November 1997, Volume 3, Number 11

Sharing Memory

Creating and Managing Memory-Mapped Files

Cover Art By: *Doug Smith*

ON THE COVER

6 **Sharing Memory** — Gregory Deatz
Sharing memory (and information) among applications was easy in 16-bit Windows. Although it's a bit more complicated in the 32-bit world of Win95, Mr Deatz demonstrates how memory-mapped files get the job done — and how to work with them in Delphi 2/3.

FEATURES

9 **Informant Spotlight**
Getting the Message — Robert Vivrette
Mr Vivrette shows us how to keep multiple Windows applications in touch, with the adept use of the Windows API and memory-mapped files.

15 **DBNavigator**
The Decision Cube — Cary Jensen, Ph.D.
The Decision Cube components are a powerful suite of controls for manipulating and displaying data for decision-support applications. Dr Jensen provides an introduction.

20 **Delphi Reports**
Avoid Report-Engine Overkill — Warren Rachele
Developers must respond to conditions; simple output calls for a simple solution. Here are some approaches for manipulating Windows printer functions, rather than loading a lumbering report engine.

24 **The API Calls**
Using the BDE API: Part I — Bill Todd
Although much of the Borland Database Engine's API is surfaced in Delphi, some useful tools are available only by calling the API directly. Mr Todd begins a two-part series.

30 **At Your Fingertips**
Disassembly View Revealed — Robert Vivrette
Activating an undocumented view of the assembler Delphi generates, defeating screen savers, and restarting applications upon Windows start-up are some of the treats Mr Vivrette has in store this month.

33 **OP Tech**
Multi-Tier Development — Ron Mashrouteh
Delphi 3's *interfaces* are the building blocks of COM. Learn how this group of semantically related routines aids multi-tier development, by following our simple database-application example.

REVIEWS

37 **WebHub**
Product Review by Peter Hyde

42 ***Rapid Development***
Book Review by Alan Moore, Ph.D.

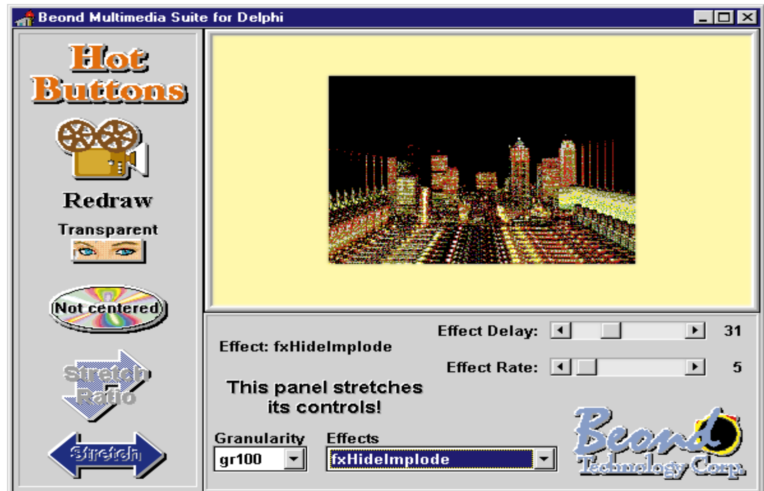
DEPARTMENTS

2 **Delphi Tools**
5 **Newsline**
44 **File | New** by Richard Wagner

Delphi
T O O L S

New Products and Solutions

Beond Technology Releases Multimedia Suite



Beond Technology Corp. announced the ***Beond Multimedia Suite***, a new component suite for Delphi that enables developers to add graphical effects and buttons to their applications. The suite includes three components: THotButton, TImageFX, and TStretchPanel.

THotButton can create 3D up/down buttons from any bitmap. It features transparency, stretch, 3D caption with word wrap, center, visual disable indication, and more.

TImageFX provides over 40 flicker-free bitmap transition effects, including explode, implode, slide, pixelation, and others. Bitmaps can have a transparent color defined by selecting a pixel location. Bitmaps can be loaded from a file, resource, or DLL. Variable granularity and speed allow effects to be optimized for bitmap size.

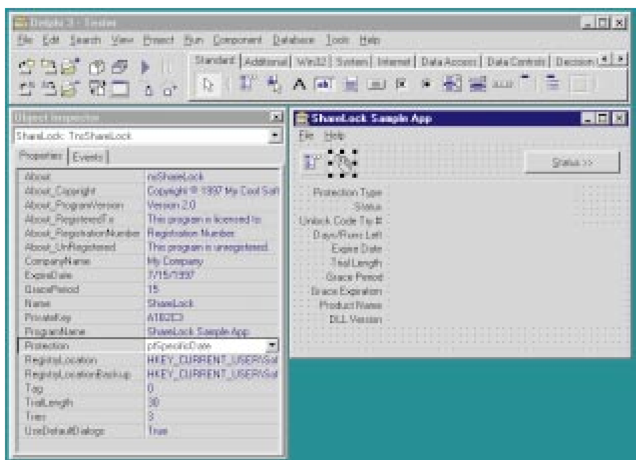
TStretchPanel is an elastic-like panel that automatically resizes the controls placed on it. This component also works with variable-sized graphic windows, such as map viewing. TStretchPanel can be nested for selective control resizing in a form.

Beond Technology Corp.
Price: Beond Multimedia Suite, US$159 (US$449 with source code); THotButton, US$69 (US$269 with source code); TImageFX, US$69 (US$269 with source code); and TStretchPanel, US$29 (US$129 with source code).
Phone: (773) 388-3771
E-Mail: brianlow@mcs.com
Web Site: http://www.mcs.net/-~brianlow/beond.html

Remember All Suite

MountainTop Systems has released the Remember All Suite, two components that simplify the entry and storage of user preference and configuration information. This information is usually a mixture of strings, booleans, checkboxes/radio buttons, or floats, dates, etc. TRememPanel, a descendant of *TPanel*, holds a PageControl with other setup recording controls and saves the data to an .INI file. Multiple configurations are also automatically handled. Other components enhance the data storage options. Remember All Suite is priced at US$38, or US$65 with source code. For a free trial version visit http://www.ozemail.com.au/~mtntop.

Nesbitt Software Offers ShareLock Online



Nesbitt Software Corp. began online sales of *ShareLock 2.0* for Windows 95 and NT. ShareLock is a component that turns any 32-bit Windows application into a trial version with as little as one line of code. ShareLock 2.0 includes ActiveX, VCL, and DLL versions for use with any Windows programming language.

ShareLock enables programmers to lock their software after a specified number of days or executions, or after an absolute date.

It also provides an optional grace period and special extension key codes. Programmers can use ShareLock's built-in data encryption, key generation, and dialog boxes, or may override any of these features to provide greater security or customized messages.

ShareLock also watches for users attempting to defeat the locking mechanism by using extension codes more than once, changing the date on their system, etc.

Nesbitt Software Corp.
Price: US$199.95
Phone: (619) 259-4700
Web Site: http://www.nesbitt.com

Delphi
T O O L S

New Products
and Solutions

DTalk Brings Speech to Delphi Applications

Out & About Productions has released DTalk, a set of speech-enabling components that use the Microsoft Speech API. These native Delphi components enable programmers to add both speech recognition and text-to-speech to applications. Expected uses include talking interface elements such as error messages, controls, online Help, and ad hoc database reports, as well as voice-driven menus, macros, and agents. Working with AT&T, DEC, IBM, and others, Microsoft has published the Speech API (SAPI) specification, which has become the standard for using speech engines and applications.

Because the DTalk components are SAPI-based, Delphi developers can select from various SAPI-compliant speech engines.

DTalk is priced at US$350. For more information, visit http://www.o2a.com or call (415) 695-9935.

Top Support Ships TopGrid

Top Support has released *TopGrid*, a new VCL grid component for Delphi 2 and 3. TopGrid can replace the DBGrid and StringGrid components. It supports standard Windows behavior, including normal scrolling and the ability to use headings as buttons.

The appearance can be customized by setting fonts, colors, alignment, and 3D effects, either on the grid level or for individual rows, columns, or cells. Headings can be multi-line, and the number of fixed rows or columns can be set. Also, rows and columns can be set to visible or invisible, and read-only or editable. The display order of columns can be set, and with the unbound version the display order of rows can also be changed.

