference is generated by a call to *GetPropInfo*:

```
function GetPropInfo(TypeInfo: PTypeInfo;
  const PropName: string): PPropInfo;
```

You then pass this *PPropInfo* reference to an appropriate set method. The following is *SetOrdProp*, which can be used with any Integer or Longint value:

```
procedure SetOrdProp(Instance: TObject;
  PropInfo: PPropInfo; Value: Longint);
```

The use of *GetPropInfo* and *SetOrdProp* are demonstrated in the PROPINFO project. This project includes a button with the following *OnClick* event handler attached to it:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  PropInfo: PPropInfo;
  i: Integer;
begin
  if ColorDialog1.Execute then
    for i := 0 to Self.ControlCount - 1 do begin
      PropInfo := GetPropInfo(Self.Controls[i].ClassInfo,
                              'Color');
      if Assigned(PropInfo) then
        SetOrdProp(Self.Controls[i], PropInfo,
                   ColorDialog1.Color);
    end;
end;
```

When you click this button, the **for** loop iterates through all *TControl* descendants on the form. For those controls that have *PropInfo* for a *Color* property, *SetOrdProp* is called to color the control. Figure 6 shows how this form looks before clicking the button, and Figure 7 shows how it looks after the button has been clicked.

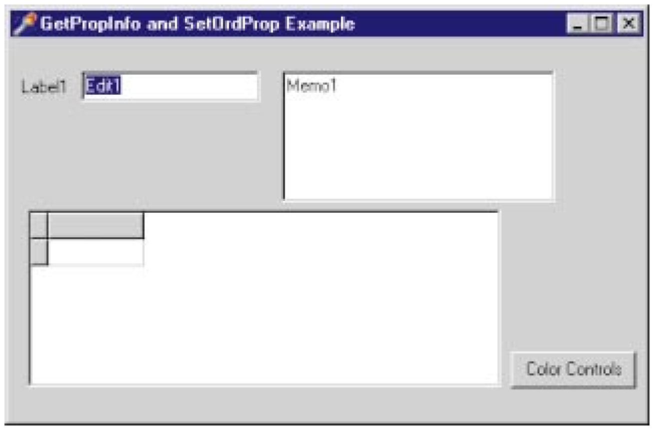


Figure 6: The PropInfo main form when it's first displayed.

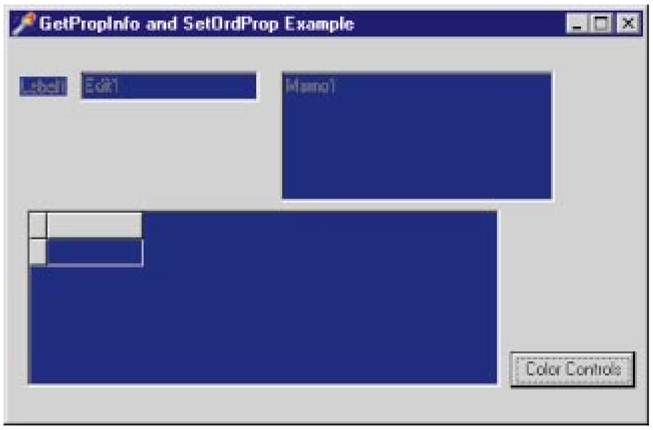


Figure 7: The PropInfo main form after clicking the button labeled **Color Controls** and selecting the color *clBlue*.

Conclusion

RTTI is information about the published properties of your classes that the compiler stores in your executable. Using the procedures and functions of the typinfo unit, you can extract this information at run time. When used effectively, RTTI can simplify your code and reduce your reliance on string constants that can be difficult to maintain. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806CJ.

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

By Binh Ly

COM Callbacks

Part I: A Hand-coded Callback Interface Manager

A COM callback interface allows a COM server component to invoke methods of objects that reside on the client application. Callbacks are usually handy for notifying the client whenever important changes on the server need to be tracked. An example where a callback may be useful is to provide data change notifications to client users in a multi-user, client/server database application.

This two-part series will demonstrate two methods of implementing callbacks in your Delphi 3 applications. This month's topic is a hand-coded callback interface manager; next month the discussion turns to connectable objects, i.e. connection points.

A Simple Chat Program

To demonstrate both methods, we'll develop a simple multi-user chat application. This application consists of a client and a server. On the server side (*ChatServer*), a chat channel object (*ChatChannel*) is used to implement a single chat channel to which several clients can connect. To be able to connect to *ChatChannel*, each client will create a connection server object (*ChatConnection*) from which it can access the single *ChatChannel* object that resides on the server. Each chat client can then, through *ChatConnection*, request *ChatChannel* to broadcast a chat message to every other client that's connected to *ChatChannel*. Figure 1 shows the interaction among the client, *ChatConnection*, and *ChatChannel* objects.

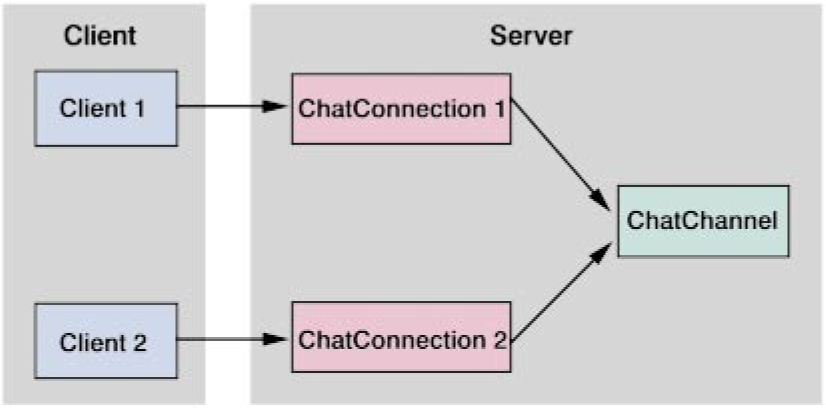


Figure 1: The chat client, *ChatConnection*, and *ChatChannel* objects.

For *ChatChannel* to broadcast chat messages to its clients, each client application will implement an *IChatEvent* interface using a *ChatEvent* object. The client will then pass a pointer to this interface (to *ChatChannel*) so *ChatChannel* can invoke a callback method of *IChatEvent*. After *ChatEvent* receives a broadcast call from *ChatChannel*, it will display the chat message on the client's main form to simulate the effect of a live chatting session, such as one of those chat services on the Web. A closer look at the chat application's objects can be found in the next section.

Chat's Application Objects

Figure 2 shows the *IChatChannel* and *IChatConnection* interfaces for the *ChatChannel* and *ChatConnection* server objects, respectively. Using COM, the client initially creates a *ChatConnection* object, then uses *ChatConnection*'s *ChatChannel* property to connect to the single *ChatChannel* object on the server.

```
IChatChannel = interface
  function ConnectUser(const Callback: IChatEvent;
                       var UserId: Integer): WordBool;
  function DisconnectUser(UserId: Integer): WordBool;
  procedure BroadcastMessage(const UserName,
                             Message: WideString);
end;

TChatChannel = class(TAutoObject, IChatChannel);

IChatConnection = interface
  procedure BroadcastMessage(
    const UserName, Message: WideString);
  property ChatChannel: IChatChannel
end;

TChatConnection = class(TAutoObject, IChatConnection);
```

Figure 2: The *IChatChannel* and *IChatConnection* interfaces.

```
IChatEvent = interface
  procedure GotMessage(const UserName,
                       Message: WideString);
end;

TChatEvent = class(TAutoIntfObject, IChatEvent);
```

Figure 3: The *IChatEvent* interface.

```
TChatUser = class
public
  UserId: Integer;
  Callback: IChatEvent;
end;

TChatUsers = class
public
  function AddUser(const Callback: IChatEvent;
                   var UserId: Integer): Boolean;
  function DeleteUser(UserId: Integer): Boolean;
  function FindUser(UserId: Integer): Integer;
  function Count: Integer;
  property Items[i: Integer]: TChatUser read GetItems;
end;
```

Figure 4: The *TChatUsers* and *TChatUser* classes.

The client then calls *ChatChannel*'s *ConnectUser* method to establish a connection to the channel so the client can start participating in a chat session. *ConnectUser* accepts a *Callback* parameter of type *IChatEvent*, which is used by *ChatConnection* to notify each of its clients whenever it needs to broadcast a chat message. The *UserId* parameter is used by *ChatChannel* to return a connection ID number, which will be used by the client when it later wants to disconnect from the channel using the *DisconnectUser* method.

Using *ChatConnection*, the client can call *BroadcastMessage* to broadcast a chat message to other chat clients connected to *ChatChannel*. *BroadcastMessage*'s *UserName* parameter is used to tell other clients the identity of the broadcasting user; *Message* is a string the client wishes to broadcast. *BroadcastMessage* internally calls *ChatChannel*'s *BroadcastMessage* method which, in turn, iterates through *ChatChannel*'s list of chat client users and calls the *GotMessage* method for each user's *IChatEvent* callback. Figure 3 shows the *IChatEvent* interface for the client's *ChatEvent* object.

Now we're ready to look at an implementation of a hand-coded callback interface manager for our chat server. The basic implementation of an interface callback manager is for the server to maintain an internal list of the callback interfaces implemented by each client object. For the server to maintain this list, the client must explicitly tell the server when it wants to connect and disconnect its callback interface. This can easily be done by exposing a method of one of the server objects where the client can pass a pointer to the callback interface as a parameter.

Implementing the Server

Our chat application server provides a few objects that can demonstrate the necessary parts in implementing a simple callback mechanism. *ChatChannel*'s *ConnectUser* method allows the client to pass in an *IChatEvent* interface pointer that *ChatChannel* stores internally so it can later use it to broadcast chat messages to the client.

ChatChannel implements *ConnectUser* using a helper class (*TChatUsers*), which is simply a list of individual chat user (*TChatUser*) items. Figure 4 shows the *TChatUsers* and *TChatUser* classes. *ChatUsers* manages the chat clients for *ChatChannel* by storing each user's callback interface together with a unique ID (which is used to identify each client connection) using the *TChatUser* class. *ChatChannel* can then simply add, remove, or find a user, and iterate all the users by implementing the methods of *TChatUsers*.

Figure 5 shows the implementation of *ChatChannel*'s methods using a contained *TChatUsers* instance, *FUsers*. Note how easy it is for *ChatChannel* to implement its *BroadcastMessage* method by using the *Count* and *Items* properties of *TChatUsers*.

If you recall, *ChatServer* provides a *ChatConnection* object the client uses to connect to *ChatChannel*. *ChatServer* contains only one instance of *ChatChannel*, but can have multiple *ChatConnection* objects that connect to this single *ChatChannel* instance. *ChatConnection* implements this mechanism by maintaining a global instance of *ChatChannel* (*MainChatChannel*), creating this instance the first time it is requested, then simply returning this global instance on succeeding requests from the client.

Because *ChatConnection* creates an instance of *ChatChannel* internally, it must take care of releasing this instance whenever it's no loner needed. Because there could be multiple *ChatConnection* objects per server, *ChatServer* tracks the count of *ChatConnections* as they are created and released, and it releases *MainChatChannel* only if the last *ChatConnection* has been released. Figure 6 shows *ChatConnection*'s implementation of this methodology using the *Initialize*, *Destroy*, and *Get_ChatChannel* methods. Also, note that *ChatConnection*'s *BroadcastMessage* method is simply a wrapper call to *ChatChannel*'s *BroadcastMessage*.

Implementing the Client

On the client side, we implement a *ChatEvent* object that supports the *IChatEvent* interface so we can handle the *GotMessage*

```
TChatChannel = class(TAutoObject, IChatChannel)
protected
  { IChatChannel }
  function ConnectUser(const Callback: IChatEvent;
                       var UserId: Integer): WordBool;
  function DisconnectUser(UserId: Integer): WordBool;
  procedure BroadcastMessage(
    const UserName, Message: WideString);
protected
  FUsers: TChatUsers;
  procedure Initialize; override;
public
  destructor Destroy; override;
  property Users: TChatUsers read FUsers;
end;

function TChatChannel.ConnectUser(
  const Callback: IChatEvent;
  var UserId: Integer): WordBool;
begin
  Result := Users.AddUser(Callback, UserId);
end;

function TChatChannel.DisconnectUser(
  UserId: Integer): WordBool;
begin
  Result := Users.DeleteUser(UserId);
end;

procedure TChatChannel.BroadcastMessage(
  const UserName, Message: WideString);
var
  i: Integer;
begin
  for i := 0 to Users.Count - 1 do
    Users[i].Callback.GotMessage(UserName, Message);
end;

procedure TChatChannel.Initialize;
begin
  inherited;
  FUsers := TChatUsers.Create;
end;

destructor TChatChannel.Destroy;
begin
  FUsers.Free;
  inherited;
end;
```

Figure 5: *TChatChannel* class definition and method implementations.

callback method. To do this, we create a *TChatEvent* class that inherits from Delphi's *TAutoIntfObject* class. We use *TAutoIntfObject* instead of *TAutoObject*, because *TChatEvent* is only an internal class to the client application; we don't really need it registered for other applications to be created.

Figure 7 shows the *TChatEvent* class. *TChatEvent* surfaces an *OnMessage* event so other objects in the client application can provide an *OnMessage* handler that does some specific processing whenever *TChatEvent* receives a chat message from the server. The first two lines in *TChatEvent's* constructor initialize *TChatEvent* so its *IChatEvent* methods can be dispatched (called) as a dual interface from the server. The constructor also calls an extra *_AddRef* so *ChatEvent* doesn't automatically get destroyed whenever the server releases the *IChatEvent* interface pointer. Note that there should be no problems about *ChatEvent* not getting destroyed when calling this extra *_AddRef* because the client application will explicitly create and free an instance of *ChatEvent*. With *TChatEvent*, our client application can easily handle chat messages coming from the server.

