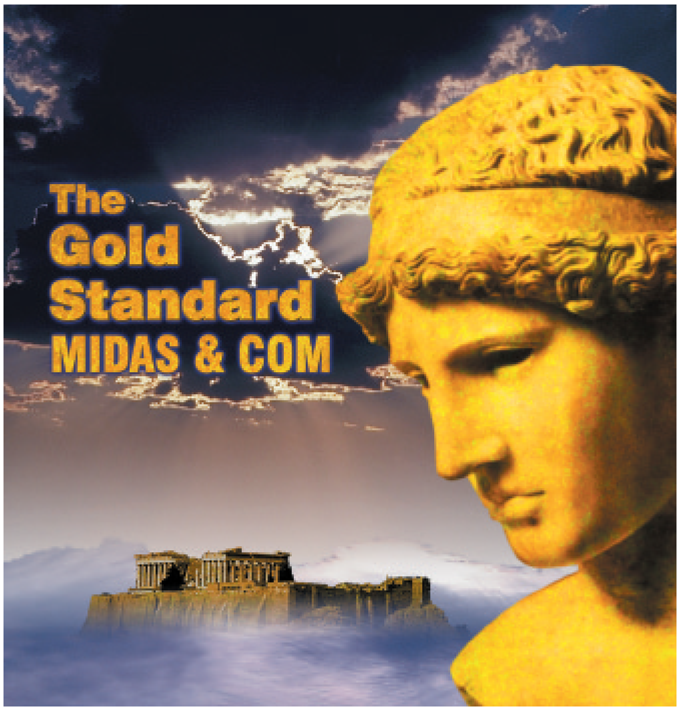
August 2000, Volume 6, Number 8



Cover Art By: *Arthur Dugoni*

Delphi TOOLS

New Products and Solutions

Dart Announces PowerTCP WebServer Tool

Dart Communications announced *PowerTCP Web-Server Tool*, an alternative for Web application developers who want to work within proven application development environments, such as Delphi, C++Builder, Visual Basic, and Visual C++. PowerTCP Web-Server Tool is a software component that developers can add to their preferred development environment, turning any application into a stand-alone Web server.

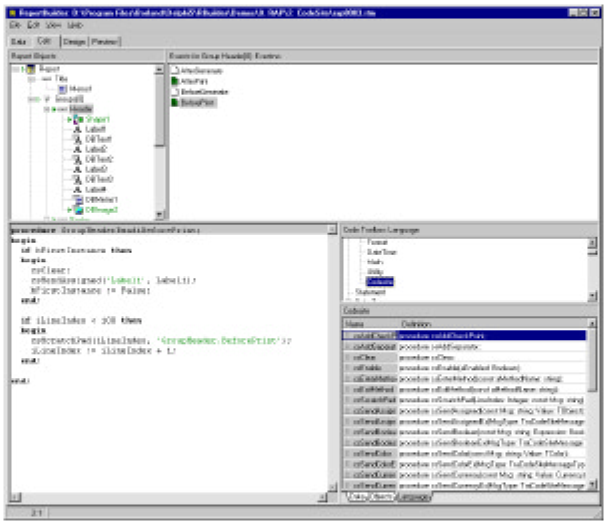
Using PowerTCP WebServer Tool, Web applications can be deployed as robust compiled applications, and the server address and port can be configured so multiple Web applications can be installed on a single NT server.

Dart Communications
Price: US$999
Phone: (315) 431-1024
Web Site: http://www.dart.com

Digital Metaphors Releases ReportBuilder 5

Digital Metaphors Corp. announced the release of ***Report-Builder 5***, the newest version of the company's reporting solution for Delphi. ReportBuilder Enterprise Edition includes the RAP (Report Application Pascal) programming language.

RAP enables developers to include Object Pascal code and Delphi-style event handlers within reports, making report layouts stand-alone entities that can simply be loaded and executed. It also allows end users to create calculations using a drag-and-drop interface.

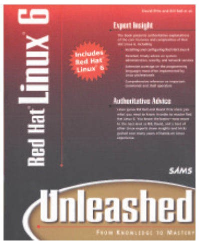

```
procedure GroupHeaderBeforePrint;
begin
  if bFirstInstance then
  begin
    [illegible]Clear;
    [illegible]Assigned('Label1', Label1);
    bFirstInstance := False;
  end;

  if iLineIndex < 100 then
  begin
    [illegible](iLineIndex, 'GroupHeader.BeforePrint');
    iLineIndex := iLineIndex + 1;
  end;

end;
```

RAP affords developers the ability to provide Delphi functionality wrapped for users as simple function calls that can be easily generated via the RAP Code Toolbox.

ReportBuilder 5 offers various enhancements, including RAP, Crosstab reports, and a new Label Template Wizard.

Digital Metaphors Corp.
Price: Standard, US$249; Professional, US$495; and Enterprise, US$749.
Phone: (972) 931-1941
Web Site: http://www.digital-metaphors.com

Red Hat Linux 6 Unleashed
David Pitts and Bill Ball, et al.
SAMS Publishing



ISBN: 0-672-31689-7
Price: US$39.99
(1,252 pages, CD-ROM)
Web Site: http://www.samspublishing.com

B&P Releases APrintDirect 3.6

B&P Technologies released *APrintDirect 3.6*, a Windows utility that allows you to manage and create a catalog listing of your files and folders.

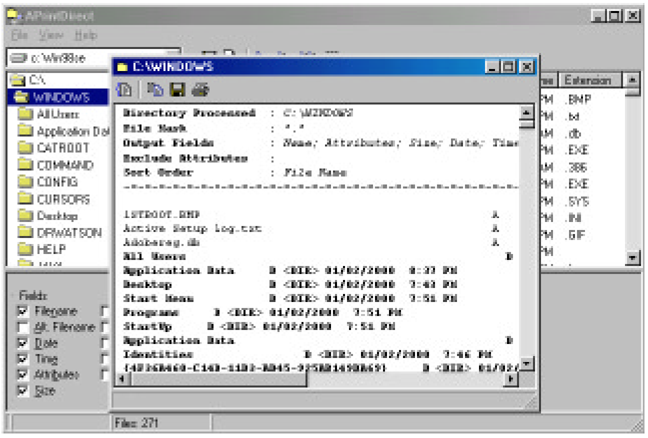
APrintDirect's interface allows for the selection of up to 11 property fields to include in each listing. This leaves you in control of selecting the information that you want included in your customized listing. You can also select by which field to sort the generated listing.

You can specify which types of files to include or exclude from the APrintDirect listing. You can also view only files and folders of particular attributes. For added precision, APrintDirect includes an output file mask. APrintDirect also features the ability to include the processing of nested folders.

APrintDirect's output options facilitate further management of your listing. The APrintDirect listing can be saved as a text file, printed to a compatible print, or displayed on screen. You may also copy text from the previewed listings to the Windows Clipboard.

Included in APrintDirect is the ability to choose from three separate output styles. The tree view depicts the structure as it originally appears. For a standard listing of files, there is the list view. The comma-separated style can be used to import your listing into many common spreadsheet and database applications for further formatting.

B&P Technologies
Price: US$14
Phone: (877) 353-7297
Web Site: http://www.bpsoftware.com


```
Directory Processed   : C:\WINDOWS
File Mask             : *.*
Output Fields         : Name; Attributes; Size; Date; Time
Exclude Attributes    :
Sort Order            : File Name
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

1STBOOT.BMP                                          A
Active Setup Log.txt                                 A
Adobereg.db                                          A
All Users                                             D
Application Data      D <DIR> 01/02/2000  8:33 PM
Desktop               D <DIR> 01/02/2000  7:43 PM
Start Menu            D <DIR> 01/02/2000  7:51 PM
Programs         D <DIR> 01/02/2000  7:51 PM
StartUp        D <DIR> 01/02/2000  7:51 PM
Application Data                                      D
Identities                  D <DIR> 01/02/2000  7:46 PM
{4E[illegible]-C140-11D3-8D45-9[illegible]}     D <DIR> 01/02/
```

Delphi
TOOLS

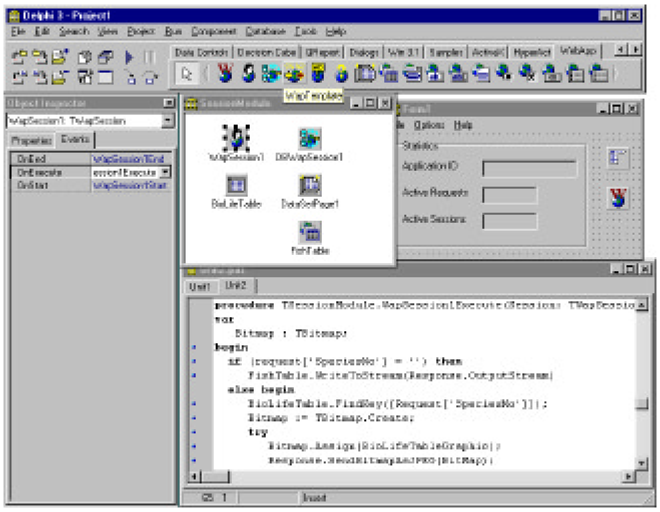
New Products and Solutions

HyperAct Announces WebApp 2.5

HyperAct, Inc. announced the availability of *WebApp 2.5*, the company's RAD framework for advanced Web server application development using Delphi. The new release offers support for Delphi 4 and 5.

New features include file upload, a Watchdog utility, remote monitor configuration, on-demand thread creation, support for non-cookies session recognition, PSWebDBGrid (a grid component), and stub application conversation through COM instead of shared memory.

WebApp supports ISAPI, NSAPI, WSAPI, CGI, and WinCGI servers. It provides routines and components that facilitate easy Web programming, such as HTML generation components, data-aware components, SMTP control for sending e-mail, on-the-fly conversion of bitmaps to GIF/JPEG, an ad-management component, and browser capabilities detection.


Delphi 3 - Project1
File Edit Search View Project Run Component Database Tools Help
Data Controls Decision Cube QReport Dialogs Win 3.1 Samples ActiveX HyperAct WebApp
Object Inspector
Properties Events
SessionModule
WapTemplate
WapSession1 DBWapSession1
BioLifeTable DataSetPage1
FishTable
Statistics
Application ID
Active Requests
Active Sessions
Unit1 Unit2
Insert

```
procedure TSessionModule.WapSession1Execute(Session: TWapSessio
var
  Bitmap : TBitmap;
begin
  if (request['SpeciesNo'] = '') then
    FishTable.WriteToStream(Response.OutputStream)
  else begin
    BiolifeTable.FindKey([Request['SpeciesNo']]);
    Bitmap := TBitmap.Create;
    try
      Bitmap.Assign(BioLifeTableGraphic);
      Response.SendBitmapAsJPEG(BitMap);
```

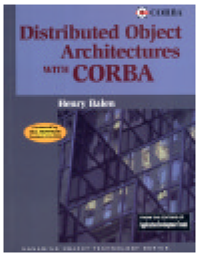
HyperAct, Inc.
Price: Standard (no source, includes one-server license), US$295; Professional (with source and two-server license), US$595; additional server license, US$195.
Phone: (402) 891-8827
Web Site: http://www.hyperact.com

Pitron Systems Announces PSWebDBGrid 1.0

Pitron Systems Ltd., in partnership with **HyperAct, Inc.**, announced the availability of *PSWebDBGrid 1.0*, a component that allows Delphi developers to create database grid applications. The new component supports Delphi 4 and 5 and its features include editable Grid and Record modes; rich layout control using events at Table, Row, and Column levels; server-side column Validation functions; extensibility using inheritance and events; built-in buttons for standard actions (Navigation, Delete, Insert, etc.); customizable buttons for developer actions; automatic scroll support based on sort order; and record tracking.

Pitron Systems Ltd./HyperAct, Inc.
Price: Standard, US$195; Professional, US$495 (includes source code).
Phone: (402) 891-8827
Web Site: http://www.pitron.co.il/main.html or http://www.hyperact.com

Distributed Object Architectures with CORBA
Henry Balen
Cambridge University Press/SIGS



ISBN: 0-521-65418-1
Price: US$39.95
(285 pages, CD-ROM)
Web Site: http://www.cup.org or http://www.sigs.com

M-Tech Releases Version 4.2 of P-Synch

M-Tech Mercury Information Technology, Inc. announced *P-Synch 4.2*, the latest version of the company's password synchronization program.

This latest version of the password management solution includes several new features, including support for managing passwords on LDAP servers, enhancing the P-Synch facility for changing and resetting passwords on directory servers; transparent password synchronization based on Netscape Directory Server password changes; native support for Novell GroupWise mail domains, adding an administrative interface and eliminating the need for scripting; native support for Lotus Domino server passwords; a simplified mechanism for Web-based password synchronization, allowing users to synchronize passwords by authenticating with one password, and then changing many passwords, with one simple GUI; and a Web-based module for enterprise-wide username management, which allows users to build a centralized database that shows their various login IDs on diverse systems.

M-Tech Mercury Information Technology, Inc.
Price: Call for pricing information.
Phone: (403) 233-0740
Web Site: http://www.psynch.com

Extended Systems Offers XTNDConnect RPM

Extended Systems, Inc. and **PenRight! Corp.** will launch *XTNDConnect RPM for MobileBuilder*, a programmable middleware for developing native Windows and wireless mobile applications. Through the new XTNDConnect RPM/MobileBuilder solution, developers can create and execute real-time, server-based processes over a wireless/wired LAN and Internet connection to Palm OS, Windows, and soon Windows CE handheld devices.

Features in XTNDConnect RPM include complete wireless API support for TCP/IP data networks, including CDPD, GSM, CDMA, and others; support for Palm modems, e.g. Minstrel, Omnisky, etc.; native real-time server access to back-end databases, including Advantage Database Server, Oracle, Sybase, DB2, and any ODBC- and OLE DB-compliant database; straightforward stored procedures concept to provide server-based data to mobile handhelds; and the ability to leverage existing Delphi code to implement business logic at the server level.

Extended Systems, Inc.
Price: Contact Extended Systems for pricing information.
Phone: (800) 235-7576 xt 5030
Web Site: http://www.extendedsystems.com

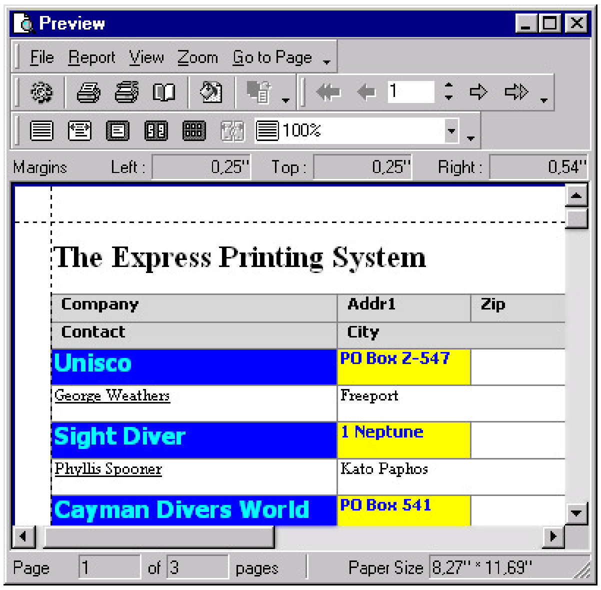
Developer Express Announces ExpressPrinting System

Developer Express Inc. announced the release of *ExpressPrinting System*, a VCL component library designed to bring the advanced presentation features of Developer Express products, such as ExpressQuantumGrid, QuantumTreeList, and other controls, to the printed page.

Via its Report Link technology, ExpressPrinting System allows developers to print and render the contents as well as the layout of ExpressQuantumGrid, ExpressQuantumTreeList, String Grid, Draw Grid, List Box, and Check List Box.

Each of the components listed above include many design- and run-time customization options to help developers and their users control the appearance of printed output. For instance, if a developer is rendering the ExpressQuantumGrid component, he/she can enable the printing of Footers, Bands, AutoPreview Pane, and more.

ExpressPrinting System includes an open architecture so that you can add extensions to print components from Borland and other third-party vendors.



Margins Left : 0,25" Top : 0,25" Right : 0,54"

The Express Printing System

Company	Addr1	Zip
Contact	City	
Unisco	PO Box Z-547	
George Weathers	Freeport	
Sight Diver	1 Neptune	
Phyllis Spooner	Kato Paphos	
Cayman Divers World	PO Box 541	

Page 1 of 3 pages Paper Size 8,27" * 11,69"

Developer Express Inc.
Price: US$179.99 (with full source code).
E-Mail: sales@devexpress.com
Web Site: http://www.devexpress.com

XML Software Releases InterAccess 1.1

XML Software Corp. released *InterAccess 1.1*, a product that provides full Internet database connectivity. Using InterAccess, you can access any ODBC/-OLE DB-compliant SQL database via the Internet.

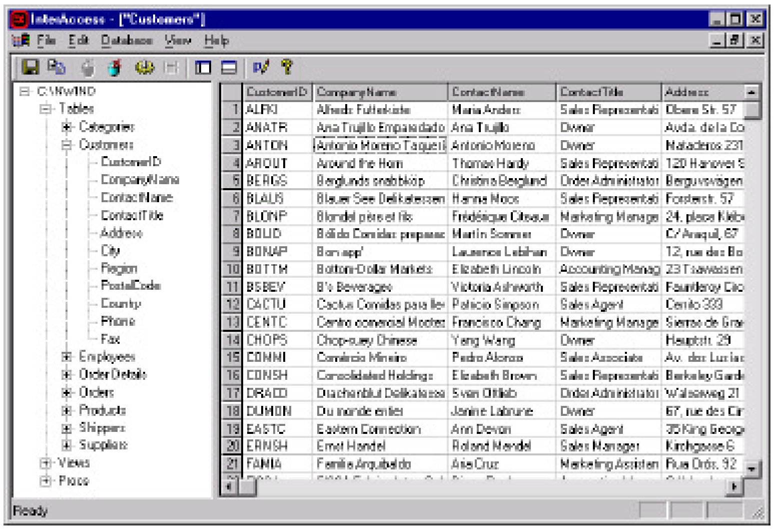
InterAccess implements a client/server architecture, much like FTP, and uses an XML protocol to provide complete access to SQL databases that have ODBC or OLE DB drivers. InterAccess consists of three components: InterAccess Server, the InterAccess client COM DLL, and the InterAccess Browser. With InterAccess Server and Browser, you can perform any SQL operations that your ODBC/OLE DB drivers provide. Use the InterAccess Browser if you only want to view/update tables and execute SQL commands. You can save any retrieved data to a file or copy/paste into other programs, such as Excel or Access.

If you need programming power for your in-house or commercial applications, use the InterAccess client COM DLL to write your own Internet-enabled database programs. The COM DLL provides COM objects and interfaces ready to drop into your Visual Basic, C++, or other RAD tool programs. The COM object model is similar to the Microsoft ADO object model.

You can optionally receive data as native ADO XML-persisted Recordsets, which enables construction of ADO disconnected Recordsets.

Using the InterAccess COM DLL, you can write in-house or commercial Internet database applications. There's no need to design and develop your own XML protocols.

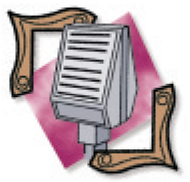
XML Software Corp.
Price: Visit Web site for licensing information.
E-Mail: info@xmlsoft.com.au
Web Site: http://www.xmlsoft.com.au



	CustomerID	CompanyName	ContactName	ContactTitle	Address
1	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representati	Obere Str. 57
2	ANATR	Ana Trujillo Emparedado	Ana Trujillo	Owner	Avda. de la Co
3	ANTON	Antonio Moreno Taquerí	Antonio Moreno	Owner	Mataderos 231
4	AROUT	Around the Horn	Thomas Hardy	Sales Representati	120 Hanover S
5	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen
6	BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representati	Forsterstr. 57
7	BLONP	Blondel père et fils	Frédérique Citeaux	Marketing Manage	24, place Kléb
8	BOLID	Bólido Comidas preparac	Martín Sommer	Owner	C/ Araquil, 67
9	BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bo
10	BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manag	23 Tsawassen
11	BSBEV	B's Beverages	Victoria Ashworth	Sales Representati	Fauntleroy Cir
12	CACTU	Cactus Comidas para lle	Patricio Simpson	Sales Agent	Cerrito 333
13	CENTC	Centro comercial Moctez	Francisco Chang	Marketing Manage	Sierras de Gra
14	CHOPS	Chop-suey Chinese	Yang Wang	Owner	Hauptstr. 29
15	COMMI	Comércio Mineiro	Pedro Afonso	Sales Associate	Av. dos Lusíad
16	CONSH	Consolidated Holdings	Elizabeth Brown	Sales Representati	Berkeley Gard
17	DRACD	Drachenblut Delikatesse	Sven Ottlieb	Order Administrator	Walserweg 21
18	DUMON	Du monde entier	Janine Labrune	Owner	67, rue des Cir
19	EASTC	Eastern Connection	Ann Devon	Sales Agent	35 King Georg
20	ERNSH	Ernst Handel	Roland Mendel	Sales Manager	Kirchgasse 6
21	FAMIA	Familia Arquibaldo	Aria Cruz	Marketing Assistan	Rua Orós, 92

News
LINE

August 2000



Troll Tech and Inprise/Borland Collaborate on Linux GUI

Scotts Valley, CA — Troll Tech and Inprise/Borland announced a technology licensing agreement covering Troll Tech's Qt graphical user interface (GUI) application framework. As a result of the agreement, Inprise/Borland can leverage Qt in its forthcoming Delphi and C/C++ rapid application development environment for Linux, code-named Kylix. Financial terms of the deal were not disclosed.

At the heart of the Kylix Project, Inprise/Borland is developing a visual component library (VCL) to speed and simplify native Linux application development. This agreement permits Inprise/Borland usage of Qt to implement the underlying GUI layer of the Borland Linux VCL for Kylix. As a result of this agreement, Kylix will include a license to develop GUI applications with the Qt-based VCL.

Inprise/Borland recently made a minority investment in Troll Tech.

Qt is a cross-platform C++ application framework that enables rapid building of GUI applications. Qt is fully object-oriented, extensible, and supports true component programming. It is supported on a range of platforms, including Windows 95/98/NT/2000, Linux, Solaris, HP-UX, IRIX, and AIX. For more information, contact Troll Tech by e-mail at info@trolltech.com, or visit the company's Web site at http://www.trolltech.com.

Bank of America Selects Inprise Application Server for E-business Platform

Scotts Valley, CA — Inprise/-Borland announced that Bank of America's Global Corporate Investment Bank (GCIB) has chosen the Inprise Application Server, which supports the J2EE standard and combines the benefits of EJB and CORBA, as a key element in its e-commerce strategy. Inprise Application Server was chosen over seven other competing application servers after a three-month trial period. Inprise Application Server will enable GCIB to expand its presence on the Web and provides the company with the technology needed to support these and other enterprise-strength Internet business applications.

Inprise/Borland Releases New MIDAS XML Server

Scotts Valley, CA — Inprise/-Borland announced the availability of Borland MIDAS 3, middleware technology and components for rapidly building Delphi and C++Builder Internet applications.

Now supporting XML and Dynamic HTML, MIDAS 3 increases the scalability and flexibility of large-scale Internet applications.

Inprise/Borland also announced modified MIDAS licensing that will now allow small- to medium-sized businesses to gain the same advantages as large corporations as they extend their businesses to the Internet.

With MIDAS 3, developers can create and deploy applications that connect users to critical business information whenever and wherever they need it. From e-commerce implementations to customer relationship management systems, users will benefit from greater control over network traffic, improved performance for mobile environments, and increased scalability.

For more information on MIDAS 3, please visit http://www.inprise.com.

Inprise/Borland Announces First Quarter 2000 Results

Scotts Valley, CA — Inprise/-Borland Corp. recently announced financial results for the first quarter of fiscal year 2000, which ended March 31, 2000.

For the first quarter, revenues were US$46.5 million, up from US$43.4 million in the same period a year ago.

The company recorded a net loss of (US$1.1) million in the first quarter, or (US$0.02) loss per share, compared to a net loss of (US$25.6) million, or (US$0.54) loss per share in the first quarter of 1999.

On an operating basis, the company recorded a net loss from operations of (US$2.3) million in the first quarter of fiscal 2000, which compares with a net loss from operations of (US$26.8) million in the same period last year. The prior year's loss included a US$15.2 million one-time charge for severance, restructuring, and other compensation-related expenses.

Cash, cash equivalents, and short-term investments as of March 31, 2000 were US$239.7 million, up from US$197.7 million on December 31, 1999. The increase in cash was due principally to the sale of the Scotts Valley campus in March of this year.

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

Inprise/Borland and Corel Terminate Proposed Merger

Scotts Valley, CA — Inprise/-Borland Corp. announced its merger agreement with Corel Corp. has been terminated by mutual agreement of the two companies without payment of any termination fees.

In addition, the reciprocal stock option agreements have also been terminated.

Dale Fuller, Inprise/Borland interim president and CEO said, "Much has changed since the merger was agreed to more than three months ago, and our board concluded that it would be best to cancel the merger on an amicable basis."

In January of 2000, Inprise/-Borland and Corel entered into a confidentiality agreement that included a standard three-year standstill covenant. That agreement remains in effect.

table of CONTENTS

By Bill Todd

The Gold Standard, MIDAS & COM

Part I: Building Modular Applications

Borland developed MIDAS (Multi-tier Distributed Applications Services Suite) for creating multi-tier distributed applications. MIDAS is also the best way to build any database application, particularly large applications, even when you don't need a distributed application. Combining MIDAS with Microsoft's Component Object Model (COM) lets you build large, complex applications from multiple COM servers that share a common database connection.

There are a number of benefits to using MIDAS and COM together. Here are some of them:

1) Team development is easier to manage, because each team member can work on a module that can be compiled and tested independently.
2) Applications that consist of many modules, such as an accounting system, are easier to deploy, because you can deploy only those modules the user needs.
3) All modules share a common database connection.
4) Modules are easily sharable across applications, regardless of the programming language used to create them.
5) Supporting multiple databases is easier. Even if you don't need to support multiple databases now, you can design your application so you can change databases more easily in the future.

The first part of this two-article series covers building a simple application that demonstrates using MIDAS and COM together. It also shows one way to implement callbacks from a COM server to its client. The second article will demonstrate two other techniques for server callbacks, and discuss deployment issues for this type of application.

To examine using MIDAS and COM to build a modular application, I will create a very simple example that consists of a MIDAS server and two MIDAS clients. The first MIDAS client will be the application's main form, and will display data from the sample Customer and Order tables. This application is an EXE. The second MIDAS client will display data from the Order table, and is implemented as an in-process Automation server.

The roles played by the three programs can be confusing. To clarify which does what, the table in Figure 1 shows each application, the roles they play, and how they're implemented (the MIDAS server and two clients are available for download; see end of article for details).

Building the MIDAS Server

The MIDAS server has only one unusual feature. It's implemented as a DLL, so it won't display a form, or show an icon on the task bar. Displaying the server on the task bar is acceptable for a distributed system where no one normally sees the screen of the machine that hosts the MIDAS server. However, it's not a good idea for an application where the server and client will run on the same PC, because the user may be confused by the extra icon and may try to close the server. The solution is to implement the MIDAS server as a DLL so it has no user interface.

To create a MIDAS server as a DLL, select **File | New** from the menu and choose the ActiveX page of the Object Repository. Double-click the ActiveX Library icon to create a new ActiveX library project. Because

Project Name	Purpose	Roles	Implemented As
DemoDllServer	Provides a connection to the database.	MIDAS Server	ActiveX Library DLL
DemoClient	Contains the Customer form.	MIDAS Client COM Client	EXE
DemoOrders	Contains the Order form.	MIDAS Client COM Server	ActiveX Library DLL

Figure 1: The demonstration MIDAS server and its two MIDAS clients.

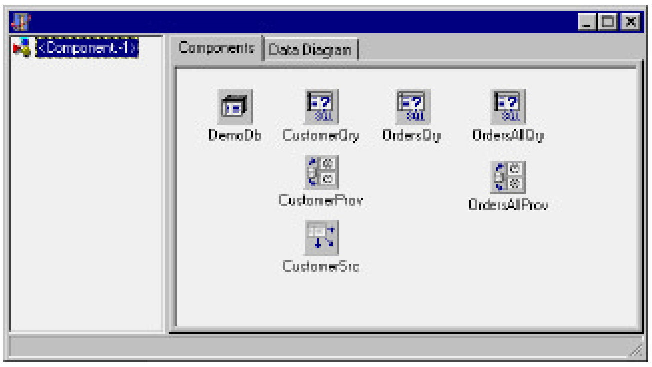
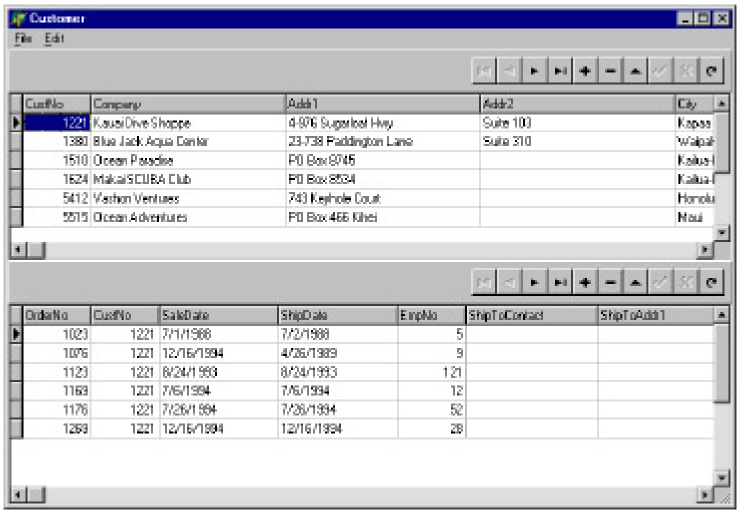


Figure 2: The remote data module.



CustNo	Company	Addr1	Addr2	City
1221	Kauai Dive Shoppe	4-976 Sugarloaf Hwy	Suite 103	Kapaa
1380	Blue Jack Aqua Center	23-738 Paddington Lane	Suite 310	Waipah
1510	Ocean Paradise	PO Box 8745		Kailua-I
1624	Makai SCUBA Club	PO Box 8534		Kailua-I
5412	Vashon Ventures	743 Keyhole Court		Honolu
5515	Ocean Adventures	PO Box 466 Kihei		Maui

OrderNo	CustNo	SaleDate	ShipDate	EmpNo	ShipToContact	ShipToAddr1
1023	1221	7/1/1988	7/2/1988	5		
1076	1221	12/16/1994	4/26/1989	9		
1123	1221	8/24/1993	8/24/1993	121		
1169	1221	7/6/1994	7/6/1994	12		
1176	1221	7/26/1994	7/26/1994	52		
1269	1221	12/16/1994	12/16/1994	28		

Figure 3: The Customer form.

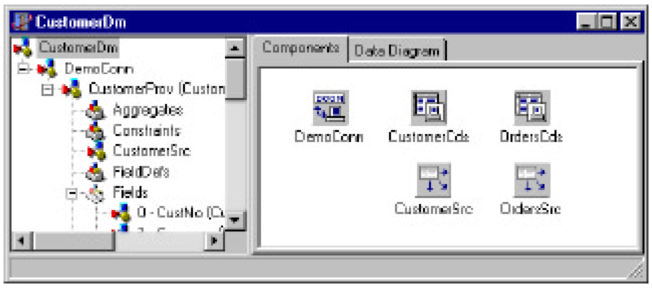


Figure 4: The Customer form's data module.