Customer Number	Large Account	Company	Flag	Address
2118		Blue Sports Club		63365 Nez Perce Street
3054		Catamaran Dive Club		Box 264 Pleasure Point
1354	✓	Cayman Divers World Unl		PO Box 541
5151		Central Underwater Suppl		PO Box 737
2156	✓	Davy Jones' Locker		246 South 16th Place
3055	✓	Diver's Grotto		24601 Universal Lane
3041	✓	Divers of Blue-green		634 Complex Ave.
2315	✓	Divers of Corfu, Inc.		Marmoset Place 54
4312	✓	Divers of Venice		220 Elm Street
5432	✓	Divers-for-Hire	Canada	
1513	✓	Fantastique Aquatica	Japan	-77 A.A.
3151	✓	Fisherman's Eye	Spain	
2135	✓	Frank's Divers Supply	Sweden	th St.
2975	✓	George Bean & Co.	UK	on Way
3042		Gold Coast Supply	US	h Place
1680		Island Finders		6133 1/3 Stone Avenue
1651	✓	Jamaica SCUBA Centre		PO Box 68
4652	✓	Jamaica Sun, Inc.		PO Box 643
1221		Kauai Dive Shoppe		4-976 Sugarloaf Hwy
2354	✓	Kirk Enterprises		42 Aqua Lane
5165	✓	Larry's Diving School		3562 NW Bruce Street
1624	✓	Makai SCUBA Club		PO Box 8534

TopGrid supports edit boxes, check boxes, combo boxes, picture boxes, and cell buttons. The combo box provides incremental searching and can be multi-column supporting the cell styles and appearance options of the grid itself. Memos can be edited directly in the grid.

TopGrid can be used as a normal grid for editing or viewing data, or as a multi-column list box with single or multi-line selection. A demonstration program and trial version can be downloaded from Top Support's Web site. At press time, Top Support planned to ship a version of TopGrid for C++Builder in early November.

Top Support
Price: US$249, with source add US$99.
Phone: 31-10-4513941
E-Mail: contact@topsupport.com
Web Site: http://www.topsupport.com

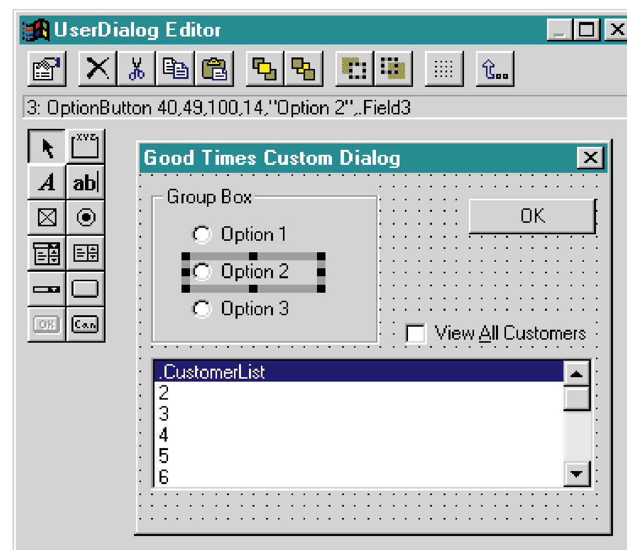
Sax Software Announces Sax Basic Engine 4.0

Sax Software is shipping *Sax Basic Engine 4.0*, an ActiveX control that integrates with Delphi, Visual Basic, and Visual C++ applications.

Version 3.0 of Sax Basic Engine added support for classes, events, and multiple modules. Version 4.0 builds on the previous version by adding support for multi-threading, support for Visual Basic 5, and enhancements to the Object Browser.

Sax Basic Engine's support for multi-threading features allows multiple macros to run simultaneously. Users of C++ and the MFC will benefit from the included documentation and C++ samples. This allows MFC developers to create applications that include a macro language, such as Microsoft Word and Excel.

MFC developers can also use the MFC Class Wizard to create ActiveX objects that extend the Sax Basic language with their own commands and functions, as well as integrate the language engine with custom applications.



Sax Software
Price: Sax Basic Engine 4.0, US$495. Sax Basic Engine ships with an unconditional 30-day money-back guarantee, and requires no royalties or run-time fees.
Phone: (800) 645-3729 or (541) 344-2235
E-Mail: info@saxsoft.com
Web Site: http://www.saxsoft.com

Sequiter Software Releases CodeBase 6.3

Sequiter Software has launched *CodeBase 6.3*, incorporating CodeControls 3.0, Sequiter's 32-bit ActiveX data-aware controls. CodeControls 3.0 includes bound edit, list, combo, slider, and button controls, as well as a multi-functional navigator control.

CodeControls 3.0 was re-written from version 2.0. In addition to the existing set of controls (edit, combo, list, and data), version 3.0 added new slider and button controls. The slider control works like a scroll bar for a database, while the button control allows you to isolate one or more of the data-control buttons in the application window.

The data control has also been expanded to include one-click functionality for such tasks as re-indexing and compressing data files, saving and refreshing the current record, traversing the data file by pages, and more. CodeControls 3.0 also boasts an improved design-time interface with the introduction of property pages for modifying a control's property set.

For developers who continue to build 16-bit applications, CodeBase 6.3 will still ship with CodeControls 2.0.

CodeBase 6.3 includes source code and updated support for the latest compilers, including Delphi 3, Visual C++ 5, C++Builder, Visual Basic 5, and the Java 1.1 SDK.

Sequiter Software
Price: US$395
Phone: (403) 437-2410
E-Mail: info@sequiter.com
Web Site: http://www.sequiter.com

Luxent Software Ships SuccessWare Database Productivity Components

Luxent Software launched new versions of its SuccessWare Database Productivity Components, including *Apollo 4.0* for Delphi and *Artemis 4.0* for Visual Studio (which are bundled as the SuccessWare Database Engine [SDE4] Suite), and *DB Power 2.0*, a set of plug-and-play DBMS components for Delphi.

SDE4 adds language-independent User-Definable Functions (UDFs) to filter and index expression strings and 32-bit thread-safe operations.

Apollo and Artemis are direct-access (i.e. non-ODBC/SQL) database engines that don't require ODBC or SQL to access Xbase file formats. Apollo and Artemis operate no-code solutions and plug-and-play replacements for the BDE and JET, respectively.

DB Power is a set of over 35 DBMS controls for Delphi and C++Builder. Consisting of source code controls for Super grids, image buttons, Quicken-style lookups, and other programmer tools, DB Power allows database creation, editing, and management in design time and run time. Both the BDE and Apollo are equally supported.

Luxent Software
Price: Apollo, Artemis, or DB Power: full version US$199; upgrade version US$99. SDE4 Suite is US$349.
Phone: (888) 4LUXENT or (909) 699-9657
E-Mail: sales@luxent.com
Web Site: http://www.luxent.com

ExceleTel Adds Telephony to Delphi Applications

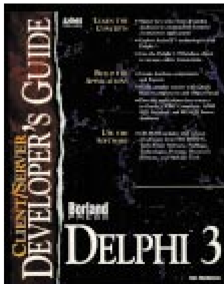
ExceleTel, Inc. offers the TeleTools product family, a Delphi VCL (and a set of ActiveX controls) for desktop computer telephony.

TeleTools can create server-based applications such as voicemail or interactive voice response, but its specialty is enabling desktop applications to work with telephones.

TeleTools works with Delphi 1, 2, or 3, C++Builder, and Microsoft Visual Basic (as well as other 32-bit environments with OCX support).