```
TChatConnection = class(TAutoObject, IChatConnection)
protected
  { IChatConnection }
  function Get_ChatChannel: IChatChannel; safecall;
  procedure BroadcastMessage(
    const UserName, Message: WideString); safecall;
protected
  procedure Initialize; override;
  destructor Destroy; override;
end;

procedure TChatConnection.Initialize;
begin
  inherited;
  ChatConnections := ChatConnections + 1;
end;

destructor TChatConnection.Destroy;
begin
  inherited;
  ChatConnections := ChatConnections - 1;
  if (ChatConnections <= 0) then
    MainChatChannel := nil;
end;

function TChatConnection.Get_ChatChannel: IChatChannel;
begin
  if (MainChatChannel = nil) then
    MainChatChannel := TChatChannel.Create;
  Result := MainChatChannel;
end;

procedure TChatConnection.BroadcastMessage(
  const UserName, Message: WideString);
begin
  if (MainChatChannel <> nil) then
    MainChatChannel.BroadcastMessage(UserName, Message);
end;
```

Figure 6: *TChatConnection* class definition and implementation.

Our client application consists of a main form, *TfrmMain* (see Figure 8), that allows a user to connect/disconnect from *ChatServer*, broadcast a chat message, and display incoming chat messages from other chat clients.

To connect to *ChatServer*, the client first creates a *ChatConnection* server object. Using *ChatConnection's* *ChatChannel* property, the client then registers itself as a chat user using the *ConnectUser* method, passing it an *IChatEvent* interface pointer of a *TChatEvent* instance. This is all shown in Figure 9. The variables *FChatConnection*, *FChatEvent*, and *FUserId* are all members of *TfrmMain*; *FChatEvent* is initialized as an instance of *TChatEvent*. Disconnecting from *ChatChannel* is as simple as calling the *DisconnectUser* method, passing the *UserId* obtained earlier from calling *ConnectUser*.

To broadcast a chat message, the client simply calls *ChatConnection's* *BroadcastMessage* method. To receive an incoming chat message, the client provides an *OnMessage* handler for the *ChatEvent* object. This handler takes the chat message and appends it to the **Chat Messages** memo on the main form (see Figure 10).

```
TChatMessageEvent = procedure(
  const UserName, Message: string) of object;

TChatEvent = class(TAutoIntfObject, IChatEvent)
protected
  { IChatEvent }
  procedure GotMessage(
    const UserName, Message: WideString); safecall;
protected
  FOnMessage: TChatMessageEvent;
public
  constructor Create;
  property OnMessage: TChatMessageEvent
    read FOnMessage write FOnMessage;
end;

procedure TChatEvent.GotMessage(
  const UserName, Message: WideString);
begin
  if Assigned(OnMessage) then
    OnMessage(UserName, Message);
end;

constructor TChatEvent.Create;
var
  ifTypeLib: ITypeLib;
begin
  OleCheck(LoadRegTypeLib(LIBID_ChatServer,
                          1, 0, 0, ifTypeLib));
  inherited Create(ifTypeLib, IChatEvent);
  _AddRef;
end;
```

Figure 7: *TChatEvent* class definition and implementation.

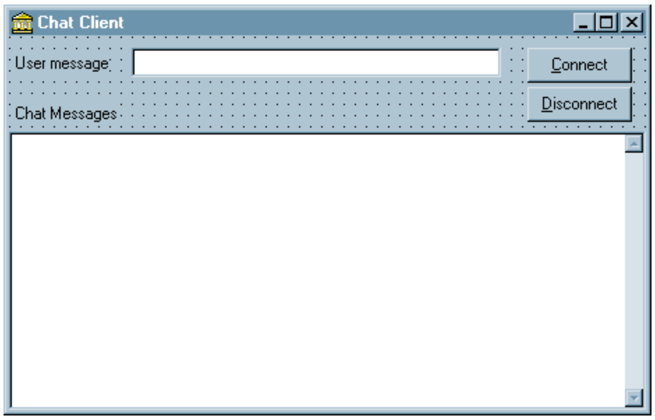


Figure 8: *ChatClient*'s main form.

Conclusion

As we've seen, a simple hand-coded interface callback manager can be implemented by writing a class that manages the list of the callback interfaces. In our chat server, *TChatUsers* is the interface callback manager. By using *TChatUsers*, *ChatChannel* is relieved of the details involved in managing each client's *IChatEvent* interface, thus making *ChatChannel*'s method implementations simple and straightforward.

On the client side, there needs to be an object that implements the callback interface passed to the server. *TChatEvent* serves this purpose, and at the same time encapsulates the callback notification method using the *OnMessage* event. This encapsulation technique ensures the code used to implement

```
procedure TfrmMain.ConnectUser;
begin
  if (FChatConnection = nil) then
    begin
      FChatConnection := CoChatConnection.Create;
      FChatConnection.ChatChannel.ConnectUser(
        FChatEvent as IChatEvent, FUserId);
    end;
end;

procedure TfrmMain.DisconnectUser;
begin
  if (FChatConnection <> nil) then
    begin
      FChatConnection.ChatChannel.DisconnectUser(FUserId);
      FChatConnection := nil;
    end;
end;
```

Figure 9: Connecting to *ChatServer* using *ChatConnection*.

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  FChatEvent := TChatEvent.Create;
  FChatEvent.OnMessage := ChatEventMessage;
end;

procedure TfrmMain.ChatEventMessage(
  const UserName, Message: string);
begin
  Memo1.Lines.Add(UserName + '> ' + Message);
end;
```

Figure 10: Providing a handler for *TChatEvent.OnMessage*.

the client application never needs to know anything about the *IChatEvent* interface details.

Because implementing client interface callbacks is a common necessity when developing distributed applications, COM provides an approach to this problem using a more generalized concept of our hand-coded interface manager. This concept is called the connection points methodology. Next month, we'll wrap up this two-part series on COM component callbacks in Delphi by introducing and implementing the connection points methodology. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806BL.

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

Delphi 1, 2, 3 / Software Piracy

By Stephen R. Broadwell

Pirates Beware!

TSeatChecker Enforces Run-time License Agreements

Sometimes, it seems as though software piracy was the third oldest profession. No matter how much effort is put into protecting software, someone always finds a way to outsmart the programmers and use or distribute the application illegally. This is especially true in Europe, where piracy is a far greater problem than in the US. The worst part is that most software pirates are themselves programmers, and should realize that by hacking software, they are simply cheating other programmers out of just compensation for their work.

Software piracy comes in different forms. There is the direct, deliberate piracy, wherein some zit-faced teenager hacks his way into the latest game, places an assembly JMP command conveniently around the algorithm that checks if the correct phrase has been typed in from the appropriate page of the manual, then uploads the resultant zip file onto forty or fifty of his favorite Internet sites. But there are also far more common and less dramatic forms of piracy. A company buys five licenses to use an application, then conveniently neglects to check and see that fifteen people have installed it and are using it. Or perhaps they have hired some new people lately, and simply haven't noticed that they've exceeded the number of users prescribed by their license agreement. And yes, maybe they deliberately cut corners by purchasing a smaller number of licenses than they knew they would use.

Whatever the reason, software license agreements are violated *en masse* on a daily basis. It's naive to think that simply displaying the agreement in a dialog box with an I Accept button in the installation program will protect a company from lost revenues. The software itself must be "aware" of the license, and must be able to enforce it.

In this article, I present *TSeatChecker*, a Delphi component for enforcing software license agreements programmatically. (The component and a demonstration program are available for download; see end of article for details.) While it doesn't present the ultimate solution, it does a pretty good job of detecting licensing violations and informing the user via an error message.

Software Licenses

There are basically two types of software licenses: seat-based and concurrent use-based. Seat-based licenses are ones in which the user pays for a certain number of seats to be distributed in a static manner. The classic example is the "like a book" licensing that was popular in the late 1980s, where a seat is assigned to a user. The user may install the software on his or her PC at home and his or her PC at work, but the software shouldn't be in use at both locations at the same time.

Concurrent use-based licenses are ones in which the user pays for a certain number of seats that may be allocated dynamically. If five licenses are purchased, any number of individuals may use the software as long as no more than five use it at the same time.

The Problem

Whether seat-based or concurrent use-based, software license agreements are meaningless unless the software is able to enforce them. The problem is that most end users are motivated to violate the license agreement. This is human nature; most people, when confronted with something they're not allowed to do, will harbor a desire to find a way to do it. Thus, any system must be better than the expected competence of the end user, multiplied by their motivation to overcome the system. I call this the Competence/Motivation rule (see Figure 1).

Unfortunately, this is easier said than done. When it is launched, the software must somehow be aware of all the other instances of itself that are in use by the same customer, then determine whether to allow itself to be used by this customer. It must do all this in such a way that the end user:

1) is unaware of how it's being done, and
2) lacks the expertise and/or motivation to figure out a way around it.

A number of techniques have been developed in an attempt to address this issue, from dongles (which users hate) to semaphores (which programmers hate). What's needed is an easily implemented method for enforcing the license agreement that's effective, yet transparent to the end user.

The Solution

For multi-user database applications, there exists such a method. The key assumption is that although the users might be using the software on different PCs, perhaps even at different sites, they will be connecting to the same database. This is the common denominator for the entire installation, and we will exploit it to enforce the license agreement at run time.

This technique involves setting up a *lurch table* in the database. A lurch table is simply a table that keeps track of who is currently logged on. If the license is seat-based, the software, when run, first checks the lurch table to see if someone else with the same user ID is currently using the software. If so, it denies access until that person exits the system. If the license is concurrent use-based, the software checks the total number of users in the lurch table. If there are no more seats available, it denies access until a seat becomes available.

A lurch table is simple to set up, and it's fairly easy to add support for it in your code. Also, it's more than enough to deter the average user with the average motivation from circumventing it.

Using a Lurch Table

The simplest form of a lurch table is defined in Figure 2. This Paradox table, Lurch.db, contains two fields: Index (the primary key) and Username. For convenience, I made Index an Autoincrement field. I also put a secondary index

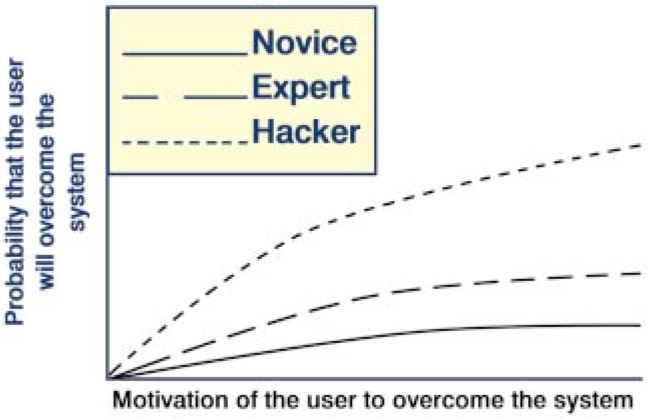


Figure 1: The Competence/Motivation rule.

on Username so I could easily search through the table to find a particular user name.

Once an application determines there is an available seat, it appends a record to the lurch table and stores the user's username in it. It posts the record, then re-edits it. The record remains in the edit state as long as the user is using the application. This is key; keeping a Paradox table's record in edit state locks it. And as long as the record is locked, it cannot be deleted from the table. It will only be unlocked if the application performs an update or a cancel, or if the user's workstation is rebooted or otherwise disconnected from the database.

When another user runs the application, it goes through the lurch table and attempts to delete each record. The ones it is able to delete are the ones that were left over from improper termination of the application, e.g. when the workstation was rebooted. The ones it's unable to delete are the ones that are being locked by other users. If this is a seat-based licensed application, the program checks if the user's username is already in the lurch table. If so, it reports an error and terminates. If this is a concurrent use-based license application, the program gets a record count of the lurch table. If there are more records than the number of concurrent users allowed by the license agreement, it reports an error and terminates.