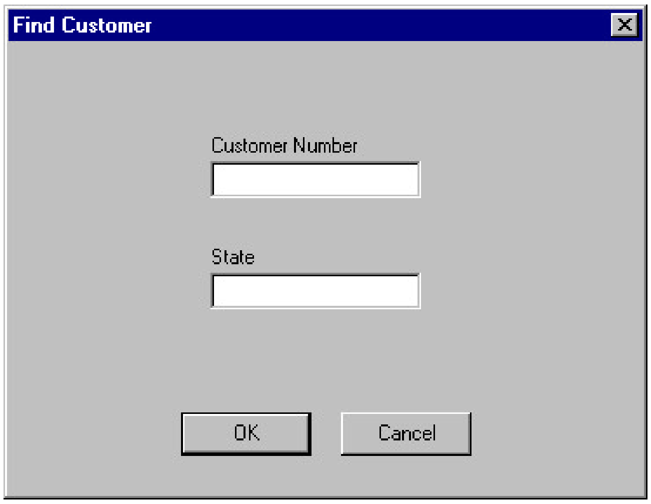


Figure 5: Selecting customer records in the Find Customer dialog box.

MIDAS uses COM to handle communications between the MIDAS client and the MIDAS server, an ActiveX library is used to provide the required COM support.

From this point on, the process is the same as creating a MIDAS server that is an EXE. Select **File | New**, go to the Multitier page, and add a Remote Data Module to the project. Figure 2 shows the remote data module for the sample application.

This application is written in typical client/server style. When the user opens the application, no data is displayed. Instead, the user must enter some selection criteria that will fetch a reasonable number of records. To implement this approach, the SQL statement for the *CustomerQry* component is:

```
SELECT * FROM Customer
WHERE CustNo = -1
```

Because there's no customer record whose customer number is minus one, this allows the Customer ClientDataSet in the DemoClient application to be opened immediately, without displaying any data. A DataSetProvider (*CustomerProv*) and a DataSource (*CustomerSrc*) are connected to the *CustomerQry* component by setting their *DataSet* property to *CustomerQry*. In the *Options* property of the DataSetProvider, *poAllowCommandText* is set to True, so the client application can change the *SQL* property of *CustomerQry* to select different sets of customer records.

OrdersQry supplies the order records for the current customer record. Its *SQL* property is set to:

```
SELECT * FROM Orders
WHERE (CustNo = :CustNo)
```

and its *DataSource* property is set to *CustomerSrc* so the *:CustNo* parameter's value will be supplied by the current record in *CustomerQry*. This will cause the order records to be stored in the customer records as a nested dataset.

The DemoOrders application allows the user to search the entire Order table, and select an order by order number, or all of the orders for a customer number. To provide access to all orders, a second Query component (not linked to the *CustomerQry*), *OrdersAllQry*, is needed. Again, the SQL statement is set to retrieve no records by selecting all columns from Order where the order number is minus one. The DataSetProvider for the *OrdersAllQry* also has its *poAllowCommandText* option set to True.

Because this MIDAS server is a DLL, you can't register it by running it. Instead, choose **Run | Register ActiveX Server** from the Delphi menu to compile and then register the MIDAS server.

In a typical three-tier distributed application, the MIDAS server not only provides the connection to the database, it also provides business rule enforcement and other services to its clients. However, in this article we're discussing a single application that consists of multiple modules. All of the modules will be MIDAS clients using the same MIDAS server, and both the clients and the server will run on the same machine. Suppose you're writing a vertical market application using this architecture. If you need to support multiple databases, you may want to limit the code in the MIDAS server to just that code that is specific to a particular database, such as Oracle or Microsoft

SQL Server, and keep all of the code that is common to all databases in the client modules. This lets you maintain multiple MIDAS servers for multiple databases with no code replication.

Building the COM Client

Figure 3 shows the application's main form. It consists of two DBGrids and two DBNavigators. The top grid and navigator display customer information, and the bottom grid and navigator display order data. Figure 4 shows the data module for this application.

The data module contains a DCOMConnection component, two ClientDataSets, and two DataSources. The DCOMConnection component's name is *DemoConn*, and its *ServerName* property is set to *DemoDllSrvr.DllDemoServer*. The *RemoteServer* property of *CustomerCds* is set to *DemoConn*, and its *ProviderName* property is set to *CustomerProv*. The *OrdersCds* component's *DataSetField* property is set to *CustomerCdsOrdersQry* so it will derive its data from the nested dataset in the *CustomerCds* records.

The **Edit** menu contains a **Find** item that displays the dialog box shown in Figure 5. This lets the user select a customer record by customer number. It also allows users to select all of the records in a specified state using the *FindCustomer* method in the data module named *CustomerDm*. If you're interested in this code, you may download the complete sample application.

On the main form, the **File | Orders** menu item lets the user open a form that can be used to search for any order by customer number or order number. The Orders grid is connected to a pop-up menu component that offers the user two choices:

1) **Show This Order** will open the Order form and show the current order record.
2) **Show All Orders For This Customer** will open the Order form, and display all orders for the customer number contained in the current order record in the grid.

Building the COM Server

Now the fun begins. The next step is to create the Order form, as well as the methods the Customer form must use to open the Order form; find the orders for a customer; and find a specific order by its order number. However, the Order form is going to be in a separate application, which is an Automation server, and the Customer form will call the Order form's methods through its interface using Automation.

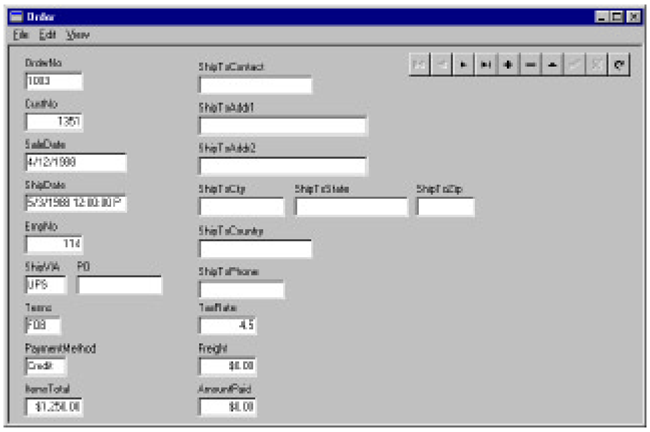
To create the Order application, go to the ActiveX page in the Object Repository and double-click ActiveX Library. Add a form and a data module to the application. The finished form is shown in Figure 6 and the data module is shown in Figure 7.

The DCOMConnection component in Figure 7, *OrdersConn*, connects to the MIDAS server, *DemoDllSrvr.DllDemoServer*, just as the DCOMConnection component in the Customer data module did. The *RemoteServer* property of *OrdersCds* is set to *OrdersConn*, and the *ProviderName* is set to *OrdersAllProv*.

The next step is to turn this DLL into an Automation server. Return to the ActiveX Page of the Object Repository, double-click the Automation Object wizard, and enter `OrdersServer` for the **CoClass Name**. Also, check the **Generate Event support code** checkbox. When the Type Library editor appears, add the methods shown in Figure 8 to the *IOrderServer* interface, then click the **Refresh** button. If you want to see captions under the Type Library editor toolbar buttons, right-click the toolbar.

The code for the *FindByOrderNo*, *FindByCustNo*, and *OpenOrdersForm* methods is straightforward (see Figure 9), and found in the OrdersAuto unit.

The first two methods, *FindByOrderNo* and *FindByCustNo*, call the methods with the same name in the Order form's data module. The **implementation** section of the Order form's data module is shown



OrderNo	1003
CustNo	1351
SaleDate	4/12/1988
ShipDate	5/3/1988 12:00:00 P
EmpNo	114
ShipVIA	UPS
PO	
Terms	FOB
PaymentMethod	Credit
ItemsTotal	$1,250.00
ShipToContact	
ShipToAddr1	
ShipToAddr2	
ShipToCity	
ShipToState	
ShipToZip	
ShipToCountry	
ShipToPhone	
TaxRate	4.5
Freight	$0.00
AmountPaid	$0.00

Figure 6: The Order form.

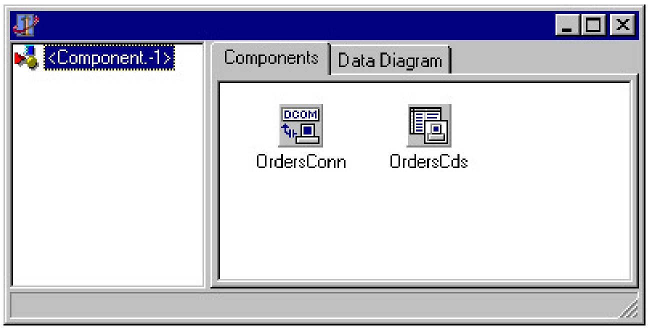


Figure 7: The Order form's data module.

Method	Param	Type
FindByOrderNo	*OrderNo*	Long
FindByCustNo	*CustNo*	Long
OpenOrdersForm		
CloseOrders		
FindCustomer		
GetCustNo	*CustNo*	Variant

Figure 8: Add these corresponding methods to the *IOrderServer* interface.

```
procedure TOrderServer.FindByOrderNo(OrderNo: Integer);
begin
  OrderDm.FindByOrderNo(OrderNo);
end;

procedure TOrderServer.FindByCustNo(CustNo: Integer);
begin
  OrderDm.FindByCustNo(CustNo);
end;

procedure TOrderServer.OpenOrdersForm;
begin
  OrderDm := TOrderDm.Create(nil);
  OrderForm := TOrderForm.Create(nil);
  OrderForm.Show;
end;
```

Figure 9: The *FindByOrderNo*, *FindByCustNo*, and *OpenOrdersForm* methods.

in Figure 10. Both methods close the order's ClientDataSet, assign a new SQL statement to its *CommandText* property, and then re-open the ClientDataSet. When the ClientDataSet is opened, the value of *CommandText* is passed to the MIDAS server and assigned to the *SQL* property of the *OrdersAllQry* component before the query is opened. The Customer EXE program calls these methods to display a particular order or orders for a specific customer in the Order form. The third method, *OpenOrdersForm*, creates the data module (*OrderDm*) and the OrdersForm, and shows the Order form. The Customer EXE program calls this method to make the Order form visible.

The *FindOrder* method of the Order form's data module is called from the Edit menu of the Order form. It displays the FindOrdersForm dialog box, which lets the user find one or more orders by order number or customer number.

Calling Back to the COM Client

With the methods described so far, the COM client application that displays the Customer form can call methods in the COM server to open the Order form, and find orders by order number or customer number. However, the COM server needs to be able to call back to

```
implementation

uses FindOrderF;

{$R *.DFM}

{ Displays the Find Order dialog. Calls appropriate find
  method based on which edit box on the Find Order dialog
  has a value. }
procedure TOrderDm.FindOrder;
begin
  FindOrderForm := TFindOrderForm.Create(Self);
  try
    with FindOrderForm do begin
      ShowModal;
      if OrderNoEdit.Text <> '' then
        FindByOrderNo(StrToInt(OrderNoEdit.Text))
      else if CustNoEdit.Text <> '' then
        FindByCustNo(StrToInt(CustNoEdit.Text))
      else
        MessageDlg('You must enter an order number or ' +
          'customer number.', mtError, [mbOK], 0);
    end;
  finally
    FindOrderForm.Free;
  end;
end;

{ Finds an Order record given its OrderNo. }
procedure TOrderDm.FindByOrderNo(OrderNo: Integer);
begin
  with OrdersCds do begin
    Close;
    CommandText := 'SELECT * FROM Orders WHERE ' +
      '(OrderNo = ' + IntToStr(OrderNo) + ')';
    Open;
  end;
end;

{ Finds all Order records for the specified Customer. }
procedure TOrderDm.FindByCustNo(CustNo: Integer);
begin
  with OrdersCds do begin
    Close;
    CommandText := 'SELECT * FROM Orders WHERE ' +
      '(CustNo = ' + IntToStr(CustNo) + ')';
    Open;
  end;
end;
```

Figure 10: The *OrdersDm* methods.

the client for two reasons. First, when a user is viewing an order, the user needs to be able to display the customer record for that order. Put another way, the Order form must be able to tell the Customer form to find a specific customer record and show itself. The second problem is that the COM server application shows the Order form modelessly. That means that the COM client has no way of knowing when it can close the COM server. The only solution is that the COM server must notify the COM client when the user closes the Order form.

There are three ways for the server to communicate with the client. The first is to add an Automation object to the client application, so the server can connect to the client and call methods of the Automation object's interface. Doing this means that the application that contains the Customer form is both a COM client of, and a COM server to, the Order application DLL. Further, the Order's DLL is both a client of, and server to, the Customer application.

The second method involves creating a callback interface to the COM client application. To do this, you must add an interface to the client, and create an object that implements that interface. When the COM client connects to the COM server, it must create an instance of the callback object, then call a method of the COM server and pass the interface reference, as a parameter, to the COM server. Using this interface reference, the server can call methods on the client.

The third technique is to let the server fire events on the client through the server's dispinterface. This is the easiest to implement in Delphi 5, thanks to wizards that do most of the work. Although this technique has some limitations, it will suffice for many applications — so we'll examine it first.

The key to using callback events is to check the Generate Event support code checkbox when adding the Automation object to the COM server. This causes two interfaces to be added to the COM server's type library. We've already added methods to the first interface, *IOrderServer*. The second interface is a dispatch interface named *IOrderServerEvents*. It's now time to open the Type Library editor again, and add two methods to the *IOrderServerEvents* interface. The first is named *OnCloseOrders*, and the second is named *OnFindCustomer*. After adding the *OnFindCustomer* event, click the Parameters tab, then click the Add button to add a new parameter. Name the parameter *CustNo*, and leave its type set to Long.

The *OnCloseOrders* event will be fired when the user closes the Order form to notify the COM client that it can close its connection to the COM server. The *OnFindCustomer* event will fire when the user selects View | Customer from the menu. This event will notify the COM client that it should find and display the customer record whose customer number matches the customer number of the current order record.

The code in Figure 11 fires the events. *CloseOrders* and *FindCustomer* are methods that were added to the *IOrderServer* interface earlier. *CloseOrders* is called from the *OnDestroy* event handler of the Order form. *FindCustomer* is called from the *OnClick* event handler of the View | Customer menu item.

To call these methods, you must have a reference to the *OrderServer* Automation object. To get this reference, two changes are made to the *OrdersAuto* unit. First, a global variable, *OrderServer*, is added to the **interface** section of the unit:

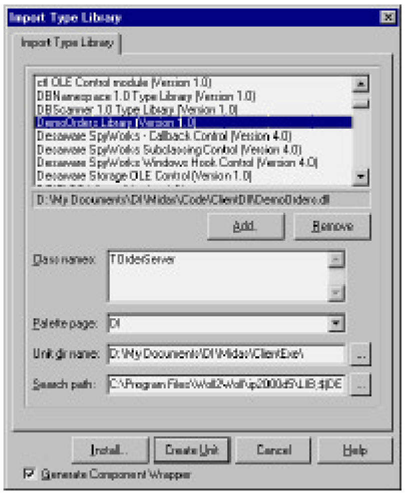
```
var
  OrderServer: TOrderServer;
```

```
procedure TOrderServer.CloseOrders;
begin
  FEvents.OnCloseOrders;
end;

procedure TOrderServer.FindCustomer;
begin
  FEvents.OnFindCustomer(
    OrderDm.OrdersCdsCustNo.AsInteger);
end;
```

Figure 11: Firing the dispinterface events.

```
procedure TOrderServer.Initialize;
begin
  inherited Initialize;
  FConnectionPoints := TConnectionPoints.Create(Self);
  if AutoFactory.EventTypeInfo <> nil then
    FConnectionPoint := FConnectionPoints.
      CreateConnectionPoint(AutoFactory.EventIID,
      ckSingle, EventConnect)
  else
    FConnectionPoint := nil;
  OrderServer := Self;
end;
```

Figure 12: The *OrderServer* reference variable is initialized.



ctl OLE Control module (Version 1.0)
DBNamespace 1.0 Type Library (Version 1.0)
DBScanner 1.0 Type Library (Version 1.0)
DemoOrders Library (Version 1.0)
Desaware SpyWorks - Callback Control (Version 4.0)
Desaware SpyWorks Subclassing Control (Version 4.0)
Desaware SpyWorks Windows Hook Control (Version 4.0)
Desaware Storage OLE Control (Version 1.0)

D:\My Documents\DI\Midas\Code\ClientDll\DemoOrders.dll

Class names: TOrderServer

Palette page: DI

Unit dir name: D:\My Documents\DI\Midas\ClientExe\

Search path: C:\Program Files\Wall2\Wall\p2000d5\LIB;$(DE

☑ Generate Component Wrapper

Figure 13: The Import Type Library dialog box.

Next, a line is added to the *TOrderServer* object's *Initialize* method to assign *Self* to the *OrderServer* global variable (see Figure 12). The *OrderServer* variable now provides a reference to the *OrderServer* Automation object, which can be used to call its methods from the Order form's *OnDestroy* event handler, the menu item's *OnClick* event handler, or from anywhere else in the DemoOrders application. Note that if you just want to fire an event from a method in the *IOrderServer* interface, you can omit these two steps. We needed a reference to the Automation object only because we needed to fire the events from elsewhere in the application.

The last step is to implement the events in the COM client. With the DemoClient project open in the IDE, select **Project | Import Type Library** from the menu to display the Import Type Library dialog box (see Figure 13). Select DemoOrders Library in the list box and make sure that **Generate Component Wrapper** is checked. This will create a component of type *TOrderServer*, and add it to your Component palette.

When you click the **Install** button, you'll be asked if you want to install this component in a new package or an existing package. You'll probably find it more convenient to put all of the server components for the project you're working on in their own packages. Whatever you do, don't install this component in one of the existing Delphi component packages. Once you've selected a package, click **OK**, then **Yes** to the dialog box informing you that the package will be built and installed. The component that is created is a wrapper around the COM server, and can be used to connect to the server and call its methods. The *OrderServer* component also has an event for each event you added to the *IOrderServerEvents* interface in the COM server.

```
procedure TCustomerForm.OrderServerCloseOrders(
  Sender: TObject);
begin
  OrderServer.Disconnect;
end;

procedure TCustomerForm.OrderServerFindCustomer(
  Sender: TObject; CustNo: Integer);
begin
  CustomerDm.FindByCustNo(CustNo);
  Show;
end;
```

Figure 14: The *OnCloseOrders* and *OnFindCustomer* event handlers.

```
procedure TCustomerForm.Orders1Click(Sender: TObject);
begin
  OrderServer.Connect;
  OrderServer.OpenOrdersForm;
end;

procedure TCustomerForm.ShowThisOrder1Click(
  Sender: TObject);
begin
  with OrderServer do begin
    Connect;
    OpenOrdersForm;
    FindByOrderNo(CustomerDm.OrdersCds.FieldByName(
      'OrderNo').AsInteger);
  end;
end;

procedure TCustomerForm.ShowAllOrdersForThisCustomer1Click(
  Sender: TObject);
begin
  with OrderServer do begin
    Connect;
    OpenOrdersForm;
    FindByCustNo(CustomerDm.OrdersCds.FieldByName(
      'CustNo').AsInteger);
  end;
end;
```

Figure 15: The menu item event handlers.

Drop an instance of the *TOrderServer* component on the Customer form, and name it *OrderServer*. Set its *AutoConnect* property to False, so the connection to the COM server won't be opened automatically when the program starts. Switch to the Events page of the Object Inspector and create event handlers for the *OnCloseOrders* and *OnFindCustomer* events. The code for both event handlers is shown in Figure 14.

All that remains is to implement the *OnClick* event handlers for the **File | Orders** menu choice, and the Order grid's pop-up menu. The code for these event handlers is shown in Figure 15.

Conclusion

With Microsoft Office and the Windows user interface, Microsoft has clearly demonstrated that large applications can be built from shared COM servers. You can use the same technique in your applications to make team development, maintenance, updates, and distribution easier. MIDAS is the ideal data access technology for this application architecture, because it makes sharing a single database connection among multiple EXEs and DLLs easy.

Next month, we'll discuss two other techniques for server callbacks, as well as deployment issues that may arise for these particular types of applications. ∆

The files accompanying this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\AUG\DI200008BT.

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*. He is a Contributing Editor to *Delphi Informant Magazine*, and a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a member of Team Borland, a nationally known trainer, and has taught Delphi programming classes across the country and overseas. He can be reached at bill@dbginc.com.

OP Basics

Form Inheritance / OOP / RAD / Delphi 2-5

By Ron Nibbelink

A Hierarchy of Forms

Save Development Time with Form Inheritance

There I sat — in a hurry and already tired. I didn't want to build those four screens for listing four kinds of support data for this application. These screens would supply the pull-down lists for data-entry fields on various other tables. I wasn't looking forward to the work involved in building edit screens to go with each list either. Then I remembered form inheritance.

Form inheritance allows programmers to build the framework of a form that will be used repeatedly. It's an extension of Delphi's *TForm*, and it significantly speeds up the routine work of programming a complete application.

The hierarchy of forms I use has evolved over time, but the basic set has served me quite well for two years. These forms have saved hundreds of hours of development and debugging time, and have given my applications a consistent look and feel. They provide sharper-looking products, and reduce the time it takes my users to learn my applications.

The Ancestor

The hierarchy starts with the *TForm* object supplied with the VCL. I added a couple of panels to it to make *TfrmGenrForm* (see Figure 1). I placed the control buttons for the descendant forms along the right side, so one panel, named *pnlControls*, is right-aligned on the base form. I gave it a width that accommodates the buttons I use on the descendant forms. (The samples available for download use a slightly modified *TBitBtn*, named *TGenrBtn*, which is 92 pixels wide, so I set *pnlControl*'s width to 112; see end of article for download details.) The other panel is named *pnlForm*, and is aligned to the remainder of the form's client area.

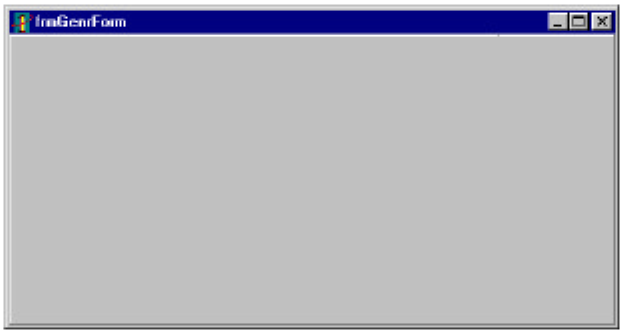


Figure 1: The first-generation form, *TfrmGenrForm*.

Of course you can arrange the panels any way you like, and you can even add company logos or other special identifiers. By placing them in an ancestor object, they can remain the same throughout the application. The form object also has certain properties set, which relieves me of having to set them consistently each time I start a new form.

I also added the stand-alone procedure *RunGenrForm*. It takes a *TfrmGenrForm* as its sole argument, so it can run any of its descendants as well. It runs the form by calling its *ShowModal* method, and then frees it from memory:

```
procedure RunGenrForm(frm: TfrmGenrForm);
begin
  with frm do begin
    ShowModal;
    Free
  end
end;
```

If the descendant form needs no specific input from the calling routine, it can be called using this procedure. The code for making such a call is simple:

```
procedure TfrmMain.actShowListExecute(Sender:
          TObject);
begin
RunGenrForm(TfrmShowList.Create(Application))
end;
```

where *actShowListExecute* is a *TAction* method within the calling form, and *TfrmShowList* is the descendant form being created.

Before adding a descendant of any of these ancestor forms to a project, it's a good idea to add its ancestor(s). Also, because they're never instantiated, you have to be sure Delphi doesn't set them up to be auto-created. In Delphi, select Project | Options, and move the form to the Available Forms list box.

The Generic List Display Screen

I like to build lists to display and maintain the support data that's used to supply the pull-down pick lists for data-entry fields elsewhere in the application. For many such lists, the user interface requirements are essentially the same: Users need ways to add, edit, and delete records, as well as ways to exit the screen. By building that common functionality into an ancestor form, the programmer can concentrate on the unique parts of the application.

I created a form, *TfrmGenrList*, that includes a DBGrid component for displaying data from a table (see Figure 2). When creating a descendant form, add a **uses** statement that includes your data module, specify the grid's DataSource, and set up the columns. Finally, add the specifics of adding, editing, and deleting records by overriding the *DoAdd*, *DoEdit*, and *DoDrop* methods. The remainder of the normal processing is done at the ancestor level.

TfrmGenrList has a *TActionList* component with actions for adding a new record, and editing and deleting the current record. The form provides several ways to access these actions. Most obvious are the pop-up menu and the series of buttons on the *pnlControls* panel. The grid has an *OnDblClick* event handler that also invokes the edit action. The *OnKeyDown* event handler invokes the add action when users press Insert, the edit action when users press Enter↵, and the delete action when users press Delete:

```
procedure TfrmGenrList.grdDBKeyDown(Sender: TObject;
  var Key: Word; Shift: TShiftState);
begin
  case Key of
    VK_RETURN : actEditExecute(Sender);
    VK_INSERT : actAddExecute(Sender);
    VK_DELETE : actDropExecute(Sender);
  else
    Exit
  end;
  Key := 0
end;
```

The form also contains a method for enabling or disabling various actions. For example, if there are no records in the underlying database table, the *SetButtons* procedure disables the edit and drop actions. If the programmer has set the *IsReadOnly* property to True, the add and drop actions are made invisible, and the edit action's caption is changed to "View" (see Figure 3).

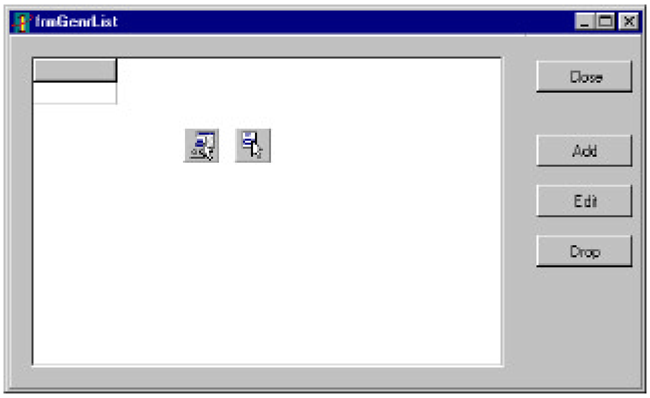


Figure 2: The *TfrmGenrList* form. The *pnlControls* buttons and the pop-up menu items reference the *TActionList* items.

Unless the table or query is already open, the form's *FormActivate* routine should activate it, and the *FormClose* method should close it. Inheriting from *TfrmGenrList* requires one additional task: implementing three methods for overriding the virtual abstract ones. The descendant form's *DoAdd*, *DoEdit*, and *DoDrop* routines actually insert, edit, and delete the records.

The Generic Edit Screen

Most of my edit screens use many of the same features as my list screens. The user may be modifying simple look-up data, complex database records, system options from an .ini file or the registry, or other information. In any case, I need an OK button to save the changes, and a Cancel button to discard them. It's also a helpful user interface feature to provide a visual indicator to show whether the data has been changed (see Figure 4).

Because there are several ways to exit the modal dialog box, I consolidated the save/cancel confirmation questions in the *FormCloseQuery* procedure (not shown). The OK button's *ModalResult* property sets the form's *ModalResult* to *mrOK*. The Cancel button and the other exit paths set it to *mrCancel*. Therefore, if *ModalResult* is *mrOK*, I give

```
procedure TfrmGenrList.SetButtons;
var
  HasRecs : Boolean;
begin
  actAdd.Enabled := not IsReadOnly;
  actAdd.Visible := actAdd.Enabled;
  HasRecs := False;
  with grdDB do
    if Assigned (DataSource) then
      with DataSource do
        if Assigned (DataSet) then
          with DataSet do
            HasRecs := not BOF or not EOF;

  actEdit.Enabled := HasRecs;
  if IsReadOnly then
    actEdit.Caption := 'View'
  else
    actEdit.Caption := 'Edit';

  actDrop.Enabled := HasRecs and not IsReadOnly;
  actDrop.Visible := actAdd.Visible
end;
```

Figure 3: Code for enabling or disabling various actions.

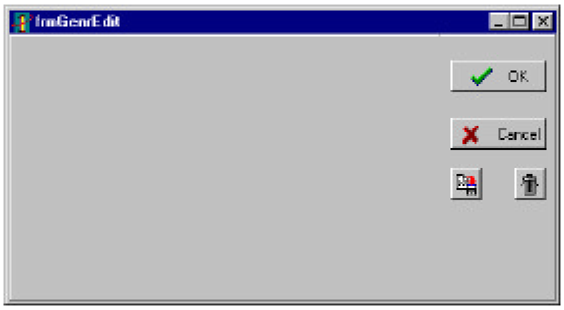


Figure 4: The *TfrmGenrEdit* form. The *TSpeedButtons* in the *pnlControls* panel serve as a visual indicator of whether the user has changed the data.

the user the option of saving or canceling. If *ModalResult* is *mrCancel*, the user is asked whether to discard the changes. If the user cancels the confirmation dialog box, I just set *FormCloseQuery*'s *CanClose* parameter to False so the form stays open.

There are times on the edit form when I don't want the user to have to confirm whether to save or cancel changes. For example, my date picker dialog box is based on *TfrmGenrEdit*. However, date picking usually occurs as part of another function, and it would be confusing to ask the user to save the date change. Therefore, *TfrmGenrEdit* has a property named *DoConfirm*. The default setting is True; the date picker sets it to False.

To use *TfrmGenrEdit*, create a new form that inherits from it, and add data-entry controls as you would normally. For each such control, set its *OnChange* or *OnClick* event handler to *DataChanging*. If you have other processing to perform in one of these event handlers, just add the call to *DataChanging* to your routine. This method enables *TfrmGenrEdit*'s save and discard speed buttons to show that something has changed, which, in turn, tells the form to confirm the exit process.

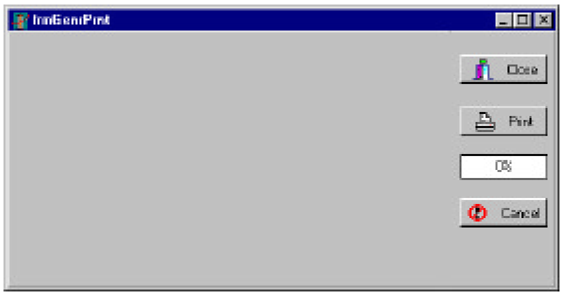


Figure 5: The *TfrmGenrPrnt* screen image. This form can be used as a basis for controlling any process that executes over a period of time.

```
procedure TfrmGenrPrnt.btnPrintClick(Sender: TObject);
begin
  btnClose.Enabled := False;
  btnPrint.Enabled := False;
  with gagProg do
    if HideGagProg then
      Visible := False
    else
      begin
        MaxValue := 100;
        Progress := 0;
        Visible := True
      end;
  btnCancel.Visible := True;
  FStopping := False;
  FStopped := False;
  Cursor := crHourglass;
  Application.ProcessMessages;
  try
    DoPrint;
  finally
    Cursor := crDefault;
    btnCancel.Visible := False;
    gagProg.Visible := False;
    btnClose.Enabled := True;
    btnPrint.Enabled := True
  end
end;
```

Figure 6: The *TfrmGenrPrnt.btnPrintClick* procedure.

Finally, the form implements two protected virtual routines: *PostChanges* and *CancelChanges*. These routines are called by *FormCloseQuery*, and by the save and discard speed buttons. They should be overridden to store any changes the user has made, or to restore the original values. At the end of your implementation of these two methods, be sure to call **inherited**, so the save and discard speed buttons will be reset.

The Generic Print Screen

Most applications generate reports or perform other tasks that take place over a period of time. The *TfrmGenrPrnt* form provides the basic functionality to support such features (see Figure 5). It supplies a Close button to exit the screen, and a Print button to run the process. You can change the Print button's caption and glyph if you're using the form for some other process, such as importing or exporting data. *TfrmGenrPrnt* also has a progress gauge and a Cancel button, both of which are hidden except when the process is running.