ExceleTel, Inc.
Price: ExceleTel TeleTools, US$199; ExceleTel TeleTools Express, US$49.
Phone: (919) 233-2232
E-Mail: sales@exceletel.com
Web Site: http://www.exceletel.com

Client/Server Developer's Guide with Delphi 3
Ken Henderson
SAMS Publishing

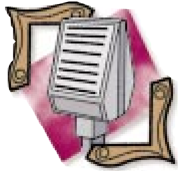


ISBN: 0-672-31024-4
Price: US$59.99
(967 pages, CD-ROM)
Phone: (800) 545-5914 or (317) 228-4231

News

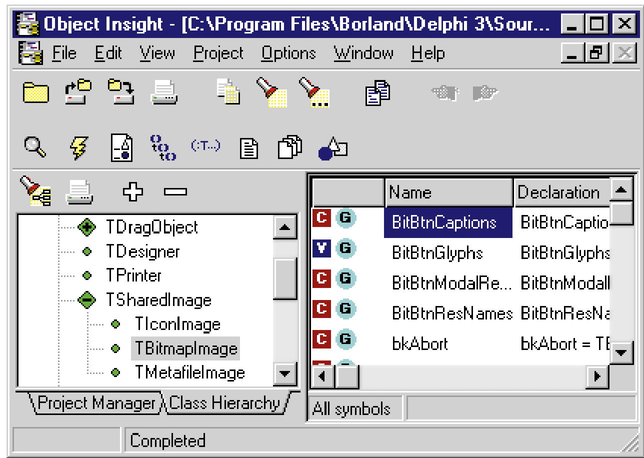
L I N E

November 1997



Borland Releases New ObjectInsight for Delphi 3

Scotts Valley, CA — Borland has shipped ObjectInsight for Delphi 3, a Delphi source-code navigation system that browses the origin and implementation of object class properties, methods, and events. An internal tool developed for the Delphi team, ObjectInsight enables users to understand their VCL and class libraries.



Navigating the Delphi Visual Component Library (VCL) source code provides insight into the techniques used by the Delphi developers when building the VCL. ObjectInsight supplements the Delphi Help system by providing a visual object framework from which to drill into the Delphi source.

ObjectInsight allows users to run queries against all information in a project. It features customized reports, allows developers to create custom project files that include source code from multiple locations in unrelated units, and maintains several views of a project at a time.

Using ObjectInsight, users can view all available variables, methods, properties, and events of a class in their order of declaration and the class hierarchy, including their implementation levels. This tool allows developers to jump to the declaration or implementation of any procedure, function, class, type, or variable using the built-in viewer or Delphi 3's editor.

In addition, ObjectInsight can compare versions of source units in a format-independent manner, reflecting changes in informational content only.

ObjectInsight is currently US$79.95. To order, call (800) 453-3375 or visit http://www.borland.com.

Borland Joins Java Development on IBM's San Francisco Project

Nashville, TN — Borland and IBM announced a combined R&D effort to develope Java applications with JBuilder and IBM's San Francisco Project.

Borland and IBM are working to improve Java development using visual tools, wizards, and object-oriented components. The partnership also allows JBuilder to be installed world-wide at IBM San Francisco education centers.

For information, visit http://www.ibm.com/java/-SanFrancisco/.

Luxent Development Merges DFL Software and SuccessWare International

Los Angeles, CA — Luxent Development Corp. has acquired DFL Software in Paris and SuccessWare International in California. DFL Software, known for its Light Lib graphical ActiveX software components and AS/400 middleware, recently released its new Magic Menus development utility. SuccessWare develops database-engine products, including Apollo for Delphi.

Luxent is an international business software integrator with principal sales, training, and support offices in California and Paris, in addition to its main research-and-development facility in Provence, France. For details, visit http://www.-luxent.com.

Borland Expands AS/400 Partner Program

Nashville, TN — Borland announced GSE Erudite, an AS/400 software consulting firm, has joined the Borland Partner/400 program. World-wide membership now includes 41 solution providers in 28 countries.

The Borland Partner/400 program is designed to provide sales, marketing, and support for IBM AS/400 customers using Borland's AS/400 development tools, including the Delphi/400 Client/Server Suite and C++Builder/400 Client/-Server Suite.

A listing of Borland/400 partners is located at http://www.borland.com/-borland400/partner.html.

GSE Erudite, headquartered in South Jordan, UT, is a wholly-owned subsidiary of GSE Systems. For details, visit http://www.erudite.com.

Corel Licenses Borland Technology

Corel announced it will license Borland's SQL Builder, SQL Links, InterBase, and Local InterBase for inclusion with Corel Paradox 8. For more information, visit the Corel Web site at http://www.corel.com.

Borland Previews Visual dBASE 7

Visual dBASE 7, an upgrade to its PC database for Windows 95 and NT, is scheduled to ship this fall. This 32-bit version of dBASE will support ActiveX components, a new two-way object-oriented report designer, and visual productivity tools, as well as native connectivity to Paradox, Access, FoxPro, and SQL database servers. For details, visit http://www.-borland.com/VdBASE/.

Delphi / Shared Memory

By Gregory Deatz

Sharing Memory

Creating and Managing Memory-Mapped Files

Sharing memory under Windows 3.1 was as simple as creating a DLL with global variables. In Win32, global memory spaces are private to each process, so it's no longer this simple; however, the Windows API provides a simple, safe, and fast mechanism for sharing memory: the memory-mapped file. Using mutexes (a way of specifying critical code between processes), the developer can create robust applications that can safely read and write shared memory.

This article demonstrates how to create memory-mapped files and manage them in critical sections identified using mutexes. It will also go one step further, showing how to wrap these API calls in easy-to-use, reusable objects, including a few surprises along the way.

The Mechanisms

Win32 introduced the flat address space to the Windows programming community; with it came the private address space. You might have seen Delphi's ShareMem unit mentioned in its online documentation. Contrary to its name, it doesn't allow you to "share memory;" rather it's a "replacement" memory-manager that allows you to pass Delphi's **string** type back and forth between DLLs and their calling applications. Delphi doesn't inherently provide a shared-memory object (i.e. there is no *TSharedMem* object packaged with Delphi); however, the Windows API provides an easy-to-use, low-level mechanism for sharing and managing memory.

These two mechanisms are called memory-mapped files (MMFs) and mutexes. MMFs enable developers to allocate sharable memory within the operating system's paging file; mutexes provide a mechanism for managing civilized access. (Actually, MMFs do much more than share memory; they provide a tool for mapping entire files into virtual memory, allowing the developer to treat large files as large arrays of bytes; but that's an entirely different discussion.) The MMF is the only mechanism in Windows that can share memory between processes. ActiveX, OLE, or any other tool you can imagine for sharing memory between applications uses MMFs at its core.

Mutexes allow the developer to specify critical sections of code that will be executing in separate processes. A critical section of code is a piece of code that performs some operation that may interfere with activities occurring in other threads. For example, when shared memory is being accessed, it's important that a given thread can operate on the shared memory exclusively; thus it is "critical."

(Mutexes can be used in a single process to manage synchronous access to global variables, but it's much cheaper to use Delphi's *TThread.Synchronize* method, or to use another suite of native API system calls: **InitializeCriticalSection**, **EnterCriticalSection**, **LeaveCriticalSection**, and **DeleteCriticalSection**. These system calls manage the application-level CRITICAL_SECTION, and are well documented in Win32.hlp.)

The Memory-Mapped File

Project Example1.dpr provides a simple example of how memory-mapped files work. We use five simple system calls. The first is **CreateFileMapping**:

```
function CreateFileMapping(hFile: THandle;
  lpFileMappingAttributes: PSecurityAttributes;
  flProtect, dwMaximumSizeHigh, dwMaximumSizeLow: DWord;
  lpName: PChar): THandle;
```

where *hFile* is a valid file handle to the file to be mapped. To map a file is to effectively load an entire file into virtual memory. This allows the developer to treat a file as a simple array of bytes. To create true shared memory, we don't want to map an actual file; we only want to use this memory-mapping feature so we can create shared memory.

The special file handle $FFFFFFFF (-1) tells Windows we want to map space from its paging file. In other words, it tells Windows, "Give me a file mapping that doesn't really map a file. Just give me a memory space that can be shared between processes."

lpFileMappingAttributes describes the security attributes the developer wishes to associate with the mapped file. (A discussion of this argument is beyond the scope of this article.) *flProtect* indicates the desired file protection for the view. This translates into opening a file read-only, write-only, or read-write. For the purposes of our article, we consider only read-writable views.

dwMaximumSizeHigh and *dwMaximumSizeLow* indicate the number of bytes the developer wishes to allocate for the MMF. It provides a total of 64 bits for this value, so you are virtually unlimited in the size of the file you wish to map. For our purposes, *dwMaximumSizeHigh* is always 0. *lpName* names the file map. Whenever the developer wishes to refer to this MMF, he or she must use this name.