The most straightforward way to manage a lurch table would be to put the process we've just described into the *OnCreate* event of your application's main form. Unfortunately, this approach is clumsy and results in code bloat in the main application. A better and more object-oriented technique would be to design a component to manage this. Ideally, you would drop such a component onto the login form at design time and modify two important properties: *DatabaseName* and *Method*. Then, at run time, all your application needs to do is feed this component a *Username*, and call its *SitDown* method to grab and hold one of the seats in the lurch table.

Name	Type	Description
Index	Autoincrement	Primary key
Username	Alphanumeric (40)	Secondary index

Figure 2: A simple lurch table has two fields: Index and Username.

Introducing *TSeatChecker*

TSeatChecker is a Delphi component I wrote that demonstrates the use of a lurch table to enforce software license agreements (see Listing One, beginning on this page). It descends directly from *TComponent* and adds the *DatabaseName*, *Method*, *Status*, and *Username* properties. It also overrides the constructor and provides two new methods: *SitDown* and *StandUp*.

The *DatabaseName* property identifies which database holds the lurch table. This is the main database for the application. *Method* specifies whether the license is seat-based or concurrent use-based. *Status* indicates whether a seat has been reserved. It's a writable property; setting it will change the status, i.e. "sit" the application down (reserve a seat), or "stand" it back up (release a seat). *Username* specifies the user's login name, which is necessary for seat-based licensing.

The *SitDown* and *StandUp* methods reserve and release a seat, respectively. *TSeatChecker* also overrides the constructor to create an instance of *TTable* called *fLurchTable*, which is used to access the lurch table and keep the seat record locked. It also defines a protected virtual method named *EmptyLurchTable*, which, as the name suggests, deletes all the unlocked records from the lurch table.

Using *TSeatChecker*

TSeatChecker is best used in your application's main form, since it must be in existence for as long as the user is using your application. Drop a copy of *TSeatChecker* onto the form, set the *DatabaseName* property to the application's database alias, and set the *Method* property to either *mByTotal* (for concurrent use-based licenses) or *mByUser* (for seat-based licenses).

Then, when the user enters the correct username and password into a password dialog box (see Figure 3), set the *Username* property to the correct value and call the *SitDown* method. An exception will be raised if *TSeatChecker* is unable to reserve a seat. When your application is finished, call the *StandUp* method (although the seat will be released even if you forget to call the method).

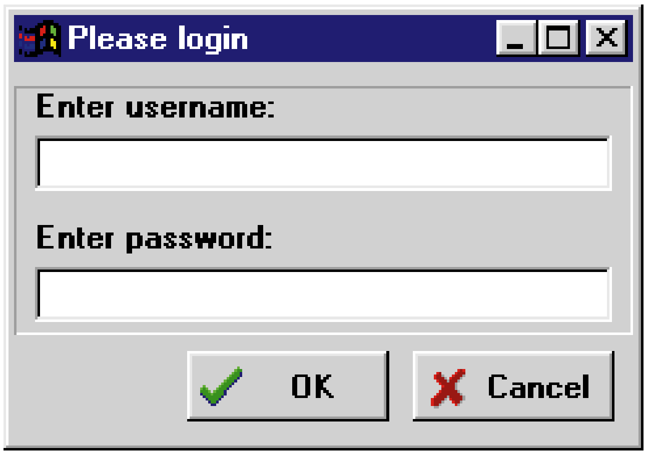


Figure 3: The *TSeatChecker* demonstration program's password dialog box.

Notes About *TSeatChecker*

TSeatChecker is presented here for academic purposes only. You will no doubt identify some areas that could easily be improved. For example, you may wish to rename the lurch table from Lurch.db to something more subtle (after all, there's no telling how many people are reading this article). You may also want to encrypt the username that's passed to the lurch table to prevent users from figuring out it's a lurch table. If you have a suite of applications that use the same database, you may wish to add a field to the lurch table (and a corresponding property to *TSeatChecker*) to identify the name of the application being used, so licenses for separate applications can be enforced using the same table.

As a programmer, I was tempted to write a property editor for the *DatabaseName* property that would display a drop-down list of all the aliases in the BDE. However, in the interest of providing clean and simple source code, I restrained myself. You will probably wish to add this feature in your implementation.

In *TSeatChecker*, I hard-coded the value for *kMaxSeats* to 3. A better way would be to load this value from a DLL, so you can easily increase the number of seats a user may use by simply sending them a new DLL.

TSeatChecker works equally well for multiple instances of an application running on different workstations or on the same workstation. It will work with any database, as long as the database supports record locking.

Conclusion

While *TSeatChecker* is not the ultimate solution, it does a pretty good job of enforcing a software license agreement programmatically. It's easy to implement, and is more than a match for the average user with the average motivation to cheat on a license agreement. And because of its simplicity, it's unlikely to be the cause of memory leaks or protection faults. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806SB.

Stephen R. Broadwell is Senior Product Manager for Bridgeway Software, Inc. in Houston, TX. He has written several articles on Delphi programming techniques and has appeared before his local Delphi users group, the Houston Association of Delphi Professionals (http://www.hadp.org). He may be reached at sbroadwell@bridge-way.com.

Begin Listing One — *TSeatChecker*

```
unit Sccomps;
{ ScComps.PAS by Stephen R. Broadwell copyright (c) 1997
  ***********************************************************
  This unit defines the component TSeatChecker, a license-
  enforcing component for database applications.  By using
  TSeatChecker in your code, you can enforce your
  licensing agreement by limiting either the total number
  of concurrent users of your software, or the number of
  seats. TSeatChecker works by making use of a lurch table
  to keep track of who is using the software.
  *****************************************************************
  Please direct any and all questions/comments to:
  sbroadwell@bridge-way.com.
  *************************************************************** }
interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, Forms, Dialogs, DB, DBTables;

const
  { This is the name of the lurch table. You may want to
    change it for security reasons. }
  kLurchTableName = 'LURCH.DB';
  { The maximum number of seats.  This is only useful for
    applications that are licensed by concurrent use. You
    may want to make this a property, or get it from a DLL
    or an ActiveX or OLE control, to make it easier to
    adjust the number of seats at a customer site, i.e.
    drop in a new DLL to increase the licensed number of
    concurrent users. }
  kMaxSeats = 3;

type
  { This is a general exception for all errors within this
    unit. }
  ESeatCheckError = class(Exception);
  { This type defines the status of the application,
    whether it is occupying one of the licensed seats or
    not. }
  TStatus = (sSitting, sStanding);
  { TMethod defines the two methods of software licensing
     -- by user (seat-based) or by total number of users
    (concurrent use-based). }
  TMethod = (mByUser, mByTotal);

  TSeatChecker = class(TComponent)
  private
    { Private declarations. }
    fDBName: TFileName;
    fStatus: TStatus;
    fMethod: TMethod;
    fUserName: string;
    fLurchTbl: TTable;
    procedure SetDBName(value: TFileName);
    procedure SetStatus(value: TStatus);
    procedure SetMethod(value: TMethod);
  protected
    { The EmptyLurchTable method empties out all of the
      'phantom' users in the table. }
    procedure EmptyLurchTable; virtual;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    { These two methods are the business end of
      TSeatChecker.  They control the occupying and
      releasing of license seats by the application. }
    procedure SitDown;
    procedure StandUp;
  published
    { DatabaseName is the name of the application database.
      It is important to set this to the database being
      used by the application, otherwise the whole purpose
      of TSeatChecker is defeated. }
    property DatabaseName: TFileName
      read fDBName write SetDBName;
    { Status indicates whether the application holds
      a seat. }
    property Status: TStatus read fStatus write SetStatus;
    { Username is the name of the user checking for
      a seat. }
    property UserName: string
      read fUserName write fUserName;
    { Method is the method of licensing being used, either
      seat-based or concurrent-use based. }
    property Method: TMethod read fMethod write SetMethod;
end;

procedure Register;

implementation

{ $IFDEF WIN32 }
  { $R SCCOMPS.D32 }
{ $ELSE }
  { $R SCCOMPS.D16 }
{ $ENDIF }

procedure Register;
begin
  RegisterComponents('SeatCheck', [TSeatChecker]);
end;

{ TSeatChecker }

constructor TSeatChecker.Create(AOwner: TComponent);
begin
  { Begin by calling the inherited constructor. }
  inherited Create(AOwner);
  { Create the lurch TTable with Self as the owner -- this
    way we don't have to worry about destroying it. }
  fLurchTbl := TTable.Create(Self);
  { Make sure we're using the right index (important during
    the FindKey operation in the SitDown method). }
  fLurchTbl.IndexName := 'Username';
  { Initialize Status to be standing,i.e. not occupying a
    seat. }
  fStatus := sStanding;
end;

procedure TSeatChecker.SetDBName(value: TFileName);
begin
  { This is your basic property access method. You may wish
    to add some extra functionality to it, such as checking
    to see that a lurch table exists in the specified
    database, and creating one if it does not. }
  fDBName := value;
end;

procedure TSeatChecker.SetStatus(value: TStatus);
begin
  { Depending on the value of Value, call the SitDown or
    the StandUp method.  I do this so that I can test my
    application in design-mode, i.e. if I want to reserve a
    seat, I just change the value in the Object Inspector.
    This is the beauty of component-oriented programming. }
  case value of
    sSitting: SitDown;
    sStanding: StandUp;
  end;
end;

procedure TSeatChecker.SetMethod(value: TMethod);
begin
  { Make sure that a seat is not being held first. }
  if fStatus = sSitting then
    raise ESeatCheckError.Create(
      'Cannot perform this action while holding a seat.');
  fMethod := value;
end;

procedure TSeatChecker.EmptyLurchTable;
{ This method is key to the whole trick.  Every time a user
  logs on to the main application, TSeatChecker creates a
  record in the lurch table for the user and holds it in
  edit state.  As long as the record remains in this state,
  it cannot be deleted.  This is true regardless of whether
  it is being locked by an application on another PC, or by
  one on the same PC.  If the user quits the application
  without giving up the seat (e.g. their PC  is rebooted,
  or the application crashes), then the record is no longer
  in edit state, but it is still present in the table.
  This is called a 'phantom' user.  Therefore, before
  attempting to reserve a seat, TSeatChecker runs through
  the lurch table and attempts to delete every record.
  'Phantom' records will be deleted, but actual occupied
  seats will not.  Once the method is done, the only
  records remaining correspond to actual seats. }
var
  i: integer;
begin
  { Check to see that the table is not empty. }
  if not (fLurchTbl.BOF and fLurchTbl.EOF) then
    begin
      { Go to the first record. }
      fLurchTbl.First;
      { Run through all the records and attempt
        to delete them. }
      for i := 0 to fLurchTbl.RecordCount-1 do begin
        try
          fLurchTbl.Delete;
        except
          { If the record is locked (i.e. if this is not a
            'phantom' user but rather an actual license
            seat being held)... }
          on e: EDBEngineError do
            if Copy(e.message,1,30) <>
                    'Record locked by another user.' then
              raise
            else
              { ...then move on to the next record. }
              fLurchTbl.Next;
        end; { try-except }
      end; { for-do }
    end; { if-then }
end;

procedure TSeatChecker.SitDown;
begin
  { Make sure we're not attempting to sit down in more than
    one seat. }
  if fStatus = sSitting then exit;
  { Make sure we've got a valid database.  You may want to
    do additional checking here, to ensure that fDBName
    corresponds to a valid alias, using
    Session.GetAliasNames. }
  if fDBName = '' then
    raise ESeatCheckError.Create('No database specified.');
  { Close the lurch TTable and set all of its properties. }
  fLurchTbl.Close;
  fLurchTbl.DatabaseName := fDBName;
  fLurchTbl.TableName := kLurchTableName;
  { Open the lurch table and empty it of 'phantom' records
    (see EmptyLurchTable or the associated documentation
    for more details). }
  fLurchTbl.Open;
  EmptyLurchTable;
  { Check to see if another seat is available.  If the
    license is based on concurrent users, then the total
    number of occupied seats must be less than kMaxSeats,
    otherwise an exception is raised. }
  case fMethod of
    mByTotal :
      if fLurchTbl.RecordCount >= kMaxSeats then
        raise ESeatCheckError.Create('No seats available');
    { On the other hand, if the license is based on seats,
      then each user may only be logged in one time.
      Therefore, if the user attempting to log in is
      already logged in, an exception is raised (see the
      note on licensing at the bottom of this file, or
      check the associated documentation). }
    mByUser :
      if fLurchTbl.FindKey([fUserName]) then
        raise ESeatCheckError.Create(
          'This userid already has a seat.');
  end;
  { If there is a seat available, reserve it.  TSeatChecker
    does this by appending a new record to the table with
    the user's name, then editing that record (and never
    posting).  This locks the record and prevents it from
    being deleted by the EmptyLurchTable method above. You
    may wish to modify this.  Primarily, you probably want
    to encrypt the username string, so that it is more
    difficult for an end user to find a way around
    TSeatChecker. }
  fLurchTbl.Append;
  fLurchTbl.FieldByName('Username').AsString := fUserName;
  fLurchTbl.Post;
  fLurchTbl.Edit;
  { Finally, record the current status. }
  fStatus := sSitting;
end;

procedure TSeatChecker.StandUp;
begin
  { Begin by making sure we're not already standing. }
  if (fStatus = sStanding) then
    Exit;
  { Also make sure that the lurch TTable is active
    and in edit state. }
  if (not fLurchTbl.Active) or
     (fLurchTbl.state <> dsEdit) then
    raise ESeatCheckError.Create('Already standing up.');
  { Cancel the edit; delete the record; close the table. }
  fLurchTbl.Cancel;
  fLurchTbl.Delete;
  fLurchTbl.Close;
  { Finally, record the current status. }
  fStatus := sStanding;
end;

end.
```

End Listing One

Delphi 1 / Delphi 2

By Vladimir Safin

Formula Compiler

A Snappy Run-time Formula Evaluation Package

This article describes a set of Delphi components that allow mathematical functions to be quickly evaluated at run time — one to two and a half times faster than hard-coded Delphi. It also describes how to create and maintain user function libraries.