To run a descendant form, the user selects options you have placed on the *pnlForm* panel of the descendant form, and then presses the Print button. The *btnPrintClick* event handler (see Figure 6) disables the Print and Close buttons, and makes the progress gauge and Cancel buttons visible. It then calls the *DoPrint* procedure. This routine is declared as a virtual abstract method, so the descendant has to declare and define an overriding *DoPrint* method.

The descendant's *DoPrint* does the real work, typically in a loop. It should first determine the number of iterations for the loop, then set the progress gauge's *MaxValue* property accordingly. At the end of the iterations, the routine should increment the *Progress* property. (For those processes that don't lend themselves to the progress gauge paradigm, the ancestor publishes a *HideGagProg* Boolean variable. It defaults to False, but if the descendant sets it to True, the *btnPrintClick* handler doesn't display the gauge. The programmer should then provide some other way to show users that something is occurring.)

Part of *DoPrint*'s loop control should also check the ancestor's *Stopping* property. It's initialized to False, but if the user clicks the Cancel button, the event handler sets *Stopping* to True. The next time the descendant checks the property, the ancestor displays a dialog box asking whether to cancel the process. If not, *Stopping* is reset to False, and the descendant's loop continues.

To iterate through a *TTable*, a descendant form's *DoPrint* method might resemble Figure 7.

```
// Confirm user's setting choices, open table(s), etc.
...
with tblXXXX do begin
  gagProg.MaxValue := RecordCount;
  First;
  while not EOF and not Stopping do begin
    // Process the record.
    ...
    with gagProg do
      Progress := Progress + 1;
    Next;
  end
end;
// Close the table(s), etc.
...
```

Figure 7: A form's *DoPrint* method iterating through a *TTable*.

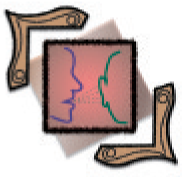
Once the descendant's *DoPrint* routine finishes, the ancestor's *btnPrintClick* handler completes the process by hiding the progress gauge and Cancel button, and re-enabling the Close and Print buttons.

Conclusion

In the two years since I started using this hierarchy of inherited forms, I have saved literally hundreds of hours of programming time. Also, my customers have found it easy to learn the applications and to move from one application to another. This process has been well worth the time it took to learn. In a nutshell, it allows true rapid application development. Δ

The files accompanying this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\AUG\DI200008RN.

Ron Nibbelink develops PC database applications for the Quality and Process Improvement organization of the Boeing Commercial Airplane Group in Wichita, KS. He has developed applications on a variety of platforms — in several languages — since 1983.



By Alexander Gofen

Recursion Excursion

Building an Advanced Expression Calculator

The goal of this article is to introduce several procedures that perform parsing and evaluation of complex arithmetic expressions with variables, parentheses, and functions. We're going to consider procedures capable of parsing and evaluating a formula or a chain of formulas, represented by a string or string list. Thus, the users of your application will be able to input various formulas and compute them within your application.

This computing may be performed either directly in an interpretive (and rather slow) way, or by "compiling" the formulas into a so-called postfix representation, and then using it for massive computing at a speed comparable to having the formulas hard-coded at design time and compiled. Naturally, it also gives us an opportunity to analyze the usefulness and elegance of recursion techniques for parsing arithmetic expressions and evaluating their postfix representations.

Finally, a new component — an advanced calculator — will be introduced that will allow users to do some math while running your application. This advanced calculator component will allow users to deal with sets of formulas and series, and is much more useful than the calculator that comes with Windows.

Interpretive Evaluation

A procedure or function is *recursive* if it calls itself within the body of its own declaration, either directly or through other procedures. It's a feature allowed in Pascal and several other high-level languages. A trivial example found everywhere is function *n!*:

```
if n = 1 then
  Factorial := 1
else
  Factorial := n*Factorial(n-1);
```

However, it's usually recommended that you implement this function via an iterative loop. The recursive form here is good only in that it exactly matches the definition of Factorial, but in practice, each function call requires a certain overhead; implementation of multiple function calls would work less efficiently than a simple iterative loop in this case. Nevertheless, there are problems whose mathematical models are recursive, and recursive algorithms for them are not only the most natural, but the preferred solution.

One of these situations is an evaluation of an arbitrary arithmetic expression with parentheses, functions, and variables in the standard mathematical notation (+, -, *, / and ^ for power). We're going to deal with lists of such expressions, where each line may be either any arithmetic expression, or an equation in the form:

```
Variable = Expression
```

where *Expression* is an arithmetic expression containing either numbers or variables defined in earlier lines. This is Linear Elaboration of Declarations (LED) in ADA terminology — the chain of formulas where each variable first has to appear in the left-hand part of an equation, before it's used in the right-hand part, as shown in Figure 1.

```
13.1 + 12.3
vol = 300
s = 13
height = vol/s
a = 10*(height - 2)
r = sqrt((a - 2)^2 + a^2)
arcsin(a/r)
e=2.718281828
x=1
y=2
e^(-x^2-y^2)
```

Figure 1: Linear Elaboration of Declarations.

It may be represented by a variable of type *TStrings*, say via the *TMemo.Lines* property. Let's define a pair of overloaded functions:

```
function Evaluate(const ValStrings: TStrings):
  Boolean; overload;
function Evaluate(const str: string; out res: Extended;
  const ValStrings: TStrings = nil): Boolean; overload;
  { Optional list of declarations. }
```

These functions will cover the following three situations:

1) The input is purely in *TStrings* type, and represents any list of expressions obeying the principle of LED — the first version of the function.
2) The input is in a simple string *str* (with only a numeric expression in this case), and the result is in a variable *res* (default parameter *ValStrings* is omitted) — the second version of the function.
3) A combination of 1 and 2 (parameter *ValStrings* contains variables and equations) — also the second version.

In all three cases, a special dynamic array, `ResultArr: array of Extended`, is associated with the lines of *ValStrings* and represents the results of the evaluation (if successful), as found in the unit ParsList.pas, in the \AdvCalc\Source folder of the included download file (see end of article for details).

The core problem solved by these functions is the parsing and evaluating of an arithmetic expression built according to Pascal syntax (more precisely, to that of ALGOL-60 because we're going to use a caret to denote raising to power, for example, 2^2^n, meaning 2^(2^n), or e^(-x^2-y^2)). Syntax of expressions in programming languages is usually presented by Backus-Naur Form (BNF), as shown in Figure 2.

```
<TermOp> ::= + | -
<Expression> ::= <Term> | <TermOp><Term> | <Expression><TermOp><Term>
<FactOp> ::= * | /
<Term> ::= <Factor> | <Term><FactOp><Factor>
<Factor > ::= <PrimaryExpr> | < Factor >^<PrimaryExpr>
<PrimaryExpr> ::= <unsigned number> | <variable> | <function call> | (<Expression>)
```

Figure 2: Backus-Naur Form (BNF).

The recursive definitions are present in all three components (Expression, Term, and Factor), but may be easily eliminated from the former two. The BNF for Expression just encodes the fact that it is a sequence of Terms delimited by the Term Operation signs (and it can possibly start with these signs). Similarly, a Term is a sequence of Factors delimited by the Factor Operation signs. Thus, the key procedure *Expression* (see Listing One beginning on page 20) uses iterative loops for parsing Terms and Factors, being recursive only because it implements the definition of the Factor and Primary Expression (the *Expression* procedure is a modification of that by Jensen and Wirth [1978]).

The *NextToken* procedure (see Figure 3) scans *WorkStr* and returns the next token (in a string variable *Tkn*), which is either an alpha-numeric string (possibly a number or a variable), a character (expected to be an operation sign + ,- ,* ,/ ,^, or parenthesis), or empty.

```
procedure NextToken;
begin
  Tkn := '';
  if cur > len then
    Exit;
  if WorkStr[cur] in AlfaNum then
    { Returns a name or number. }
    while (cur <= len) and
          (WorkStr[cur] in AlfaNum) do begin
      AppendStr(Tkn, WorkStr[cur]);
      Inc(cur);
    end
  else  { Returns an operation sign. }
    begin
      Tkn := WorkStr[cur];
      Inc(cur)
    end
end;
```

Figure 3: The *NextToken* procedure.

Any procedure call requires allocation of the stack memory for the parameters, local variables (if any), and for the result (in the case of a function). To improve performance, one has to minimize this overhead. Thus, the functions *Expression*, *Term*, and *Factor* are designed without parameters, and with a minimal number of local variables.

Different exceptions may occur and be raised while parsing. The user can either handle them in his/her **try..except** clause, or just analyze the Boolean result of the evaluation after acknowledging the warning message in the ShowMessage box.

The recursion here will not be infinite; it ends when the scanning reaches the end of *WorkStr*, and the *Tkn* string is empty.

This procedure is used to build the advanced calculator and works pretty well, but how fast is it? As a benchmark, I used a sample where repetitive calculation of equations was run, first hard-coded and compiled, and second, via the parsing procedure, *Evaluate*, described previously. The sample equations are shown in Figure 4.

```
x := 1
y := 2
z := 3
f := sqrt(x*x + y*y + z*z)
```

Figure 4: Sample equations.

As you might predict, the parsing works much slower than the hard-coded compiled code; with a 100MHz Pentium, it took 1870 µs versus 2.6 µs, or 720 times slower. Fortunately, this speed doesn't really matter. It serves only user-driven dialog boxes like that in the advanced calculator, where something near 2 ms per formula is still fast enough. This isn't always the case, however.

Suppose the evaluation of the formulas inputted by the user needs to be performed massively for different values of the arguments, rather than just once. For example, this is necessary for drawing a complex 2D or 3D image defined by the given equations. In this situation, a decrease in speed of a thousand times less than that of the compiled code is absolutely unacceptable. Thus, we will consider an alternative approach.

Compiled Evaluation

Processing the user-defined equations involves parsing, searching variables and function names in the corresponding lists, and evaluating ASCII-coded numbers. That is what causes such a dramatic decrease in speed. Is it possible to do this time-consuming processing just once and encode all operations, their sequence, and intermediate binary values into a certain structure so that later an efficient algorithm could evaluate this structure for different arguments at a much greater speed? One such structure is known as the *postfix notation sequence* (Polish notation). Postfix notation sequence is a representation without parentheses, where the operation codes follow their corresponding operands. For example, the postfix notation for a + b*c is a b c * + ; for $e^{-(x^2 + y^2)}$, it is x 2 ^ y 2 ^ + - exp.

The idea is that first we compile the source list of formulas into such a structure, and, if successful, we apply an efficient algorithm evaluating these postfix notations. Now let's define the corresponding structures and procedures.

The arithmetic expressions are built from operations requiring either two arguments, or just one (such as the elementary functions and unary minus). For simplicity's sake, we accept the convention that any operation requires two arguments; if there is an excess argument, it should always be 0. Then, in the case of unary minus (or plus), it works correctly (-x = 0 - x, or, in postfix notation, 0 x -), while for the predefined elementary functions, the second argument is simply ignored.

Our input data — a list of formulas — comes as *TStrings*. The output data types representing the postfix notation are shown in Figure 5. They are dynamic arrays of type *object*. For each formula (a string with index *i* of *TStrings*), there is exactly one real number in dynamic array *Parameters[i]*, and one postfix notation line, *PostfLines[i]*, in the object *TPostfixList* corresponding to this formula. Each element of *PostfLines*

```
var Parameters: array of Extended;

type
  TOperand = object
    { If not negative, it points to elem. of Parameters. }
    ParIndex: Integer;
    { Actual numeric value if ParIndex < 0. }
    ActualVal: Extended;
    function Value: Extended;
  end;

  TPostfixItem = object
    op: Char;          { Operation code or nop. }
    Operand: TOperand; { Valid only if op = nop. }
    function myFunctions(const y, x: Extended): Extended;
  end;

  TPostfix = object { One postfix notation sequence. }
    Count: Integer;
    Items: array of TPostfixItem;
    procedure Init(const curLine: Integer);
    function Evaluate: Extended;
  end;

  { List of postfix notation sequences. }
  TPostfixList = object
    Count: Integer;
    IsCorrect: Boolean;
    PostfLines: array of TPostfix;
    function Compile(const ValStrings: TStrings;
      { optional. } PfixLookList: TStrings = nil): Boolean;
    function Evaluate(
      const GivenParams: array of Extended): Extended;
  end;
```

Figure 5: Data structures to represent a list of postfix notation sequences.

includes a dynamic array of object *TPostfixItems*. Each postfix item represents either an operation code *op*, or an *Operand*. The latter is represented either by a real value *ActualVal* (which contains numeric constants from the formulas), or an integer index *ParIndex*, referring to the global dynamic array *Parameters* (containing values of the already evaluated formulas in the previous lines).

Users usually deal only with two methods of the object *TPostfixList*. First is:

```
function TPostfixList.Compile(const ValStrings: TStrings;
  { optional. } PfixLookList: TStrings = nil): Boolean;
  { just to show postfix notation.}
```

to compile the source list of formulas into the list of postfix sequences; and then:

```
function TPostfixList.Evaluate(
  const GivenParams: array of Extended): Extended;
```

for massive calculations with this list for different sets of the parameters.

For example, suppose that given the formulas in Figure 4 and default values of x, y, z are in *Memo1.Lines*. Then if the declaration:

```
var MyPList: TPostfixList
```

if *MyPList.Compile(Memo1.Lines)* is successful, we can call:

```
MyPList.Evaluate([5,6,7]), MyPList.Evaluate([8,9])
```

or do that in some loop. In other words, we can now override dynamically a certain number of the beginning lines in a list of linearly elaborated declarations.

Project1 in folder Postfix allows users to enter different formulas to see the corresponding postfix notations and to evaluate them. The parsing is performed in unit Pars.pas by a procedure (see Listing Two on page 21) similar to that of Jensen and Wirth (1978). The postfix processing code is in the unit Postfix.pas. (Both are in the folder Postfix of the files included with this article.)

The key method here is the function *Evaluate* of object *TPostfix*, containing an elegant and essentially recursive function named *GetResult*. In general, there are many ways to evaluate a postfix notation sequence. Select any triplet of consecutive items (Operand Operand Operation), substitute it with their result as a new Operand, and so on, until the process ends with just one Operand — the result of the evaluation. But we need an algorithm that:

- does not change the source postfix sequence, leaving it intact for multiple use; and
- does not create new copies of the source sequence (which would be costly).

The recursive function, *GetResult*, in *TPostfix.Evaluate* was designed with these in mind (see Figure 6).

A non-local variable, *Pos*, is assumed pointing to the position next to where we want the processing of the postfix sequence to start. Every call to *GetResult* decrements *Pos* (as a side effect). Depending on what type the *Items[Pos]* is, *GetResult* returns either the value of the *Operand* and immediately ends, or it calls the universal function *MyFunction* to compute the required operation over the two actual parameters — recursive calls to *GetResult*. Left-to-Right parameter evaluation here is crucial; otherwise, the procedure will not work. (The Left-to-Right evaluation order corresponds to the parameter passing conventions register and Pascal, and the default convention in Delphi is register.)

First, we have to call it with *Pos* pointing to the position after the end of a postfix sequence, i.e.

```
Pos := Count
```

and the end of a correct postfix sequence must be an operation code. Further actions evolve depending on what items precede the current position in the postfix sequence, building a sophisticated tree of recursive calls. But it is easy to prove that it works correctly, using mathematical induction.

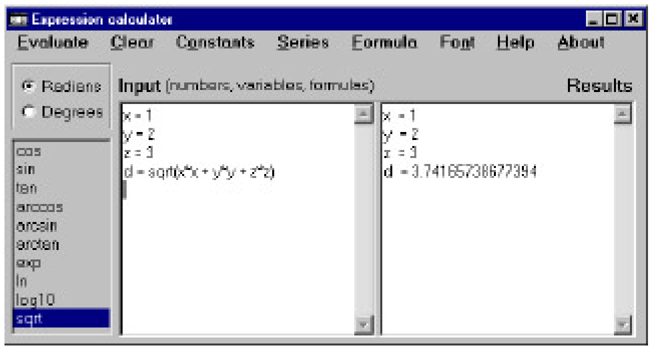
The shortest (trivial) sequence consists of just one Operand. The next by length is a triplet Operand Operand Operation. It's easy to see that in both cases, *GetResult* returns a correct result, with *Pos* pointing to the last processed item. This is the basis of the induction. Now, assuming that *GetResult* works correctly for sequences of length *n* and less, we have to prove that it works for sequences of the length *n*+1.

```
function TPostfix.Evaluate: Extended;
var
  Pos: Cardinal; { Must point to position next to the
                   desired processing start. }
  function GetResult: Extended;
  begin
    Dec(Pos);   { Pos < 0 may happen only in case of
                  mistakes in postfix sequence. }
    with Items[Pos] do
      if op = nop then
        { Items[Pos] is an operand. }
        Result := Operand.Value
      else
        { Items[Pos] is an operation. }
        Result := MyFunctions(GetResult, GetResult)
    end;
    { Left-to-Right parameter evaluation is essential! }
begin
  Pos := Count;
  Result := GetResult
end; { TPostfix.Evaluate.}
```

Figure 6: The recursive *GetResult* function.

Pos	**1**	**...**	***n*-1**	***n***	***n*+1**
Case 1	...	...	Operation	Operand	Operation
Case 2	...	...	...	Operation	Operation

Figure 7: Two possible cases for the item at the position *n*.



```
x = 1
y = 2
z = 3
d = sqrt(x*x + y*y + z*z)
```

```
x = 1
y = 2
z = 3
d = 3.74165738677394
```

Figure 8: The advanced calculator.

Then, the last item, *Items[n+1]*, must be an Operation. Therefore, at the beginning, *GetResult* goes to the branch:

```
Result := MyFunctions(GetResult, GetResult)
```

with *Pos* pointing to *n*. There are two possible cases for the item at the position *n*, and they're presented in the table in Figure 7.

In Case 1, the first-parameter call immediately returns the value of the Operand. The second-parameter call, starting with position *n*-1, must return the correct result by the induction assumption, "consuming" all remaining items.

In Case 2, the first-parameter call must return the correct result by the induction assumption also, stopping at some position *m*, where 0 < *m* < *n*. Otherwise, the postfix sequence would be wrong. The remaining items from 0 to *m*-1, again by the induction assumption, must present a correct postfix sequence, and be processed successfully by the second-parameter call. This concludes the proof.

To test the efficiency of the compiled evaluation, I used as a benchmark the same formulas used earlier, and it took 11.6 μs per formula. If the same, but hard-coded formula sqrt(x*x + y*y + z*z) in the format:

```
sqrt(Parameters[0]*Parameters[0]+Parameters[1]*
  Parameters[1]+Parameters[2]*Parameters[2])
```

is substituted into the method *TPostfList.Evaluate* (instead of the *PostfLines[i].Evaluate*), it takes 2.6 μs. Our postfix evaluation requires 4.45 times more than the hard-coded version of this formula, but this is quite reasonable. The recursive function, *GetResult*, although itself without parameters, calls *MyFunctions* (which has two parameters) and therefore initiates two more recursive calls and corresponding records in the stack. Will it grow exponentially (like in the case of the notorious Ackerman function)? Fortunately, each call decrements the counter of postfix items, and the total number of calls cannot exceed the initial number of items in the postfix sequence — at least for a correct postfix sequence. The method *TPostfList.Evaluate* prevents attempts to do this if the field IsCorrect = False in the compiled object.

Features of the Advanced Calculator

Unlike other software imitations of the old days, where (scientific) calculators had only a one-line display and a hidden register, this advanced calculator takes full advantage of the computer screen, implementing the standard mathematical notation for lists of formulas (see Figure 8).

When editing the lines in the **Input** box, every time you press Enter↵ or click **Evaluate** on the menu, the computer performs an evaluation of the whole list of lines, displaying the results in the **Results** box. Conversely, if you have changed the **Input**, but not pressed Enter↵ or clicked **Evaluate**, the **Results** box is cleared.

The advanced calculator is an object of type *TForm*. You can use it as either a part of your application, adding the required forms and units to your project (the entire content of the folder AdvCalc\Source), or as a separate application (the folder AdvCalc\Exec). In the former case, when adding the forms and the units, the forms HelpForm and AboutBox should not be auto-created (the form CalcForm creates, opens, and releases them).

Calculations with Series

To perform calculations listed on the **Series** item of the menu, you must first input the terms (one to a line) of the desired series in the **Input** box (each term may be a number, a formula, or an equation). Then highlight the lines and select the necessary function from the **Series** drop-down menu. With the following six lines, for example, if you highlight the five lines beginning with "100":

```
123 + 456
100
11^2
12^2
k = 13^2
14^2
```

and click **Series | Sum** in the menu, it allows you to obtain `100 + 11^2 + 12^2 + 13^2 + 14^2 = 730`.

Calculations Using a Formula

To perform calculations using the same formula for a series of values of a certain variable, say *var*, first enter the list of the values, followed by the formula, highlight them all together, and click **Formula** in the menu. For example, if you highlight this:

```
123 + 456
var = 1
2
3
4
f = var^3
```

or this:

```
123 + 456
var = 1
2
3
4
var^3
```

and click **Formula** in the menu, it allows you to obtain function var^3 for var = 1, 2, 3, 4.

Conclusion

We've considered the problem of parsing and evaluating arithmetic expressions for lists of linearly elaborated declarations, and discovered that recursive procedures provide elegant and efficient solutions. We also introduced the structures — a hierarchy of objects — for representing and processing postfix notation sequences. Incidentally, these structures are of type *object* (rather than *class*), proving that the "old style" objects may still deliver the simplest solution for certain problems. (In this particular case, only the encapsulation feature of the OOP was used, because inheritance and polymorphism were not required.) Δ

References

K. Jensen, N. Wirth (1978). PASCAL. User Manual and Report.

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\AUG\DI200008AG.

Alexander Gofen is a programmer at the Smith-Kettlewell Eye Research Institute in San Francisco, CA where he has been working since 1995. Previously, he was a Senior Researcher for the Institute of Computer Science (Academy of Sciences, Russia) and Hydro-Meteorological Center in Moscow, Russia. Gofen has been developing scientific applications in all versions of Delphi and Borland's Pascal, varying from the Numeric Weather Forecast and the Taylor Solver to the Macula Mapping Test (Eye Research Institute). He can be reached via phone at (415) 345-2119, or e-mail at galex@ski.org.

Begin Listing One — Interpretive parsing and evaluation

```
function Expression: Extended;
var
  TermOp, FactOp: Char; i: Shortint;

  function Term: Extended;

    function Factor: Extended;
    begin
      i := Functions.IndexOf(Tkn);
      if i >= 0 then  { Function name correct. }
        begin
          NextToken;
          if Tkn <> '(' then
            raise Exception.CreateFmt(
              '%s'#10#13' "(" expected but "%s" found',
              [CurSegm,Tkn])
        end;
      if Tkn = '(' then
        { Factor is a function or expression. }
        begin
          NextToken; Result := Expression; { recursive! }
          if Tkn <> ')' then
            raise Exception.CreateFmt(
              '%s'#10#13' ")" expected but "%s" found',
              [CurSegm,Tkn]);
          NextToken;
          if i >= 0 then
            { For expression i < 0. }
            Result := MyFunctions(Result, i)
        end { Factor was a function or expression. }
      else { Factor is a variable or number. }
        begin
          if Tkn = '' then
            raise Exception.CreateFmt(
           '%s'#10#13' operand expected but nothing found',
              [CurSegm]);
            if EvaluVar(Tkn, Result) then
              NextToken
            else
              raise Exception.CreateFmt(
                '%s'#10#13' cannot evaluate %s',
                [CurSegm, Tkn])
        end;  { Factor is a variable or number,
                possibly followed by power index. }
      while Tkn = '^' do begin
        NextToken;
        realExp := Factor;  { recursive! }
        if Frac(realExp) <> 0.0 then
          Result := Power(Result, realExp)
        else
          Result := IntPower(Result, Floor(realExp))
      end
    end; { Factor. }

  begin { Term. }
    Result := Factor;
    while (Tkn = '*') or (Tkn = '/') do begin
      FactOp := Tkn[1]; NextToken;
      case FactOp of
        '*': Result := Result * Factor;
        '/': Result := Result / Factor
      end
    end
  end; { Term. }

begin { Expression. }
  Result := 0.0; { In case of unary + or -. }
  if (Tkn <> '+') and (Tkn <> '-') then
    Result := Term; { No unary + or -. }
  while (Tkn = '+') or (Tkn = '-') do begin
    TermOp := Tkn[1]; NextToken;
    case TermOp of
      '+': Result := Result + Term;
      '-': Result := Result - Term
    end
  end
end; { Expression. }
```

End Listing One

Begin Listing Two — Parse and create postfix notation sequence

```
procedure Expression;
var
  TermOp, FactOp: Char; i: Shortint;
  procedure Term;

    procedure Factor;
    begin
      i := Functions.IndexOf(Tkn);
      if i >= 0 then { Correct function name. }
        begin
          NextToken;
          if Tkn <> '(' then
            raise Exception.CreateFmt(
              '%s'#10#13' "(" expected but "%s" found',
              [CurSegm,Tkn]);
        end;
      if Tkn = '(' then
        { Factor is a function or expression. }
        begin
          NextToken; Expression; { Recursive! }
          if Tkn <> ')' then
            raise Exception.CreateFmt(
              '%s'#10#13' ")" expected but "%s" found',
              [CurSegm,Tkn]);
          NextToken;
          if i >= 0 then { Function #i. }
            begin
              TempPfxLine.Add('0'); { 2nd operand 0. }
              TempPfxLine.Add(Chr(i))
            end
        end { Factor was a function or expression. }
      else { Factor is a variable or number. }
        begin
          if Tkn = '' then
            raise Exception.CreateFmt(
              '%s'#10#13' operand expected but not found',
              [CurSegm]);
          TempPfxLine.Add(Tkn); NextToken
        end; { Factor is a variable or number,
               possibly followed by power index. }
      while Tkn = '^' do begin
        NextToken; Factor; { Recursive! }
        TempPfxLine.Add('^')
      end
    end; { Factor. }

  begin { Term. }
    Factor;
    while (Tkn = '*') or (Tkn = '/') do begin
      FactOp := Tkn[1]; NextToken; Factor;
      TempPfxLine.Add(FactOp)
    end
  end; { Term. }

begin { Expression. }
  if (Tkn = '+') or (Tkn = '-') then
    TempPfxLine.Add('0') { 0 +... or 0 -... }
  else
    Term;
  while (Tkn = '+') or (Tkn = '-') do begin
    TermOp := Tkn[1]; NextToken; Term;
    TempPfxLine.Add(TermOp)
  end
end; { Expression. }
```

End Listing Two

By Mike Riley

WAP Apps

Writing WAP-WML Applications in Delphi

Since its third version, Delphi has provided programmers with easy tools and language enhancements designed to connect data to the Web. The form of data delivery has been primarily intended for Web browsers running on PCs physically connected to the Internet via a wire. Ubiquitous Internet connectivity has recently evolved to the next level, with the introduction of wireless connectivity for TCP/IP-intended data to portable, hand-held displays. The two recently established standards that have allowed this to occur are Wireless Application Protocol (WAP) and Wireless Markup Language (WML).

WAP-WML for an HTTP-HTML World

WAP and WML are analogous to HTTP and HTML, respectively. The protocol and markup languages were designed for optimal delivery of short bursts of data traveling along a wireless network. These are traditionally Cellular Digital Packet Data (CDPD) and Mobitex Wireless Data (a.k.a. Bell South Wireless Data [BSWD]) networks, due to the extensive infrastructure previously constructed for digital mobile phone service. Because digital wireless communication is still being deployed, WAP-WML functionality may not be available in all areas for some time. However, most major metropolitan cities already have these networks operational, with additional coverage areas being added daily. It's only a matter of time before the entire planet will be accessible through these wireless digital pathways.

So why develop another Web protocol? WAP is optimized for wireless network delivery. Hence, gateways provided by companies such as Phone.com (developers of the Unwired Planet wireless Web browser) convert TCP/IP Web transmissions into WAP delivery over the CDPA-BPMA wireless networks. Doing so further allows the data packets to minimize the delivery time over the network, while using the extensive infrastructure already installed for wireless voice communications.

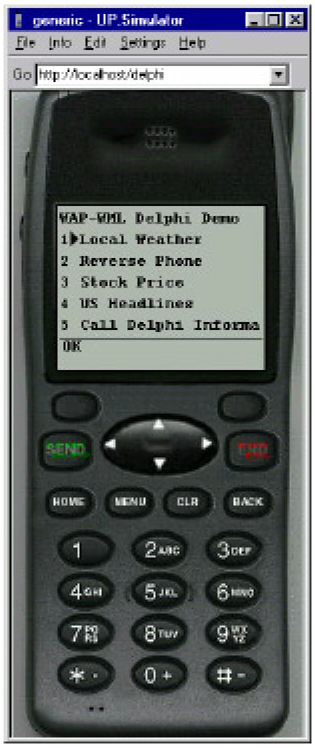
So why develop another markup language? One glance at the display on a digital mobile phone or pager is convincing enough. The screen real estate on many of these devices is often limited to 80 characters or less. To imbue additional formatting rules, such as style sheets and absolute positioning, would not only require a more expensive processor and increased memory in the device to handle the advanced rendering needs, but it would also be impractical from a designer's standpoint, given the tiny view of data the user will be able to see at one time. To help offset the reduced screen size and lengthy data packet transmission time, WML employs the concept of a "card deck" to provide multiple page views in a single transmission. This reduces the amount of traffic between the client and server, and provides the user with a perceived dramatic increase in access speed.

WAP/WML-enabled devices are just now hitting the consumer market. Many of the Web-enabled phones that were previously released are embedded with an older markup methodology developed by Unwired Planet, called Handheld Device Markup Language (HDML). With the advent of XML and the need for an open consortium-developed standard, Unwired Planet submitted HDML as a template from which to construct WML. Hence, HDML may still be required for "legacy" devices for some time. Rather than dwell on the constructs of this antiquated markup language, I leave it to the reader to pursue the variations between the two languages. Because WML was developed from the seed of HDML, it's often easier to initially develop content in HDML format and upsize it to WML. Unfortunately, there is currently no way for older HDML browsers to gracefully degrade WML to a renderable state. The UP WML-enabled browsers being embedded in the new line of portable communication devices can render HDML for precisely these legacy code reasons.

The Path to Delphi WAP-WML Enlightenment

First, research the links listed in the References section at the end of this article. The first stop should be http://updev.phone.com to obtain the free Phone.com UP.SDK 4.0. The SDK provides WML reference documentation, samples, and a phone simulator. Using the simulator is much less expensive than developing on a live device. It features a number of good debugging services as well.

Next, install Personal Web Server 4.0 (or higher) if you're isolated to a Windows 95/98 workstation. Otherwise, install IIS 4.0 (or higher) on an accessible Windows NT Server platform. While I could have just as easily written this demo as an ISAPI or NSAPI DLL using Delphi's Web dispatcher technology, I decided to adopt the ASP object approach introduced by Ron Loewy in his March, 1999 *Delphi Informant Magazine* article, "Active Server Pages." I did this because most developers running Win32-based Web services these days are running them from PWS or IIS because they're free, integrate well within the Windows OS, provide easier debugging, and are well supported. Naturally, the other advantages that Mr Loewy asserted for ASP-based Delphi COM objects continue to hold true.