The next call is to **MapViewOfFile**:

```
function MapViewOfFile(hFileMappingObject: THandle;
  dwDesiredAccess, dwFileOffsetHigh, dwFileOffsetLow,
  dwNumberOfBytesToMap: DWord): Pointer;
```

where *hFileMappingObject* is a valid handle to an MMF returned by **CreateFileMapping**, and *dwDesiredAccess* is the access the developer desires to have against the particular MMF. This is roughly equivalent to opening a file read-only, write-only, or read-write. Again, we concern ourselves with read-writable data only. For the purposes of this article, *dwFileOffsetHigh*, *dwFileOffsetLow*, and *dwNumberOfBytesToMap* are all to be set to zero.

Given a valid pointer to an address returned by **MapViewOfFile**, the **UnmapViewOfFile** function unmaps the view of the MMF:

```
function UnmapViewOfFile(lpBaseAddress: Pointer): BOOL;
```

The **CloseHandle** function is used to close the *hFileMap* returned by **CreateFileMapping**:

```
function CloseHandle(hObject: THandle): BOOL;
```

and **GetLastError** returns the most recent error that occurred in Windows.

```
function GetLastError: Integer;
```

This function is used specifically to determine the error status of our previously-mentioned API calls. Use this function to determine the status of the **MapViewOfFile** function.

In Unit1.pas of Example1.dpr, the form's *OnCreate* event is used to initialize and map a view of the MMF:

```
hFileMap := CreateFileMapping($FFFFFFFF, nil,
              PAGE_READWRITE, 0, share_len + 1,
              'MyMemoryMappedFile');
...
SharedPChar := PChar(MapViewOfFile(hFileMap,
                 FILE_MAP_READ + FILE_MAP_WRITE, 0, 0, 0));
```

As you can see, we map *share_len* + 1 bytes of memory into the paging file and specify that we want to be able to read and write it. We call this MMF "MyMemoryMappedFile." Then, using **MapViewOfFile**, we can map a view of the file into the application's address space, thus making *SharedPChar* a shared null-terminated string of length *share_len*. The extra byte is for the null character (#0).

Using the **GetLastError** function, we can determine if the MMF was created, or if it already exists. If the MMF was just created, we can initialize it like this:

```
if GetLastError <> ERROR_ALREADY_EXISTS then
  StrPLCopy(SharedPChar, 'Hello world', share_len);
```

Then, we set the value in our Edit control:

```
Edit1.Text := SharedPChar;
```

Now that the MMF has been created and initialized, we can treat *SharedPChar* as if it's just another pre-allocated character array. The click event for *btnSet* merely copies the data in the Edit control to *SharedPChar*, and the click event for *btnRefresh* copies the data in *SharedPChar* to the Edit control.

By launching two or more instances of Example1.dpr, you will see that, indeed, the MMF called *MyMemoryMappedFile* shares its data between processes.

Now, we want to clean up after ourselves; the form's *OnDestroy* event unmaps and closes the MMF:

```
UnmapViewOfFile(SharedPChar);

CloseHandle(hFileMap);
```

The Mutex

Now that we've seen how we can share memory between processes, the next question is, "How do we manage this shared memory to ensure orderly access to it?" Obviously, we want predictable results from our MMF, so the prospect that two processes could be writing to our MMF at the same time is unacceptable.

Mutexes provide mutually exclusive access to critical sections of code across threads and across processes. Example2.dpr shows very minor modifications to the original project, Example1.dpr. We introduce a new handle, *hAccessMutex* (which allows us to instantiate a mutex), and two new system calls, **WaitForSingleObject** and **ReleaseMutex**.

We create the mutex using **CreateMutex**:

```
hAccessMutex := CreateMutex(nil, False,'MyAccessMutex').
```

The first argument tells the function simply to use default security, and the final argument names the mutex. The interesting argument here is the second argument, which tells **CreateMutex** if it should make this thread the initial owner of the mutex.

When a thread owns a mutex, any other thread (in any process) waiting on the mutex cannot execute. This means we can wrap all access to the MMF in two system calls. In the following code:

```
WaitForSingleObject(hAccessMutex, INFINITE);
try
  ... // do your stuff.
finally
  ReleaseMutex(hAccessMutex);
end;
```

the **WaitForSingleObject** function will wait as long as necessary (potentially forever, as indicated by the INFINITE constant) for *hAccessMutex* to be "disowned." Windows guarantees that only one process or thread can own a mutex at a time, so as soon as **WaitForSingleObject** returns, we are confident we have exclusive access to the MMF. Because all other threads that wait on *hAccessMutex* will wait for this process to relinquish ownership of the mutex, we must guarantee that **ReleaseMutex** will be called. Hence, we use the **try..finally** construct.

Getting Fancy

At this point, we can stop. We have created an MMF and we have protected access to the MMF using mutexes. However, it would be nice if we could wrap this up in an easy-to-use Delphi object. Our goals for the object are to:

1) share a piece of memory,
2) manage access to the shared memory, and
3) be alerted when the shared memory has been altered by another thread.

We already have what we need to build an object that does items one and two, but accomplishing three requires a few more tricks from the Windows API. In particular, we need to use threads and events.

As the reader already knows, a thread is the most basic Windows object that is given CPU time. All processes have at least one thread, but Win32 allows a single process to have many threads of execution. As the reader may not already know, events allow a thread to wait for a specified event to occur.

Example3.dpr implements an object called *TSharedMem*, which provides the base functionality for sharing any data type, as well as two end-user objects, *TSharedString* and *TSharedInt*, which are both descended from *TSharedMem*.

The code available for download (see the end of the article for details) is heavily commented so that examination of the source will reveal all the intricacies involved. We will provide a cursory explanation of the code here, but it's left to the reader to examine the code for a complete understanding.

When an object of type *TSharedMem* is instantiated, it creates handles for an event, an MMF, a mutex, a thread, and a critical section. The thread responds to broadcasts of the event (an event is broadcast using the Windows API **PulseEvent** function), which indicates that a process or thread has modified the shared object. This thread then locks the shared buffer, calls an *OnChange* event if one was provided, and subsequently unlocks the shared buffer.

Conclusion

Sharing memory in Win32 is a bit more complicated than it was in Windows 3.1, but it's still a relatively simple process. In fact, we have successfully navigated the terrain of quite a few API tools (MMFs, mutexes, events, threads, and critical sections), all of which have played an integral role in implementing a robust, easy-to-use object that shares memory across processes. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\NOV\DI9711GD.

Gregory Deatz is a senior programmer/analyst at Hoagland, Longo, Moran, Dunst & Doukas, a law firm in New Brunswick, NJ. He has been working with Delphi and InterBase for approximately two years and has been developing under the Windows API for approximately five years. His current focus is in legal billing and case-management applications. He is the author of FreeUDFLib, a free UDF library for InterBase 4.x written entirely in Delphi and hosted at http://www.borland.com/-devsupport/interbase/download/FreeUDFLib.html. In addition to being published in *Delphi Informant,* he has also been published in *Optical Computing.* He can be reached via e-mail at gdeatz@skyweb.net, by voice at (732) 545-4717, or by fax at (732) 545-4579.

By *Robert Vivrette*

Getting the Message

Techniques for Communicating between Applications

Do you remember passing messages in school? You would scribble a love note or tidbit of gossip on a little scrap of paper, fold it up, and write a name on it. Then you would hand it to a friend, who would hand it to another friend, and the note would wend its way across the room until it (hopefully) arrived at its intended destination (and not on the teacher's desk!).

But passing messages isn't just something grade-schoolers do. All good Delphi programmers know that Windows is a message-based operating system and that message-passing is the name of the game in the Windows programming world.

This article will discuss some basic principles for communications between applications. A term you might often hear applied to this discussion would be *Interprocess Communications*, or IPC. This term is a pretty broad description of any kind of communication between separate applications or processes. Some of these techniques include (but are not limited to) Windows messaging, shared memory, shared files, DDE, OLE, memory-mapped files, Windows atoms, the Clipboard, and others. All of these are mechanisms by which one process can send data to another. In light of how applications are becoming more modular, it's only natural that we see increasing attention being placed on the ability of applications to talk to one another.