Formula parsing is becoming commonplace in scientific, engineering, statistical, and report-generating software packages. Run-time formula evaluation is indispensable for these tasks; applications such as MathCad or Mathematica couldn't work without it.

But formula parsing isn't only helpful in these fields; it's required when we need to create a flexible application. Let's consider an example; perhaps a complex database application, or even a simple graph plotting program. In such an application, some values depend on others, but the formulas used in their calculation vary. In this case, it's better to develop a way to change formulas than to change the source code and have to recompile the entire program. In fact, the latter is impossible for developing commercial products.

In addition, many applications require intensive calculations. Some scientific tasks bring the total number of function evaluations into the billions. Therefore, it would be desirable if the parser's speed was comparable to the execution speed of a hard-coded, compiled function. This can only be achieved if the parser is a real compiler (not an interpreter), since only compilation would provide adequate execution speed.

Property	Description
Lib	The name of the user function library, or *TFormulaLib* component. By setting this property, you can allow the use of functions defined in a particular library.
Source	The string containing the formula. When changed, the new formula is checked and compiled.
Args	The number of the formula's arguments; a run-time, read-only property.
UnknownNames	Determines whether unknown variable and function names are allowed.

Figure 1: The properties in the *TFormulaCompiler* component.

All these tasks can be accomplished using the Formula Compiler (FC) package. FC is a set of Delphi components that include the formula parser. There are 16- and 32-bit versions of this package. The package itself consists of three components: *TFormulaCompilerD*, *TFormulaCompilerE*, and *TFormulaLib*. FC is a real compiler that generates optimized code to evaluate the expression.

The *TFormulaCompilerD* and *TFormulaCompilerE* Components

The *TFormulaCompilerD* and *TFormulaCompilerE* components are the heart of the package. What can we do with them? We can pass the string which contains an arithmetic expression to them. It will then perform a full syntax check of the string and confirm that the string represents a valid formula. If this is true, it will then allocate memory and generate optimized machine code for evaluating the result. After the formula has been compiled, we can supply arguments and evaluate the function.

These components are similar. Only one method differs, so we won't differentiate between them in most cases; instead, we'll refer to them as *TFormulaCompiler* components. The *TFormulaCompiler* components consist of four properties, three methods, and two events. The properties are listed in Figure 1.

Let's consider the *Source* property more closely. When we assign a string, FC checks the validity of the formula and generates code if it's valid, or raises an exception if it's invalid. The syntax of valid expressions is similar to Pascal syntax. The source string can consist of numbers (they can be written in the usual form or scientific notation), arithmetic operation signs [+, -, *, /, ^], relational operators [=,<,>,<>,<=,>=,!], braces [( )], built-in and user-defined functions and their arguments, and the main arguments. The only difference between classical Pascal and FC syntax is that main arguments are integer numbers, preceded by the percent character (%). The main argument list is zero-based, hence %0 is the first argument. Built-in and user-defined functions' arguments should be placed within brackets and separated by commas. FC is case-insensitive, so it doesn't distinguish between setting the source to SIN(4) or SiN(4), for example.

Built-in Function	Number of Arguments	Meaning
SIN(X)	1	Sine of X
COS (X)	1	Cosine of X
TAN (X)	1	Tangent of X
COTAN (X)	1	Cotangent of X
ATAN (X)	1	Arc tangent of X
ACOS (X)	1	Arc cosine of X
ASIN (X)	1	Arc sine of X
ABS (X)	1	Absolute value of X
SQRT (X)	1	Square root of X
EXP (X)	1	Exponential of X
LN (X)	1	Natural logarithm of X
LG (X)	1	Decimal logarithm of X
INT (X)	1	Integer part of X
ROUND (X)	1	Rounds X to nearest integer
FRAC (X)	1	Fractional part of X
SQR (X)	1	Quadrate of X = X*X
CUBE (X)	1	Cube of X = X*X*X
POW (X, Y)	2	X in a power Y
IIF(X, Y, Z)	3	if X<>0 return Y; otherwise return Z
NOT(X)	1	if X = 0 return 1; otherwise return 0
PI	0	PI number
LN2	0	Natural logarithm of 2
CHS(X)	1	Change sign of X
FACT(X)	1	Factorial of X
BINOM(X, Y)	2	Binomial coefficient X over Y
COSH(X)	1	Hyperbolic cosine of X
SINH(X)	1	Hyperbolic sine of X
TANH(X)	1	Hyperbolic tangent of X
ASINH(X)	1	Arc hyperbolic sine of X
ACOSH(X)	1	Arc hyperbolic cosine of X
ATANH(X)	1	Arc hyperbolic tangent of X
MIN(X, Y, Z, ...)	unlimited	Minimum of arguments
MAX(X, Y, Z, ...)	unlimited	Maximum of arguments
SUM(X, Y, Z, ...)	unlimited	Sum of arguments
PROD(X, Y, Z,...)	unlimited	Product of arguments

Figure 2: Built-in Formula Compiler functions.

There are more than 30 built-in functions (see Figure 2). They are the building blocks for constructing virtually any formula. Most of FC's built-in functions are common and are used in many different programs, but there are some functions that don't have a Pascal equivalent. Some of the new functions correspond to composite pieces of code. For example, the IIF function corresponds to an **if..then..else** statement, which can be used to create branch formulas or formulas with a given condition.

Another important property of FC is that it permits an unlimited number of arguments, which neither Pascal nor Delphi allow. As we see from Figure 2, FC has four built-in functions with unlimited parameters; we can pass any number of parameters to these functions. Thus, the *MIN* function returns the minimum from two numbers, as well as from 10 numbers, without additional code or problems.

As previously mentioned, *TFormulaCompilerD* and *TFormulaCompilerE* have three methods. The main method, *F*, is used to obtain the result of a calculation. This method is different in the *TFormulaCompilerE* and *TFormulaCompilerD* components:

```
function TFormulaCompilerD.F(
  const X: array of Double): Double;
function TFormulaCompilerE.F(
  const X: array of Extended): Extended;
```

The function arguments should be supplied in the *X* array. The first number corresponds to the %0 argument, second to %1, and so on. For the correct result to be returned, the number of arguments passed to the *F* method should be equal to or greater than the value of the *Args* property (see Figure 3).

```
{ FC1: TFormulaCompilerD; }
{ X: Double }
FC1.Source := 'Frac(%1 + %0)';
{ X := Frac(3.3+2.2) = 0.5 }
X := FC1.F([2.2, 3.3]);
FC1.Source := 'Sqr(Max(%0 + 3, %1, %2))';
{ X := Sqr(Max(4, 2, 3)) = 16 }
X := FC1.F([1, 2, 3]);
FC1.Source := 'IIF(-1, 1, 2)';
{ Main arguments are unnecessary. }
{ X := IIF(-1, 1, 2) = 1 }
X := FC1.F([2, 5]);
```

Figure 3: Examples of using the *Source* property and *F* method.

In most cases, setting the *Source* property and *F* method is enough to build an application using FC. The other two methods of *TFormulaCompiler* components provide an additional service to the programmer. Using the *Recompile* and *NewFunction* methods, the programmer can provide additional checking of formulae and build it into his or her own error handler.

The *Recompile* method checks the validity of the *Source* property. It recompiles the expression if it's valid, or sets *Source* to default value 1 if it isn't. Why would you need to check the validity of a formula again? Imagine we have a user function library with some functions defined, and the *Source* property contains a reference to a user function from this library. When we delete this function from the library, *Source* will now contain an invalid formula and we may want to reset *Source* to its default value. In this case, it's convenient to use the *Recompile* procedure. The last method is *NewFunction*:

```
function NewFunction(S: string): Integer;
```

It checks the validity of the formula given in *S*; if *S* is valid, it sets the *Source* property to *S* and returns 0. Otherwise, it returns to the position in string *S*, where the first error was detected. *NewFunction* can also return a negative value if any unexpected error has occurred, e.g. a memory allocation error.

Now we know all about the properties and methods of the *TFormulaCompilerD* and *TFormulaCompilerE* components. Usually, we need to extend their capabilities (for example, by adding new user functions and constants). The *TFormulaLib* component provides this, as well as other new features.

The *TFormulaLib* Component

This component allows us to create and maintain libraries of user functions and user aliases. Because it's a non-visual component, it's derived directly from *TComponent*. Therefore, *TFormulaLib*, as well as *TFormulaCompilerD* and *TFormulaCompilerE*, can be placed on the Component palette and easily incorporated in a project. To allow user functions and aliases, which are defined in a library, to be entered into the *Source* property of *TFormulaCompiler* components, we need to set the *Lib* property to the particular library name.

TFormulaLib consists of only two methods and two properties. The first method is the *Assign* procedure, which assigns the content of another library (see Delphi Help to get more information about this method). The *Aliases* property holds a list of user aliases, and the *Functions* property is a list of user functions. Both properties are of *TFunctionList* type. We'll discuss this class and its functionality later.

Let's discuss user aliases and user functions, their proposed usage, and the differences between them. Both aliases and functions are referenced in the library by their name and have their own source. Let's consider an example of an alias and function already defined in the library:

```
{ Alias name is X, source is '%0' }
Alias   : X = '%0'
{ Function name is XX, source is '%0 }
Function : XX = '%0'
```

Once the functions and aliases are defined in the library, they can be used in the *Source* property (e.g. setting *Source* to COS(x) gives the same result as COS(%0), but is more readable). All aliases are replaced by their sources during compilation. So, using aliases allows us to define user constants, gives short names for long subexpressions, and avoids %*nn* notation (we can define new variables *X*, *Y*, *Z*, or *X0*, *X1*, *X2* to use instead of %*nn* arguments).

When we utilize user functions in the *Source* property, we should supply arguments if the function depends on some other parameters. In the case of the *XX* function, it should be written as `COS(XX(%0))`. It looks a little odd and gives the same result as COS(%0), so we should define more substantial functions, for example SIN(2*%0). User constants also can be defined as functions, but only when we don't need to provide any arguments to it.

Aliases and user functions can be defined at design time, as well as at run time. In the first case, we will use two dialog boxes: Formula library editor, and Formula editor (see Figure 4). By using them, we can perform all the operations on the library contents. In the second case, simply use the *Edit* method of *TFormulaLib*. The application was designed to be user-friendly, and shouldn't need any explanation.

To manipulate the library at run time, we have to use the *TFunctionList* class functionality. This class has four properties and three methods. The properties are:

- *Names[Index]* — Holds the names of user functions. The first function has *Index* equal to 0.
- *Sources[Index]* — Holds the sources of the functions.
- *ArgCount[Index]* — Holds the numbers of arguments in the functions.
- *Count* — A read-only property that holds the total number of entries in the library. (Note that the last entry has an *Index* of *Count* -1.)