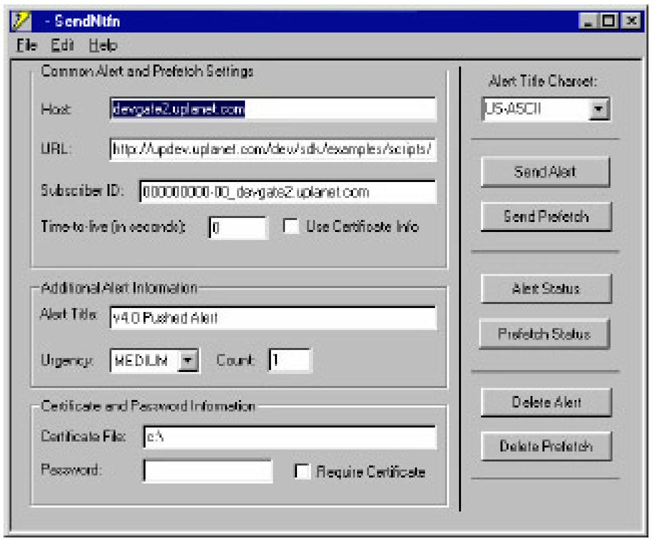
generic - UP.Simulator

Go http://localhost/delphi

WAP-WML Delphi Demo
1 Local Weather
2 Reverse Phone
3 Stock Price
4 US Headlines
5 Call Delphi Informa
OK

Figure 1: The Phone.com simulator.

Once the Web server is operational, create a directory named WML in the Web server's root directory, and mark it for read and scripting privileges. To immediately test the installation, download the code included with this article (see end of article for details), unzip it, and place the contents of the archived WML directory into the WML directory you created within your Web server's root directory. Register DIWML.DLL via the regsvr32.exe program, launch the UP.Simulator, and target the directory via http://localhost/wml/. The results displayed should be identical to Figure 1.



- SendNtfn

Common Alert and Prefetch Settings

Host: devgate2.uplanet.com

URL: http://updev.uplanet.com/dev/sdk/examples/scripts/

Subscriber ID: 000000000-00_devgate2.uplanet.com

Time-to-live (in seconds): 0 ☐ Use Certificate Info

Alert Title Charset: US-ASCII

Additional Alert Information

Alert Title: v4.0 Pushed Alert

Urgency: MEDIUM Count: 1

Certificate and Password Information

Certificate File: c:\

Password: ☐ Require Certificate

Figure 2: The Phone.com SDK gateway testing application.

I have also included an hdml directory, which provides the demonstration project outputting content in HDML format. This can be immediately employed to deliver to UP.Browser-enabled phones in consumers' hands today, including mobile phones offered through service providers, such as Sprint PCS.

In addition, the Phone.com SDK includes a gateway testing application, illustrated in Figure 2. To use this application, a free developer account must be established with Phone.com's "devgate" gateway. This gateway provides alert messaging capabilities

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
  "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <card  title=" WAP-WML Demo">
  <p mode="nowrap"> WAP-WML Delphi Demo
  <select>
    <option onpick=
      "http://www.accuweather.com/uwp/weather/95624/menu">
      Local Weather</option>
    <option  onpick=
      "http://guides3.infospace.com/
        _1_40JOUNF04MP191Y__phone.att/reverse.hdml">
      Reverse Phone</option>
    <option onpick=
      "http://ff5.quote.com/fq/uplanet/quote">
      Stock Price</option>
    <option onpick=
      "http://204.202.137.120/up/sections/us/index.hdml">
      US Headlines</option>
    <option  onpick=" wtai://wp/mc;9166866610">
      Call Delphi Informant</option>
    <option onpick="#server">Server Info</option>
    <option onpick="#about">About About this demo</option>
    <option onpick="#mike">About Mike Riley</option>
  </select>
  </p>
  </card>
  <card  id="server">
    <p>
    Available space on drive c: 917MB of 6266MB.
    </p>
  </card>
  <card  id="about">
    <p>
      Welcome to the wireless version RileyFAN,
      Mike Riley's Family Area Network.
    </p>
  </card>
  <card  id="mike">
    <p>
      Mike Riley is currently working for RR Donnelley
      &amp; Sons as the company's Director of Internet
      Application Development.
    </p>
  </card>
</wml>
```

Figure 3: A formatted version of sample WML output sent to the UP.Simulator.

to the developer for event notification applications requirements. Although only VB and VC++ code examples of the SendNtfn alert messaging application is included in the SDK, porting key components of this application to Delphi is a painless endeavor.

The Code

The code is straightforward. The ASP COM details have already been discussed in Mr Loewy's article, so I will focus primarily on the *Main* procedure (see Listing One). One notable line of code is properly setting the *ContentType* equal to text/vnd.wap.wml, not the default text/html that is intended for Web browsers.

The other significant code fragment is the line containing the *DiskFree* and *DiskSize* Delphi System unit calls. This demonstrates the primary reason why developers may prefer to write ASP-based COM DLLs in Delphi rather than rely on Microsoft-provided ASP function libraries. Review the code in Figure 3 for a formatted version of sample WML output sent to the UP.Simulator.

Conclusion

Once developers become comfortable with WML, the accessibility and convenience of untethered client/server communication, and the portability of these Web-enabled devices, the creative opportunities really blossom. Although the demonstration code provided in this article was limited to checking disk space, the possibilities of combining the rich programmatic strengths of Delphi with remote communication are endless. Imagine processing a batch billing cycle with the press of a button, or sending off Web-clipping service bots to report on news, stock, and weather conditions, or even linking into X-10, Jini, or Microsoft's Universal Plug-and-Play home devices to remotely configure and inquire on your home's status.

And to think, all of this can be exercised today while relaxing on a beach with a cool drink in one hand, and a cool phone in the other. ∆

References

- HDML specification and Phone.com's developer site, http://updev.phone.com
- WAP and WML specifications, http://www.wapforum.org/what/technical.htm
- Wireless simulators — UP.Simulator, available at http://updev.phone.com
- RIM Developer Zone, http://developers.rim.net/handhelds/index.html
- Ron Loewy's ASP article, http://www.DelphiZine.com/features/1999/03/di199903rl_f/di199903rl_d.asp
- Microsoft Personal Web Server 4.0, http://www.microsoft.com/Windows/ie/pws/default.htm
- Microsoft IIS 4.0, http://www.microsoft.com/ntserver/nts/downloads/recommended/NT4OptPk/default.asp

Recommended WML device service providers:

- Sprint PCS for UP.Browser-enabled phones, http://www.sprintpcs.com
- GoAmerica for RIM-enabled WML messaging units, http://www.goamerica.net

The project referenced in this article is available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\AUG\DI200008MR.

Mike Riley is the Director of Internet Application Development for RR Donnelley & Sons, North America's largest printer. He actively participates in the company's Internet, intranet, and extranet strategies using a wide variety of Web-enabled technologies, including Delphi 4. Mike can be reached via his spam-shielding e-mail address, mike_riley_@hotmail.com.

Begin Listing One — The *Main* procedure

```
{ This procedure first validates the browser type via the
  Validate procedure call. If the browser is the correct
  type, the WML code is sent. Note the ContentType must be
  assigned to the data in order for the WML browser to
  render the content. Also, while Chr(10)+Chr(13) character
  combinations could have been added to each line to
  improve source readability in a standard browser, these
  characters are stripped and all white space is removed
  when WML code passes over a WAP gateway to further
  compress the packet size being delivered to the wireless
  device. Hence, including them would only be useful for
  more legible debugging purposes.

  Lastly, note the Delphi System unit call in the 'server'
  card. This is where the power of Win32-based commands is
  realized. Other possibilities include connecting to and
  displaying data sources, programatically activating
  server processes such as triggering backups, inquiring
  about event monitor status, sending e-mail, activating
  Web bots, sending data to the COM ports, and a slew of
  other possibilities. }
procedure TASPObject.Main;
begin
  Validate;
  if isvalid then begin
    ASPResponse.ContentType := 'text/vnd.wap.wml';
    ASPResponse.Write('<?xml version="1.0"?>');
    ASPResponse.Write('<!DOCTYPE wml PUBLIC' +
      ' "-//WAPFORUM//DTD WML 1.1//EN"');
    ASPResponse.Write(
      '"http://www.wapforum.org/DTD/wml_1.1.xml">');
    ASPResponse.Write('<wml>');
    ASPResponse.Write('<card  title="WAP-WML Demo">');
    ASPResponse.Write('<p mode="nowrap">');
    ASPResponse.Write('WAP-WML Delphi Demo');
    ASPResponse.Write('<select>');

    // Replace Delphi Informant's 95624 office ZIP Code
    // with your local zip code.
    ASPResponse.Write('<option onpick="http://www.' +
      'accuweather.com/uwp/weather/95624/menu">' +
      'Local Weather</option>');
    ASPResponse.Write('<option onpick="http://guides3. ' +
      'infospace. com/_1_40JOUNF04MP191Y__phone.att/' +
      'reverse.hdml">Reverse Phone</option>');
    ASPResponse.Write('<option onpick="http://ff5.quote' +
      '.com/fq/uplanet/quote">Stock Price</option>');
    ASPResponse.Write('<option onpick="http://204.202. ' +
      '137.120/up/sections/us/index.hdml">US Headlines' +
      '</option>');
    // Replace Delphi Informant's 9166866610 office phone
    // number with your phone number.
    ASPResponse.Write('<option  onpick="wtai://wp/mc; ' +
      '9166866610">Call Delphi Informant</option>');
    ASPResponse.Write('<option onpick="#server">' +
      'Server Info</option>');
    ASPResponse.Write('<option onpick="#about">' +
      'About this demo</option>');
    ASPResponse.Write('<option onpick="#mike">' +
      'About Mike Riley</option>');
    ASPResponse.Write('</select>');
    ASPResponse.Write('</p>');
    ASPResponse.Write('</card>');
    ASPResponse.Write('<card id="server">');
    ASPResponse.Write('<p>');
    ASPResponse.Write('Available space on drive c: ');
    ASPResponse.Write(IntToStr(DiskFree(3) div 1000000) +
      'MB of ' + IntToStr(DiskSize(3) div 1000000) +'MB.');
    ASPResponse.Write('</p>');
    ASPResponse.Write('</card>');
    ASPResponse.Write('<card id="about">');
    ASPResponse.Write('<p>');
    ASPResponse.Write('This WAP-WML session was' +
      ' generated by a Delphi-constructed ASP-enabled' +
      ' COM object.');
    ASPResponse.Write('</p>');
    ASPResponse.Write('</card>');
    ASPResponse.Write('<card  id="mike">');
    ASPResponse.Write('<p>');
    ASPResponse.Write('Mike Riley is currently working' +
      ' for RR Donnelley &amp; Sons as the company's ');
    ASPResponse.Write('Director of Internet Application' +
      ' Development.');
    ASPResponse.Write('</p>');
    ASPResponse.Write('</card>');
    ASPResponse.Write('</wml>');
  end;
end;
```

End Listing One

By Michael L. Perry

Dependency Tracking

Managing State, Not Process

When I was studying computer science in college, I took a FORTRAN course from an engineer. He taught that the model for a program was input, calculation, and output. The model worked well for all the problems he had solved; he couldn't envision a program apart from process.

Then I graduated and immediately went to work for an oil and gas company porting a DOS application to Windows. I quickly learned that the procedural model of application development no longer applied. The modern application is not input, calculation, and output. It is state dependent.

For example, take a financial ledger, such as a checkbook. The underlying data model is essentially a list of transactions. The order of the transactions depends upon their dates and check numbers. The balance of each transaction depends upon its amount and the balance of the previous transaction. The view, furthermore, depends upon the transactions, order, and balances. The most important task of the checkbook program is to update these dependent states as the user makes changes to the underlying data model.

The traditional approach is to design one process that sorts transactions, another that calculates balances, and another to draw the view. The designer then must consider every circumstance in which the order, balances, or view could change, and then update accordingly.

One popular mechanism for doing this is the Observer pattern, whose structure is shown in Figure 1 (for further reference, see *Design Patterns*, Gamma, et al. [Addison-Wesley, 1995]). This requires observers to register with their subjects, which imposes three restrictions.

- First, dependency relationships are static. An observer may only care about a particular subject under certain circumstances, but it must still register and respond to it in all cases.
- Second, care must be taken to destroy objects in the proper order. If the subject is destroyed before the observer, the observer will crash when it tries to un-register.
- Third, the observer might respond to many simultaneous changes. One action by the user could affect many subjects with which the same observer has registered, causing redundant updates.

Another mechanism is the document/view architecture (see Figure 2). A document template mediates the relationship between a document and its many views. Such a system requires three maintenance tasks:

1) Hint messages must be manually routed.

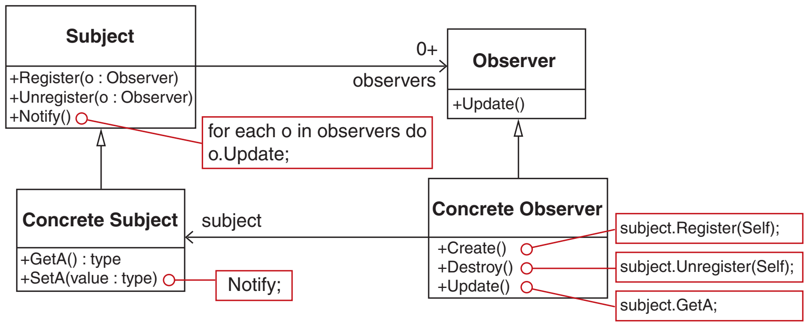


Figure 1: The Observer pattern.

2) A unique hint identifier is required for each possible state change.
3) Views must consider each possible circumstance in which they could be affected, and respond to all associated hints.

These three tasks make maintenance difficult. Developers spend more time keeping the hint mechanism operating than they do working on the actual problem.

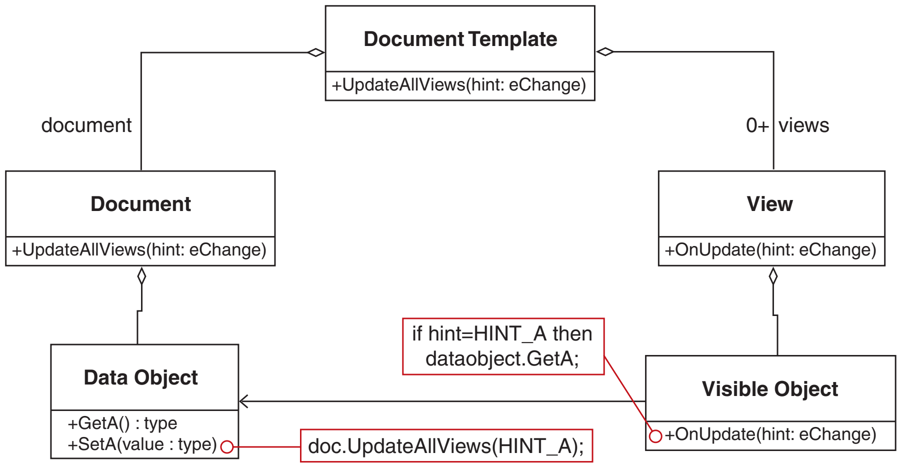


Figure 2: Document/view architecture.

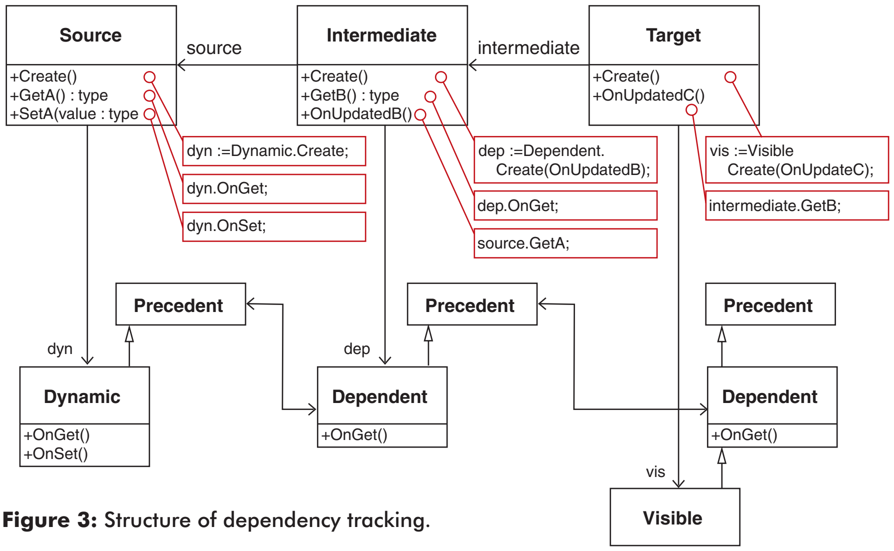
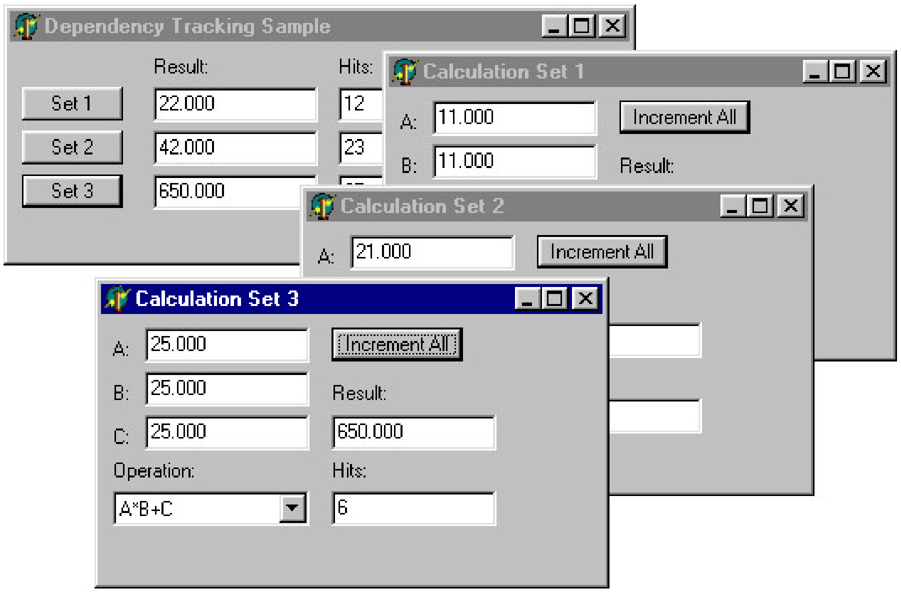


Figure 3: Structure of dependency tracking.


Dependency Tracking Sample

	Result:	Hits:
Set 1	22.000	12
Set 2	42.000	23
Set 3	650.000	

Calculation Set 1
A: 11.000
B: 11.000
Result:

Calculation Set 2
A: 21.000

Calculation Set 3
A: 25.000
B: 25.000
C: 25.000
Result: 650.000
Operation: A*B+C
Hits: 6

Figure 4: The demonstration application.

The ideal mechanism would be similar to a spreadsheet. The spreadsheet user simply enters a formula into a cell, and the program figures out when to calculate the value. It's easy to combine several simple calculations to form one complex system, yet a local change doesn't have a global impact. Of course, for the spreadsheet paradigm to work for complex software problems, we require an object-oriented, dependency-tracking framework.

The Goals of Dependency Tracking

Dependency tracking is a way of describing different kinds of state of a system and letting the framework take care of their relationships. Like a spreadsheet, it discovers the relationships among attributes, and determines the proper order of calculation. Unlike a spreadsheet, it does so in an object-oriented environment.

The key to dependency tracking is different kinds of state. Consider the checkbook problem to see examples of each: dynamic, dependent, and visible. The list of transactions is dynamic. Transactions can be added, deleted, or changed at any time. Dynamic state is the ultimate source of information in the system. The order of the transactions and running balances are dependent. They're determined via calculation dependent upon the dynamic attributes. Dependent state represents an intermediate point in a system's information flow. The view of the ledger is visible. It depends upon other attributes, and it directly affects what the user sees. Visible state is the ultimate destination of information in the system.

Dependency tracking uses these three kinds of state to discover relationships and determine the timing of calculations. This mechanism has the following advantages:

- **No registration.** All relationships are discovered automatically.
- **Dynamic relationships.** A dependent only responds to the attributes it depends upon at that moment.
- **No redundant updates.** A dependent responds only once to simultaneous changes.
- **Arbitrary order of destruction.** Dependency tracking will function no matter which object is destroyed first.
- **Simple maintenance.** The developer

simply identifies dynamic, dependent, and visible states; the system discovers their relationships.

Of course, this mechanism isn't applicable to every situation. Dependency tracking has the following disadvantages:

- **Slight performance hit.** It takes time and memory to discover and record dependency relationships.
- **Must be end-to-end.** Unless the developer identifies the dynamic state, its dependents will not be updated.
- **Doesn't work across thread or process boundaries.** For the framework to discover dependency relationships, the calculation can't rely upon other threads.
- **Single language.** Both end points of a dependency relationship must be implemented in the same language.
- **Data must be live.** Without the help of additional programming, dependency tracking can't discover relationships on database records, files, or other forms of persistent data.

The Application of Dependency Tracking

To identify dynamic, dependent, and visible attributes, an application pairs each one with a *sentry* object (see the sidebar "Glossary" on page 32). Sentries discover and maintain the dependency relationships among the attributes by forming connections among themselves. The structure diagram in Figure 3 illustrates this interconnection among sentries.

The application accompanying this article, DT, illustrates the power and ease of dependency tracking (this application is available for download; see end of article for details). DT contains a model with three calculation sets. From the main form (see Figure 4), the user can open a calculation form for any of these three. The main form also displays the result of each of the three calculations. On the calculation form, the user can enter three values and choose one of three operations. The form shows the result of the calculation.

```
  TCalculationSet = class(TObject)
  public
    ...
    { Access methods. }
    function GetA: Real;
    procedure SetA(value: Double);
    ...
  private
    { Attributes. }
    m_dA: Double;
    ...
    { Sentries. }
    m_dynA: TDTDynamic;
    ...
  end;
...
function TCalculationSet.GetA: Double;
begin
  { Notify the sentry. }
  m_dynA.OnGet;
  Result := m_dA;
end;

procedure TCalculationSet.SetA(value: Double);
begin
  if value <> m_dA then begin
    { Notify the sentry. }
    m_dynA.OnSet;
    m_dA := value;
  end;
end;
```

Figure 5: Declaration and use of a dynamic sentry.

This example illustrates several features of dependency tracking. The user can open multiple calculation forms based on the same set. When the user makes changes in one of these, the changes are reflected in all the others. This is the main goal of dependency tracking and the feature for which an application would use the mechanism.

The example also contains some features strictly to illustrate some additional benefits of dependency tracking. Each result keeps a hit count, which shows how often it's been calculated. Observing this hit count, the user can see that the result depends only upon the inputs used to calculate its current value. For instance, when the operation "A+B" is selected, changing C doesn't increment the hit count. Change the operation to "B+C" and notice that changing A now has no effect. This illustrates that dependencies are dynamic.

Finally, the calculation form features a button that simultaneously increments all three inputs. Observing the hit count, the user can see that the result is updated only once. Had we used the Observer pattern, the result would have been updated three times: one for each input change. This illustrates that dependency tracking avoids redundant updates.

Though these features of dependency tracking are exciting, the most valuable asset of this mechanism is its maintainability. The applica-

```
  TCalculationSet = class(TObject)
  public
    ...
    function GetResult: Double;
  private
    { Attributes. }
    ...
    m_dResult: Double;
    { Sentries. }
    ...
    m_depResult: TDTDependent;
    { Update procedure for the dependent attribute. }
    procedure OnUpdateResult;
  end;
...
constructor TCalculationSet.Create;
begin
  inherited;
  { Create the sentry objects. }
  ...
  m_depResult := TDTDependent.Create(OnUpdateResult);
  ...
end;
...
function TCalculationSet.GetResult: Double;
begin
  { Notify the sentry. }
  m_depResult.OnGet;
  Result := m_dResult;
end;

procedure TCalculationSet.OnUpdateResult;
var
  operation: TCalculationSetOperation;
begin
  operation := GetOperation;
  if operation <> nil then
    { Calculate the result. }
    m_dResult := operation.Calculate(Self)
  else
    m_dResult := -1.0;
end;
```

Figure 6: Declaration and use of a dependent sentry.

tion developer doesn't need to specify relationships between dependent and dynamic state, only to identify different types of state and let the system discover the relationships. Take a closer look at how the sample application does this.

To identify dynamic attributes, *TCalculationSet* declares a sentry and pair of access methods for each (see Figure 5). For instance, in addition to the numeric attribute *m_dA*, the class defines the sentry *m_dynA*. In accordance with proper encapsulation, the class also defines a pair of access methods. The *GetA* access method calls the sentry's *OnGet*, and *SetA* calls *OnSet*.

To identify a dependent attribute, *TCalculationSet* declares a sentry, an update procedure, and a single access method (see Figure 6). For the result of the calculation, the class defines both the numeric *m_dResult*, and the sentry *m_depResult*. The class further defines the procedure *OnUpdateResult* to calculate this attribute, and passes the procedure to the sentry via its constructor. Because the update procedure determines the attribute's value, the class declares only one access method, *GetResult*, which calls the sentry's *OnGet* method.

Visible attributes, being a special type of dependent attribute, receive similar treatment. To identify a visible attribute, *TMainForm* declares a sentry and an update procedure (see Figure 7). For example, it passes the update procedure *OnUpdateResult1* to the sentry *m_depResult1* via its constructor. Because visible attributes are rarely accessed by other classes, an access method isn't necessary. Instead, the update procedure puts its results directly into the *Text* property of a control, thereby making it visible to the user.

One final step is required to get dependency tracking working in this application. To bring visible attributes up-to-date, the application must periodically call *DTOnIdle*. *TMainForm* does this within the *AppOnIdle* procedure, which is assigned to the *Application.OnIdle* event (see Figure 8).

```
  TMainForm = class(TForm)
    ...
  private
    { Sentries. }
    ...
    m_visResult1: TDTVisible;
    ...
    { Update procedures for the results. }
    procedure OnUpdateResult1;
    ...
  end;
...
procedure TMainForm.FormCreate(Sender: TObject);
begin
  ...
  { Create the sentries. }
  m_visResult1 := TDTVisible.Create(OnUpdateResult1);
  ...
end;
...
procedure TMainForm.OnUpdateResult1;
begin
  { Update the edit box with the calculation results. }
  ebResult1.Text := FloatToStrF(
    m_Model.GetSet1.GetResult, ffFixed, 7, 3);
  { Increment the hit count (for illustration only). }
  Inc(m_nHits1);
  ebHits1.Text := IntToStr(m_nHits1);
end;
```

Figure 7: Declaration and use of a visible sentry.

As this example shows, the steps for implementing dependency tracking in an application are quite simple. More importantly, these steps are local, making maintenance much easier. For each dynamic, dependent, or visible attribute that a class contains, it must also include the appropriate kind of sentry. They maintain these sentries through access methods and update procedures, completely embracing the concept of encapsulation. The application doesn't need to consider the relationships among its various attributes; that is left to the dependency-tracking framework.

The Rules of Dependency Tracking

To discover dependency relationships, the framework recognizes three rules. These rules form the fundamental basis of the mechanism, and they are necessary and sufficient to solve the problem.

The first rule of dependency tracking is that a dependent attribute must be up-to-date whenever its value is needed. If no one cares

```
  TMainForm = class(TForm)
    ...
  private
    ...
    { The idle procedure. }
    procedure AppOnIdle(
      Sender: TObject; var Done: Boolean);
  end;
...
procedure TMainForm.FormCreate(Sender: TObject);
begin
  ...
  { Set up the idle procedure. }
  Application.OnIdle := AppOnIdle;
  ...
end;
...
procedure TMainForm.AppOnIdle(
  Sender: TObject; var Done: Boolean);
begin
  { Allow dependency system to perform idle processing. }
  DTOnIdle;
end;
```

Figure 8: Implementation of the *OnIdle* event.

```
constructor TDTVisible.Create(OnUpdate: TUpdateProcedure);
begin
  inherited;
  { Add visible dependents to the list. }
  g_lpdVisible.Add(Self);
end;
...
procedure DTOnIdle;
begin
  { Make all visible dependent attributes up to date. }
  ...
  g_nIndex := 0;
  while g_nIndex < g_lpdVisible.Count do begin
    TDTDependent(g_lpdVisible[g_nIndex]).MakeUpToDate;
    Inc(g_nIndex);
  end;
  ...
end;
...
procedure TDTDependent.OnGet;
begin
  MakeUpToDate;
  ...
end;
```

Figure 9: Implementation of the first rule. Dependents are brought up-to-date when necessary.

```
procedure TDTDependent.MakeUpToDate;
var
  pStack: TDTDependent;
begin
  { Check update status. }
  case m_eStatus of
    DT_UPDATING:
      OutputDebugString(
        'Cycle discovered during update.\n');
    DT_OUT_OF_DATE:
    begin
      { Push myself to the update stack. }
      pStack := g_pUpdate;
      g_pUpdate := Self;
      { Update the attribute. }
      m_eStatus := DT_UPDATING;
      try
        m_OnUpdate
      finally
        m_eStatus := DT_UP_TO_DATE;
        { Pop myself off the update stack. }
        Assert(g_pUpdate = Self);
        g_pUpdate := pStack;
      end;
    end;
    { DT_UP_TO_DATE: }
    { No action required. }
  end;
  Assert(m_eStatus = DT_UP_TO_DATE);
end;
```

Figure 10: The first half of the implementation of the second rule. A dependent sentry pushes itself onto a stack, and calls its update procedure.

what its value is, an attribute can simply remain out-of-date. However, when someone shows interest, the attribute must be recalculated. For non-visible, dependent attributes, this implies that the value is updated only when referenced. For visible attributes, however, the value must be updated regularly.

The second rule is that a dependent only depends upon the attributes used to calculate its current value. If an attribute wasn't referenced during the previous update, then it had no effect on the outcome, but the attributes that were referenced probably did. These attributes are called precedents. A precedent may be a dynamic attribute, or it may be another dependent.

The third rule is that a dynamic attribute, when changed, makes all of its direct and indirect dependents out-of-date. Those dependents will remain out-of-date until their value is required, as per the first rule. Direct dependents, as you may infer, are those that depend directly on the dynamic attribute. Indirect dependents are those that depend upon other dependents.

The implementation of the dependency-tracking framework follows directly from these three rules. In the absence of these rules, the code can be difficult to understand. However, when seen in context, the code falls neatly into place.

```
procedure TDTDynamic.OnGet;
begin
  { Establish dependency between the current update
    and this attribute. }
  RecordDependent;
end;
...
procedure TDTDependent.OnGet;
begin
  ...
  { Establish dependency between the current update
    and this attribute. }
  RecordDependent;
end;
...
procedure TDTPrecedent.RecordDependent;
begin
  { Get the active dependent. }
  { Verify that the link does not already exist. }
  if (g_pUpdate <> nil) and
     (m_lpdDependents.IndexOf(g_pUpdate) = -1) then begin
    { Establish a two-way link. }
    g_pUpdate.AddPrecedent(Self);
    m_lpdDependents.Add(g_pUpdate);
  end;
end;
```

Figure 11: The second half of the implementation of the second rule. When a precedent is referenced, it creates a link to the active dependent.

The Implementation of Dependency Tracking

The framework implements one sentry class each for dynamic, dependent, and visible state. Because visible state is a type of dynamic state, the visible sentry inherits much of its implementation from the dynamic sentry. Because a dependent attribute can depend upon any of the three types of attributes, all three share a common base class as precedents.

To implement the first rule, the framework responds when someone requires the value of a dependent attribute (see Figure 9). It maintains a list of visible dependents to be updated at idle time. *DTOnIdle* traverses this list, calling *TDTDependent.MakeUpToDate* for each member. Similarly, when an access method calls *TDTDependent.OnGet*, the framework again responds by calling *TDTDependent.MakeUpToDate*.

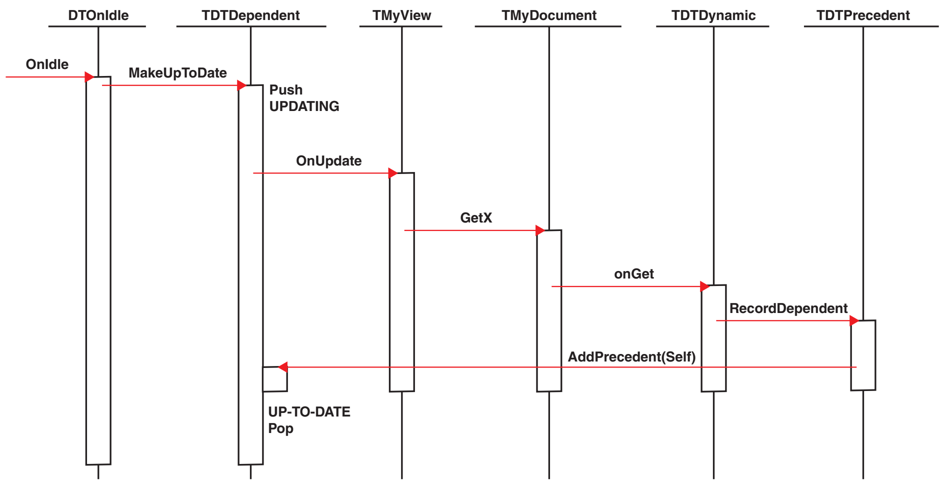


Figure 12: Interaction diagram of the implementation of the first and second rules. The system discovers the dependency relationship between a dependent attribute and its precedent, while the dependent is brought up-to-date.