This article will discuss, in some depth, the topic of Windows messaging. I will also touch on the topic of memory-mapped files, but for more details on this second technique, refer to Gregory Deatz's article "Sharing Memory" on page 6. Both techniques are powerful and easy to implement for sending data between applications. Before we start, however, let me say that there are many ways of achieving the effects in the samples I have provided. However, my intention is to give an easy-to-understand tutorial on just these two techniques.

Windows Messages

No doubt many of you are familiar with how Windows talks to itself. Essentially, every aspect of a Windows program's behavior is the result of a Windows message in one form or another. For example, when you enter data into the Notepad application, there are messages flying all over the place. There are messages that track when keys go down, when keys go up, when the mouse moves a pixel, or when its buttons are clicked, just to name a few.

For the most part, however, the average Delphi programmer generally only deals with messages passed around within a single application; even then you might not really think they are messages. For example, your form might have a message handler called *OnMouseDown*, but Delphi is really responding to a WM_MOUSEDOWN Windows message, and is passing it along through a hierarchy of message-handling locations. Ultimately, the message gets to your form, and control is passed into your *OnMouseDown* routine. You could construct a fake WM_MOUSEDOWN message and send it to your form, by using the **SendMessage** API call, and your form wouldn't know the difference. It got a message; that's all it's concerned about. Sending

Figure 1: Test Application #1.

messages between applications is only slightly different than sending messages inside an application.

Before we get too far in this discussion, let's define a Windows message. A *Windows message* is simply a record structure that Windows uses to communicate information between controls, forms, windows, etc. The message structure first has a field named *HWnd*, which is a handle to the window or control to which the message is directed. Next comes the *Msg* field, which is the actual message number being sent. In Delphi, these messages are defined in the Messages.pas unit. When you see messages such as WM_MOUSEDOWN, this is really only a constant that has been defined to make the purpose of the message more intelligible.

Next come two fields for passing message-specific data along with the message. These are named *wParam* and *lParam*. In 16-bit Windows, *wParam* was a Word and *lParam* was a Longint; in Win32, they are both Longints. At different times in a message's life, it might have additional data associated with it — such as when the message was created, where the mouse cursor was at the time, and so on. However, I don't want to make this discussion too complex, so I'm going to skip these features.

As I said earlier, a message can be sent between applications just as easily as it can to controls of forms within an application. All you need to know is the window handle of the receiver. In the case of sending a message to another application, you only need to know that application's window handle, to identify it as the recipient of the message.

Talk Amongst Yourselves

To illustrate the techniques used in this article, I put together a sample application. And to make it clearer that we are communicating between two distinct applications, I have established two Char constants: *ThisApp* and *OtherApp*. I then made two copies of this application and its supporting .DFM and .PAS files. In the first copy, I set *ThisApp* to "1" and *OtherApp* to "2." In the second copy, I set *ThisApp* to "2" and *OtherApp* to "1." When these two programs are compiled, they will be functionally the same, but one will know it is

```
procedure TForm1.LookForApp(StartLooking: Boolean);
begin
  if StartLooking then
    lblOtherApp.Caption := 'Looking...'
  else
    lblOtherApp.Caption := 'Test Application #' + OtherApp;
  if StartLooking then
    TheHWnd := 0;
  Timer1.Enabled := StartLooking;
  btnStringSend.Enabled := not StartLooking;
  btnValueSend.Enabled :=  not StartLooking;
  btnBitmapSend.Enabled := not StartLooking;
  OtherAppFound := (TheHWnd <> 0);
  Image1.Refresh;
  Image2.Refresh;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  TheHWnd := FindWindow('TForm1',
                        'Test Application #' + OtherApp);
  if TheHWnd <> 0 then
    LookForApp(False);
end;
```

Figure 2: Looking for another application.

Test Application #1 and the other is Test Application #2. To highlight this difference at run time, this constant is used to create the title of the application. Figure 1 shows Test Application #1; of course, Test Application #2 would look the same, but would have Test Application #2 as its title. Note also that each application is designed to both send and receive data. The left side of the application shows the data it is sending, and the right side shows the data it is receiving.

So our first step — in setting up some kind of communication from one program to the other — is to find the other application. You can't send a message to someone if you don't know who or where they are. We shouldn't require the user to be involved in this process, so let's have the programs do the work. To automate this process, we will use the **FindWindow** API call within a timer loop. When one application is launched, it starts the timer, and begins looking for its partner using **FindWindow**. Once the second application is launched, it starts doing the same thing.

At this point, they are both running, so the next tick of each one's timer will cause them to find each other. The two routines responsible for this search are called *LookForApp* and *Timer1Timer* (see Figure 2).

When the application first runs, it calls *LookForApp* in its *FormCreate* method, with a *StartLooking* parameter set to *True*. Inside *LookForApp*, we set a Label on the form to indicate we're looking, and also enable or disable the three **Send** buttons on the form. The timer is also turned on. On each tick of the timer, the *Timer1Timer* function is called. Here we make a call to **FindWindow** to look for the other application. The first parameter is the form's *ClassName*. In this case, both of them are the class *TForm1*. The second parameter is the form's banner title. If Test Application #1 was running, it would be looking for Test Application #2. If the window is found, **FindWindow** returns its handle. If not found, it returns zero. Therefore, if we get back a non-zero number, it

```
TForm1 = class(TForm)
  ...
  public
    procedure WndProc(var TheMsg: TMessage); override;
  end;

procedure TForm1.WndProc(var TheMsg: TMessage);
begin
  if TheMsg.Msg = TheMsgVal then
    begin
      { It's our custom message -- deal with it! }
    end;
  inherited WndProc(TheMsg);
end;
```

Figure 3: Overriding the *WndProc* method.

means the other application was found, and we need to turn off the timer. We do this by calling *LookForApp* again with the *StartLooking* parameter set to *False*. This enables the three **Send** buttons, and sets a Boolean variable indicating the other application was found. The handle of the other application is saved in *TheHWnd*.

At this point, both applications are running, and each knows the window handle of the other. In a Windows application, there is a procedure that's used as a kind of clearinghouse for all message traffic. This is the *WndProc* procedure. If you want to create your own inter-process message, you will need to override this procedure, and listen for the message you have defined. That brings us to the next topic: defining our message.

Defining the Message

Earlier I mentioned that all the standard Windows message names are really just constants. For example, WM_MOUSEMOVE is the hex value $0200. Of course, $0200 doesn't tell us much. To define our own message, we must get such a value, but we need to make sure it doesn't collide with any of the pre-defined values. If it's a message that stays within our application (perhaps a message to or from a control), message collision isn't really a problem; we simply make sure we don't define a message constant that conflicts with another in that single application. However, if you're communicating to another application, you should make sure the message constant is unique to those two applications. So what value do we pick for our message constant?

Microsoft has defined that user messages should always reside above hex $0400 (as defined by the constant WM_USER). You might then be inclined to just pick a number above WM_USER and use that as your message constant. However, when communicating between applications, this is not good practice, and in rare circumstances can cause collisions between user-defined messages that other applications are using. Windows resolves this problem with the **RegisterWindowMessage** function. It takes a single unique string parameter, and returns an unused message identifier. The valuable aspect of this function is that if two applications register the same string, the function will return the same message identifier to each one.

Therefore, at the top of the sample application, I defined a constant called *TheMsgConstant* and set it equal to "MyUniqueMessageConstant." The message identifier is then obtained in the application's *FormCreate* procedure by calling **RegisterWindowMessage**, and passing it this constant. The returned value is our unique message identifier and is saved in *TheMsgVal*.

Many of you have created message handlers for code in a single application. You defined your message constant at the top of the application as something like:

```
WM_MyMsg = WM_USER+1;
```

then wrote a message handler for it defined like this:

```
procedure MyVerySpecialMessage(var Message: TMessage);
  message WM_MyMsg;
```

We can write a message handler that specifically handles a message with the value *MyMsgConst*. Because this is a constant, the compiler knows how to compile it into the final executable. But in our case, we don't know the value until the program runs; we obtain it from **RegisterWindowMessage**.