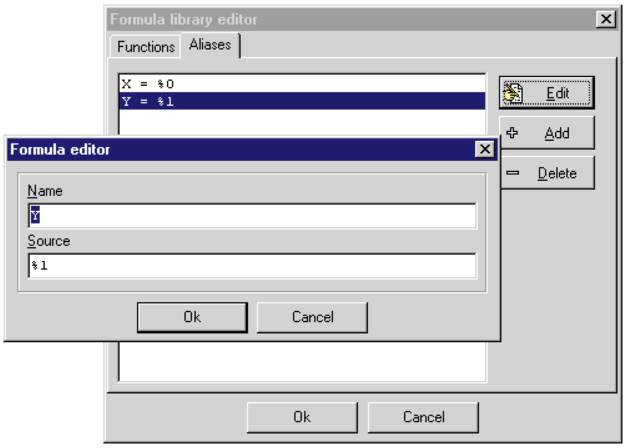


Figure 4: Formula library editor and Formula editor dialog boxes. They allows us to manipulate the library.

```
{ This is how to store library contents to the file. }
Ini := TIniFile.Create('EXAMPLE.INI');
if Ini <> nil then
  with FormulaLib do begin
    { Store user aliases. }
    for i := 0 to Aliases.Count - 1 do
      WriteString('Aliases', Aliases.Names[I],
                  Aliases.Sources[I]);
    { Store user functions. }
    for i := 0 to Functions.Count - 1 do
      WriteString('Functions', Functions.Names[I],
                  Functions.Sources[I]);
    Ini.Free;
  end;

{ This is how to load library contents from the file. }
Ini := TIniFile.Create('EXAMPLE.INI');
if Ini <> nil then
  with FormulaLib do begin
    FL := TStringList.Create;
    { Fill string list with aliases names. }
    Ini.ReadSection('Aliases', FL);
    { Load aliases. }
    for i := 0 to FL.Count - 1 do
      Aliases.Add(FL.Strings[I], Ini.ReadString(
                 'Aliases', FL.String[I], ''));
    { Clear string list. }
    FL.Clear;
    { Fill string list with user functions names. }
    Ini.ReadSection('Functions', FL);
    { Load functions. }
    for i := 0 to FL.Count - 1 do
      Functions.Add(FL.Strings[i], Ini.ReadString(
                   'Functions', FL.String[i], ''));
    { Free string list. }
    FL.Free;
    Ini.Free;
  end;
```

Figure 5: Example of storing the library contents to a file and then loading it.

Using these properties allows us to receive complete information about the user functions or aliases, which are contained in the library, as well as being able to rename them and edit their sources. It's important that the user function name satisfies the following rule: It should begin with a letter, followed by zero or more characters from the set of letters, numbers, and underscore (_) sign. In other words, the user function names follow the same syntax as Object Pascal identifiers.

The *TFunctionList* class has three methods. The first is the *Add* method, which adds a function or alias with a given name and source to the library. This method returns 0 if the addition was successful, or an error code otherwise. Error codes are described in the section "FC Errors." The *Delete* method deletes the specified function from the library. The *IndexOf* method returns the position of the specified function in the library. The following code deletes the function with the name *SampleFunction* from the library:

```
with FCLib.Functions do begin
  Temp := IndexOf('SampleFunction');
  if Temp >= 0 then
    Delete(Temp);
end;
```

Figure 5 is an example of how we can store the contents of the library to a file and load it again later. To make working with functions and aliases easier, it's a good idea to store them in an .INI file (in the example, EXAMPLE.INI) with two sections (Aliases) and (Functions), where each entry is stored in the form Name=Source (for example: SIN2x=SIN[2*%0]).

```
function TMainForm.FCFunction(const FuncName: string;
  const X: array of Extended): Extended;
var
  i: Integer;
begin
  Result := 0;
  { Compare function name. }
  if CompareText(FuncName, 'AVERAGE') = 0 then
    begin
      { AVERAGE evaluation. }
      for i := 0 to High(X) do
        Result := Result + X[i];
      Result := Result / (High(X) + 1)
    end;
end;

function TMainForm.FCVariable(
  const VarName: string): Extended;
var
  i: Integer;
begin
  Result := 0;
  { Compare variable name. }
  if CompareText(VarName, 'RNDVAR1') = 0 then
    Result := Random(1000)/1000
  else if CompareText(VarName, 'RNDVAR10') = 0 then
    Result := Random(1000)/100;
end;
```

Figure 6: Using *OnFunction* and *OnVariable* events.

Unknown User Functions and Variables

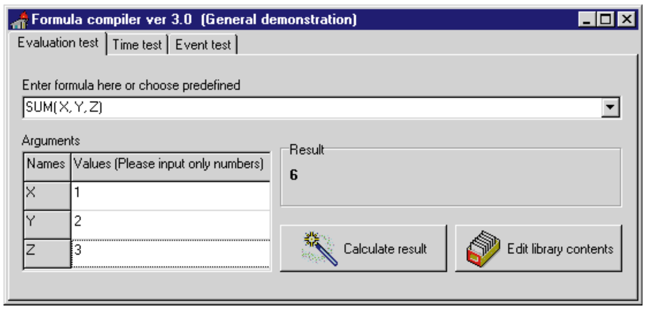
It's convenient to create and use new user functions and constants by means of *TFormulaLib*. But sometimes it becomes necessary to evaluate something unique or complicated that can't be expressed by only FC built-in functions (e.g. random values or different statistic functions). In this case, *TFormulaCompiler* components give us the opportunity to use such custom functions or variables. In other words, FC can use procedures written in Delphi.

There are two events: *OnFunction* and *OnVariable*. The first one occurs when the parser should evaluate an unknown user function. The second occurs when FC needs an unknown variable value. The example in Figure 6 illustrates how we can use this feature. Within the *OnFunction* event, the *AVERAGE* function, with an unlimited number of arguments, is evaluated. The *OnVariable* event is used to assign a random value. Now, we can set *Source* to:

```
AVERAGE(RNDVAR1, 2, 3, RNDVAR10)
```

FC Errors

When we work with a library, some errors can occur while adding or editing aliases or functions. Errors can also occur during the formula compilation process or when an unknown function or variable name is used, but an appropriate event handler isn't assigned. Most programmers prefer to maintain errors by themselves, so later we will consider the different errors which can occur during the use of FC.



Enter formula here or choose predefined: SUM(X,Y,Z)

Arguments

Names	Values (Please input only numbers)
X	1
Y	2
Z	3

Result: 6

Figure 7: A general demonstration involving the advanced calculator, user library manager, and time test.

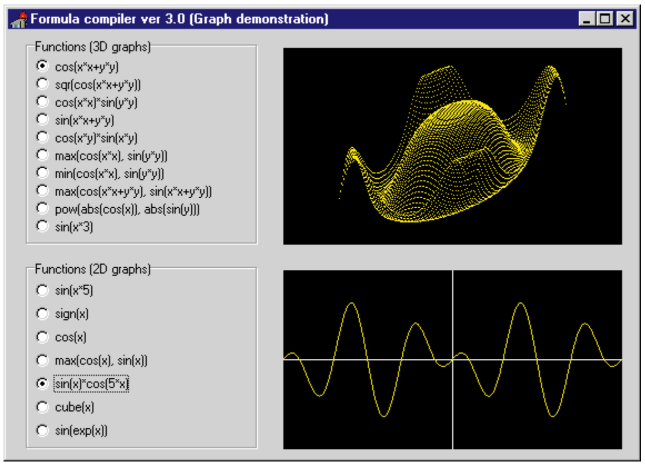
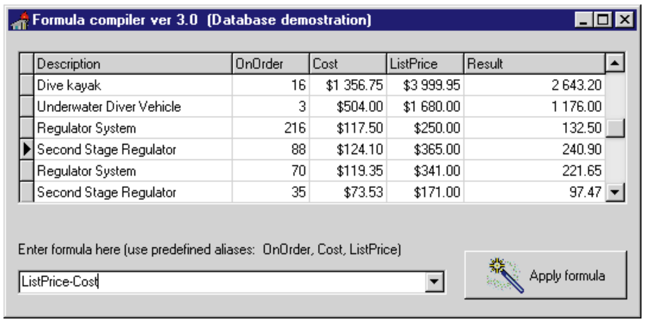


Figure 8: A graph demonstration.



Description	OnOrder	Cost	ListPrice	Result
Dive kayak	16	$1 356.75	$3 999.95	2 643.20
Underwater Diver Vehicle	3	$504.00	$1 680.00	1 176.00
Regulator System	216	$117.50	$250.00	132.50
Second Stage Regulator	88	$124.10	$365.00	240.90
Regulator System	70	$119.35	$341.00	221.65
Second Stage Regulator	35	$73.53	$171.00	97.47

Enter formula here (use predefined aliases: OnOrder, Cost, ListPrice): ListPrice-Cost

Figure 9: Simple example of using FC in a database application.

Several integer constants are defined in the unit. These codes are returned by the *Add* and *NewFunction* methods, allowing us to trap, identify, and handle errors as necessary. These methods can also return a positive value position — indicating the position where the first syntax error was detected. So, using these error codes, we can create custom error handlers. However, FC has a built-in error handler, the *FCCheck* procedure:

```
procedure FCCheck(ErrorCode: Integer);
```

This procedure raises the *EFCError* exception with the message, according to the *ErrorCode* parameter value. Thus, this feature allows us to write the following Object Pascal code:

```
FCCheck(NewFunction('COS(4*)'));
```

which, in turn, generates an exception with the message "Syntax error at pos 7."

Demonstration

The demonstration projects included with this article show some ways of using the *TFormulaCompiler* and *TFormulaLib* components. (The projects are available for download; see end of article for details.)

The first demonstration (see Figure 7) includes an advanced calculator, user function, aliases manager, and time and events test. The calculator gives the user an opportunity to input an expression in the edit field, change parameters, and evaluate the result, while reporting any error that has occurred. The function manager allows the user to enter new functions and aliases, and then use them in the calculator. The time test compares the calculation speed of a built-in function and the same function coded in the unit. To change the time test function, we would have to change the source code a little.

The graph demonstration (see Figure 8) allows the user to choose between 10 3D surfaces, and seven 2D graphs. After choosing a surface or graph equation, it will be immediately calculated and plotted. All calculations are performed by the *TFormulaCompilerE* component.

The database demonstration (see Figure 9) shows how we can use FC with databases. This project uses the Parts.db table (from the Delphi example MASTAPP application). Some fields are displayed in the grid, and one field is user-defined (calculated). The field value is calculated by the *TFormulaCompilerE* component. We can input the formula using field names, and the result field will be recalculated according to this formula. It's possible to use a formula from the predefined functions.

Conclusion

The FormulaCompiler package allows us to evaluate arithmetic expressions on-the-fly, at run time. Because it's a real expression compiler, it provides very quick function evaluation. It performs full syntax checking of the function to be compiled and detects all possible errors. The *TFormulaLib* components provide a convenient tool for creating and maintaining user function libraries. Add the *TFormulaCompiler* component to a project for a fast and flexible database application. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806VS.

Vladimir is a programmer at JSC Slavneft-YANOS in Russia. He has been working with Delphi since version 1.0 and is the author of several Delphi components that can be found at http://www.uniyar.ac.ru/lads/dpage.htm. Vladimir can also be reached via e-mail at vlads@univ.uniyar.ac.ru or safin@yorp.yaroslavl.su.

By Alan C. Moore, Ph.D.

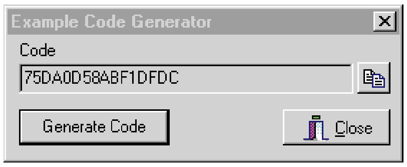
OnGuard

Software Protection for Your Delphi Applications

The shareware concept has certainly had its impact on the software industry. The ability to try a program before paying for it is extremely attractive to software users. The problem with shareware (from the developer's point of view) is equally well known: The percentage of users willing to register, as a matter of conscience, has always been small.

Software developers soon realized other "incentives" were needed to ensure that people would register the software they were using. As the shareware concept evolved, commercial software producers took a cue from the shareware developers and began offering demonstration versions of their software. Unfortunately, some of the early demonstration versions did little more than indicate the look and feel of the software. Today, you can produce nearly- or fully-functional evaluation versions of your software using a variety of protection schemes — protection that is essential if you plan to distribute your applications over the Internet, a practice which is becoming commonplace.

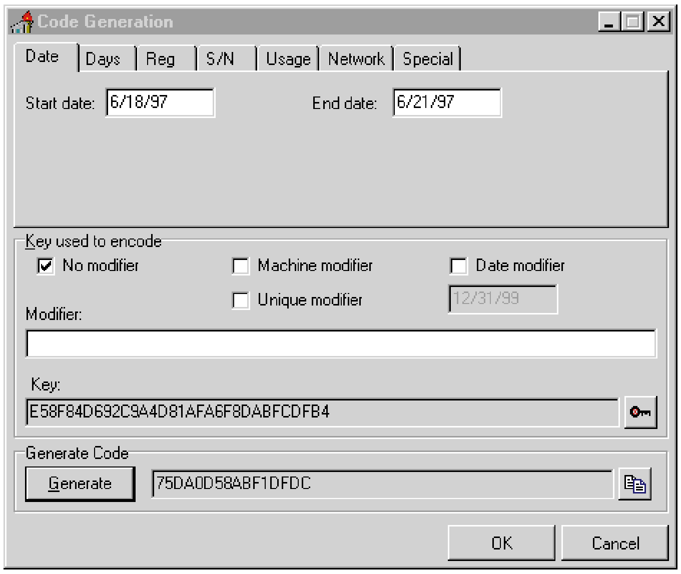
OnGuard, TurboPower Software's new library of non-visual components, allows you to easily choose from a variety of protection methods. As we'll see in the example program I've written to accompany this review (see end of article for download details), you can even combine protection schemes to gain more security.