```
procedure TDTDynamic.OnSet;
begin
  { When a dynamic attribute changes,
    its dependents become out-of-date. }
  MakeDependentsOutOfDate;
end;
...
procedure TDTPrecedent.MakeDependentsOutOfDate;
begin
  { When I make a dependent out-of-date, it will call
    RemoveDependent, thereby removing it from the list. }
  while m_lpdDependents.Count > 0 do
    TDTDependent(m_lpdDependents.First).MakeOutOfDate;
end;
...
procedure TDTDependent.MakeOutOfDate;
var
  nCount: Integer;
  nIndex: Integer;
begin
  Assert(m_eStatus = DT_UP_TO_DATE);
  { Tell all precedents to forget about me. }
  nCount := m_lpdPrecedents.Count;
  for nIndex := 0 to nCount-1 do
    TDTPrecedent(m_lpdPrecedents[nIndex]).
      RemoveDependent(Self);
  m_lpdPrecedents.Clear;
  { Make all indirect dependents out-of-date, too. }
  MakeDependentsOutOfDate;
  m_eStatus := DT_OUT_OF_DATE;
end;
```

Figure 13: Implementation of the third rule. When changed, a dynamic attribute makes its dependent out-of-date.

```
destructor TDTPrecedent.Destroy;
begin
  MakeDependentsOutOfDate;
  m_lpdDependents.Free;
  inherited;
end;
...
destructor TDTDependent.Destroy;
begin
  if m_eStatus = DT_UP_TO_DATE then
    { This will make all precedents forget about me. }
    MakeOutOfDate;
  Assert(m_eStatus = DT_OUT_OF_DATE);
  m_lpdPrecedents.Free;
  inherited;
end;
```

Figure 15: Implementation of sentry destructors. Dependents become out-of-date to disconnect the sentries.

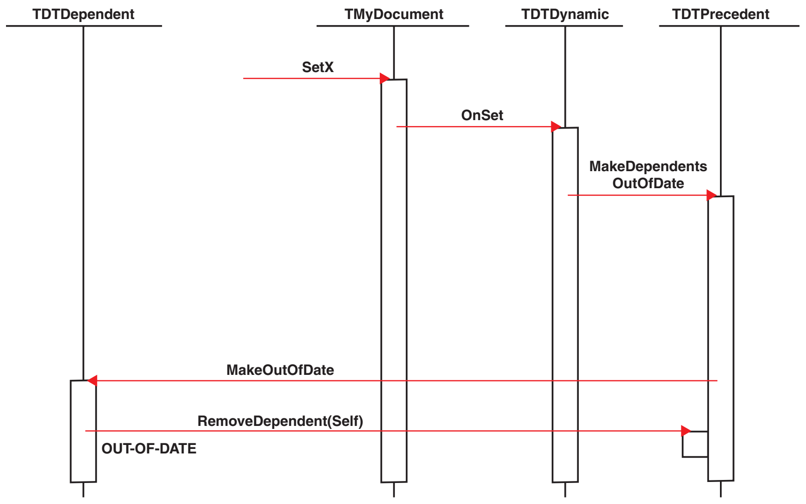


Figure 14: Interaction diagram of the implementation of the third rule. A change to a dynamic attribute signals its dependent attribute to become out-of-date.

To implement the second rule, the framework keeps track of the dependent currently being updated (see Figure 10). In *TDTDependent.MakeUpToDate*, the sentry pushes itself onto a stack, thereby establishing itself as the active sentry. It records its status as "updating" to detect cyclic dependencies, which would otherwise cause an infinite loop. It then calls the update procedure before popping itself off the stack.

The update procedure gathers information from other attributes in the system, ultimately calling their access methods. Each access method calls *TDTDynamic.OnGet* or *TDTDependent.OnGet*, each of which calls *TDTPrecedent.RecordDependent*. This method looks at the top of the stack, and establishes a two-way link between the active dependent and the precedent.

By the time the update procedure is finished, the new value has been calculated, and the framework has discovered and connected all precedents (see Figure 11).

The interaction diagram shown in Figure 12 illustrates the process flow that results.

To implement the third rule, the framework responds when a dynamic attribute is changed (see Figure 13). *TDTDynamic.OnSet* calls *TDTPrecedent.MakeDependentsOutOfDate*, which traverses the precedent's list of dependents, calling *TDTDependent.MakeOutOfDate* for each. This method breaks the two-way link between the dependent and each of its precedents, and then calls *TDTPrecedent.MakeDependentsOutOfDate* to do the same for indirect dependents. After the message has propagated through the dependency network, all direct and indirect dependents are out-of-date.

The interaction diagram shown in Figure 14 illustrates the implementation of the third rule.

When making a dependent out-of-date, it's essential that the framework disconnect it from its precedents. As per the second rule, the dependent will re-discover its precedents when it's brought up-to-date, so it's unnecessary to retain this information. Furthermore, because the set of precedents may change (as the example illustrates with "A+B" versus "B+C"), it's not even desirable to retain this information. Finally, discarding these connections prevents other precedents from redundantly making an out-of-date dependent out-of-date.

Taking advantage of disconnection, the framework makes dependents out-of-date in both *TDTPrecedent.Destroy* and *TDTDependent.Destroy*. When dependents are out-of-date, they're disconnected. When objects are disconnected, they can be destroyed in any order. Thus the framework eliminates one more potential problem area (see Figure 15).

Conclusion

Dependency tracking is a powerful mechanism for keeping a system up-to-date. Unlike the Observer pattern or document/view architecture, this mechanism requires only local code changes. The application developer simply identifies dynamic, dependent, and visible attributes and lets the framework discover the relationships among them. This localization embraces the concept of encapsulation and makes maintenance easier. Δ

The project referenced in this article is available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\AUG\DI200008MP.

Michael L. Perry formed Mallard Software Designs, Inc. to practice a new model of software development. He uses top-down design, patterns, and mathematical proofs to ensure the quality and maintainability of code. You can reach Michael at mperry@mallardsoft.com, or visit the Mallard Web site at http://www.mallardsoft.com.

Glossary

Dynamic attribute — An attribute whose value can be changed externally at any time.
Dependent attribute — An attribute whose value is determined by an internal update procedure.
Visible attribute — A dependent attribute that affects the user's view of the system.
Direct dependency — The situation in which a dependent or visible attribute depends upon a dynamic attribute with no intervention.
Indirect dependency — The situation in which a dependent or visible attribute depends upon another dependent attribute.
Cyclic dependency — The situation in which a dependent depends either directly or indirectly upon itself. Because many such situations are ambiguous, the dependency tracking system identifies them as errors.
Precedent — An attribute upon which a dependent or visible attribute directly depends.
Sentry — An object that performs a task on behalf of a sibling. In this context, dynamic dependent and visible sentries track dependency relationships on behalf of the attributes.
Update procedure — The procedure that gathers information and calculates the value of a dependent or visible attribute.
OnIdle — An event fired when the application's message queue is empty. This is a convenient time to bring visible attributes up-to-date.
Encapsulation — The object-oriented concept of hiding data or relationships within an object. Proper encapsulation reduces coupling and facilitates maintenance.
Access method — A public method used to get or set private data. Proper encapsulation requires the use of access methods.

By Deepak Shenoy

An Open Dialog

Using the Windows 2000 Open Dialog Box

Windows 2000 — the operating system we love to hate — is in full swing. Applications have had to conform to the new user interface and concepts that Windows 2000 introduces: COM+, Active Directory, Kerberos authentication, and more. I won't delve into these, however. I'll limit the discussion to the use of the Windows 2000 Open dialog box.

There's a marked change in the appearance of the Windows 2000 Open dialog box. Figure 1 shows the Windows 9x version. The sporty, new Windows 2000 version appears in Figure 2. The area on the left is known as the Places Bar; its buttons provide one-click access to select folders. If you use Office 2000, it should look all too familiar. In this article, we'll build a component to encapsulate its functionality, and make it convenient to use.

Calling the Open Dialog Box

The Open dialog box is a part of the Windows operating system. This means you don't have to create a form for this purpose; an API call is enough. This API function, *GetOpenFileName*, is implemented in Comdlg32.dll. In Delphi, you'll find the function defined in commdlg.pas. In Windows 9x, the function expects an argument of the type shown in Figure 3.

The Open dialog box is displayed when *GetOpenFileName* is called. Callbacks are sent to the hook procedure in *lpfnHook* (if present) when the user selects a file, changes a directory, loads the dialog box, etc.

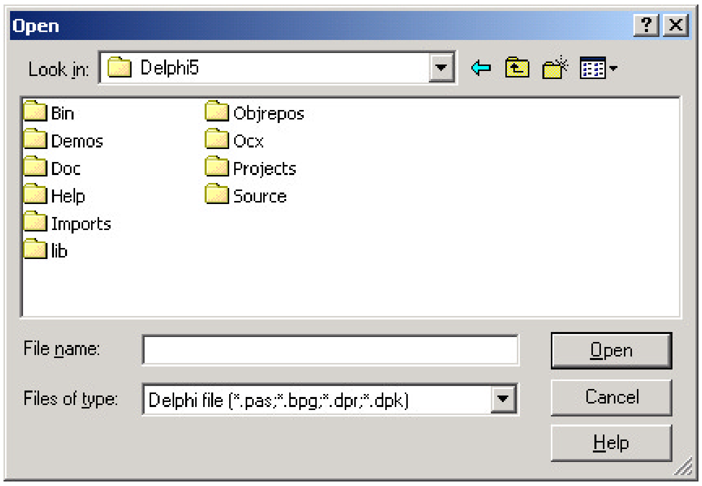


Figure 1: The Open dialog box of old.

Delphi hides this complexity in its OpenDialog component (*TOpenDialog* class). Among other functions, the component initializes the structure and handles the messages in the hook procedure. It also sets all necessary flags based on its property settings.

Windows 2000 introduces an extension to the argument's structure (see Figure 4).

The differences are minor. If *Flags* contains OFN_EX_NOPLACESBAR, the left bar won't appear; otherwise it will. The only other changes are to set the *lStructSize* member to the size of the extended structure, and to set the *FlagsEx* parameter.

Unfortunately, *TOpenDialog* cannot be customized for this purpose. We have no access to the actual structure being passed; we can only call *Execute*. The component then takes complete control, as it was designed to. There's another problem in the Commdlg.pas definition:

```
function GetOpenFileName(var OpenFile:
         TOpenFilename):
  Bool; stdcall;
```

The Windows API function expects a pointer to the structure *TOpenFileName*. Borland has defined the parameter as a **var**, which allows a programmer to pass a structure without using an @ operator, i.e. without passing a pointer to the structure. Unfortunately, with **var** parameters, the compiler expects you to pass the same structure, and complains when you try to pass anything else, including an extension. So we'll need to redefine the function as:

```
function GetOpenFileNameEx(var Open File: TOpenFilenameEx):
  Bool; stdcall;
...
implementation
...
function GetOpenFileNameEx; external 'comdlg32.dll'
  name 'GetOpenFileNameA';
```

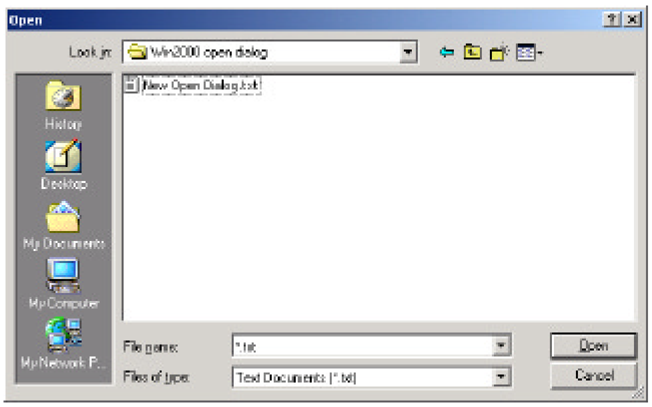


Figure 2: The Windows 2000 Open dialog box.

```
tagOFNA = packed record
  // Size of the structure in bytes.
  lStructSize: DWORD;
  // Handle that is the parent of the dialog.
  hWndOwner: HWND;
  // Application instance handle.
  hInstance: HINST;
  // String containing filter information.
  lpstrFilter: PAnsiChar;
  // Will hold the filter chosen by the user.
  lpstrCustomFilter: PAnsiChar;
  // Size of lpstrCustomFilter, in bytes.
  nMaxCustFilter: DWORD;
  // Index of the filter to be shown.
  nFilterIndex: DWORD;
  // File name to start with (and retrieve).
  lpstrFile: PAnsiChar;
  // Size of lpstrFile, in bytes.
  nMaxFile: DWORD;
  // File name without path will be returned.
  lpstrFileTitle: PAnsiChar;
  // Size of lpstrFileTitle, in bytes.
  nMaxFileTitle: DWORD;
  // Starting directory.
  lpstrInitialDir: PansiChar;
  // Title of the open dialog.
  lpstrTitle: PAnsiChar;
  // Controls user selection options.
  Flags: DWORD;
  // Offset of file name in filepath=lpstrFile.
  nFileOffset: Word;
  // Offset of extension in filepath=lpstrFile.
  nFileExtension: Word;
  // Default extension if no extension typed.
  lpstrDefExt: PAnsiChar;
  // Custom data to be passed to hook.
  lCustData: LPARAM;
  lpfnHook: function(Wnd: HWND; Msg: UINT; wParam: WPARAM;
    lParam: LPARAM): UINT stdcall;  // Hook.
  // Template dialog, if applicable.
  lpTemplateName: PAnsiChar;
end;

TOpenFilenameA = tagOFNA;
TOpenFilename = TOpenFilenameA;
```

Figure 3: An argument of this record type must be passed to the Windows 9x *GetOpenFileName* API function.

The New Component

TAgOpenDialog is the class we'll construct ("Ag" is short for Agni Software, the company I work for) to extend *TOpenDialog*. We won't write a completely new implementation. A new property, *ShowPlacesBar*, will control the display of the Places Bar on the left. The dialog box should display unchanged in Windows 9x and Windows NT 4.0. First, we'll define the component:

```
TAgOpenDialog = class(TOpenDialog)
protected
  FShowPlacesBar: Boolean;
public
  constructor Create(AOwner: TComponent); override;
  function Execute: Boolean; override;
published
  property ShowPlacesBar: Boolean
    read FShowPlacesBar write FShowPlacesBar;
end;
```

The *ShowPlacesBar* property controls the display of the Places Bar. Let's see how we can modify *TOpenDialog*. Looking at the source

```
TOpenFileNameEx = packed record
  // Size of the structure in bytes.
  lStructSize: DWORD;
  // Handle that is the parent of the dialog.
  hWndOwner: HWND;
  // Application instance handle.
  hInstance: HINST;
  // String containing filter information.
  lpstrFilter: PAnsiChar;
  // Will hold the filter chosen by the user.
  lpstrCustomFilter: PAnsiChar;
  // Size of lpstrCustomFilter, in bytes.
  nMaxCustFilter: DWORD;
  // Index of the filter to be shown.
  nFilterIndex: DWORD;
  // File name to start with (and retrieve).
  lpstrFile: PAnsiChar;
  // Size of lpstrFile, in bytes.
  nMaxFile: DWORD;
  // File name without path will be returned.
  lpstrFileTitle: PAnsiChar;
  // Size of lpstrFileTitle, in bytes.
  nMaxFileTitle: DWORD;
  // Starting directory.
  lpstrInitialDir: PansiChar;
  // Title of the open dialog.
  lpstrTitle: PAnsiChar;
  // Controls user selection options.
  Flags: DWORD;
  // Offset of file name in filepath=lpstrFile.
  nFileOffset: Word;
  // Offset of extension in filepath=lpstrFile.
  nFileExtension: Word;
  // Default extension if no extension typed.
  lpstrDefExt: PAnsiChar;
  // Custom data to be passed to hook.
  lCustData: LPARAM;
  lpfnHook: function(Wnd: HWND; Msg: UINT; wParam: WPARAM;
    lParam: LPARAM): UINT stdcall;  // Hook.
  // Template dialog, if applicable.
  lpTemplateName: PAnsiChar;
  // Extended structure starts here.
  pvReserved: Pointer;  // Reserved, use nil.
  dwReserved: DWORD;    // Reserved, use 0.
  FlagsEx: DWORD;       // Extended Flags.
end;

// FlagsEx of TopenFileNameEx.
const OFN_EX_NOPLACESBAR = 1;
```

Figure 4: This extended record must be passed to the Windows 2000 *GetOpenFileName* API function.

code in Dialogs.pas, we discover that *TOpenDialog* has a virtual *Execute* function that calls:

```
DoExecute(@GetOpenFileName);
```

And *DoExecute* (not virtual, unfortunately) builds the *TOpenFilename* structure, and passes it to *GetOpenFileName*. *DoExecute* takes a parameter, because the common File Save dialog box has exactly the same structure passed to it, except the Windows API function is named

```
var
  CurInstanceShowPlacesBar : Boolean;

// Global function.
function OpenInterceptor(var DialogData: TOpenFileName):
  Bool; stdcall;
var
  DialogDataEx : TOpenFileNameEx;
begin
  // Copy the structure to DialogDataEx.
  Move(DialogData, DialogDataEx, SizeOf(DialogData));
  if CurInstanceShowPlacesBar then
    DialogDataEx.FlagsEx := 0
  else
    DialogDataEx.FlagsEx := OFN_EX_NOPLACESBAR;
  // Set the new size.
  DialogDataEx.lStructSize := SizeOf(TOpenFileNameEx);
  Result := GetOpenFileNameEx(DialogDataEx);
end;

function TAgOpenDialog.Execute: Boolean;
begin
  if IsWin2000 then
    begin
      CurInstanceShowPlacesBar := FShowPlacesBar;
      Result := DoExecute(@OpenInterceptor);
    end
  else
    Result := inherited Execute;
end;
```

Figure 5: Calling *GetOpenFileNameEx*.

```
function TAgOpenDialog.IsWin2000: Boolean;
var
  ver : TOSVersionInfo;
begin
  Result := False;
  ver.dwOSVersionInfoSize := SizeOf(TOSVersionInfo);
  if not GetVersionEx(ver) then
    Exit;
  if (ver.dwPlatformId = VER_PLATFORM_WIN32_NT) then
    if (ver.dwMajorVersion >= 5) then
      Result := True;
end;
```

Figure 6: Testing if a user is running Windows 2000.

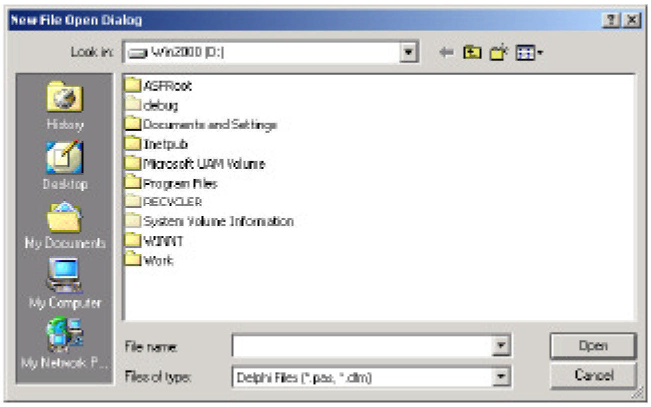


Figure 7: A sample output of the new dialog box.

GetSaveFileName. The *Execute* function in *TSaveDialog* (which is derived from *TOpenDialog*) calls *DoExecute*(*@GetSaveFileName*).

Here's where we step in and take control. We override *TOpenDialog.Execute* and pass a global function to *DoExecute*, which will now get the parameter that *GetOpenFileName* would have received. In the global function, we'll extend the structure and call *GetOpenFileNameEx* instead (see Figure 5).

There's a global variable declared in the **implementation** section that offers a little security. This variable, *CurInstanceShowPlacesBar*, is required because the global function, *OpenInterceptor*, has no way to access the current instance of *TAgOpenDialog*. We can't pass a member function to *DoExecute*. A global function is different from a class method in that there is no implicit *Self* parameter.

We also need to check if the user is running Windows 2000. The test is shown in Figure 6. We use *ver.dwMajorVersion >= 5* because we want to support further versions of Windows 2000. Include this component in a package and use it. See Figure 7 for a sample output. That's about it. The code accompanying this article contains a *TAgSaveDialog*, with some minor modifications.

Conclusion

This implementation isn't thread-safe. Because it uses global variables, you can't have multiple threads create *TAgOpenDialog* components and execute them without synchronized access to the global variable. Because the VCL isn't thread-safe, and it's recommended to have user-interface elements in the same thread, this doesn't sound like such a bad restriction (see *Effective COM: 50 Ways to Improve Your COM and MTS-Based Applications*, Don Box, ed. [Addison-Wesley, 1998]). The fact remains: Don't try to use this component across threads.

It would have been simpler to create this component had Delphi provided access to the structure being sent to *GetOpenFileName* — say through an event called just before calling *GetOpenFileName*. We wouldn't need to go through the rigmarole of the global function and variable. Most Delphi developers won't need it, but I would argue that access to the structure should be available for component developers.

You may use the *TAgOpenDialog* and *TAgSaveDialog* in your applications, commercial or otherwise. Tools like GExperts (http://www.gexperts.com) provide a way to replace components in an entire project, so you might want to convert *TOpenDialog* and *TSaveDialog* to *TAgOpenDialog* and *TAgSaveDialog*, respectively. This way your users with Windows 2000 will see the new Open dialog box, without you having to write a single line of code. Δ

Information regarding the Microsoft Platform Software Development Kit (SDK) is available at http://www.microsoft.com/msdownload/platformsdk/setuplauncher.htm.

The files accompanying this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\AUG\DI200008DS.

Deepak Shenoy is the technical director at Agni Software, a startup in Bangalore, India. Agni Software (http://www.agnisoft.com) builds products and provides services in Delphi, focusing on business solutions. Deepak has worked with Delphi since 1997, his experience largely in *n*-tier applications using Delphi and COM. Deepak has been working on emerging technologies such as ADSI, XML, and Windows 2000. You can contact Deepak at shenoy@agnisoft.com.

New & Used

By Bill Todd

Wise InstallMaster 8.0

A Do-it-all Installation Solution

If you need an installation tool that does it all, look no further than the Wise family of installation products. Wise Solutions offers three different products: InstallMaker 8.0, InstallBuilder 8.0, and InstallMaster 8.0. These products are at three different price points with three different feature sets, so you can pick the combination of features that meets your needs. All three products have the same user interface; only the features vary, so this review will focus on InstallMaster, the high-end product.

InstallMaster doesn't force you to choose between a fast, easy-to-use, wizard-based interface and a scripting language; both are provided. The interactive interface generates scripts that you can modify in the script editor, or you can write the entire installation using the scripting language if you need to.

When you start InstallMaster, you're greeted by the wizard interface shown in Figure 1. Wise breaks the process of creating your installation into the six steps listed across the top of the screen. As you click on each successive step, the list of options down the left side of the screen changes. You can use the **Next** and **Back** buttons to move through the screens in typical wizard fashion, or click directly on the screens you want to go to, bypassing the ones you don't need for the project you're working on. The **Steps** menu choice lets you customize the wizard by selecting the steps and screens within each step you will visit when using the **Next** and **Back** buttons.

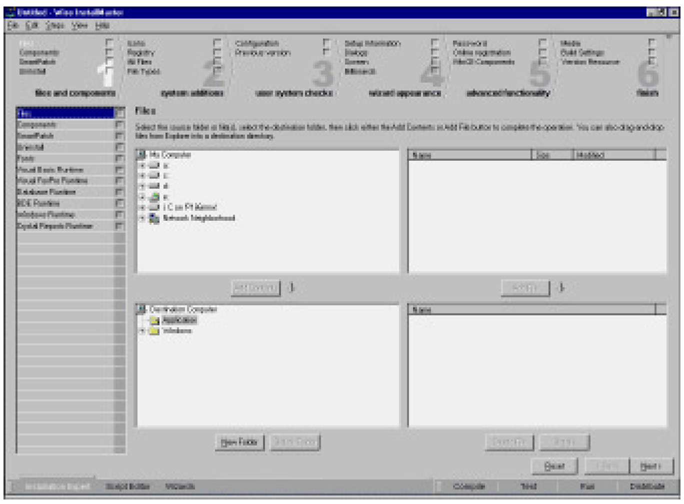


Figure 1: The InstallMaster user interface.

Choosing What Will Be Installed

The first option in step 1 is **Files**. Using this screen, you can choose which files will be installed on the target computer, and how they will be organized. The lower-left pane provides a tree view of the destination computer with two branches, named Application and Windows. The Application folder is the folder in which the user chooses to install your application. Windows represents the user's Windows directory (regardless of its actual name). You can create additional folders under either the Application or Windows folders to hold the files you are installing.

Once you've defined your folder structure, select the folder whose files you want to specify, then use the tree in the upper-left pane to navigate to the folder that contains the files on your computer. Select files you want to add to the selected destination folder, and click the **Add File** button. You can also select a folder in the upper-left pane, and click the **Add Contents** button to have the entire contents of the folder added to the selected destination folder, or added as a subfolder beneath the destination folder.

If you have optional files, you can divide your installation into multiple components. When you add components, you can specify whether that component will be installed by default. Each component you add has its own folder tree in the Files view. Figure 2 shows the Files view after two components, named Database and Tutorial, were added. This lets you define the files and folders associated with each component in a single screen. If you're installing an empty set of database files, you might want to make them a separate component, so the user can reinstall the program files without overwriting an existing database.

One of the great features of the Wise installation products is SmartPatch. Instead of creating an installation that contains all of the files in your application, you can create a self-installing patch file that will update an existing installation. This is an excellent way to distribute updates on diskette or via the Internet, because it reduces the size of the distribution file. Distributing patches also provides security, because the patch cannot be installed by anyone who doesn't have the prior version installed.

Creating a patch is easy. First, specify the files in the new version, just as you would to create a complete installation. Next, select SmartPatch and enter the directory that contains each of the prior versions you want to patch. Complete the installation settings as you would for a full installation, and InstallMaster will generate a patch EXE that will patch any of the prior versions you've specified.

You have the option of including uninstall capability, and will probably want to do so on your original installation disks. Install-

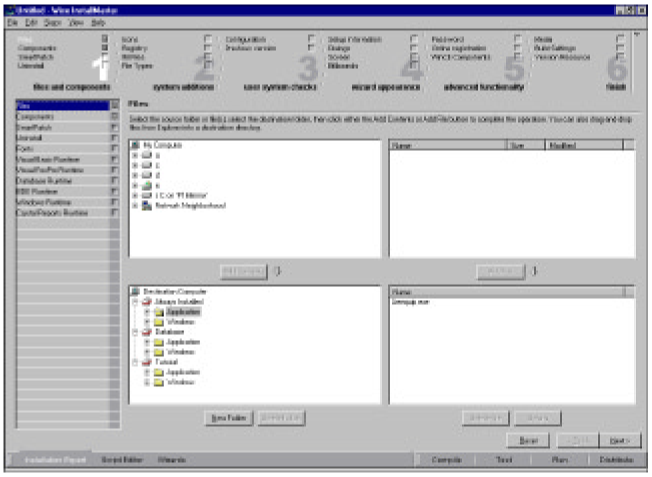


Figure 2: The Files screen with multiple components.

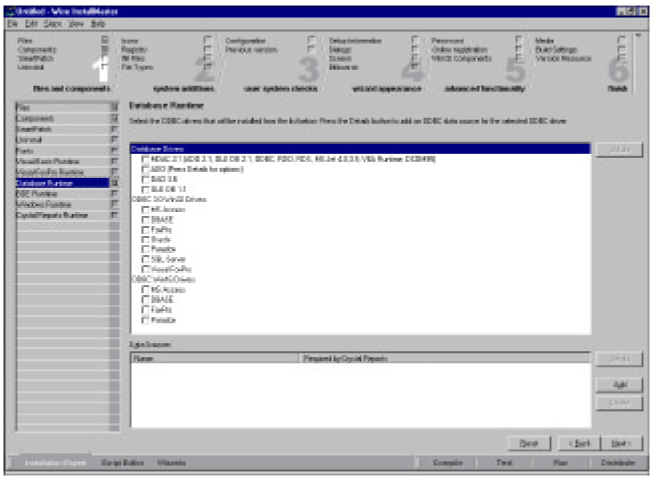


Figure 3: The Database Runtime page.

Master will generate the uninstall script for you. You can also add commands to remove registry entries or files that your application creates after installation. You can also elect to remove files that are in use.

Database and Run-time Engines

InstallMaster will also include Visual Basic Runtime, Visual FoxPro Runtime, the BDE, and Crystal Reports Runtime in your installation with the click of a mouse. The Windows Runtime option lets you include DirectX or MFC 4.2. InstallMaster also includes an option called Database Runtime, shown in Figure 3. This page allows you to include any of the Microsoft data access technologies in your installation. Options include MDAC 2.1, ADO, DAO 3.6, OLE DB 1.1, seven 32-bit ODBC drivers, and four 16-bit ODBC drivers. Use the lower pane to add any ODBC DSNs you need to create on the destination computer.

One feature that sets InstallMaster apart is its support for installing the BDE. It will install both the 16-bit and 32-bit BDE, and it will install the BDE in any location. It even has an option to let the user select the installation location. This is perfect for installing multi-user BDE applications, where you want to install the BDE on a file server, and have all of the clients share this installation. You can also install the BDE in one location, and the BDE configuration file in a different location. Another option lets you create BDE aliases without installing the BDE. This makes creating installations for machines where the BDE is already installed a snap.

InstallMaster also lets you change any setting in the BDE configuration file — even if the BDE is already installed on the target machine. To change BDE configurations settings, simply include the settings as parameters for any alias you create. The syntax is the same as that used for the BDE API function *DbiOpenCfgInfoList*. For example, to set Local Share and the Paradox driver's NetDir, use the following parameters:

```
\SYSTEM\INIT\LOCAL SHARE: TRUE
\DRIVERS\PARADOX\INIT\NET DIR: %MAINDIR%\BDENET
```

These lines will set Local Share to True and the NetDir to the BDENET directory under the user's main installation directory.

Operating System Setup

Move to step 2, system additions, and you can name the program group for your application, and list the files to appear as icons in the group. Choosing Registry provides a dual-pane view of the Windows registry on your computer, and the registry on the destination computer. You can create registry keys on the destination computer by entering them manually, by copying them from your registry, or by importing a registry file. You can also use variables, such as %MAINDIR%, to insert variable information, such as the path to the folder where your application will be installed.

The INI Files option allows you to create or update INI files. You can add, delete, and change sections and entries using fixed values or variables. The File Types option lets you define associations between file extensions and the programs that open them. There is also a Services option that allows you to install Windows NT services. Other options in step 2 let you add device drivers and modify autoexec.bat and config.sys. The last option lets you control whether an installation log is created, where it's located, and the name of the file. Because the installation log is used by the uninstall feature, you don't want to overwrite the original log

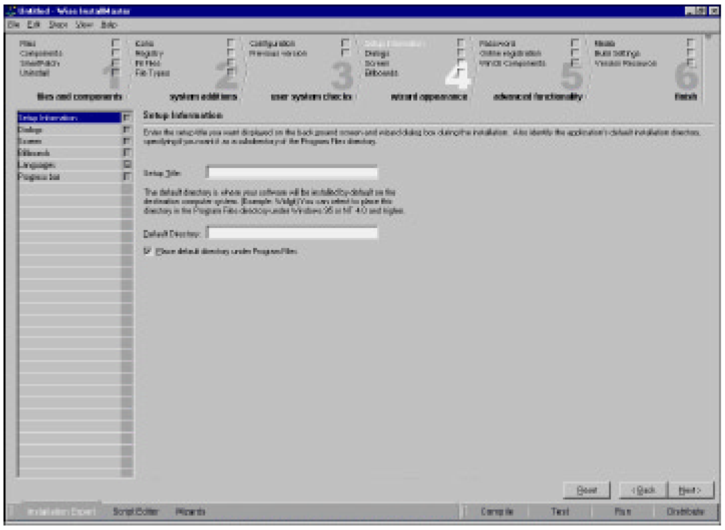


Figure 4: The Setup Information screen.

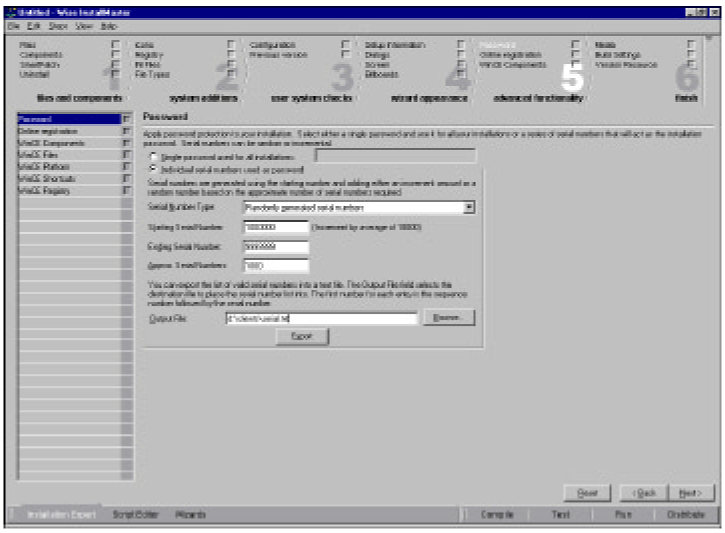


Figure 5: The Password screen.

when installing an update that only replaces a few files, because this would prevent uninstall from working correctly.

Will Your Application Run Here?

Step 3 in the installation process, user system checks, lets you define checks on the user's system, including the minimum Windows or Windows NT version, screen resolution, color depth, whether a sound card is installed, and whether WAV or MIDI support is installed. Each setting can be either required or recommended, and you can enter the message that will be displayed if the user's system doesn't meet the requirements. You can also require that a previous version of your application be installed. There are three ways to do this. First, you can search for one or more files. Second, you can check for one or more values in one or more INI files. Finally, you can check for one or more registry entries. You can also use any combination of these checks.

Controlling What the User Sees

The fourth step in creating your installation is called wizard appearance and begins by asking you for the title text to display while the installation is running, and the default directory name into which your files will be installed (see Figure 4). This is the directory that corresponds to Application in the Files screen in Figure 1.

However, you can only enter a directory name here, not a full path. That creates a problem if you create installations that must install to a specific directory on a corporate network. For example, there is no way, using the wizard, to set the default directory to K:\SALES\APPS\BUDGET. All you can do is enter a directory name, and, via a checkbox, choose whether that directory will be created in the root of C:, or in C:\Program Files. If you need to specify a full path, you must switch to the script editor and modify the script the wizard creates. This is an unfortunate shortcoming in an otherwise excellent product.

The Dialogs option in step 4 lets you select which of the standard dialog boxes the user will see during installation. Other options in step 4 let you set the screen colors and fonts, and define a list of billboard graphics that will be displayed during the installation.

Providing Security

Step 5 is titled advanced functionality and provides options to set passwords, enable online registration, and install Win CE components. The Password screen in Figure 5 offers two options for securing your application. The first is a simple password the user must enter during installation. The second is serial numbers used as a password.

If you choose serial numbers, once again there are two options. The first is a series of sequential numbers, any one of which will be accepted during installation. The second is a set of randomly generated numbers, any one of which will be accepted. For sequential serial numbers, you specify a beginning and an ending value. For random numbers, you specify a beginning and an ending value, and the number of serial numbers you want generated within that range. For example, you might set the starting value to 1000000 and the ending value to 9999999 and request 1000 random serial numbers in this range. While any one of the 1,000 serial numbers will work, the chance of someone guessing one of the 1,000 valid values out of nine million possibilities is quite small. If you use serial numbers, an export option lets you export the list of valid numbers to a text file that you can use to print labels or import into a database.

You can use the Online registration option by setting up a Web site to accept HTTP Post commands containing registration information in a predefined set of fields. By entering the name of an INI file, you can also store the registration information on the user's system for future use. For example, if the user purchases an update, the registration screen in the update installation will automatically be filled in using the information from the INI file created by the original installation.

Distribution Options

The sixth and final step controls the distribution settings, beginning with media options. If you will distribute your application via the Internet or a corporate intranet, or if users will install from a file server, choose Single File Installation. InstallMaster will create a single EXE file containing your entire application. To distribute your application on media, choose Media Based Installation and select the type of media. InstallMaster supports 5.25" floppy disks, 3.5" 720MB and 1.44MB diskettes, 100MB Zip disks, 120MB LS-120 disks, CD-ROM, DVD-ROM, and custom sizes. If you choose custom, you can enter any media size.

Choose **Build Settings** to set a variety of options, including whether you are deploying to the Windows 3.1 or Windows 95/98/NT platforms. Perhaps the most important setting on this screen is the **Maximum Compression** checkbox. This option is unchecked by default to reduce the time required to compile your installation during development and testing. You should always check this option before the final build to make the installation files as small as possible. InstallMaster supports silent installations by starting the installation EXE with the /s command line switch. Selecting **No Reboot Message During Silent Installs** will make a silent installation completely silent even if a reboot is required. **Replace In-Use Files** controls whether a reboot will be used to free files that are in use when the installation is run. You can also choose to have the compressed installation files in Zip-compatible format, specify the name for the installation EXE file, enter the location of the icon file for the installation EXE, supply the path to any custom dialog boxes you created with the InstallMaster dialog editor, and select a temporary directory other than the Windows \temp directory to be used when creating your installation.

If you plan to distribute your application over the Internet or a company intranet, InstallMaster offers another approach, called WebDeploy. Using WebDeploy, the user downloads and runs a small program that checks the user's configuration, determines which files need to be downloaded from the server, downloads them, and completes the installation. WebDeploy works well with unstable network connections, such as dial-up lines, because it will automatically resume an interrupted installation without having to restart the download from the beginning.

WiseUpdate is a companion technology that lets you install a small update program with your application. Based on your settings, this program will periodically offer the user the option to check for updates on the Internet or your intranet. If an update is found, WiseUpdate downloads and installs it.

Other Ways to Create an Installation

If you want to create your installation the easy way, InstallMaster offers two wizards that will do just that. The Run Application and Watch For Loaded Files wizard does exactly what its name implies. You start the wizard, run your application, make sure you use all of the features that will load additional files, such as DLLs or ActiveX controls, then let the wizard build an installation that will install your application's EXE and all of the files it used.

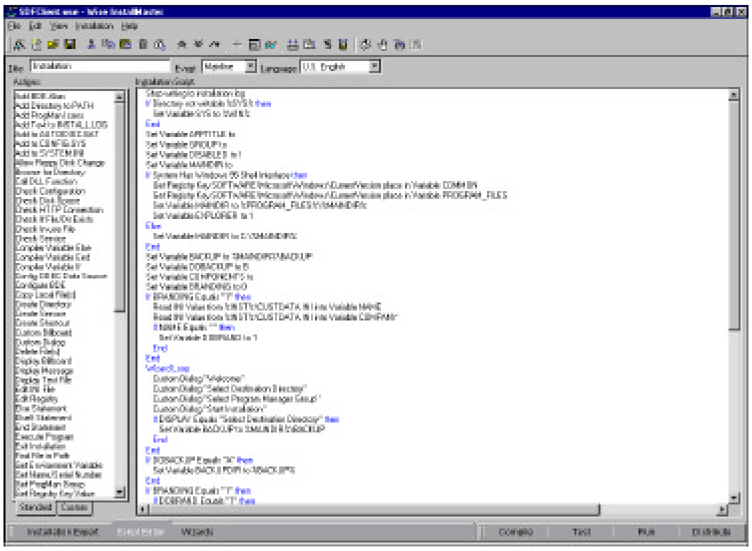
The second wizard is the Setup Capture wizard. Setup Capture creates a snapshot of your system before and after an application has been installed and creates a script to replicate the installation, including registry entries. Another time-saving feature is the ability to create installation templates. If you find yourself setting the same options in many of the installations you create, start with a blank project, change the settings that are common to many installations, then save the installation script to the Templates directory. From then on, your template will be listed in the New Installation dialog box that appears when you choose **File | New** from the main menu.

If you need installation features that you cannot get using the installation wizard, you can use the script editor to modify a script created by the wizard, or write your own from scratch. The script editor provides an integrated debugger and a dialog box editor for creating custom dialog boxes. One of the unique features of the script editor, shown in Figure 6, is that you don't need to type code as you would in a traditional programming environment. Instead, the script commands are listed in the left pane of the editor, and you add them to your script by drag-and-drop. When you drop a new command on your script, a dialog box opens automatically to prompt you for any variable information the command requires. This approach works well and virtually eliminates syntax and typographical errors. The scripting language also lets you call any Windows API function, call functions in your own DLL, or run an executable as part of your installation.

If you distribute applications internationally, InstallMaster provides multi-language support, including support for multi-byte character sets. SMS support is also included if you use Microsoft's System Management Server to manage a large number of PCs.

Conclusion

InstallMaster 8.0 does it all. Whether you're distributing commercial or corporate applications over a LAN,



```
Add BDE Alias
Add Directory to PATH
Add ProgMan Icons
Add Text to INSTALL.LOG
Add to AUTOEXEC.BAT
Add to CONFIG.SYS
Add to SYSTEM.INI
Allow Floppy Disk Change
Browse for Directory
Call DLL Function
Check Configuration
Check Disk Space
Check HTTP Connection
Check If File/Dir Exists
Check In-use File
Check Service
Compiler Variable Else
Compiler Variable End
Compiler Variable If
Config ODBC Data Source
Configure DDE
Copy Local File(s)
Create Directory
Create Service
Create Shortcut
Custom Billboard
Custom Dialog
Delete File(s)
Display Billboard
Display Message
Display Text File
Edit INI File
Edit Registry
Else Statement
ElseIf Statement
End Statement
Execute Program
Exit Installation
Find File in Path
Get Environment Variable
Get Name/Serial Number
Get ProgMan Group
Get Registry Key Value
```