Enter the *WndProc* method. As I mentioned before, this is a sort of clearinghouse for the application's messages. To listen in on a custom message, all we have to do is override the application's *WndProc* method and look for our message value (which is stored in the variable *TheMsgVal*). If it is that value, we handle it accordingly. If it isn't, it gets passed on to the inherited *WndProc* handler (the one that was overridden). Figure 3 shows pseudo-code for this overridden method.

So at this point, we have two running applications that are aware of each other and have registered a Windows message to communicate with. We have overridden the *WndProc* method, so we know when one of the messages comes along.

Sending a Number

In our first test — the easiest one — we're going to just send a number to the other application. As mentioned earlier, messages have two fields that can be used to pass custom data. In this case, let's use the *lParam* field of the message. As mentioned, *lParam* is a Longint, so we can pass any number that would fit in a long integer. Because I designed the test application to be capable of passing other things with this message, we'll also need to provide something that says this message is holding a numeric value in its *lParam* field. Therefore, we'll use the *wParam* field to hold one of four constants we have defined: *wpString*, *wpValue*, *wpBitmap*, or *wpShutdown*. (Keep in mind that we could have defined four different messages rather than just one. That would have made this mechanism of categorizing the message by means of the *wParam* field unnecessary.)

If you again look at Figure 1, you'll see one of our test applications running. To pass a number over to the second application, you simply enter a value in the **Value To Send** edit box and click on its associated **Send** button. When that button is clicked, the *btnValueSendClick* method is called; it performs two simple lines of code. The first is to verify that we have a

```
procedure TForm1.WndProc(var TheMsg: TMessage);
var
  ThePtr : PChar;
  TmpBmp : TBitmap;
begin

  if TheMsg.Msg = TheMsgVal then

    case TheMsg.wParam of

      wpString :
        if TheMapHnd <> 0 then
          begin
            ThePtr := MapViewOfFile(
                        TheMapHnd,FILE_MAP_WRITE,0,0,0);
            edtStringRec.Text := ThePtr;
            UnmapViewOfFile(ThePtr);
          end;

      wpValue :
        edtValueRec.Text := IntToStr(TheMsg.lParam);

      wpBitmap :
        begin
          TmpBmp := TBitmap.Create;
          try
            TmpBmp.Handle := TheMsg.lParam;
            Image2.Picture.Bitmap.Width := TmpBmp.Width;
            Image2.Picture.Bitmap.Height := TmpBmp.Height;
            Image2.Picture.Bitmap.Canvas.Draw(0,0,TmpBmp);
          finally
            TmpBmp.ReleaseHandle;
            TmpBmp.Free;
          end;
        end;

      wpShutDown :
        LookForApp(True);
    end;

  inherited WndProc(TheMsg);

end;
```

Figure 4: Completing the *WndProc* method.

valid handle to the other application. If so, we then send the message. The first parameter is the other application's window handle. The second is the custom message value we obtained from **RegisterWindowMessage**. The next value is the *wParam* field. In this case we are sending *wpValue,* which will tell the receiving application that it should expect a Longint in the *lParam* field. Lastly, we have the *lParam* field itself. All we do is convert the value of the edit box from a string to an integer, and send that value.

What happens on the receiving side? First, the message is delivered by Windows to the receiving application. When it gets there, it ultimately passes through the *WndProc* method that we have overridden. Now all we need to do is put a little more meat into that method. Figure 4 shows how we examine the incoming message to determine what it contains. After determining that it is indeed our message, we run the *wParam* field through a **case** statement to determine which flavor of our custom message we have received. In this case, the value in *wParam* will be *wpValue*, indicating that *lParam* is holding a Longint. Then, as you can see in the *wpValue* case in our *WndProc* method, we simply cast the message's *lParam* into a string, and place it in the receiver's edit box. Simple! Figure 5 shows both applications running, and we

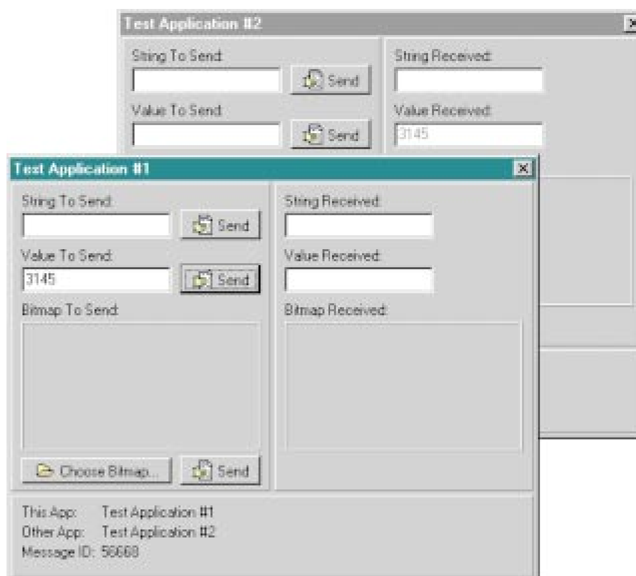


Figure 5: Sending the value 3145 to Test Application #2.

have sent the value 3145 from Test Application #1 to Test Application #2.

Sending a Bitmap

Now let's try something a little more complex: sending a bitmap from one application to another. The key component of a bitmap (as far as Windows is concerned) is its handle. With this handle, Windows can access the rest of the bitmap's data. Because a window's handle is just a Longint value, we are going to pass the bitmap's handle across applications in the same manner as the previous example. However, there is just a little more to this one.

As you can see in our test application, there is a button that allows you to select a bitmap. After you have made your selection, the bitmap is placed in a *TImage* under the label **Bitmap To Send**. When we click on the **Send** button, the *btnBitmapSendClick* method is called. It's essentially identical to the *btnValueSendClick* we discussed earlier; the only difference is that we pass the value *wpBitmap* as the *wParam* parameter, and the handle of the *TImage*'s bitmap as the *lParam* parameter.

If we refer back to Figure 4, we can look at how the *wpBitmap* case is handled. First, though, we must understand that this handle that has just been sent to us from the other application is still the handle of that *TImage*'s bitmap. Whatever we do to it will impact the other program. To ensure that we accept the other bitmap gracefully, we will first create a temporary bitmap, and assign its handle to the handle that we received in the message. Then we set the receiving *TImage*'s bitmap width and height to match. Then we copy the graphic.

Once we're done, we want to free our temporary bitmap. However, we don't want that handle to be destroyed, so we first call *ReleaseHandle*. This separates the handle from the bitmap, so when the bitmap is destroyed, the handle is unaffected. If we didn't include this statement, we'd destroy the

```
function TForm1.ObtainMappingHandle: THandle;
begin
  Result := CreateFileMapping($FFFFFFFF,nil,PAGE_READWRITE,
                              0,2000,TheMappingConstant);
  if Result <> 0 then
    // Did it return a valid handle?
    if GetLastError = ERROR_ALREADY_EXISTS then
      // Did it already exist?
      begin
        CloseHandle(Result);
        // Close this and we will open an existing.
        Result := OpenFileMapping(FILE_MAP_WRITE,False,
                                  TheMappingConstant);
      end;
end;

procedure TForm1.btnStringSendClick(Sender: TObject);
var
  ThePtr : PChar;
begin
  if (TheHWnd = 0) or
     (TheMapHnd = 0) then
    Exit;

  ThePtr := MapViewOfFile(TheMapHnd,FILE_MAP_WRITE,0,0,0);
  StrPCopy(ThePtr,edtStringSend.Text);
  SendMessage(TheHWnd,TheMsgVal,wpString,0);
  UnmapViewOfFile(ThePtr);
end;
```

Figure 6: Obtaining a mapping handle and sending a string using an MMF.

other application's handle to its bitmap when our temporary bitmap was destroyed. Doing it the right way, however, provides the result of both *TImages* containing the same image.

Memory-Mapped Files

Our last example is the use of memory-mapped files (MMF), which is simply a way of sending an arbitrary block of data between two cooperating applications. This is done by obtaining a handle to a common piece of memory space from the operating system. When both applications have a handle to this space, they can write information back and forth to each other. The concept of an MMF is similar to creating a disk file and writing out the data while letting the other application open it up and read it. However, MMFs have a number of advantages over this kind of mechanism. As I mentioned earlier, Gregory Deatz's article covers this topic in much greater detail, so I'm just going to hit the high points.