Example Code Generator

Code: 75DA0D58ABF1DFDC

Generate Code | Close



Code Generation

Date | Days | Reg | S/N | Usage | Network | Special

Start date: 6/18/97 End date: 6/21/97

Key used to encode

[✓] No modifier [ ] Machine modifier [ ] Date modifier

[ ] Unique modifier 12/31/99

Modifier:

Key: E58F84D692C9A4D81AFA6F8DABFCDFB4

Generate Code

Generate 75DA0D58ABF1DFDC

OK | Cancel

Figure 1: The *TOgMakeCode* component editor in action.

The Key to Protection

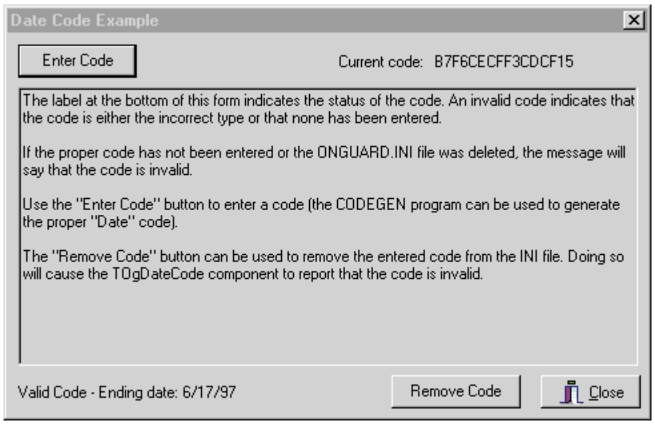
Like the other, more general, software-protection libraries, OnGuard uses hidden keys to mask its release codes. While it contains all the power of some more expensive products, it differs in one important respect: Because it's built with Object Pascal and uses native Delphi components, it's fully integrated into the Delphi environment. Therefore, it's customizable and extendable.

Using OnGuard's tools, you can build an application that is partially- or fully-functional during a trial period in which your customers evaluate it. If they decide to purchase your application, you can simply give them a release code to unlock any disabled features and make the program completely functional. This is just one of many scenarios that OnGuard supports. Others include releasing several different versions of the same software in a single .EXE file, establishing a leasing system, controlling the numbers of users on a network, and more.

As already mentioned, OnGuard's protection system is based on the use of keys and release codes. There's an important difference between these two crucial data items: Keys are

Component Name	Description
TOgDateCode	A start/end date release code with both dates encoded; used to limit features or when an application can be used.
TOgDaysCode	This release code limits the days (times unlimited) that an application (or specific features of it) can be run.
TOgUsageCode	This release code limits the number of times an application can be executed.
TOgRegistrationCode	Text-based release code derived from a user's name, a company name, or any data that could be stored in a string.
TOgSerialNumberCode	Integer-based release code that checks for a serial number.
TOgNetCode	Release code to monitor and restrict the number of simultaneous instances that can run on a network.
TOgSpecialCode	Special release code for other programmer-defined schemes.

Figure 2: The OnGuard release-code components.


Date Code Example

Current code: B7F6CECFF3CDCF15

The label at the bottom of this form indicates the status of the code. An invalid code indicates that the code is either the incorrect type or that none has been entered.

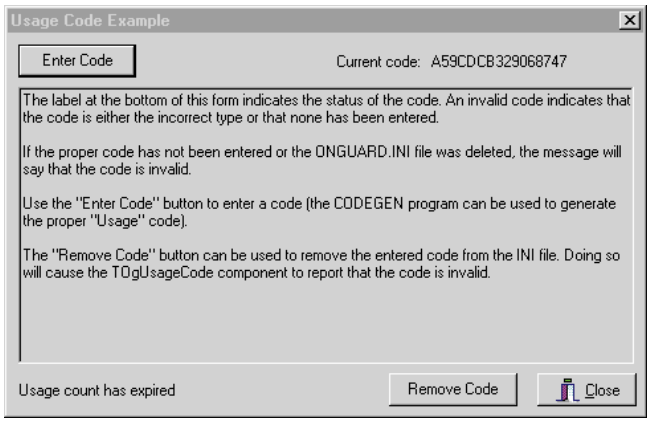
If the proper code has not been entered or the ONGUARD.INI file was deleted, the message will say that the code is invalid.

Use the "Enter Code" button to enter a code (the CODEGEN program can be used to generate the proper "Date" code).

The "Remove Code" button can be used to remove the entered code from the INI file. Doing so will cause the TOgDateCode component to report that the code is invalid.

Valid Code - Ending date: 6/17/97

Figure 3: An example program that uses the *TOgDateCode* component.


Usage Code Example

Current code: A59CDCB329068747

The label at the bottom of this form indicates the status of the code. An invalid code indicates that the code is either the incorrect type or that none has been entered.

If the proper code has not been entered or the ONGUARD.INI file was deleted, the message will say that the code is invalid.

Use the "Enter Code" button to enter a code (the CODEGEN program can be used to generate the proper "Usage" code).

The "Remove Code" button can be used to remove the entered code from the INI file. Doing so will cause the TOgUsageCode component to report that the code is invalid.

Usage count has expired

Figure 4: An example program that uses the *TOgUsageCode* component.

generally random in nature, their sole function being to mask the release codes. The data within a release code is meaningful, e.g. the first two bytes indicate the particular type of OnGuard component with which the release code is involved. The remaining bytes provide component-specific information.

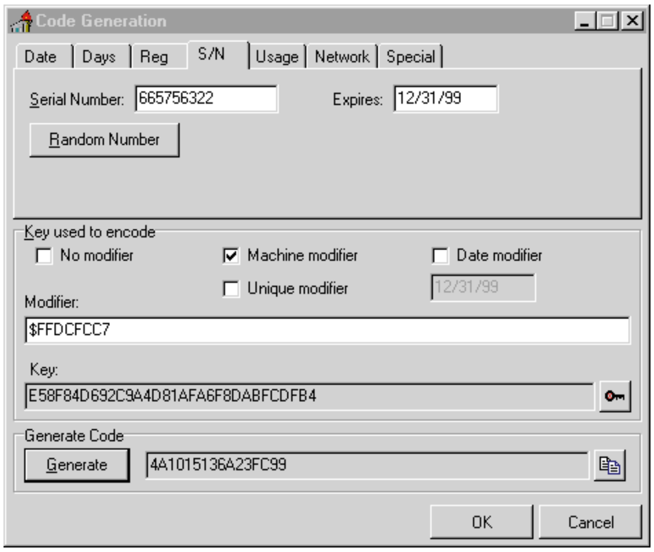
Two non-visual components are central to the process of creating keys and release codes: *TOgMakeKeys* and *TOgMakeCodes*. The former allows you to create and maintain keys for all your applications, while *TOgMakeCodes* enables you to produce the 8-byte release codes. These keys are used to encode and decode the release codes that the main OnGuard components use. Best of all, OnGuard provides two helpful component editors to manage keys and release codes. There is one utility — an example Code Generator program that, given a key and protection-type information, will give the resulting release code. The example program and the component editor that it brings up are both shown in Figure 1. The component editor is related to the *TOgMakeCodes* component that we will now discuss.

TOgMakeCodes' component editor displays a multi-page dialog box when its *Execute* method is called (by double-clicking on any of the main components we'll discuss presently). The dialog box allows you to create any of the several types of release codes. Each release code consists of 8 bytes, and is viewed and entered as 16 hexadecimal digits.

Similarly, *TOgMakeKeys* is another non-visual component that displays a small dialog box when its *Execute* method is called. You can choose from three types: random, standard text, and case-sensitive text. Random is the most common. Using a speed button, you can save the new key to the Clipboard for later use in the application.

Let's take a look at the release-code components. *TOgCodeBase*, the ancestor class for OnGuard's seven specialized release-code components, descends directly from *TComponent*. Several of its properties and methods are commonly used in its descendants. These include the important *AutoCheck* property and the corresponding *CheckCode* method. The former makes sure your application automatically checks for a proper release code before running; the latter checks for a valid release code. Since the latter method is abstract and virtual, each descendant overrides it and implements the checking in an appropriate way.

Let's examine each of the descendent components; their names and descriptions are given in Figure 2. You'll notice the first three components all have to do with a time frame in which an application can be run: *TOgDateCode* uses a start date and an end date, *TOgDaysCode* keeps track of days run, and *TOgUsageCode* simply limits the number of times an application can be run. *TOgDateCode* isn't especially secure, as a user could change a computer's clock so the date is within the acceptable date range. Still, it might be useful in some situations, or in combinations with other protection strategies. Figure 3 shows an example program that uses the *TOgDateCode* component. Figure 4 shows another example program that uses the *TOgUsageCode* component.


Code Generation

Date | Days | Reg | S/N | Usage | Network | Special

Serial Number: 665756322   Expires: 12/31/99

Random Number

Key used to encode

☐ No modifier   ☑ Machine modifier   ☐ Date modifier

☐ Unique modifier   12/31/99

Modifier:

$FFDCFCC7

Key:

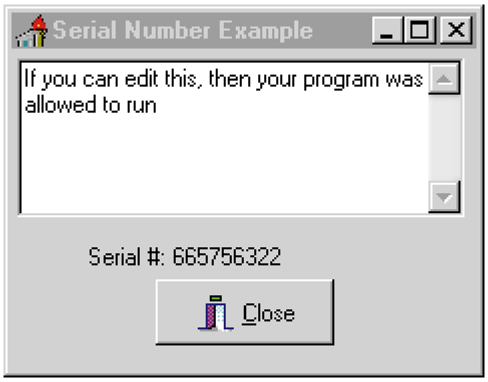
E58F84D692C9A4D81AFA6F8DABFCDFB4

Generate Code

Generate   4A1015136A23FC99

OK   Cancel

Figure 5: The component editor page used to set up a serial number.

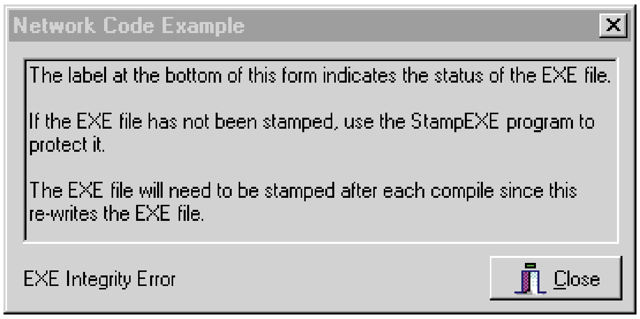

Serial Number Example

If you can edit this, then your program was allowed to run

Serial #: 665756322

Close

Figure 6: An example program using the code established in Figure 5.


Network Code Example

The label at the bottom of this form indicates the status of the EXE file.

If the EXE file has not been stamped, use the StampEXE program to protect it.

The EXE file will need to be stamped after each compile since this re-writes the EXE file.

EXE Integrity Error

Close

Figure 7: Look at what happened when I ran this example program after altering just one byte!

TOgDaysCode, which keeps track of the number of days that an application, or some of its features, can run, is also open to user tampering. Here, someone would probably just need to restore an .INI file, or the registry entry to a previous day's setting, to gain access to the application. Both of these methods call for additional security measures. With *TOgDaysCode*, an application can be run an unlimited number of times each day.

To limit the actual number of times an application can be run, use the *TOgUsageCode*. This protection method is more secure than some of the others, but a clever user could still defeat it with some effort. It has a built-in feature that keeps track of the last date the counter was decremented, adding further security.

The next two components provide tools for setting up a registration process. Based on strings or textual data, *TOgRegistrationCode* can be used to create a simple-but-effective type of release code, particularly if used in combination with other approaches. The text used in the release code could be based on a user's name, a company name, or similar data that can be stored in a string.

Similar to this, but based on numerical data, *TOgSerialNumberCode* can be used to create a serial-number release code. An obvious difference is in the data that is used as part of the code generation process. The Serial Number Registration release code uses a number instead of a text string. This is one of the more secure methods. Figure 5 shows the component editor that sets up a serial number. Figure 6 shows an example program that makes use of the release code established in Figure 5.