```
[illegible] installation log
If Directory exists/variable %SYS% then
  Set Variable SYS to %WIN%
End
Set Variable APPTITLE to
Set Variable GROUP to
Set Variable DISABLED to !
Set Variable MAINDIR to
If System Has Windows 95 Shell Interface then
  Get Registry Key SOFTWARE\Microsoft\Windows\CurrentVersion place in Variable COMMON
  Get Registry Key SOFTWARE\Microsoft\Windows\CurrentVersion place in Variable PROGRAM_FILES
  Set Variable MAINDIR to %PROGRAM_FILES%\%MAINDIR%
  Set Variable EXPLORER to 1
Else
  Set Variable MAINDIR to C:\%MAINDIR%
End
Set Variable BACKUP to %MAINDIR%\BACKUP
Set Variable DOBACKUP to B
Set Variable COMPONENTS to
Set Variable BRANDING to 0
If BRANDING Equals "1" then
  Read INI Value from %INST%\CUSTDATA.INI into Variable NAME
  Read INI Value from %INST%\CUSTDATA.INI into Variable COMPANY
  If NAME Equals "" then
    Set Variable DOBRAND to 1
  End
End
Wizard Loop
  Custom Dialog "Welcome"
  Custom Dialog "Select Destination Directory"
  Custom Dialog "Select Program Manager Group"
  Custom Dialog "Start Installation"
  If DISPLAY Equals "Select Destination Directory" then
    Set Variable BACKUP to %MAINDIR%\BACKUP
  End
End
If DOBACKUP Equals "A" then
  Set Variable BACKUPDIR to %BACKUP%
End
If BRANDING Equals "1" then
  If DOBRAND Equals "1" then
```

Figure 6: The script editor.

Informant Fact File

InstallMaster 8.0 does it all. Whether you are distributing commercial or corporate applications over a LAN, WAN, the Internet, or on media, InstallMaster 8.0 provides the right combination of distribution and update features to meet your needs. No matter how large or small your applications are, this is the only installation program you'll need.

Wise Solutions, Inc.
5880 N. Canton Center Rd., Suite 450
Canton, MI 48187

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

WAN, the Internet, or on media, InstallMaster provides the right combination of distribution and update features to meet your needs. The general rule in software is that the more powerful the product the more difficult it is to use, but InstallMaster is the exception. I can do 99 percent of my installations using the installation expert alone. When I do have to modify an installation in the script editor, the list of commands, combined with drag-and-drop editing, makes it easy to enhance a script and get it right the first time. No matter how large or small your applications are, this is the only installation program you'll need. Δ

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, including *Delphi: A Developer's Guide*. He is a Contributing Editor to *Delphi Informant Magazine*, and a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a member of Team Borland, a nationally known trainer, and has taught Delphi programming classes across the country and overseas. He can be reached at bill@dbginc.com.

Delphi in a Nutshell

I always think of O'Reilly's *...in a Nutshell* books as the motorcycle repair books of the software world. These books provide clear, to-the-point, and in-depth reference information. If you're wondering about the motorcycle repair book reference, you haven't had the pleasure (or misfortune?) of owning a motorcycle. Publishers like Clymer offer books for your specific machine that provide in-depth, step-by-step procedures for many of the common maintenance and repair tasks you will need to perform.

In the past, O'Reilly has concentrated on subjects like Unix, C/C++, Java, Windows NT, and Web technologies — so seeing the first Delphi-related *...in a Nutshell* book is cause for celebration. I was even more excited to find that Ray Lischner is the author. With the possible exception of the original beta manuals for Delphi 1, I consider Lischner's *Secrets of Delphi 2* [Waite Group Press, 1996] the most inspiring Delphi book I ever read. After going through that tome, I got a much better understanding of how Delphi handles forms, objects, and the like, and I was able to use an architecture in two major applications that resembles the way Delphi handles its components and deals with properties.

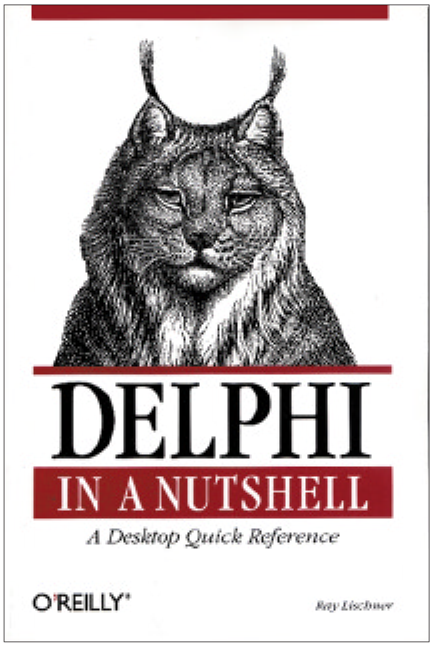
Before you run out to purchase *Delphi in a Nutshell*, however, you must know that it doesn't discuss the VCL that is a large part of what makes Delphi ... well ... Delphi. It's a disappointment that this information is excluded. However, once you start going through all the information, it becomes clear that a comparable *VCL in a Nutshell* would probably be a 3,000-page book (or more). As it is, *Delphi in a Nutshell* — which discusses "only" the Object Pascal language, compiler-related information, and the system libraries — is over 550 pages.

The Clymer manual that came with my Kawasaki EX500 spends no less than four pages discussing the various designs and uses of nuts, bolts, washers, screws, snap rings, and the tools you need to deal with them. Likewise, *Delphi in a Nutshell* starts with a chapter that discusses Delphi Pascal. The nuts and bolts of the language are described in detail, including units, programs, packages, data types, variables, I/O, exception handling, and the like.

The second chapter discusses the Delphi Object Model; don't confuse the name of this chapter with the VCL hierarchy. The chapter discusses classes and objects (including the VMT structures, dynamic method calling, etc.), interfaces, the life-cycle of objects, reference counting, messages, and memory management. Although it seems the information to this point targets novice Delphi programmers, the reality is that there is a lot of in-depth information of which you are probably not aware unless you took the time to dissect the system units as Lischner has. For example, when discussing memory management, the author describes how to override the *NewInstance* system procedure to create alternate memory allocation algorithms, and points to some interesting behavior of components (or forms) when freeing "owned" components that are referenced in the **published** section.

If the first two chapters provide in-depth information about subjects with which many of us are familiar, Chapter 3 dissects and explains a subject that usually *isn't* discussed: RTTI. The structure of a VMT is examined, followed by detailed information of what kind of information is available for the different things you can declare in the **published** section of a class. The undocumented typinfo.pas unit is explained as well. The chapter also discusses VMT vs. DMT tables, **initialization** and **finalization**, and interfaces.

Chapter 4 is the last "non-reference" chapter in the book. And what a chapter it is! Lischner discusses concurrent programming, the basics



of processes and threads are introduced, and the common problems of multi-threaded programming (race conditions and deadlocks) are discussed, with simple suggestions for ways to overcome these issues. Elements like a mutex, semaphore, critical sections, synchronization, and the like are introduced.

Next, Delphi's *TThread* class is introduced, and a discussion about thread local storage and inter-process communication is included. The chapter ends with the introduction of a *Futures* class that can be used to simplify concurrent programming. Of all the Delphi books and articles I've seen, this chapter provides the most complete and useful information about concurrent programming. For my needs, this chapter alone is worth the price of admission.

The reference chapters include a long chapter devoted to the language, including every keyword, class, function, and variable supported in the "system" RTL (including system.pas and sysinit.pas). Every entry in the chapter includes a "Syntax," "Description," and "See Also" section, and when appropriate, an "Example" and "Tips and Tricks" section. For example, the *TextFile* type entry (used to define a text file in Delphi) includes a tips-and-tricks section that shows the internal structure of the type, and discusses the issue of writing a custom text file driver. The example writes such a driver that maps a stream to a text file and allows you to *writeln* and *readln* from any *TStream* — be it in memory, disk, or elsewhere — as if it were a standard text file.

Additional reference chapters cover system constants, operators, and compiler directives, as well as an appendix that documents the command-line tools, including dcc32, brcc32, the DFM converter, tdump, the object file dumper, and the IDE itself. Surprisingly, tlibimp (the most important command-line tool in my opinion) isn't mentioned.

The last appendix documents the sysutils.pas unit. This is not an integral part of the system RTL, but it's an important addition used by almost every Delphi application.

This book is a great reference, with some excellent non-reference chapters. For advanced Delphi developers, the chapters about RTTI and concurrent programming provide important information that is hard to come by. For programmers moving to Delphi after stints as C, C++, or Java programmers (and there will probably be a nice amount of those when Delphi's Linux version appears), Lischner includes comparisons between elements in these languages and their Delphi counterparts.

My gripes are small and insignificant. I would have loved to see the same categorical separation of functions in the language reference chapter that is used for the sysutils.pas appendix and, although a bit out of the scope of the book, a chapter about the classes.pas unit would have been appreciated.

This book is another masterful tome from Lischner. It includes so much in-depth, undocumented information, that it earns a place of honor next to the all-time great Delphi books every advanced programmer should own.

— Ron Loewy

Delphi in a Nutshell by Ray Lischner, O'Reilly & Associates, Inc., 101 Morris St., Sebastopol, CA 95472, http://www.oreilly.com.

ISBN: 1-56592-659-5
Price: US$24.95 (561 pages)

table of CONTENTS

Standards and Conventions

Coders love to code. The profession is difficult enough that those who try to become programmers simply because it's a "good job" usually don't last long. They switch to basket weaving, lion taming, politics, or something else requiring a little less skill, courage, or commitment. You have to really love coding to invest the necessary time and energy to program well, as well as endure the inevitable aggravation that comes with the learning curve.

For this reason, when coders confront a challenging problem, we tend to jump right in and gleefully figure out our own "solution." Many times, however, we end up spending hours and hours on an algorithm that has already been written, i.e. "re-inventing the wheel."

Although educational, and possibly fun, to adopt the "Not Invented Here" syndrome and make sure that every bit of code is completely original (as if our needs are completely unique and nobody before in the history of programming ever had to write a procedure or function addressing them), it boils down to a colossal waste of time if the algorithm already exists, has been tested, and is available for the taking. Adapting existing code for one's own use has been called code reuse of the highest order.

In accounting circles, some people use the acronym OPM (Other People's Money), which they like to use to make money for themselves. In the programming arena, we should adopt a similar fondness for using OPC (Other People's Code).

For example, you may need to write a sorting algorithm. Should you write it yourself "from scratch" so to speak, or see if you can use someone else's code? The answer depends on whether you're coding for coding's sake, or if your focus is to get the work done in the most efficient manner. I've written several programs using string grids. In many cases I wanted to allow the user to sort by any column, ascending or descending. I suppose I could have written the sorting algorithm myself, but as I was more interested in getting the program done than delving into the intricacies of sorting, I searched on the newsgroups for sorting algorithms. It didn't take me long at all to find one that somebody said they had "slapped together" (some Delphi guru, obviously) that I pasted into my program, modified a little to fit my particular situation, and *voilà*! Who knows how much time it saved me? Time I could then use designing my next software masterpiece!

Besides searching the newsgroups for algorithms, there are many Delphi publications full of example code, as well as numerous Web sites devoted to Delphi, with gobs of Delphi code samples. Both *Delphi Informant Magazine* (http://www.DelphiZine.com) and *The Delphi Magazine* (http://www.itecuk.com/delmag/index.htm) are chock full of good code. There are also dozens of Delphi books full of example code (search for "Delphi" on Amazon.com [or ComputerBookstore.com]). For Delphi-centric Web sites, go to http://www.undu.com (an electronic Delphi magazine in its own right) and follow its **Delphi Links** link.

Another benefit of using existing code, especially from "official" sources such as publications, is that the code usually adheres to Delphi/Pascal conventions. What's the benefit of that? Isn't programming about individuality, freedom, the right to code things however you want (as long as it works)? Well ... if you perform an operation that's normally done another way by the Delphi community, or name a variable something unusual, it will be confusing to other programmers who must maintain your code.

Of course, if you're writing programs for yourself, and are certain that nobody else will ever see the code, you're welcome to code them any way you want. Even then, you can do yourself a favor by following standards and conventions — even if they're your own conventions.

For example, Object Pascal keywords should be in lowercase, e.g. **for**, **begin**, **with**, **if**, etc., and class names should begin with an uppercase T. And you should follow some type of indenting rule: two spaces (not a tab) is standard. A good source of Delphi/Pascal conventions (other than the Delphi source code itself) is Chapter 6, "Coding Standards Document," in *Delphi 5 Developer's Guide* by Steve Teixeira and Xavier Pacheco [SAMS, 2000]. It's available online at http://www.xapware.com/ddg.

It is also very helpful to adopt a component-naming convention. If you name one button *ButtonClose*, another *SaveButton*, another as the default *Button1*, and another as *Post*, it makes it more difficult — for one thing — to locate your various buttons in the Object Inspector. The January, 1997 issue of *Delphi Informant Magazine* contained an article by Mark Ostroff, "What's in a Name?" on component-naming conventions. If everyone were to follow these suggestions, it would help greatly in seeing at a glance what type of component is being referred to in code. As an example of what *not* to do, I have seen a *TQuery* component that was given the name *QueryUpdate*. And it wasn't an UpdateQuery component!

Will this stifle your need for expression through code? Not at all; following conventions will simply help you apply your brainpower and creativity to the truly unique aspects of your project. The "standard" parts will be easily recognizable, and require less of your attention. Unorthodox coding practices only confuse and aggravate.

The more often standards and conventions are adhered to, the easier it will be for the Delphi programming community at large to understand — and yes, reuse — each other's code. This will make all of us more productive and successful in our endeavors to be the world's greatest community of software developers, using the world's greatest software development tool. Δ

— *Clay Shannon*

Clay Shannon is a Delphi developer for eMake, Inc. in Post Falls, ID. Having visited 49 states (all but Hawaii) and lived in seven, he and his family have finally settled in northern Idaho, near beautiful Lake Coeur d'Alene. The only spuds he has seen in Idaho have been in the grocery, and most of those are from Oregon and Washington. Clay has been working (almost) exclusively with Delphi since the release of version 1, and is the author of *Developer's Guide to Delphi Troubleshooting* [Wordware, 1999]. You can reach him at BClayShannon@aol.com.

Open Source and the Delphi Community

Open source. No doubt the expression brings a sense of excitement and anticipation to some, fear and loathing to others. But what does open source mean? In the broadest sense, it could be interpreted as software whose source code is "open" or generally available for study. However, the various interpretations of what open source means in the Linux community (and elsewhere) have led to different understandings, definitions, and licenses. We'll begin with a brief history of the open-source movement, review some of the licensing issues, and consider the implications for Delphi.

The Nature and Development of Open Source

As a means of developing and distributing software, it has been around for a long time — much longer than the expression "open source." In the early days of the Internet, the idea of "free software" was popular among developers. That spirit has continued to this day, with developers publishing the code to their software, and people sharing code in newsgroups, list servers, and on message boards. The Linux movement (see my three recent columns on Delphi and Linux), based on an operating system that was completely open source from the beginning, was a great catalyst for the movement.

A specific Open Source Initiative was organized in 1998. Their Web site (http://OpenSource.org) presents an open-source philosophy, some history, a particular open-source specification, and more. This approach is closely aligned to one of the major licenses, GNU, which we'll discuss presently.

Not surprisingly, OpenSource.org views open source as the future of software development, an approach "whose time has finally come." In tracing its history, they point out that "for twenty years [the open source approach] has been building momentum in the technical cultures that built the Internet and the World Wide Web."

But what is open source all about? The following argument from OpenSource.org explains how open source works to produce more robust software in less time: "The basic idea behind open source is very simple. When programmers on the Internet can read, redistribute, and modify the source for a piece of software, it evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing."

A major exposition of open-source philosophy, "The Cathedral and the Bazaar" by Eric S. Raymond (http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar.txt), includes this argument in support of open source: "Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone." He goes on to restate it in less technical language: "Given enough eyeballs, all bugs are shallow." This seminal essay by one of the prime movers in the open-source movement is a must-read for anyone who wants to get a good sense of how the movement evolved, as well as its major strengths.

Of Licenses, Left and Right

As in the world of politics, the open-source movement has its left wing and right wing, both of whom would claim an exclusive understanding of the word "freedom." We'll examine two of the popular licenses: the GNU General Public License (GPL) and the Mozilla Public License (MPL). Going back to the late 1980s, the former license is quite popular with many in the open-source movement, particularly those connected with Linux development. A colleague of mine in Project JEDI described this type of licensing in these words: "GPL is 'copyleft,' which means if you compile GNU-licensed code in a project, all other code in the project must be GNU-licensed also. So GNU-licensed code is not really an option for, say, a software house that sells the binaries at one price and the binaries plus [the] source at another (higher) price. Nor is it an option if you want to use the GPL code but don't release your entire source code to GPL." She concluded by informing me that "this is known on the boards as 'viral licensing.'" You can read a critique of GPL at http://www.gnu.org/copyleft/gpl.html. When I examined the actual GPL license itself, I found the following definition of the source code that you must redistribute if you use any source code that is GPL-licensed: "For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable."

The implications for Delphi developers are obvious. In the most literal interpretation, you would be required to make available the source code to portions of the Visual Component Library (VCL) you used in an application. This would be patently illegal, unless, of course, Inprise were to release the VCL under GNU! While

GPL-licensed software would be problematic for Delphi developers, there are other more appropriate licenses. MPL, which we decided to use in Project JEDI, is a good example of a less restrictive approach. With MPL you can mix open source and non-open source code in the same project. Compare the following excerpt from MPL to the previous GPL statement: "Any Modification which You create or to which You contribute must be made available in Source Code form under the terms of this License either on the same media as an Executable version or via an accepted Electronic Distribution Mechanism to anyone to whom you made an Executable version available." So, you're only required to make available the code that you write.

Open Source in Delphi and Commercial Software

Netscape was the major force in developing MPL. Its decision to release the code to its browser became a major catalyst for the open-source movement. Of course, not everyone was overjoyed by the move to promote an open approach to software development. You might be wondering how Microsoft, for example, reacted to all of this. To satisfy your curiosity, I suggest you take a look at the "Halloween Papers" on the OpenSource.org Web site. These articles, written by Eric S. Raymond, are based on internal memos leaked by employees at Microsoft. Other companies, like Inprise, have shown a genuine interest in the open-source approach. Some, but not all, versions of Delphi include the source code for the VCL. Late last year, after a period of confusion and concern, Inprise announced that it would release InterBase to open source. And there are developments in the larger Delphi community.

Delphree: The Delphi Connection

The central location for Delphi open-source projects is Delphree (http://delphree.clexpert.com/pages/initiative.htm), where you can find information and links to the major Delphi open-source initiatives. Included are some of the Pascal links I've discussed in past columns, such as the Free Pascal Compiler and the Lazarus Project. Delphree is a close partner with Project JEDI. It has links to many Project JEDI projects, including the JEDI Component Creator, the JEDI Dolphin educational project, and the JEDI Program Editor. There are also links to JEDI's API translation projects, its utility library, and its component projects.

Delphree also has links to several well-known Delphi open-source initiatives, including Gerald Nunn's GExperts (http://www.gexperts.org) and Chad Hower's Winshoes (http://www.pbe.com/Winshoes/). Both products (projects is probably a better term) have had interesting histories. Winshoes is an Internet library that was started as shareware in 1995. It later became part of two different commercial libraries, and was finally released as freeware and an open-source project in 1998.

GExperts is a series of Delphi IDE enhancers and programming tools. Developers are encouraged to download the source code, submit bug reports or fixes, and create new features for inclusion in the GExperts distribution. In 1998, Gerald turned the project over to Erik Berry, allowing him to make it open source to speed development, and further enhance the quality of its experts. Gerald shared the following observations with me. To be successful he felt that an open-source project should have the following characteristics:
1) possess an existing, working code base;
2) possess a strong, capable administrator;
3) not be too large in scope; and,
4) have a target market that includes developers.

He was quite candid in his assessment of open source, indicating that he thought it was "more hype than reality," and that he didn't think it would "supplant commercial software development any time soon."

While the two Delphi open-source projects I mentioned have been around for a while, others are in an early stage of development, so I expect to revisit this topic. In the meantime, I'd like to hear from readers who have been involved with open-source projects, and hear your views and experiences. Until next time ... Δ

— *Alan C. Moore, Ph.D.*

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

A Special Supplement on Delphi for the Linux Platform

www.DelphiZine.com

Delphi™ INFORMANT® MAGAZINE

August 2000

THE DISH ON KYLIX

Cross-platform Controls

Is Linux Ready for Delphi?

An Interview with R&D Leaders: Michael Swindell and Chuck Jazdzewski

AN INFORMANT® COMMUNICATIONS GROUP PUBLICATION

SYMPOSIUM

You could not step twice into the same river; for other waters are ever flowing onto you.
— Heraclitus (c.540-c.480 B.C.), *On the Universe*

At the Threshold
If you're a hard-core Delphiphile, as I am, there's a good chance you're reading this at the 11th annual Borland Conference in San Diego, California. And you've also no doubt heard of a "secret" project named Kylix — the effort to bring Delphi and C++Builder to the Linux platform. So you know these are heady times. Its cross-platform nature makes Kylix far more than yet-another RAD environment; it's a unique opportunity for Borland and its Windows — and soon Linux — customers.

In short, it hasn't been this exciting to be a Delphi developer since Delphi 1 shipped.

Therefore, we're especially proud to be able to bring you some very early Kylix coverage, including a look at the first Kylix custom control (built by Kylix R&D team member Robert Kozak) and its underlying code.

Perhaps the most important thing right now, however, is orientation. The questions are myriad. Why port Delphi to Linux? What will Kylix be good for? What are the benefits and pitfalls of Linux? Will Delphi for Windows be left behind? There's a lot to get your mind around, and opinions in the Delphi community run the gamut from euphoric to paranoiac.

We have two offerings in this regard. First, well-known Borland Engineer Danny Thorpe does an outstanding job of putting Linux and Kylix into perspective — dousing giddy exuberance and quelling baseless fears in equal measure. Danny is an accomplished author, having written the classic *Delphi Component Design* [Addison-Wesley, 1997], and is experienced and clear-headed enough to separate hyperbole from fact and write about them clearly.

We've also had the good fortune of interviewing two senior members of the Kylix R&D team: Michael Swindell and Chuck Jazdzewski. Michael is a Director of Product Management, and director of the Kylix Project; Chuck was one of the two original developers of Delphi, is Chief Architect of Kylix — and filled the shoes of legendary Anders Hejlsberg as Chief Architect of Delphi.

Kylix is nothing less than a bridge to the Linux/Unix programming world. It's a two-way bridge, as well, so we can expect an influx of Linux developers who will see Kylix not only as the best development environment for Linux, but also as their portal to Windows. I don't know which prospect is more exciting; fortunately, we don't have to choose.

Delphi Informant Magazine is committed to covering Kylix to the extent that you, the reader, are interested in hearing about it. Online surveys from our Web site (http://www.DelphiZine.com) indicate strong interest, but I want to be sure we're on the right track. This is an important crossroads for the Delphi community, so please let me hear from you.

How important is Linux and Kylix to you?

Thanks for reading.

[illegible]

Jerry Coffey, Editor-in-Chief
jcoffey@informant.com

Special Supplement to
Delphi Informant Magazine
August 2000

The Dish on Kylix

An Early Look at Delphi for Linux

Delphi INFORMANT MAGAZINE

Kylix / Linux / Windows

Is Linux Ready for Delphi?

Fears, Misconceptions, and Misplaced Euphoria

By Danny Thorpe

Change is a great way to polarize any community. It should come as no great surprise that when Inprise announced plans to develop RAD tools for the Linux platform, the responses from the Borland developer community were stereo cries of "Oh Yes!" and "Oh No!" with a background channel of "uh, what?"

I'd like to address some of the commonly expressed fears, misconceptions, and even misplaced euphoria I've heard from various quarters since the announcement. These are my personal opinions, not Borland/Inprise company policy. The questions may be paraphrased from actual conversations with actual customers, and my responses may contain actual opinions and/or sarcasm. Be prepared to wash your eyes out with soap. Safe harbor statement: Forward-looking statements are the fantasy we endeavor to make reality. If you buy high and sell low, well, that's pretty dumb isn't it?

Why Linux? Why not BeOS, or Mac OS10, or Solaris, or <your pet OS here>?
Linux was the fastest-growing OS in commercial installations in 1998 and 1999, and those commercial installations are buying every means of support and technology they can find. Even when the software can be obtained for free (or perhaps because of it), corporations buy support. It makes perfect business sense to spend money to protect and fortify your mission-critical and business-critical systems. It doesn't matter to your business that the stuff running your critical systems is a free download — except when it's 100,000 times cheaper than the alternatives. (Some of the "big iron" Unix OS licenses cost hundreds of thousands of dollars a year, every year. Linux runs on the same hardware, at the cost of a free download, or less than $100 for shrink-wrap retail. You do the math.)

From those Linux growth numbers, I seem to recall that Windows NT was still well ahead of Linux in terms of installed base (like 45 percent vs. 15 percent), and Windows NT grew a healthy amount, too. I believe Linux's growth is due more to cannibalizing the traditional Unix installed base (Solaris or HP Unix converting to Linux) than to Linux taking over the Windows NT market. Unix folks tend to be more comfortable switching between flavors of Unix than switching to NT. People who choose NT for their servers do it for the Microsoft name and perception of corporate stability and safety of investment. They aren't likely to move from NT to Linux anytime soon.

I'm not really that interested in where Linux's growth is coming from. Linux is the fastest growing OS (albeit starting from zero), and it's the only Unix flavor showing any growth at all.

Why would anyone pay money for development tools for a "free software" OS like Linux?
On first blush, the notion of taking a commercial product like Delphi to the so-called "free software" Linux platform sounds crazy. Why would anyone pay money for Linux development tools when Linux ships with a free built-in C compiler?

Answer: Quality, features, support, and — most of all — choice. Linux is about choice. Any Linux advocate who says Delphi isn't welcome in the Linux space is a hypocrite. Some people feel that commercial software for Linux is long overdue. Others feel strongly that commercial software containing proprietary technology should be burned at the stake. Regardless of your position in that particular jihad, you have to admit that choice of development tools (even those you wouldn't choose for yourself) is better for the Linux community than a police state model where one tool should be good enough for everyone, and nothing else is permitted.

Have you actually evaluated the benefits of Linux?
Yes, we have. We wouldn't have embarked on the Kylix project if we didn't feel there was a significant business opportunity in that space. We investigate lots of technology ideas for market opportunities all the time, but we only jump in to make a product where the numbers indicate we have a better than good chance of making a home-run product.

Microsoft has kept Borland/Inprise alive. Microsoft gave Borland/-Inprise millions.
Whoa, slow down there, Tex. Although the success of our Windows development tools is certainly linked to Microsoft's success in making Windows the dominant PC operating system, that's the limit of our endearment to the Microsoft Marketing Machine. Borland identified Windows as an emerging market way back in the early 1990s and built tools to capture revenue from that market. We're now doing the same for Linux, but this time we're starting earlier in the OS emergence cycle. (We almost missed the boat for Windows.) Starting earlier involves greater risk, but also offers greater rewards. More on that later.

The $125 million was a settlement for a patent infringement lawsuit between Microsoft and Borland. As a settlement, one could reasonably assume it is less than what Microsoft feared could be awarded at the end of a long and expensive legal battle. Borland/Inprise executives made public remarks at the time about the irony that Microsoft "blood money" would be poured into making products that did not support the Microsoft agenda, specifically: CORBA, Java, and Kylix.

Borland RAD tools for Linux! This is so cool! It'll destroy Windows! Linux rules the world!
That's nice. Now please enter your credit card number and your language preference: Java, Delphi, or C++? Would you like CORBA with that?

Okay folks, here's a radical concept to tattoo on your eyelids: "Our success does not require the destruction of Microsoft." Say it out loud. Say it slowly. Lather, rinse, repeat. Linux is succeeding in spite of the Microsoft monopoly. That's what makes Linux interesting. If there were no Microsoft, if Linux were the only OS in town, we'd have no reason to get excited about Linux, would we?

Linux can continue to grow by leaps and bounds and be successful in the presence of Microsoft. Borland tools for Linux can be a financial success, even if Linux always runs a distant second to Windows NT, and even if Linux never breaks into the desktop OS market. "There can be only one" doesn't apply. Borland's interest in the Linux market isn't that Linux will replace the Windows market, but that Linux is a market in addition to the Windows market.

There are so many Linux flavors, e.g. Red Hat, Corel, Slackware, SUSE, TurboLinux, Bob's Linux, etc. Testing our applications will be a nightmare!
Yeah, so? How is this different from testing Windows applications on the many flavors of the Win32 platform? Proper testing of Win32 applications today should include testing on the following distinct platforms: "virgin" Windows 95, Windows 95 OSR2, Windows 95 with IE4, Windows 95 with IE5, Windows 95 with DCOM, Windows 98, Windows 98 Second Edition, Windows 98 with IE5, Windows NT 4.0 SP3, Windows NT 4.0 with IE5, and Windows 2000. Linux is no worse. Ultimately, it comes down to defining what platforms you must support, and have the resources to test against. Anything else is then technically not supported.

As for Borland's testing of its Linux tools, Linux offers new opportunities. Unlike the Windows realm, chances are good that the many purveyors of Linux variants (er, distributions) will assist us with testing our products on their platforms. If they don't understand the incentives to help developers support their Linux flavors, they won't last very long. Linux distributions are becoming a commodity market, and commodities are distinguished more by name, endorsements, and availability (placement) than by actual feature differences.

Have you ever mused: "Wouldn't it be cool if the VCL core packages were distributed with the OS? Then I really could distribute full GUI applications in just 50K, and not have to worry about shipping the VCL packages." Realistically, the chances of getting Borland run-time packages included in the Windows platform distributions are pretty slim (trust me, we've asked), or at least very expensive. Linux, however, is not Windows ...

Who manages Linux? How are any major architectural changes going to be implemented over the next five years? Who do I write to ask for improvements?
Same response as before: How is this different from Windows? Who do you write to for improvements to Windows? And how responsive is Microsoft to your personal requests? Linux's lack of central control poses different challenges, but it also eliminates some of the steamroller effect that Microsoft is famous for.

The biggest risk for Linux is that true control of Linux features and progress is in the hands of an elite few who manage the Linux source-code archives. Personalities and egos can be as effective at steamrolling personal agendas into Linux as Microsoft's corporate OS agenda is for Windows. The Linux community must be ever vigilant of abuse of power.

Microsoft copies ideas from everybody. Linux is built on true innovation.
Baloney. Linux is a Unix clone. Call it what you want — derivative, compatible, whatever — it's still a clone of something that started long ago in a galaxy far, far away. The leading graphical desktops for Linux, KDE and Gnome, bear far greater resemblance to Windows 95 than to their Unix kin, Motif and NextStep.

Granted, Windows 95's look wasn't all that new either. Apple tried to sue Microsoft for copying the Macintosh UI/trashcan icon, until Microsoft pointed out that Apple got many of its Mac ideas — including the trashcan icon — from Xerox PARC. Xerox is probably still wondering why people are interested in their trashcans.

The greatest tragedy in the Linux community is that so many of its most energetic and vocal advocates know so little about their own technological heritage. Any new term or technology discovered while surfing around in the Linux source code must certainly be unique to Linux, or created in Linux and copied by Microsoft. Right? And heaven help anyone who says anything critical (or merely factual) about Linux, for flaming message threads shall rain down upon the heads of the heretics. Yea, verily.

With no knowledge of the past — or worse, revisionist history — the Linux community is at risk of realizing its own form of "1984."

Linux has done wonders for our community, but let us not run and jump until Linux has actually proven itself.

Markets grow on an "S" curve: flat at the bottom, steep slope up, and then flat at the top as the market reaches saturation. Opportunity starts at the base of the growth curve. If we wait until the top of the growth curve when Linux is established and there are a plethora of development tools available, then we're stuck in "the flats" — a plain-old 10 percent a year or less growth situation in a very crowded market space. Sounds like Windows, doesn't it?

It's very difficult to make a runaway success story from a standing start in the flats at either end of the growth curve. However, if you can hitch your wagon to a market "big bang" — get into a market soon after it starts its explosive growth — then your product can be swept up in the market's growth. That's how you get a triple-digit return on investment and market share. It's a relativity thing; you can't travel faster than the speed of light through normal space, but what happens if your space is expanding faster than the speed of light? When you take a step forward on an aircraft in flight, are you walking at 500 miles per hour? Would you rather have 10 percent of a large, stable market, or 10 percent of a smaller market that's doubling in size every year?

Remember that development tools have to venture ahead of applications by 18 months or more. You won't have a lot of really good applications on a particular platform until you have several really good sets of tools to choose from. Certainly, it's a little early yet to commit resources to developing end-user desktop applications for Linux. Linux has yet to really crack into the end-user desktop OS space. Whether or not Linux finds a foothold in the desktop space or remains a server OS, the time is right for development tools to move in.

If we do it right, Kylix has the potential to open the floodgates for Linux applications and Linux acceptance in the consumer markets. As noble as that may sound, we intend to make a buck on it too. These situations are very rare, but I firmly believe that Kylix is a market-maker opportunity — for Borland and for Linux.

Borland is a Windows shop. How are you going to survive a platform shift to Linux?

Excuse me, Borland hasn't always been a Windows tools vendor. Borland has produced development tools for CP/M, MSDOS, Macintosh, OS/2, Windows 3.1, protected mode DOS, Windows 95, Windows NT, and most recently, Java. (I don't consider Windows 95/Windows 98 a complete Win32 implementation.) The senior staff of the Kylix development team has first-hand experience developing native tools for seven distinct platforms: real mode MSDOS, 16-bit Windows 3.1, 16-bit protected mode DOS, 32-bit protected mode DOS, Windows 95, Windows NT, and Java. And I don't mean one staff member per platform; I mean nearly all the senior staff has worked on products on all these platforms.

Given this broad base of experience in the Kylix team, Linux is more of another walk around the block than some radical departure never before attempted. There are different faces to greet, different sights to see, but the same streets to walk, and same shoes to walk them.

It's amazing (and frustrating) how many of the "new" platform issues we're discovering in our Linux work bear haunting resemblance to platform challenges discovered and solved in past Borland products. For example, Linux's Position Independent Code (PIC) specification for shared object libraries will require compiler code generation treatments that are conceptually identical to the DS segment switching required in exported functions in 16-bit Windows DLLs. Spooky, huh?

A lot of the Delphi team is off doing Linux stuff now. Has Borland abandoned Delphi 6/C++Builder 6 development for Windows?

No. "Deep cycle" R&D for the Delphi 6/C++Builder 6 Windows products is proceeding in parallel with the Kylix effort. Some Delphi team members are dedicated to Delphi 6/C++Builder 6 work, some are dedicated to Kylix work, and some (like me) work on technology that applies to both platforms, and/or consult and advise for both projects simultaneously.

We've hired a lot of new folks to fill new openings in the Kylix effort, or backfill openings created on the Windows side by senior team members shifting focus to Kylix. Shifting resources around isn't the end of the world: We do it all the time here at Borland. It's called resource management — putting the talent and manpower where it's needed to complete projects. As one release of Delphi ships, the team shifts focus to the sister release of C++Builder. When that ships, resources refocus on the next version of Delphi. Now we have a third ball to juggle: Kylix in two flavors: Delphi and C++Builder.

Some folks have questioned the wisdom of having the majority of the senior staff "distracted" by this so-called Kylix "side-project." I wouldn't have it any other way. Who is best qualified to deliver a new product in an alien environment on an insane schedule that will still be recognizable as a child of Delphi? The Delphi team, that's who. We've built a few of these component architectures and IDE things before, ya know.

Why?

As you might have guessed from these opinions, I'm not a rabid Linux fan. Nor am I a rabid Windows fan (but don't ask me on a bad Linux day). Although I see the business opportunities opening up in the Linux market, that's not what keeps me going.

I volunteered to work on Kylix for a number of reasons:

1) I didn't know a thing about Linux, and I felt I should. It's a personal growth opportunity. Lots of neat stuff to explore. To better appreciate what you're familiar with, get familiar with something completely different.
2) To be a critical voice of reason in an otherwise enthusiastic Linux fan club.
3) To ensure the end result is Delphi.

I'm working on Kylix, not because I believe in Linux, but because I believe in Delphi.

This article originally appeared on the Borland Community Web site. This article describes features of software products that are in development and subject to change without notice. Description of such features here is speculative and does not constitute a binding contract or commitment of service.

AUTHOR BIO

Danny Thorpe is Staff Engineer for Delphi R&D at Borland/Inprise Corp.

table of CONTENTS

CLX / Custom Controls / Cross-platform

Cross-platform Controls

From Windows to Linux, and Back

These are exciting times for Borland. Not since the first whisper of Delphi has there been this much excitement about a Borland product. I'm talking, of course, about Kylix, the project to bring Delphi and C++Builder to the Linux operating system. The Delphi version will be available first, so for the rest of this article, Kylix refers to Delphi for Linux.

By Robert Kozak

We're developing a new VCL that will work with the Windows *and* Linux versions of Delphi. This means you can write an application in Windows, then move the source to a Linux box and recompile it — or vice versa. This new VCL is named CLX, for Component Library Cross-Platform. CLX encompasses the entire cross-platform library distributed with Kylix. There are a few sub-categories, which, as of this writing, break down as follows:

- baseCLX is the RTL, up to, and including, Classes.pas.
- visualCLX includes the user interface classes, i.e. the usual controls.
- dbCLX comprises the cross-platform database components.
- interCLX includes the Internet stuff, e.g. Apache, etc.

At the time of this writing [early May 2000], the first Field Test for Kylix is just beginning. By the time you read this, there will be a big difference between the Kylix I'm using and working on, and the version you'll see when it's available. This makes my job all that more difficult. It would be easy to talk in generalities, waxing eloquent about the underlying architecture. I'd much rather discuss the details, however, so you can get a head start producing CLX controls. Just keep in mind that it's likely some of the particulars discussed in this article will have changed by the time you read it.

visualCLX

This article is a primer on writing custom visualCLX (vCLX) controls. Essentially, the vCLX is what you know and love about the VCL. When you think about it, Visual Component Library is a bit of a misnomer; there's a lot more to it than the visual components. In this article, however, I'm only going to write about the visual controls. The Button, Edit, ListBox, PageControl, StatusBar, ProgressBar, etc. controls, have all been re-implemented to be cross-platform. How did we do this when the current VCL relies so much on Windows? In brief, we ripped out all the Windows stuff, and replaced it with another toolkit.

In Linux, there are a number of toolkits that contain the standard windowing controls, such as Buttons. They're called widgets, and GTK and Qt (pronounced "cute") are two of the more popular. Qt is a Linux widget toolkit that works on Windows and Linux. Because it aligned most closely with our goals, Qt was chosen as the basis for CLX. In other words, Qt is to CLX what the Windows API and common controls are to the VCL. Qt has some definite positives for the Delphi custom component developer on Linux:

- It's a prevalent Linux widget, used by the popular KDE desktop.
- It's similar to the Windows API style of development.
- Its graphics model is close to the VCL's graphics model.
- It introduces many standard widgets, and handles the message loop.

This begs two questions: Does this mean that Kylix supports only KDE, and no other desktops, such as Gnome? And how does using Qt as the basis of CLX affect me? The answer to the first question is that Kylix applications will run under any Linux desktop — particularly Gnome and KDE. The rest of this article answers the second question.

Methods
CreateParams
CreateSubClass
CreateWindowHandle
CreateWnd
DestroyWindowHandle
DestroyWnd
DoAddDockClient
DockOver
DoDockOver
DoRemoveDockClient
DoUnDock
GetDeviceContext
MainWndProc
ResetIme
ResetImeComposition
SetIme
SetImeCompositionWindow
WndProc
Properties
Ctl3D
DefWndProc
DockManager
DockSite
ImeMode
ImeName
ParentCtl3D
UseDockManager
WheelAccumulator

Figure 1: Methods and properties missing from *TWidgetControl* (formerly known as *TWinControl*).

Don't Want You Back

The goal is to make it easy for developers to port their applications to Linux with the least amount of trouble. Most of the component names are the same, and most of the properties are the same. Although a few properties will be missing from some components, and a few new ones will be added, for the most part, it should be fairly painless to port your applications.

For component writers, it's a different story. For starters, there is no Windows.pas, nor Windows API (see Figure 1). You can say goodbye to the **message** directive, and all of the CN and CM notifications. These have been changed to dynamics. There is also no docking, bi-directional (BiDi) methods/properties, input method editor (IME), or Asian support in the first release. Of course, there's no ActiveX, COM, or OLE support. Windows 3.1 components are also out as I write this.

By now I bet you're thinking, "That's not so bad; porting my components doesn't sound too difficult." But wait — there's more. At the time of this writing, the CLX unit names have all been changed to include "Q" as a prefix. So StdCtrls is now QStdCtrls, some classes have shuffled around a bit, and there are some subtle differences in the hierarchy (see Figure 2).

The CLX prefix of "Q" may or may not end up as the permanent prefix in the final release. *TWinControl* is now a *TWidgetControl*, but to ease the pain we added a *TWinControl* alias to *TWidgetControl*. *TWidgetControl* and descendants all have a *Handle* property that is an opaque reference to the Qt object; and a *Hooks* property, which is a reference to the hook objects that handle the event mechanism. (Hooks are part of a complex topic that is outside the scope of this article.)

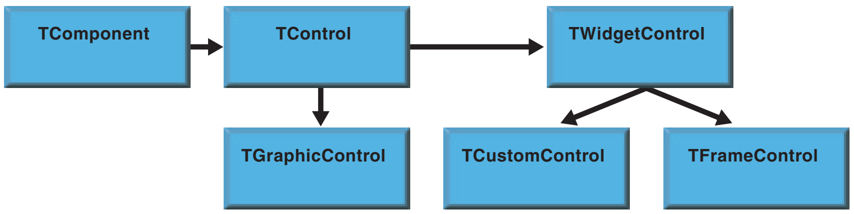


Figure 2: The differences in the class hierarchy are subtle.

OwnerDraw will be replaced with a new idea called Styles. Styles are basically a mechanism whereby a widget or application can take on a whole new look, similar to skins in Windows. This is something that's still in development, so I'm not going to discuss it further in this article. I will say this: It's way cool.

Is anything the same? Sure. *TCanvas* is the same as you remember, with its collection of *Pens* and *Brushes*. As I mentioned, the class hierarchy is basically the same, and events, such as *OnMouseDown*, *OnMouseMove*, *OnClick*, etc., are still there.

What Does It All Mean?

Let's move on to the meat of CLX, and see how it works. Qt is a C++ toolkit, so all of its widgets are C++ objects. On the other hand, CLX is written in Object Pascal, and Object Pascal can't talk directly to C++ objects. To make matters a little more difficult, Qt uses multiple inheritance in a few places. So we created an interface layer that takes all of the Qt classes and reduces them to a series of straight C functions. These are then wrapped up in a DLL in Windows and a shared object in Linux.

Every *TWidgetControl* has *CreateWidget*, *InitWidget*, and *HookEvents* virtual methods that almost always have to be overridden. *CreateWidget* creates the Qt widget, and assigns the *Handle* to the *FHandle* private field variable. *InitWidget* gets called after the widget is constructed, and the *Handle* is valid. Some of your property assignments will move from the *Create* constructor to *InitWidget*. This will allow delayed construction of the Qt object until it's really needed. For example, say you have a property named *Color*. In *SetColor*, you can check with *HandleAllocated* to see if you have a Qt handle. If the *Handle* is allocated, you can make the proper call to Qt to set the color. If not, you can store the value in a private field variable, and, in *InitWidget*, make the call to the Qt function to set the color.

There are two types of events: Widget and System. *HookEvents* is a virtual method that hooks the CLX controls event methods to a special hook object that communicates with the Qt object. (At least that's how I like to look at it.) The hook object is really just a set of method pointers. System events now go through *EventHandler*, which is basically a replacement for *WndProc*.

Larger Than Life

All of this is just background information, because you really don't need to know it in order to write cross-platform custom controls. It helps, but with CLX, writing cross-platform controls is a snap. Just as you didn't have to understand the complexities of the Windows API to write a VCL control, the same goes for CLX and Qt. Listing One (beginning on page viii) shows a custom control written with CLX. [It's available for download; see end of article for details.]

The project file, CalcTest.dpr, is shown in Figure 3. It's shown at design time in Figure 4. The run-time result, a calculator control (shown in Figure 5), looks much like the standard Microsoft Windows calculator.

As you can see, *TCalculator* is a descendant of *TFrameControl*. *TFrameControl* is a new control introduced to the hierarchy under

TWidgetControl that provides a frame for your controls. The two properties we're most interested in are *FrameBorderStyle* and *ShadowStyle*:

```
TFrameBorderStyle = (fbsNone, fbsBox, fbsPanel,
  fbsWinPanel, fbsHLine, fbsVLine, fbsStyledPanel,
  fbsPopupPanel);
TShadowStyle = (ssPlain, ssRaised, ssSunken);
```

```
program CalcTest;

uses
  SysUtils, Classes, QControls, QForms, QStdCtrls, Qt,
  QComCtrls, QCalc, Types;

type
  TTestForm = class(TForm)
    Calc: TCalculator;
  public
    constructor Create(AOwner: TComponent); override;
  end;

var
  TestForm: TTestForm;

{ TTestForm }
constructor TTestForm.Create(AOwner: TComponent);
begin
  inherited CreateNew(AOwner);
  SetBounds(10,100,640,480);

  Calc := TCalculator.Create(Self);
  // Don't forget: we have to set the parent.
  Calc.Parent := Self;
  Calc.Top := 100;
  Calc.Left := 200;
  // Uncomment these to try other Border effects:
  // Calc.FrameBorderStyle := fbsBox;
  // Calc.ShadowStyle := ssSunken;
end;

begin
  Application := TApplication.Create(nil);
  Application.CreateForm(TTestForm, TestForm);
  TestForm.Show;
  Application.Run;
end.
```

Figure 3: The project file for the CLX calculator control.

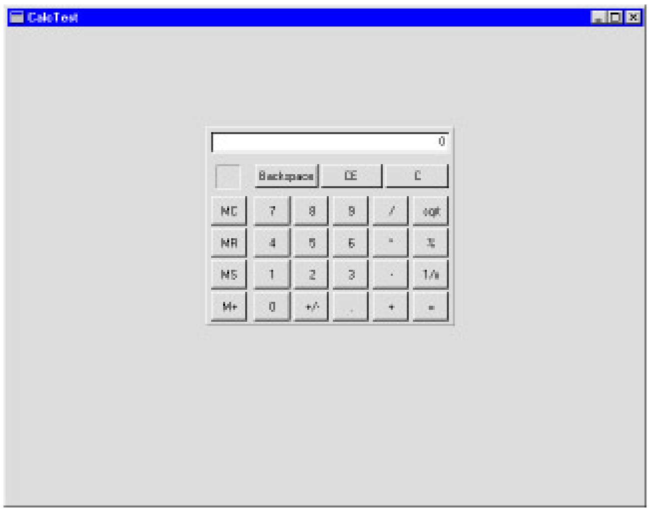


Figure 4: The calculator control at design time.

There are two important methods in this control. *BuildCalc* creates all of the buttons, and places them in their proper locations. As you can see, I used an enumerator named *TButtonType* to hold the "function" of the button, and this tidbit of information is stored as an integer in the *Tag* property. I refer to this later in the *Calc* method. All of the calculator buttons are stored in a protected array of *TButtonRecord* records named *Btns*:

```
TButtonRecord = record
  Top: Integer;
  Left: Integer;
  Width: Integer;
  Height: Integer;
  Caption: string;
  Color: TColor;
end;
```

This makes it easy to set up all of the buttons in a loop, rather than using an ugly bunch of *TButton.Create* calls. Notice that the buttons' *OnClick* handlers get assigned to the *TCalculator*'s *Calc* method. It's alright to do a direct assignment to what is typically a user event, because all of these buttons are internal to the calculator, and these events won't be published (see Figure 6).

I'm sure you've noticed a control I've created named *TStatusLabel*. I made this for the *TStatusBar*. It's basically a *TLabel* that publishes the properties from *TFrameControl*. I wanted it in the calculator, so I could get the "sunken box" look for the memory display like the Windows calculator. The Qt label widget is really a lot like the VCL *TPanel* component. For the *TLabel* in CLX, we don't publish the frame properties, but that doesn't stop you from using them in your descendants.

The last thing I do in *BuildCalc* is to create the edit control to display the results of the calculation. As you can see, the *Text* property of the calculator hooks directly to the *Text* property of the *Edit* control.

The other main method is *Calc*, which is essentially a huge **case** statement that evaluates which button was pushed, and decides what to do about it. I use the private field variables *FCurrentValue*, *FLastValue*, and *FRepeatValue* to handle the value of the calculations, so I don't have to implement a stack. The idea was to show how to create a cross-platform control, not how to write a calculator.

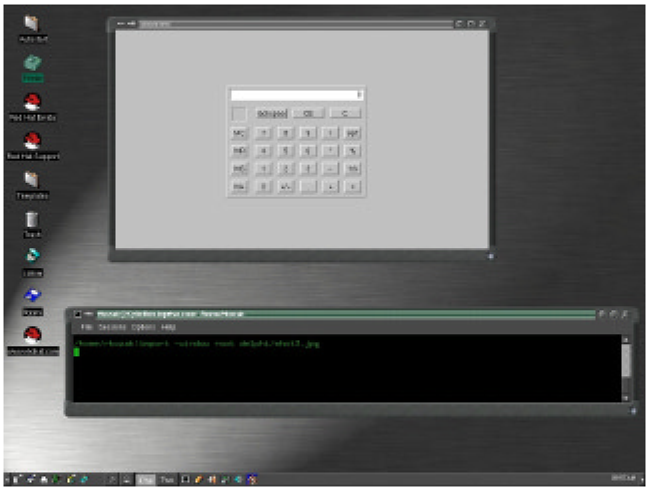


Figure 5: The control at run time, as it appears on Red Hat Linux.