To illustrate this feature, we're going to send a string between the test applications. This string will be a Delphi 2/3 HugeString rather than the 256-character Delphi 1 string. The first step we must take when preparing to move data with an MMF is to create the file mapping. This will be a handle that Windows will use to access our mapped file space. I did this in the *FormCreate* method using a call to a small function I created called *ObtainMappingHandle* (see Figure 6).

The two Win32 API calls I use in this function are **CreateFileMapping** and **OpenFileMapping**. **CreateFileMapping** creates a new mapping, and **OpenFileMapping** opens one that already exists in the system. What I've done here is to try to create one first. If it succeeds, then I have my handle. If it fails, however, it returns the value ERROR_ALREADY_EXISTS by means of the **GetLastError** function. In this case, I then call **OpenFileMapping** to open the existing one. The result of my *ObtainMappingHandle* function is that it will open an existing mapping handle or create a new one if necessary.

When I call **CreateFileMapping**, I must pass in parameters to define the type of MMF I'm interested in. The second-to-last parameter in **CreateFileMapping** is *MaxMapLen,* which is a constant I defined to specify the maximum allowable length for our file. In the **const** section at the top of the application, I defined this arbitrarily as 2000, so at most we will be able to send a string no more than 2,000 characters long. The last parameter is *TheMappingConstant*, another unique string constant we have defined in both applications — specifically, the string "MyUniqueMappingConstant." Because both applications will use the same string, they will ultimately obtain a handle to the same MMF.

The return result of the function is a handle we'll use to refer to the MMF; the value is stored as *TheMapHnd*. I get the mapping handle in this fashion because the first application that runs is going to create the file mapping under the name of our unique string constant. When the second application runs, it won't be able to create one under the same name because it already exists in the system. Therefore, it opens the existing one instead.

Looking again at Figure 6, you will see the *btnStringSendClick* method. This method is called when the user enters a string, then clicks on its associated **Send** button. We first ensure that we have a valid handle to the other application and also have a valid handle to our file mapping. Then we open a view to the file with **MapViewOfFile**. Think of creating the file mapping as using the *AssignFile* method on a disk file, and the **MapViewOfFile** function like using *Reset* or *Rewrite* on that file. The return result from **MapViewOfFile**, however, is an actual pointer to the memory space used by the file mapping.

All we need to do now is put the string in it and let the other application know it's there. We first copy the string to the mapped area using *StrPLCopy* (which just copies the string, up to a maximum number of characters). Next we send a message to the other application telling it that a string is waiting for it in the MMF. You will note that we don't actually pass any data in the *lParam* field of our "string" message; the message simply notifies the other application that a string is available. After the message is sent, we close our view to the MMF with **UnmapViewOfFile**.

In this example, I'm using a Windows message to synchronize the transfer of data to and from the MMF. I place the data in the MMF's data area, then send a message to the other application telling it to go check. Because we're only dealing with two applications, this method will work well.

In a more complex environment, however, with many applications trying to hit the MMF at the same time, this

Figure 7: Our second test application after receiving the data.

simple mechanism would have to be replaced by using mutexes (again, see Gregory Deatz's article).

On the receiving application, we return to the *WndProc* method illustrated in Figure 4. Here the *wpString* case has been triggered by our new message. That tells us that the other application has placed a string in the MMF and is waiting for us to retrieve it. All we do is open up a view to the file with **MapViewOfFile** that returns a pointer to the start of the string. Then we assign that pointer to the *Text* property of our edit box. And presto! The string appears in our receiving application. The only thing left is to close our view of the file with **UnmapViewOfFile**. Figure 7 shows our second test application after it has received all three types of messages sent to it.

To give you one final thing to think about, ask yourself this question: If I send a string from Test Application #1 to Test Application #2, and then send a different string from Test Application #2 to Test Application #1, wouldn't the strings write over each other in the mapped file? Well, as it turns out, when you assign the pointer into the edit box's *Text* property, it makes a unique copy of the string on its own. Another of the wonders of Delphi!

Shutting Things Down

The last item I want to briefly cover is the fourth message type I defined, namely *wpShutdown*. Whenever one application closes, I first send this message to the other application to let it know that its sibling has terminated. That tells the receiving application to start up the timer looking for the other application again. That means that you can shut down one application and the other will immediately disable its buttons and start looking for the second application again. If you restart it, they will re-synchronize with each other.

Hopefully this article has provided a deeper understanding of how to communicate various types of data between applications. Believe me, there are plenty of other ways to accomplish these same tasks, but passing messages and using memory-mapped files are two of the easier ways your applications can talk. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\NOV\DI9711RV.

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

By Cary Jensen, Ph.D.

The Decision Cube

An In-depth Review of Its Features

The Client/Server edition of Delphi 3 introduces a collection of components designed for decision support. Collectively they are referred to as the Decision Cube. The core to these components is a non-visual component named DecisionCube, which stores data for display in grids and graphs.

While the role of a DecisionCube at first appears to be similar to a DataSet, it's different in a number of important ways. First, a DecisionCube stores data in memory, unlike a DataSet, which points to data. Second, a DecisionCube is designed to store only certain types of data — specifically, dimensions and summary statistics.

Dimensions are typically discrete, non-ordinal values without a natural order, such as Department within a company (accounting, research and development, and so forth), or religious affiliation (Muslim, Catholic, Protestant, and so forth). Summary statistics are measurements of those dimensions (e.g. departments may be measured for sum of annual budget or total number of employees, while religious affiliation may be measured for number of members or average member age).

A DecisionCube gets its data from a DataSet, which can be a Table, Query, or StoredProc component. It can also receive data from a specialized query component, called a DecisionQuery. The DecisionQuery provides a component editor that simplifies the process of defining the query. *TDecisionQuery* descends directly from *TQuery*, and doesn't introduce new properties or methods.

The data stored in a DecisionCube can be displayed in either a DecisionGrid or a DecisionGraph (i.e. a TeeChart component). The DecisionGrid is a tabular control that displays rows and columns. A minimum of one row or column within the DecisionGrid is associated with at least one dimension in the DecisionCube. One or more cells in the DecisionGrid display the summary statistics calculated on the displayed dimension(s). Data of this form, displayed in this format, is often called a crosstab, a cross-tabulation, an xtab, or another, similar name.

An Overview

DecisionGraph is used to graph the summary statistic by the dimensions stored in the DecisionCube. You can use one or more DecisionGrids and DecisionGraphs with a single DecisionCube. The DecisionGrids and DecisionGraphs don't connect directly to a DecisionCube. Instead, there is an intermediary component called a DecisionSource. The role played by a DecisionSource between a DecisionGrid (or DecisionGraph) and a DecisionCube mimics a DataSource component working between data-aware controls and a DataSet.

There is only one more Decision Cube component: the DecisionPivot, which is analogous to a DBNavigator, in that it permits the user to more easily manipulate a non-visual control (the DecisionCube, in this case).

There are a number of steps to consider when using the Decision Cube components:

- Define a DataSet that includes at least one dimension field and one summary field.
- Configure a DecisionCube using the data from the DataSet.
- Point a DecisionSource to the DecisionCube.

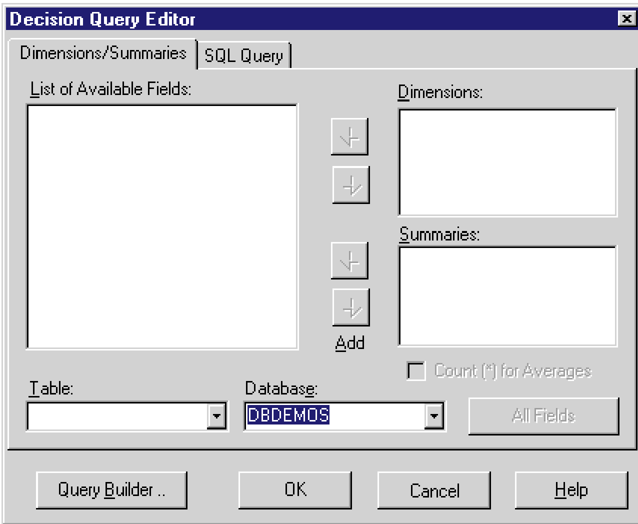


Figure 1: The Decision Query Editor.