The last two release code components are different from those we've looked at so far. *TOgNetCode* creates a release code that limits the number of instances of an application that can run on a network at the same time. It uses a network release code and a network access file to accomplish this. This release-code component is a bit more involved than the others since it needs to monitor your application's status on the network and take appropriate actions. For example, in addition to keeping track of the actual number of instances running, it needs to deal with instances that have not been terminated properly. Unless these are dealt with properly, they might prevent a legitimate instance from running. Obviously, this component is useful for applications that are intended to run on a network. For example, with *TOgNetCode* you could create a demonstration that could run on just one or two stations concurrently, or you could enforce a licensing scenario.

The final release code component is special; in fact, it's called *TOgSpecialCode*. Based on a special value — a long integer — it lets you construct a release code containing any kind of data you want. For example, you can use the individual byte fields within the Longint to store information about specific features of your application that should or should not be implemented. For additional security, you can store a specific build number tied to a specific serial number. Because these components' release code is masked with the secret key embedded in your application, you increase the difficulty in hacking your security system when you use more than one.

Non-release-code Components

OnGuard includes two additional components that are useful in other situations where a different kind of security is needed: *TOgProtectExe* detects any changes to an .EXE file, and *TOgFirst* provides a means to determine if a second instance of an application is being run. The first of these components, *TOgProtectExe*, is particularly useful in guarding against a virus being attached to an executable file, or preventing a user from tampering with a file in an attempt to defeat a protection scheme. Like the code-maintenance components, it provides an *AutoCheck* property and an *OnChecked* event to check the .EXE file every time it's run (see Figure 7).

Some applications are intended to be run as single instances. On the other hand, you may want to restrict a demonstration application to running as a single instance, which provides the option of multiple instances once the product has been registered. The *TOgFirst* component provides an easy way to detect whether an application is running, and optionally bring the currently running application into focus if the user attempts to run another instance (see Figure 8). The *IsFirstUnit* routine determines if the .EXE file is running. The *ActivateFirstInstance* procedure facilitates bringing the first instance of the application into focus whenever the user attempts to run another instance.

Two Ways to Use the OnGuard Library

OnGuard not only provides many protection options, it also allows you to choose the level at which you use its protection tools: 1) the component level that we have been discussing, and 2) the procedural level. The latter approach provides a good deal more control but (as you've already guessed) also imposes more responsibility on the programmer. You become responsible for creating, checking, and, in some cases, updating release codes. Particularly in cases where you need to determine when and how checking is done, it makes a lot of sense to use this lower-level approach. The example program for this article provides one specific example. Let's begin with an overview of the program, review the low-level routines used, then take a more detailed look.

The main form file uses the *TOgSerialNumberCode* to first check for a valid serial number. Then, if an invalid number or no number at all is entered, it uses procedures associated with the *TOgUsageCode* component to enforce a demonstration version of the application. Once the application is properly registered, these procedures are no longer needed. The logic of this protection scheme is shown in the flow chart in Figure 8. The procedures and functions used are shown in Figure 9.

Please note that these are the low-level procedures for just one of the seven means of protection. Similar procedures are provided for the others. Most are located in a single unit, appropriately called OnGuard, that also implements most of the components. This makes it extremely easy to use the components and the low-level procedures in an application. So, how does the new example application work?

A New Example Program

The common practice used in OnGuard is to use an .INI file to keep track of registration and other protection data. While it is somewhat safe to store a release code in an .INI file, it's out of the question to store a key file there. The enumerated type, *ProgramStatus*, is used to keep track of registration data with three possible values: *psRegistered*, *psDemo*, or *psInvalid*. The property *CurrentProgramStatus* keeps track of the current setting.

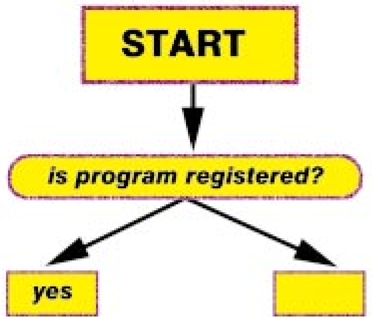


Figure 8: This flow chart shows one protection scheme implemented in the example program.

The first procedure, *SetDefaultDirectory*, ensures that the .INI file will be in the same directory as the application. If there is no .INI file found, this indicates the user hasn't entered registration information. If so, the user has the option of entering the release code immediately, which takes place in the *EnterSerialCode* procedure. Otherwise, the third procedure, *CreateIniFile*, is called to create a new .INI file, and to write a usage code to that file to limit the application to just five runs. (Of course, the user could delete the .INI file after the five runs and get five more demonstration runs. Further protection is called for!)

Every release-code component includes four events, the first three of which are required to have handlers. In this application the event handlers are as follows:

- *GetKey*, which reads the value of the embedded key;
- *GetCode,* which checks for the existence of the .INI file and reads the release code from it; and,
- *CodeChecked,* which determines if the code is valid. Because we are not using any modifier in this application, there is no need for an *OnGetModifier* event handler.

The *GetCode* procedure calls *SetDefaultDirectory*. Then, if no .INI is found, it calls *CreateIniFile* which, in turn, calls low-level routines to set up the usage code, particularly *InitUsageCode*. This procedure creates the release code from the key and the number of uses you want to allow for the demonstration version. Likewise, the *CodeChecked* procedure, which takes appropriate actions based on the status of the code, also calls a low-level usage code procedure, *CheckUsageCode*. This only happens in the case of an invalid code, indicating that the proper Serial Number Code has not been entered. You'll note that other usage code procedures are used, including *DecUsageCode*, which

Procedure/Function	Purpose
InitUsageCode	Using *key*, a *usage count*, and an *expiration date* initializes an appropriate *release code*.
IsUsageCodeValid	Boolean function that returns *True* if usage code is valid.
IsUsageCodeExpired	Boolean function that returns *True* if usage code is expired.
GetUsageCodeValue	Longint function that returns current value of usage code.
DecUsageCode	Using *key* and current *release code*, decrements remaining usages and returns a new value in the *release code* field.

Figure 9: *TOgUsageCode*'s low-level procedures and functions.

decrements the usage counter every time the demonstration version of the application is run. Hopefully, the remainder of the sample application will become obvious by studying the source code. (The project is available for download; see end of article for details.)

Manuals, Examples, and Free Help

TurboPower is noted for its excellent manuals and example programs. The OnGuard library continues that tradition. When I began working with this product, something interesting happened: Completely by surprise, and with no request from me, a new supplement to the manual and an updated program disk arrived via Federal Express. As the letter accompanying the materials explained, there had been some complaints about the original manual (I'm not convinced these were justified) that were addressed in the supplement.

This new, 34-page document included a more detailed explanation of keys and release codes for those unfamiliar with such concepts, an expanded tutorial, and detailed information on the low-level routines I mentioned previously. The disk provided seven new example programs to accompany the large collection included in the original release. Proactive, customer-service actions like this are a hallmark of this excellent company, and continue to ensure a loyal customer base.

TurboPower continues to provide free support for all its products, and free update patches via its Web site. Also, you can download demonstration versions of all of their programs, including OnGuard. To really get a good idea of what this product can do, you can download it, install it into Delphi (any version), then test it with the demonstration program I've written.

Conclusion

Needless to say, I recommend this product highly for all Delphi developers writing and distributing their applications. You work hard to produce the fine software that bears your name. You should be compensated properly for that hard work. By providing excellent tools to help you produce effective demonstration versions of your software, OnGuard can help make that happen. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\98\JUNE\DI9806AM.

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.

INFORMANT FACT FILE

OnGuard is a powerful collection of components, component editors, classes, and low-level routines that provide an effective means of protecting applications in many ways. It can be used to check for proper registration, implement demonstration versions, and monitor the running status of an application on a network or a single computer. With its excellent documentation and large number of example programs, OnGuard's components are very easy to integrate into Delphi applications.

TurboPower Software
P.O. Box 49009
Colorado Springs, CO 80949-9009
Phone: (800) 333-4160 or (719) 260-9136
Fax: (719) 260-7151
E-Mail: info@TurboPower.com
Web Site: http://www.turbopower.com
Price: US$199

table of CONTENTS

Collaborative Computing with Delphi 3

This book is, perhaps, the most daring Delphi book I've read. It asks the truly obscure questions: "What is organic computing?" "What can the Borg (alien nemesis in *Star Trek: First Contact*) teach us about the future of computing?" Then there's my question: "How far can an author run with his passion for alliteration without losing his readers?" More seriously, the central question is: "If client/server is dead, what's next?" The answer: collaborative computing, of course.

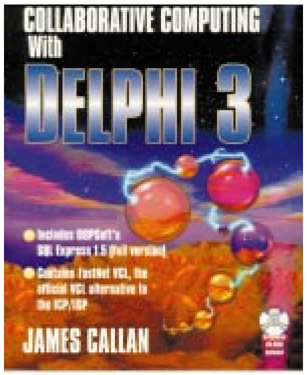
James Callan's *Collaborative Computing with Delphi 3* is also daring in its target audience, which includes client/server consultants, business analysts, SQL developers, Web developers, and IT managers. In other words, it's for just about anyone connected with Windows/Delphi development. Remarkably, it goes a long way on the path to achieving that goal.

Collaborative computing, as Callan presents it, has more to do with communication among computers, applications, and components than with people engaging in team programming. The author discusses COM, DCOM, ActiveX, and other newer technologies that help make such collaboration easier, but his major emphasis is on database and client/server development. He discusses normalization, cached updates, SQL joins, security, and other similar topics.

Callan presents his new computing paradigm with seven principles he calls "demandments." For example, under the first one, "Sell Savvy Strategies," he presents an interesting philosophy of computing in which he gives practical advice on dealing with the increasingly rapid rate of change in our field.

Before we examine the contents of the book, let's consider the definition of collaborative computing: "a model for creating software based on role-driven components, which communicate with each other to achieve highly synergistic outcomes." Callan points out that "Like people, computer collaborators must be managed." One of the most effective ways of performing this management is through OOP techniques, which allow us to define the roles of program elements and organize them logically. He covers a wide range of OOP topics, from basic elements to the differences between class methods and object (instance) methods. Now that we have a better notion of what collaborative computing is, let's take a quick tour of the book.

The first 60 pages will certainly aggravate some readers. I can hear it now: "Where's the code?" There's not even much discussion of Delphi. That all changes in chapter



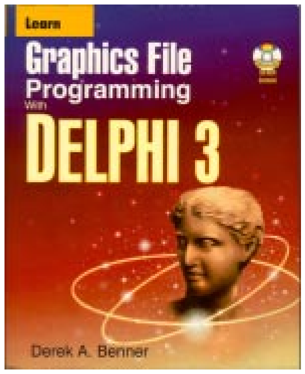
five. Callan starts to outline a fairly involved database application to enable collaborative communication between people. Refreshingly, he constructs a Paradox database from the ground up, and deals with topics you don't always find in works that rely on Delphi's sample databases,

"Collaborative Computing with Delphi 3"
continued on page 44

Learn Graphics File Programming with Delphi 3

Derek A. Benner has mined a very narrow niche in the Delphi books market with *Learn Graphics File Programming with Delphi 3*. Citing the inherent Delphi shortcoming — limiting image support to the Windows .BMP, .WMF, and .ICO formats — as a driving force behind his research, Mr Benner made the effort to compile this research into book form. Though the title indicates that programmers will learn graphics file programming, they must be prepared to approach this advanced text with additional resources to absorb the material.

The objective of the three initial chapters is to provide programmers with the background needed to understand how graphics are displayed in the Windows environment. Unfortunately, the reader might be left unfulfilled after seeing the material. Chapter two, "Windows Graphics Basics," contains six scant pages of information on the topic. Relevance is also an important consideration; the CGA and EGA display modes have limited value in today's environment. One important topic, crucial to successfully absorbing the information, is the Device Independent Bitmap (DIB), and it should receive much more attention. The acronym is used heavily throughout the



book, referenced as the middle layer between the application and the device driver. However, it's difficult to locate an expansion on the subject. DIB files should have been fully explained in the first few pages.

"Learn Graphics File Programming with Delphi 3"
continued on page 45

Collaborative Computing with Delphi 3 (cont. from page 43)

e.g. secondary indexes. The emphasis here is on good database design and creating an effective user interface. You'll encounter many useful tips and approaches during these five chapters (nearly 150 pages).

Much of the remainder of the book builds the basic framework of another ambitious application, providing another, more complete model for collaborative computing. In chapter 16, "Bier Busting in Belize," you find yourself transported to Germany as the IT manager of Perfect Plastics, Inc. Your task is to effectively manage your company's business expenses. Here you might be reminded of *Delphi Programming Explorer* [Coriolis Group Books, 1995] — by Jeff Duntemann, Jim Mischel, and Don Taylor — with its excursion into the realm of fiction. However, Callan's approach is different. His fictional narrative is less for the sake of entertainment, and more to provide a real-world scenario that demonstrates collaborative computing. In fact, it reminds me of some of the case histories in Steve McConnell's *Rapid Development* [Microsoft Press, 1996].