```
for i := Low(TButtonType) to High(TButtonType) do
  with TButton.Create(Self) do begin
    Parent := Self;
    SetBounds(Btns[i].Left, Btns[i].Top, Btns[i].Width,
              Btns[i].Height);
    Caption := Btns[i].Caption;
    Color := Btns[i].Color;
    OnClick := Calc;
    Tag := Ord(i);
  end;
```

Figure 6: A direct assignment to a user event is okay in this case.

Oh yeah! Remember that I used the *Tag* property in *BuildCalc* to hold its function? That's retrieved in this method by casting the *Sender* to a *TButton*, and casting the *Tag* back to a *TButtonType*. *ButtonType* is the selector expression of the case statement:

```
ButtonType := TButtonType(TButton(Sender).Tag);
```

Are you wondering how we convert this to a cross-platform control? No? Good! That means you've been paying attention. This code will compile in Windows and Linux with absolutely no changes. There are no extra steps involved. Just by the virtue of using CLX, this control is ready to go.

Conclusion

As you can see, writing a cross-platform control isn't all that different from writing a VCL component. If you're a new component developer, it won't be difficult to learn. If you're an experienced VCL component builder, most of your knowledge will transfer to Kylix nicely.

As I said earlier, there are a lot of differences, but that should only affect developers who have components that rely on the Windows API. If you wrote a control that was a descendant of a VCL control, an aggregate of a few controls (as I did here with *TCalculator*), a non-visual component that doesn't rely on the Windows API, or was a *TGraphic* control, then you shouldn't have much trouble porting it to Linux.

This article describes features of software products that are in development and are subject to change without notice. Description of such features here is speculative and does not constitute a binding contract or commitment of service.

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD located in INFORM\00\AUG\DI200008RK.

AUTHOR BIO

Involved as a user of Delphi since the initial beta, Robert Kozak is a member of the Kylix R&D team and has been with Borland since the later half of 1999. Since he joined Borland, he has been involved in the development of C++Builder 5 and Kylix. Robert was involved with the start of TaDDA! (Toronto Area Delphi Developers Association), which later merged with TDUG (Toronto Delphi Users Group). Robert continues to stay active in the user community, and is active on the Borland newsgroups.

Begin Listing One — QCalc.pas

```
{ ***************************************************** }
{                                                       }
{      Borland Delphi Visual Component Library          }
{   Borland Delphi Component Library (X)Crossplatform   }
{                                                       }
{      Copyright (c) 2000 Borland International         }
{                                                       }
{ ***************************************************** }
unit QCalc;

// This is the very first Custom control written for CLX.
interface

uses
  Sysutils, Classes, QT, QControls, QStdCtrls, QComCtrls,
  QGraphics;

type
  TButtonType = (bt0, bt1, bt2, bt3, bt4, bt5, bt6, bt7,
    bt8, bt9, btDecimal, btPlusMinus, btMultiply, btDivide,
    btAdd, btSubtract, btSqrt, btPercent, btInverse,
    btEquals, btBackspace, btClear, btClearAll,
    btMemoryRecall, btMemoryStore, btMemoryClear,
    btMemoryAdd);

  TCalcState = (csNone, csAdd, csSubtract, csMultiply,
    csDivide);

  TButtonRecord = record
    Top: Integer;
    Left: Integer;
    Width: Integer;
    Height: Integer;
    Caption: string;
    Color: TColor;
  end;

  TCalculator = class(TFrameControl)
  private
    FResultEdit: TEdit;
    FStatus: TStatusLabel;
    FMemoryValue: Single;
    FCurrentValue: Single;
    FLastValue: Single;
    FRepeatValue: Single;
    FState: TCalcState;
    FBackSpaceValid: Boolean;
  protected
    Btns: array [TButtonType] of TButtonRecord;
    procedure BuildCalc;
    procedure Calc(Sender: TObject);
    function GetText : string; override;
    procedure SetText(const Value : string); override;
  public
    constructor Create(AOwner: TComponent); override;
    property Value : Single read FCurrentValue;
  published
    property Text : string read GetText write SetText;
    property FrameBorderStyle;
    property ShadowStyle;
    property LineWidth;
    property Margin;
    property MidLineWidth;
    property FrameRect;
  end;

implementation

function ButtonRecord(aTop, aLeft, aWidth,
  aHeight: Integer; aCaption: string;
  aColor: TColor = clBlack): TButtonRecord;
```

```
begin
  Result.Top := aTop;
  Result.Left := aLeft;
  Result.Width := aWidth;
  Result.Height := aHeight;
  Result.Caption := aCaption;
  Result.Color := aColor;
end;

{ TCalculator }
constructor TCalculator.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  SetBounds(0,0,250,200);
  FMemoryValue := 0;
  FCurrentValue := 0;
  FLastValue := 0;
  FRepeatValue := 0;
  ShadowStyle := ssRaised;
  FrameBorderStyle := fbsStyledPanel;
  BuildCalc;
end;

procedure TCalculator.BuildCalc;
var
  i: TButtonType;
begin
  Btns[bt7] := ButtonRecord(70, 48, 36, 29, '7');
  Btns[bt4] := ButtonRecord(102, 48, 36, 29, '4');
  Btns[bt1] := ButtonRecord(134, 48, 36, 29, '1');
  Btns[bt0] := ButtonRecord(166, 48, 36, 29, '0');
  Btns[bt8] := ButtonRecord(70, 88, 36, 29, '8');
  Btns[bt5] := ButtonRecord(102, 88, 36, 29, '5');
  Btns[bt2] := ButtonRecord(134, 88, 36, 29, '2');
  Btns[btPlusMinus] :=
    ButtonRecord(166, 88, 36, 29, '+/-');
  Btns[bt9] := ButtonRecord(70, 128, 36, 29, '9');
  Btns[bt6] := ButtonRecord(102, 128, 36, 29, '6');
  Btns[bt3] := ButtonRecord(134, 128, 36, 29, '3');
  Btns[btDecimal] := ButtonRecord(166, 128, 36, 29, '.');
  Btns[btDivide] := ButtonRecord(70, 168, 36, 29, '/');
  Btns[btMultiply] := ButtonRecord(102, 168, 36, 29, '*');
  Btns[btSubtract] := ButtonRecord(134, 168, 36, 29, '-');
  Btns[btAdd] := ButtonRecord(166, 168, 36, 29, '+');
  Btns[btBackspace] :=
    ButtonRecord(37, 49, 63, 25, 'Backspace');
  Btns[btClear] := ButtonRecord(37, 115, 63, 25, 'CE');
  Btns[btClearAll] := ButtonRecord(37, 181, 63, 25, 'C');
  Btns[btsqrt] := ButtonRecord(70, 208, 36, 29, 'sqrt');
  Btns[btPercent] := ButtonRecord(102, 208, 36, 29, '%');
  Btns[btInverse] := ButtonRecord(134, 208, 36, 29, '1/x');
  Btns[btEquals] := ButtonRecord(166, 208, 36, 29, '=');
  Btns[btMemoryAdd] := ButtonRecord(166, 5, 36, 29, 'M+');
  Btns[btMemoryStore] :=
    ButtonRecord(134, 5, 36, 29, 'MS');
  Btns[btMemoryRecall] :=
    ButtonRecord(102, 5, 36, 29, 'MR');
  Btns[btMemoryClear] := ButtonRecord(70, 5, 36, 29, 'MC');

  for i := Low(TButtonType) to High(TButtonType) do
    with TButton.Create(Self) do begin
      Parent := Self;
      SetBounds(Btns[i].Left, Btns[i].Top, Btns[i].Width,
                Btns[i].Height);
      Caption := Btns[i].Caption;
      Color := Btns[i].Color;
      OnClick := Calc;
      Tag := Ord(i);
    end;

  FStatus := TStatusLabel.Create(Self);
  with FStatus do begin
    Parent := Self;
    SetBounds(10, 38, 25, 25);
    FrameBorderStyle := fbsStyledPanel;
    ShadowStyle := ssSunken;
  end;

  FResultEdit := TEdit.Create(Self);
  FResultEdit.Parent := Self;
  FResultEdit.SetBounds(5, 5, 240, 25);
  FResultEdit.Alignment := taRightJustify;
  FResultEdit.Font.Height := -13;
  FResultEdit.Font.Name := 'Arial';
  FResultEdit.Text := '0';
end;

procedure TCalculator.Calc(Sender: TObject);
const
  MemoryStoreMap: array [Boolean] of string = (' M','');
var
  ButtonType: TButtonType;
  Temp: string;
  TempValue: Single;
begin
  ButtonType := TButtonType(TButton(Sender).Tag);

  try
  case ButtonType of
    bt0..bt9:
    begin
      FBackSpaceValid := True;
      if (FResultEdit.Text = '0') or
         (FCurrentValue = 0) then
        FResultEdit.Text := '';
      FResultEdit.Text :=
        FResultEdit.Text + Btns[ButtonType].Caption;
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FRepeatValue := 0;
    end;

    btDecimal:
    if Pos('.', FResultEdit.Text) < 1 then begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FLastValue := 0;
      FResultEdit.Text :=
        FResultEdit.Text + Btns[ButtonType].Caption;
    end;

    btPlusMinus:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FCurrentValue := FCurrentValue * -1;
      FResultEdit.Text := FloatToStr(FCurrentValue);
    end;

    btClearAll:
    begin
      FCurrentValue := 0;
      FLastValue := 0;
      FResultEdit.Text := '0';
      FState := csNone;
    end;

    btClear:
    begin
      FCurrentValue := 0;
      FResultEdit.Text := '0';
    end;

    btAdd:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FState := csAdd;
      FLastValue := FCurrentValue;
      FCurrentValue := 0;
    end;

    btSubtract:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FState := csSubtract;
      FLastValue := FCurrentValue;
      FCurrentValue := 0;
    end;

    btDivide:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FState := csDivide;
      FLastValue := FCurrentValue;
      FCurrentValue := 0;
    end;

    btMultiply:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FState := csMultiply;
      FLastValue := FCurrentValue;
      FCurrentValue := 0;
    end;

    btBackSpace:
    if FBackSpaceValid then begin
      Temp := FResultEdit.Text;
      Delete(Temp, Length(Temp),1);
      if Temp = '' then
        Temp := '0';
      FCurrentValue := StrToFloat(Temp);
      FResultEdit.Text := FloatToStr(FCurrentValue);
    end;

    btInverse:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FCurrentValue := 1 / FCurrentValue;
      FResultEdit.Text := FloatToStr(FCurrentValue);
    end;

    btPercent:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FCurrentValue := FCurrentValue / 100;
      FResultEdit.Text := FloatToStr(FCurrentValue);
    end;

    btSqrt:
    begin
      FCurrentValue := StrToFloat(FResultEdit.Text);
      FCurrentValue := Sqrt(FCurrentValue);
      FResultEdit.Text := FloatToStr(FCurrentValue);
    end;

    btMemoryStore:
    begin
      FMemoryValue := StrToFloat(FResultEdit.Text);
      FMemoryValue := FMemoryValue * 1;
      FCurrentValue := 0;
    end;

    btMemoryAdd:
    begin
      TempValue := FMemoryValue;
      FMemoryValue := StrToFloat(FResultEdit.Text);
      FMemoryValue := (FMemoryValue * 1) + TempValue;
    end;

    btMemoryRecall:
    begin
      FResultEdit.Text := FloatToStr(FMemoryValue);
      FCurrentValue := 0;
    end;

    btMemoryClear:
    begin
      FMemoryValue := 0;
    end;

    btEquals:
    if FState <> csNone then begin
      FBackSpaceValid := False;
      FCurrentValue := StrToFloat(FResultEdit.Text);
      if FRepeatValue = 0 then begin
        FRepeatValue := FCurrentValue;
        FCurrentValue := FLastValue;
      end;
      FLastValue := FRepeatValue;
      case FState of
        csAdd:
          FCurrentValue := FCurrentValue + FLastValue;
        csMultiply:
          FCurrentValue := FCurrentValue * FLastValue;
        csSubtract:
          FCurrentValue := FCurrentValue - FLastValue;
        csDivide:
          FCurrentValue := FCurrentValue / FLastValue;
      end;
      FLastValue := FCurrentValue;
      FResultEdit.Text := FloatToStr(FCurrentValue);
      FCurrentValue := 0;
    end;
  end;  // case  ButtonType of...

  except
    on E: Exception do begin
      FResultEdit.Text := E.Message;
      FLastValue := 0;
      FCurrentValue := 0;
      FRepeatValue := 0;
      FState := csNone;
    end;
  end;
  FStatus.Caption := MemoryStoreMap[FMemoryValue = 0];
end;

function TCalculator.GetText: string;
begin
  Result := FResultEdit.Text;
end;

procedure TCalculator.SetText(const Value: string);
begin
  FResultEdit.Text := Value;
end;

end.
```

End Listing One

R&D Interview

Some Q&A with R&D

An Interview with Borland's Michael Swindell and Chuck Jazdzewski

D*elphi Informant Magazine* recently had the good fortune of being able to interview two senior members of the Borland research and development team working on the Kylix Project. Michael Swindell is a Director of Product Management, and director of the Kylix Project. Chuck Jazdzewski was one of the two original developers of Delphi, and is Chief Architect of Delphi and Kylix.

What will Kylix be named when it's released? "Delphi for Linux" seems the obvious choice.

Michael: We haven't announced the official name yet. It'll have to remain a secret for a little while longer. At this time, Kylix is a code name, but what we are building is definitely Delphi.

Will there be separate editions of Kylix, e.g. Standard, Enterprise, etc.?

Michael: Yes, there will be different editions for different levels of developer and target development types, just as we now offer with the Windows editions. However, we do have some fairly substantial changes in the works in the area of editions.

How will Kylix be priced? Will Kylix be open source?

Michael: Pricing has not yet been announced, but there will be pricing. <smile> There are no plans to open source the Kylix IDE and compiler. Our business model still includes selling our products. However, we do understand and appreciate free and open source software development. I'm confident that Linux — not to mention Kylix — would not exist today without the GPL (GNU General Public License), so we're working on a solution that will give us the freedom to support our business model, and arm developers with tools to build proprietary applications or open-source applications. It's going to be up to the developer to decide. A major piece of what we are working on is what and how in Kylix to open source, and what to make freely available for download.

Delphi is famous for being developed in Delphi. Will this still be the case for Kylix?

Chuck: Delphi for Linux is being developed in Delphi for Linux. We are using the same compiler and the same fundamental CLX run-time library to build Kylix that application users will be using. However, the initial version of the Linux IDE won't be using CLX itself for its user interface. This is due mainly to the parallel IDE/CLX development model we are taking with the first version of Kylix in order to ship in 2000. We plan to move the IDE's UI to CLX in a future release.

What has been the biggest challenge in developing Kylix? What, if anything, turned out to be unexpectedly easy?

Chuck: Our biggest challenge so far has been modifying the compiler to emit ELF-format, instead of PE-format, executables and dealing with the differences between Linux source object files (.so files) and Windows DLLs. We've also encountered incompatibilities between various desktops, although we're working with various standards groups in the Linux community to help standardize the application interface to the desktop.

What's been a pleasant surprise so far is how portable code written with the VCL has been. We've found that if the code is written in straight Delphi, with no implicit Windows assumptions, the code moves very well into CLX. A substantial number of components that we coded ourselves have moved over with little or no changes.

Michael: We were also surprised when a few third-party component developers in the beta program had rebuilt their custom components natively for Linux, after having the first Kylix beta installed for just a few hours. That was very validating for us.

Users

You've been surveying the Delphi community, which is ostensibly a Windows community. How many of them have Unix/Linux experience?

Michael: It's definitely a mix. There are Windows developers who cut their teeth on Unix in the 70s and 80s, and there are Windows developers who have never needed to edit a config.sys. We haven't done any specific studies on this, but anecdotally it looks like about a quarter of our Windows developers have Unix/Linux experience. However, about three-quarters plan to start developing on Linux with Kylix — which has us very excited.

Do you anticipate attracting new customers to Kylix, or do you expect most to come from your Delphi/C++Builder base?

Michael: Absolutely yes! About five years ago developers flocked to Delphi from Visual Basic, simply because it did RAD better. It was a native code compiler, it was blazingly fast, it included a vast component library, and it allowed developers to derive from our library and build their own custom components. Five years later, we're seeing a whole new rush of Visual Basic developers coming over to Delphi in order to get to Linux. It's unlikely that Microsoft will take VB and VC++ to Linux anytime soon, so if you're a VB or VC++ developer, and you want platform flexibility, suddenly Delphi and C++Builder look mighty attractive.

Is Kylix particularly well-suited for corporate developers, independent developers, or third-party developers?

Michael: I wouldn't say more for one type or the other. Kylix is well suited for "application" developers. Kylix is all about applications — whether you're a fortune-50 bank, or one guy in a garage. If you're re-building the kernel, or a device driver for the latest 3D accelerator, then the GCC (GNU C compiler), Emacs, Cygnus, or CodeWarrior is probably your best bet. The tools we're used to seeing on Linux have been well-suited for building the Linux infrastructure, but haven't done much to help solve the application problems. The complicated technologies that no one likes to deal with are exactly what Kylix does so well — making the hard stuff easy. So if you want to build anything with a GUI, or an Apache Web application, or anything that remotely has to do with a database, then Kylix is the tool you need.

Kylix is also a tool that's tailor made for third-party developers. Delphi is a testament to this, with tens of thousands of third-party tools and components, commercially and freely available online and in catalogs. Kylix will deliver a lot out of the box, but will also make it easy to encapsulate more vertical solutions, and present them to a developer in a way that's both easy to use and understand. I'm not a telephony expert for example, but I can throw a set of third-party Delphi telephony components down on a form and have a working application that integrates with our PBX in minutes. Kylix makes it easy for the third-party, and easy for the application developer, to leverage those vertical solutions.

Are there particular types of applications for which Kylix will be especially well-suited?

Michael: Delphi targets a wide range of application development and Kylix will be no exception. The first and most obvious that Kylix will target will be visual applications, applications with graphical user interfaces. From a desktop application perspective, Linux is in a similar situation as Windows was in the early 90s. Until, of course, when Visual Basic and Delphi changed everything. Beyond that, Kylix specifically targets database and Internet applications, making both far faster and easier to develop on Linux than with any other tool. Then there's the added benefit of cross-platform compatibility with Windows.

Chuck: Keeping in mind that Kylix is Delphi and C++Builder for Linux, Kylix is well-suited for any application you would use Delphi or C++Builder for, including utilities, client/server applications, multi-tier applications, Web servers, etc. The only difference is that Kylix applications are now native Linux applications.

Database

Which databases do you plan to support with version 1? Do you plan to ship a database with it?

Michael: We're working with all the major database vendors on Kylix support. We hope to support MySQL, InterBase, Oracle, DB2, and perhaps a few more. At this time we have MySQL and InterBase in beta, and are getting ready to go into beta with the next driver. If all continues as planned, Kylix will have the fastest, most flexible, and most diverse DB support of any development tool on Linux and Windows. We plan to ship the Open Source InterBase 6.0 with Kylix.

How will the connectivity to different databases be handled? Will there be a middleware layer that fulfills the same function as the BDE or ADO in the Windows world?

Michael: There is a new data-access layer named dbDirect that is both blazingly fast and fully cross platform. The BDE is a heavyweight because it's a desktop RDBMS in itself, with a lot of overhead and configuration issues that are really unnecessary if you're using anything other than Paradox or dBASE tables. In Kylix the access layer is just that, an optimized data-access layer. We'll be supplying native drivers for the popular databases we mentioned a minute ago, that will plug into dbDirect. We also plan to open up the specification for dbDirect, so others can easily build native drivers for other data sources, and so it can be used in non-Kylix solutions.

Will the database component architecture resemble the *TTable*, *TQuery*, *TStoredProc* architecture? Or will it be more MIDAS-like, i.e. some type of provider/consumer architecture such as JBuilder's?

Chuck: The architecture for database access in Kylix will be more like MIDAS than the BDE. If you're already using cached-updates, *TStoredProc*, *TQuery*, and/or MIDAS, your application will port very quickly to the new components. If you rely on BDE-specific features of *TTable*, you'll have to rewrite portions of your application. We're making the Kylix database-access components cross-platform, and will make them available in the next release of our Windows version of Delphi as well. This will allow you to make the change once to the new architecture, and then keep the Windows and Linux code substantially the same.

Interface

Does Kylix use an existing Linux GUI, such as the KDE Desktop Environment (KDE), or GNU Network Object Model Environment (GNOME), or are you rolling your own?

Chuck: We certainly won't be doing our own desktop or window manager; we'll be relying on the existing desktops such as KDE and GNOME.

Michael: Or no desktop at all for that matter. If a user just wants to run their favorite window manager without GNOME or KDE, that will be fine. As for the GUI widgets themselves, CLX will be using the native Linux Qt widgets. We don't roll our own in Windows either, thank goodness. In Windows the VCL uses the Windows common controls and GDI for drawing functions and widgets. On Linux, CLX will be using Qt's widgets and drawing functions in place of the Windows API.

Which Linux interfaces (e.g. GNOME, KDE, etc.) will Kylix support?
Chuck: We'd like to support all desktops, but we'll probably only certify against GNOME and KDE. The applications you write, however, will work on any desktop.

Is Kylix going to support Linux themes? For example, if I develop under GNOME and ship my application to someone running KDE, is it going to look like a GNOME application, or a KDE application?
Chuck: We'll try to make your application feel as natural as possible. We plan to support both KDE and GNOME themes eventually, but will probably only ship with KDE theme support in the first version. We don't like having to make a choice between KDE and GNOME support, and are working with the GNOME and KDE teams to make theme support more transparent. Unfortunately, I don't expect to see fruit from our effort in the first version of Kylix.

Michael: Philosophically we don't believe it's the place of a tools vendor to dictate which desktop environment the end user of an application should be using. It should be up to the Linux user to decide which desktop he or she prefers. We don't want to get in the way of that decision. At the same time, both KDE and GNOME are exposing some very cool APIs, and we intend to support them when possible — either automatically or at the discretion of the developer.

Windows-Linux Compatibility

Will I be able to move my projects back and forth between Delphi and Kylix?
Chuck: There are two ways we plan to support this. First, if you need a high degree of source compatibility, you can use CLX components under Windows. Second, we will also make the form designer capable of parsing IFDEF statements in the source, so you can design two form images, one for CLX and one for VCL, in the same file. This will allow you to have a fair degree of source compatibility without sacrificing Windows features or the Windows look and feel.

Will the Delphi and Kylix IDEs be the same?
Chuck: The Kylix IDE is based on the Delphi 5 IDE, so it will look and feel substantially the same. There will probably be a few Delphi 6 features that make their way into the Kylix release, but we'll try not to steal too much thunder from Delphi 6.

Michael: We plan to develop both IDEs simultaneously, so a new feature in one will be a new feature in the other.

What level of interoperability is planned between the Windows and Linux versions? For example, are the DFMs compatible at the binary (or even text) level?
Chuck: The format of the DFM files will be exactly the same, but that really doesn't answer the question. The biggest problem users will face moving from Windows to Linux is finding the equivalent components. We'll provide the standard components Button, Label, etc., but if you're using a third-party component, or a component you wrote yourself, you'll need to find or create a Linux equivalent. We're working with our third-party vendors now to help them get a leg up on the process of moving their components to Linux. Since we feel this is one of the biggest issues users will face, we're spending a lot of time on it.

Michael: We're focusing on providing a high degree of compatibility between Kylix and Delphi from the application developer's perspective, but things are different for the third-party component developers who often stray outside the VCL today. There are basically two levels of compatibility for the application developer: VCL-to-CLX compatibility, and CLX-to-CLX compatibility. Yes, I said CLX to CLX. CLX will be coming to a Windows version of Delphi in the near future. Moving from VCL to CLX should be straightforward for a Delphi developer; the basics should just "move over" so to speak. There will be minor differences here and there, but the classes, properties, and methods should be the same for the most part. Of course, it's always the minor differences that have the biggest impact. Once a developer is using CLX, then it really becomes possible to single-source an application between the platforms — provided the developer keeps some basic rules in mind.

Will current Delphi applications have to be substantially rewritten to compile and run on Linux via Kylix?
Chuck: I would like to say "Absolutely not," but it really depends. If your application is a traditional client/server application, WebBroker application, etc., then you won't have to change much, if anything. You just need to find the equivalent components, many of which will be named the same, and recompile the application. If you're using MAPI, ActiveX, or have written a desktop extension, however, you have your work cut out for you. The more Windows-specific features you use, the more trouble you'll have porting to Linux.

Will the CLX have the same class structure as the VCL?
Chuck: The CLX hierarchy and the VCL hierarchy are very similar. Anyone familiar with VCL will feel right at home with CLX. Some of the names have changed, but not drastically.

The plan was to create a version of Delphi for Linux that was compatible at the VCL level, with some components available for Linux only, some for Windows only, and some for both. How does it look like those percentages will shake out?
Chuck: Our original goal was to create a native class library for Linux. We did not set out, initially, to create a cross-platform library at all. We wanted to make sure that the application you wrote using Kylix looked and felt like a Linux application — even to a die-hard Linux user. As it turns out, since we based CLX on top of an existing cross-platform widget set, Qt, we found we could also easily produce a Windows version of CLX, thus making it cross-platform. Also, since most of the abstraction provided by the VCL was already independent of Windows, a lot of that abstraction is common between Linux and Windows. We are quite pleased, so far, with the degree of compatibility between

VCL and CLX, but we won't know exact percentages until we are closer to shipping. Even then, the percentages will not be very meaningful, since compatibility really depends so much on how much your individual applications depend on Windows and Windows-specific things.

Aside from using the Windows API and Windows-specific calls (e.g. DirectX), is there going to be any cross-platform programming differences? For example, if I use OpenGL and standard Delphi commands, will I be able to just make a change such as the OpenGL.pas unit, and then recompile in Kylix?

Chuck: This is really hard to say. Obviously things like DirectX, MAPI, etc. will not exist on Linux. OpenGL and the socket API, will. The general rule of thumb is if the API originally came from Unix, you should be fine. If it was originated by Microsoft, then you'll need to find a Linux equivalent.

The biggest single non-API issue we have found is differences between file names in Linux and file names in Windows. Linux is case-sensitive, Windows is not. Linux uses a forward slash to delimit directories, where Windows uses a backslash. Linux doesn't have a device prefix, while Windows does. We're modifying our RTL to allow you to code independent of these issues, so you'll need to examine your code carefully for subtle assumptions regarding the format of file names, and possibly make changes with regard to the new API.

How about kernel and library compatibility? If I create my application on a 2.2.10 kernel, supplied by Red Hat, will it run on a 2.2.12 kernel I download from Slackware? Do I need to ship libraries, depend on standard glibc, or are the Kylix executables stand-alone, as Delphi executables are?

Chuck: We expect a high degree of compatibility between the various distributions, once you get your application installed. Unfortunately, installing the application on the various distributions remains one of Linux's biggest problems. It's a problem that Red Hat, Debian, and others have made great strides in, but the installation process is still not standardized across all Linux distributions. We will be working with various companies to deal with this problem for our own installation, and we hope to make those solutions available to our customers as well.

The operating system changes, but the chip remains the same. Does this mean that the assembler code in Delphi — and in Delphi applications — can be ported unscathed?

Chuck: My advice is to stay away from assembler if you want to have your code portable. Recode it in Pascal if you can. We are doing just that for substantial portions of the RTL. The main reason is that, as opposed to Windows, the assembler used in a Linux .so file is different from that in an executable. Shared objects require the code to be position independent without fixups. This means that all code referring to a global variable needs to be through an indirection relative to what is referred to as the global offset table. This is too complicated a topic to describe here, but we'll describe this in our documentation. Linux is a different world. The chip is the same, but the assumptions made by the OS are different.

Linux

Which flavors of Linux will Kylix support in version 1. Which one do you use?

Michael: We haven't yet announced which distributions will be recommended "on the box" so to speak, but we're testing on a handful of different popular distributions, and our beta testers are also using different distributions. We also have representatives from all the major distributions participating in the beta program, and interacting with the beta testers. In house it's a fairly mixed bag; we use Red Hat, Mandrake, Debian, SuSE, Caldera, TurboLinux, and Corel.

Once Linux is supported, it doesn't seem a long step to Unix. If Kylix takes off, will we see a Delphi for Unix?

Michael: If we see a market demand, we'll certainly explore it. Kylix came about primarily because our customers spoke up, and we were able to verify the validity of the demand. Technically, the work we are doing in Kylix, removing the Windows-specific dependencies, will enable us to take Kylix to just about anywhere we'd like. Where we'd like to go and what would make good business sense are sometimes two different things. <laugh>

Many Linux developers are accustomed to getting their software for free. Are there plans to make a free version of Kylix available? Does Inprise anticipate having problems actually selling Kylix?

Michael: We haven't seen this kind of demand for a product in development since Delphi 1 five years ago. The demand is coming from both the Windows and Linux communities, so we don't see any obstacles in selling Kylix.

I can say without blinking that, next to Apache, I believe Kylix will be the most important application for the Linux platform to date, and probably for the next three to five years. That said, will Linux developers pay for Kylix? We believe so. Linux developers have told us that they aren't looking for any handouts; they're looking for a killer application-development tool. That doesn't mean we won't support open-source development, or that we won't be offering a form of freely available tools. These are things we're definitely working on.

Are there particular features of Linux that are proving to be especially helpful in the development of Kylix? For example, is there some feature of the Linux OS that has helped the team overcome an obstacle found in Windows?

Chuck: Stability is the biggest feature of Linux that has helped us. The only time I reboot my Linux machine is to upgrade the kernel. People ask me what the "blue screen" equivalent is under Linux. I smile and respond, "I don't know; I haven't seen it yet." Although Windows has made great strides in stability, especially with Windows 2000, I feel that Linux sets the standard for workstation stability.

Miscellaneous

How do the size of the executables compare to ones written with standard Linux compilers, e.g. the GCC?

Michael: It's definitely too early to say for sure, but it's our intention that executables be on par with the size of Delphi's Windows executables.

How do size and performance measure up to a Delphi application on a similarly equipped Windows box?

Michael: Again, it's still too early to tell, but we expect the system requirements to be similar to Delphi/Windows to achieve the same level of performance.

Is the CLX thread-safe?

Chuck: No, but it will be thread-aware. We will provide a standard mechanism that will allow multiple threads to operate in the same application and use CLX.

How's access to other libraries? Can I use third-party libraries, for example, to make enhancements to or take advantage of existing software like the GNOME/KDE configuration software, or StarOffice?

Chuck: We don't have any built-in limitation to what you can use or link to, but you will need to have the API translated into Pascal, just as we did for Windows, and now have done with Qt, and many of the Linux APIs such as libc.

Michael: Libraries built with the GCC should be fine, but — as Chuck said — the headers will need to be translated into Pascal. In the near future, however, C++Builder will be a different story, and should be able to use GCC libraries directly.

Are OpenGL wrappers included out-of-the-box?

Chuck: We will probably not have OpenGL support available out-of-the-box. We do expect organizations, such as Project JEDI (http://www.delphi-jedi.org), to contribute these types of translations.

What support is included for using multiple X terminals in the same program? How easy will it be to use?

Chuck: CLX will not have any built-in support for multiple X-terminal connections, but that doesn't limit a single X server from rendering to multiple monitors, which we will support. Your application could also start multiple X connections as well, but they would have to be handled separately and independently from CLX. We've found that multiple X-terminal connections aren't supported very often among applications, nor is it typically required.

Is there going to be a set of components comparable to the Internet and FastNet components in Delphi 5?

Michael: Yes, there will be client and server Internet protocol components. There has been a lot of progress in the area of Delphi third-party Internet protocol components lately, and we expect to see several available very soon, in addition to what we include with Kylix.

Have you considered adding limited JavaBean support, say, the ability to import them?

Chuck: We haven't specifically looked into supporting JavaBeans, but we have looked into coming up with an independent component specification for Linux that would be similar to ActiveX under Windows. We're working on this specification with Troll Tech, the makers of Qt, who employ some of the maintainers of KDE; and Red Hat, the primary maintainers of GTK+ and GNOME. Trolltech and Red Hat have taken the lead on this specification, and we've made some contributions as well. We plan to implement the specification when it's finalized. It should then be possible to implement the specification for a Java virtual machine that would allow JavaBeans to be usable in any container that supports the specification, Delphi being just one of them.

Conclusion

We have to ask: When will Kylix be available?

Michael: We aren't yet ready to announce release dates, but I can say that it will be after BorCon.

What message concerning Kylix and Delphi do you want the developers attending the Inprise/Borland Conference in San Diego to share when they return to their companies?

Michael: That real rapid application development no longer means being tied to a single platform, or relying on a run-time interpreter, or virtual machine. Native rapid cross-platform development is here, and it is Delphi.

Questions for this interview were contributed by *Delphi Informant Magazine's* Editor-in-Chief, Jerry Coffey; *Delphi Informant Magazine's* Technical Editor, Robert Vivrette; *Delphi Informant Magazine's* Contributing Editors, Bill Todd and Cary Jensen, Ph.D.; and Richard Porter.

This article describes features of software products that are in development and are subject to change without notice. Description of such features here is speculative and does not constitute a binding contract or commitment of service.

BIO

Michael Swindell is a Director of Product Management at Inprise/Borland. He joined Inprise/Borland in 1997 working on C++Builder versions 3 through 5, and is currently directing the Kylix Project: Delphi and C++Builder for Linux. Previously, Michael was Senior Product Manager for raster imaging systems at Imation Software Publishing, where he specialized in PostScript processing and color printing systems. As Director of Software Development at RIPit Technologies, Michael managed the research and development of server-based digital pre-press imaging systems, and authored such technologies as the Enhanced Rational Tangent Screening algorithm (ERT), and the OnTarget halftone linearization system. When Michael isn't developing products, or speaking to developers, he is listening to CDs and MP3s, or surfing at Pleasure Point in Santa Cruz, California.

BIO

Chuck Jazdzewski was one of the two original developers of Borland Delphi, and is Chief Architect of Delphi and Kylix. He has worked on several products for Inprise, including C++Builder, JBuilder, Borland Pascal, and Turbo Pascal. Chuck joined Inprise/Borland right out of college and has worked at the company for 13 years. He lives with his wife Kristin and three children, Jonathan, Joseph, and Rebekah, in Soquel, California.

table of CONTENTS