This chapter develops various solutions to the problem of using Entity Relationship Diagrams (ERDs). After refining the basic ERD, the author uses it to design a database in the following chapter, then builds a user interface to provide access to the database. Using Borland's Local InterBase Server, he creates a prototype on the desktop. Then, through a nice exploration of SQL, he shows how to scale up beyond the desktop. Finally, he shows how to deploy it on the Web.

Many of the techniques demonstrated in these chapters rely on technologies beyond Delphi. Many third-party (freeware, shareware, and demonstration) tools are included on the CD-ROM, and are referenced in the text. I found this approach helpful.

This overview of the book provides some indication of its contents. However, an important question remains: Who will benefit from reading this book? Although the publisher lists it as "Advanced," I can't completely agree. First, I don't believe a really advanced database programmer will find that much new information here except, perhaps, in the final chapters, which provide an introduction to deployment on the Web.

On the other hand, I do feel this book could benefit a lot of intermediate-level developers who are just beginning to explore some of these technologies. This book is ideal for someone who wants to quickly get up to speed in client/server programming, and develop an understanding of the large context in which such development operates. Clearly, Callan is a programmer with a wealth of experience. In these pages, he freely shares insights gained through many years of work in this field. Even if you don't agree with his views, I think you'll find them interesting and worth pondering.

— Alan C. Moore, Ph.D.

Collaborative Computing with Delphi 3 by James Callan, Wordware Publishing, Inc., 2320 Los Rios Blvd. #200, Plano, TX 75074, (972) 423-0090.
ISBN: 1-55622-554-7
Price: US$59.95
(827 pages, CD-ROM)

Learn Graphics File Programming with Delphi 3 (cont. from page 43)

The two chapters that follow are entitled "Using and Modifying the Windows System Palette" and "File Compression Basics," respectively. Working with the system palette is covered in a brief reference to some of the API functions needed for manipulation purposes. The programmer's education is supported by the presentation of a palette viewer project. File compression is given a fair bit of attention that includes lucid explanations of several popular compression schemes, such as LZW and Huffman encoding. This is excellent information for all programmers, regardless of their graphics interests. Chapter five, "Dithering and Color Quantization," covers topics important to the programmer, but difficult to understand without knowing the basics of Windows display schemes.

The remaining chapters are devoted to working with specific file formats grouped by their storage scheme. Coverage includes uncompressed files (.BMP and .TGA), run-length encoding (MacPaint, .PCX, and .IMG), dictionary-based (.GIF, .PNG, and .TIF), and a miscellaneous files section that covers .JPG and FlashPix formats. Some of these image file formats are superfluous and out of place in this volume; there is little chance that the Windows programmer working on modern platforms will encounter them. The format-specific chapters share a similar presentation format: A brief text presentation of the file format is followed by extensive code listings meant to be integrated in the graphics viewer project.

Of greatest interest to the general business applications programmers are the chapters pertaining to the .BMP and .JPG file formats. The Windows .BMP is probably the easiest display format to understand for two reasons. First, it's the default Windows bitmapped image file format, and is directly addressed by the Delphi *TImage* component. Secondly, the image is stored internally bit-for-bit as it was created, so there are no compression algorithms to complicate the process of reading or writing the file. The author adds to his explanation by presenting code that exposes the reader to the process of reading and manipulating the bitmap. This is more educational than just using the methods of the *TImage* component.

Readers will find the chapter dealing with the .JPG file format interesting because of its unique trait of using a lossy compression algorithm, a technique that compresses the file size by omitting pieces of the image. This type of image was designed from the start to support photographic images and the varying degrees of color contained within them. A lossy compression algorithm operates on the theory that parts of an image can be deleted without serious visual degradation. The author includes an important fact sometimes lost on those working with graphics files: Repeatedly writing a JPEG image to a .JPG file will result in continued image degradation. He reminds the reader that JPEG is not the proper image choice for an image that must stay the same as the day it was written. The chapter closes as all others do: with the code necessary to read and write the file type.

Despite its title, this book won't allow readers to lay claim to being a graphics file expert; nor is it a primer. And the primer material that *is* provided isn't sufficient to bring programmers to the point where they can begin to produce the code included in the book. The amount of text contained in the book on all topics measures less than half of the number of printed pages. The majority of the 422 pages are composed of code listings taken directly from the CD-ROM included with the book. With a title such as this, much more space should be devoted to step-by-step explanations of file manipulation. The code that is presented should be heavily commented snippets of the programs, printing a complete listing only when a contextual perspective is necessary. If a second edition of this book is being planned, more attention should be paid to the editing, because the text and the code listings don't match in a number of cases. The author's writing style also makes for a difficult read. There is little transition between chapters, and many thoughts end abruptly.

The code included with the book is difficult to work with. It shows up in the listings without explanation, and occasionally doesn't match the explanatory text. If the programmer is typing the code, should the procedures be included? This isn't explained. More likely, programmers will choose to load the code from the CD-ROM, but there is a caveat there as well: The chapter directories don't match the printed chapters. All the code appears to work as long as the reader carefully emulates the directory structure implied by the text. Don't try to place these programs in your project's directory without being prepared to debug the problems that will appear. The CD-ROM also contains a directory filled with shareware components.

The material in the book would be better presented in an anthology-type presentation rather than presenting such topics as the MacPaint file format and endless pages of code listings to fill space. An even more appropriate setting would be a series of magazine article installments. The book is not a good value if you're seeking an in-depth learning experience on the topic of graphics file programming techniques. On the other hand, if your project needs to add compatibility with the file types discussed in the book, this would be a good source for the code alone. Perhaps a second edition will consider the more descriptive title: *Graphics File Formats Handbook*.

— *Warren Rachele*

Learn Graphics File Programming with Delphi 3 by Derek A. Benner, Wordware Publishing, Inc., 2320 Los Rios Blvd. #200, Plano, TX 75074, (972) 423-0090.
ISBN: 1-55622-558-X
Price: US$49.95
(422 pages, CD-ROM)

What RAD Really Means

I've seen the term *RAD* thrown around a lot. Usually, it's in reference to some problem that Delphi didn't solve with a wave of the mouse and a double-click. This is then followed by a statement along the lines of, "This isn't RAD."

RAD, which stands for Rapid Application Development, is a lot of things. It can go to the very heart of the development process to help teams reach their goals in a more timely manner by combining many different techniques, such as prototyping and CASE tools.

RAD is *not* a silver bullet that transforms anyone who can open a development tool into a productive programmer. As much as Delphi does, there is always an entry price. Delphi does as much — or more — to obscure the gory details of Windows API programming as any tool on the market. Especially when you consider the flexibility that Delphi allows; it gives you the power to actually delve into the depths of the API, if you so desire.

Perhaps the amount of success a developer obtains by getting a prototype up instantly actually works against Delphi. By this, I mean someone codes an application that's "done" almost immediately. When they have trouble bringing the application to completion, the tool is suddenly looked at with disdain. "This isn't RAD."

When the initial euphoria of the prototype wears off, and the programmer wants to go beyond the cursory, a higher level of knowledge is required. A programming tool such as Delphi cannot shield programmers from knowing how to program.

As in life, Delphi's strength probably contributes greatly to its weakness. Because Delphi successfully prevents developers from having to recall vast tidbits of trivia relating to the Windows API, a functioning application can be built in record time. Because Delphi can take them so far so easily, some developers feel cheated when it can't take them further without becoming familiar with the tool and environment.

The 80/20 rule in software programming has existed for many years. This rule states that 80 percent of the application will be coded in 20 percent of the time. My argument is that Delphi gets you to the 80-percent point quicker than any other tool out there. Period. Furthermore, I also believe that finishing the last 20 percent of the project with Delphi is at least as fast as other tools. The results are magnified if you compare the same development techniques and planning across tools. It's not fair to point the finger at Delphi when there wasn't any planning. A project like this would inevitably encounter a rough spot no matter what tool was used.

Perhaps an example using the classic building analogy would more clearly illustrate what I'm talking about. If I was a building engineer and a tool came along claiming to build houses in record time, I would definitely evaluate, and probably use, that tool. But that would not preclude me from having to know the underlying fundamentals of building design and construction, especially as it pertains to the local building codes. After all, I wouldn't want to build a house in Wisconsin based on Southern California needs. I would also have to know how to solve problems by picking up a hammer and pounding a nail in the right place if the tool didn't do it properly.

Using Delphi, programmers will typically see an increase in productivity. However, to fully realize the benefits that Delphi can bring to your shop, you must look beyond what a tool provides. You must look at how that tool can fit into your overall development efforts. For more information on how to turn your shop into a "rapid development" shop, refer to Steve McConnell's *Rapid Development* [Microsoft Press, 1996]. ∆

— Dan Miser

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

Delphi's Own API

With its expert interface and Open Tools API, Delphi is incredibly extendable and customizable. In fact, it's the polymorphic programming environment *par excellence* of the late 1990s. Taking advantage of that power, however, is another matter. Until now, you had to learn the language of Delphi's API by either studying the source files (VirtIntf.pas, ExptIntf.pas, etc.), or reading Ray Lischner's comprehensive treatises: *Secrets of Delphi 2* [Waite Group Press, 1996] and *Hidden Paths of Delphi 3* [Informant Press, 1997].

Now there's an easier way. Last October, I wrote a review of a powerful Delphi add-on named Raptor, then in beta and subsequently released as CodeRush. There, I concentrated on its extensive keyboard templates and its panels. What prompted its creation? Its main architect, Mark Miller, explains: "High-speed tools are the nature of our business. Once [new programmers begin] to master Delphi, [they] begin to realize the need for new tools and features."

But there's another side to CodeRush — a powerful API called RAPID — which gives us much easier access to Delphi's API. With RAPID you can create your own CodeRush add-ons or panels, and extend Delphi in just about any way imaginable. It also gives you access to undocumented Delphi capabilities that could otherwise involve weeks of hacking. Miller encouraged me to write my own CodeRush plug-in, which I did. With its bookmarked, heavily-commented templates for each type of add-on, I found the whole process remarkably easy. I wrote a plug-in to make project identifiers readily available at any time.

While Delphi 3 and CodeRush both provide tools for quickly inserting text into code, neither provided the exact functionality I was looking for. I needed a means of storing application-specific identifiers (variables, procedure names, etc.), which I could select from a drop-down list and paste into code.

My First CodeRush Plug-In

Using a CodeRush template file as a basis, I quickly wrote a basic outline. Since I needed to add a Combo box to the CodeRush toolbar, I also studied another template file that explained how to add new toolbar controls. As with Delphi experts, CodeRush plug-ins require the implementation of certain basic housekeeping functions. Since I planned to add entries to the **Editor** menu, I also had to create command constants and resource string captions for each menu item — and make sure everything was properly registered with CodeRush.

You may be wondering, "Is it possible to add new commands to Delphi's **Editor** menu without CodeRush?" The answer is, "Yes, but not easily." I had read Lischner's discussion of menu interfaces in *Hidden Paths*; I checked again, but found nothing concerning this menu. I wrote to both Lischner and Miller and got essentially the same answer: "You can do this, it's been done, but it's undocumented and fraught with danger." They both provided hints on how to proceed, but I decided to concentrate instead on building this plug-in.

Next, I added the new Combo box to the toolbar. I needed to create handlers for the added functionality (double-click and keypress handlers) and register everything. Now, whenever I double-clicked on the Combo box, the focused identifier would be pasted into the code at the current pointer position. Just what I wanted!

There's more detail and features, but this gives you an idea of what's involved. You can download the plug-in's full source code and compiled package from the *Delphi Informant* site (you'll need CodeRush to run it). Also, I plan to write a more detailed account of this plug-in and publish it on the Eagle Software site (http://www.eagle-software.com).

You may be thinking that none of this really applies to you as an application developer. Consider what Danny Thorpe wrote in his book *Delphi Component Design* [Addison-Wesley, 1995]: "[Writing custom components] is an occasional task that should be in the repertoire of a sharp application writer." Today, I would extend this to writing experts and IDE add-ons. Mark Miller says, "I will always trade two weeks of tool development for a 10-second savings in time, providing that savings can be applied several times, every day, ideally by every member of your development team." Beyond the total savings of time is "preventing an interruption to the flow of creativity which could easily cost 20 minutes or significantly more [with the introduction of bugs]."

I think developing new productivity tools is definitely something worth considering for all of us. If you would like to hear more on this topic, please let me know. Δ

— Alan C. Moore, Ph.D.

The files referenced in this article are available on the Delphi Informant Companion Disk, or for download from the Informant Web site at http://www.informant.com/delphi/dinewupl.htm. File name: DI9806FN.ZIP.

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