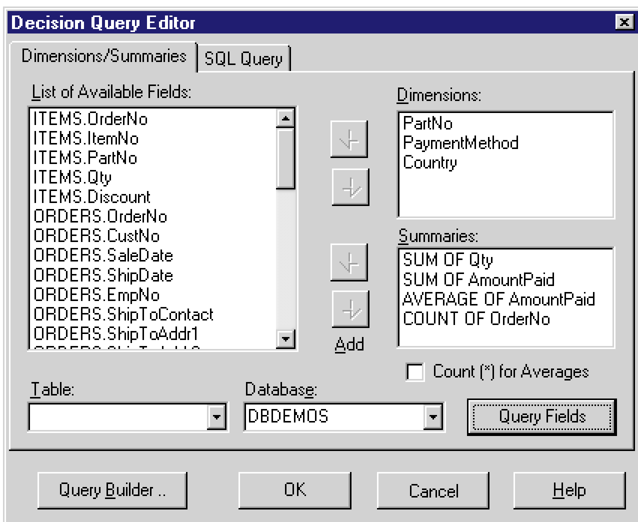


Figure 2: The Dimensions/Summaries page showing all fields from the tables participating in the query.

- Attach at least one DecisionGrid or DecisionGraph to the DecisionSource.
- Optionally, you can attach a DecisionPivot to the DecisionSource.

These steps are described in the following sections.

Defining the DecisionQuery

The data loaded into a DecisionCube must possess certain characteristics. There must be one or more fields representing dimensions of interest, and there must be one or more fields that represent simple, descriptive statistics about those dimensions. In some cases a Table or StoredProc contains or produces data in this form, but you'll usually have to query the data to get what you need. Because of its powerful component editor, the DecisionQuery is best suited for this purpose.

Begin by placing a DecisionQuery component on your form, and set its *DatabaseName* property to *DBDEMOS*. Next, either double-click the DecisionQuery, or right-click it and

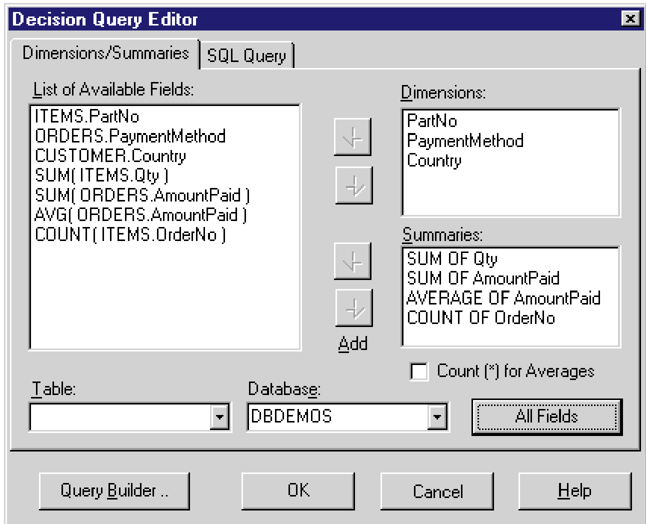


Figure 3: The Decision Query Editor showing only the query fields.

select **Decision Query Editor** to display the Decision Query Editor shown in Figure 1. Now, construct a query that produces one or more dimensions and their associated calculations. You can click **Query Builder** and use it to define the query. Otherwise, you can select the SQL Query page of the Decision Query Editor and enter:

```
SELECT ITEMS.PartNo, ORDERS.PaymentMethod,
       CUSTOMER.Country, SUM(ITEMS.Qty),
       SUM(ORDERS.AmountPaid), AVG(ORDERS.AmountPaid),
       COUNT(ORDERS.OrderNo)
FROM "ITEMS.DB" ITEMS
INNER JOIN "ORDERS.DB" ORDERS
        ON (ITEMS.OrderNo = ORDERS.OrderNo)
INNER JOIN "CUSTOMER.DB" CUSTOMER
        ON (ORDERS.CustNo = CUSTOMER.CustNo)
GROUP BY ITEMS.PartNo, ORDERS.PaymentMethod,
         CUSTOMER.Country
```

Note one important characteristic of the query. You must have a GROUP BY clause, and the order of the fields in the GROUP BY statement must match the order of field selection in the SELECT clause. Note also the use of SUM, AVG, and COUNT operators to calculate the simple descriptive statistics. Once you have entered the query, select **Edit Done**, and choose the Dimensions/Summaries tab. Your form should resemble Figure 2.

In many cases, only the fields selected in the query are of interest. To display only those fields, click the **Query Fields** button on the Decision Query Editor. As shown in Figure 3, this will often greatly reduce the number of fields displayed. When done, click **OK** to close the Decision Query Editor and save your changes.

Defining the DecisionCube

Once the DataSet is producing the correct data, link the DecisionCube to the DataSet. Place a DecisionCube on your form, and set its *DataSet* property to *DecisionQuery1*. Next, use the Decision Cube Editor to define how the decision data is loaded and displayed, as well as to control the DecisionCube's memory use (see Figure 4). To display the Decision Cube Editor, either double-click on the

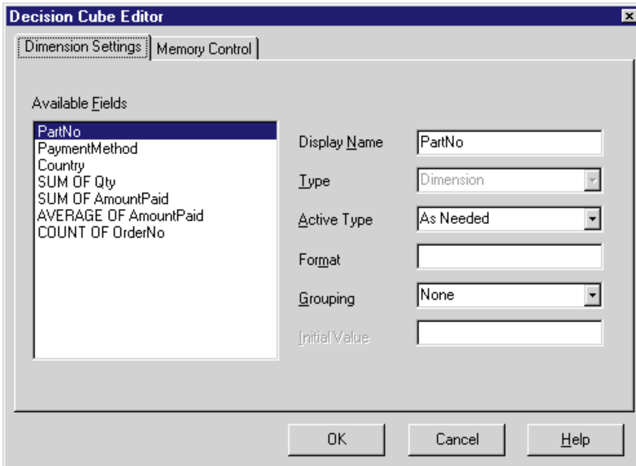


Figure 4: The Decision Cube Editor.

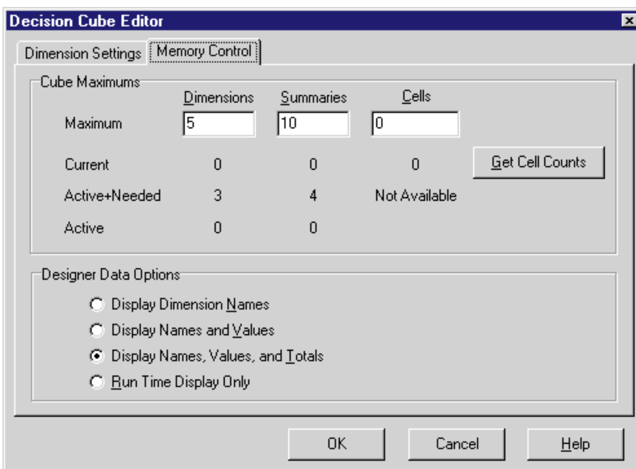


Figure 5: The Memory Control page of the Decision Cube Editor permits you to manage the memory requirements of a DecisionCube.

DecisionCube, right-click the Decision Cube and select **Decision Cube Editor**, or choose the DecisionCube and invoke the DimensionMap property editor.

Use the Dimension Settings page of the Decision Cube Editor to control when data is loaded from the DataSet, as well as the display characteristics of the data. To do this, select a dimension or summary field from the **Available Fields** list. Next, use the **Display Name** field to specify a label. If you didn't use a DecisionQuery to select the data (in other words, if you used a Table, Query, or StoredProc), use the **Type** field to define whether the field is a dimension or a summary. The **Active Type** field permits you to define when the data is loaded (as needed, immediately, or never). By setting a field to **As Needed**, you reduce overall memory use, but increase the DecisionCube's response time to changes. Use **Format** to specify a Delphi format statement to format the data, **Grouping** to define the grouping of a date-related dimension (by year, month, and so forth), and **Initial Value** to set the beginning value of a range (used when you limit dimensions on the Memory Control page).

Controlling Memory Use

Because a DecisionCube stores information (as opposed to pointing to it as the Table component does), the amount of memory it requires depends on the number and size of the dimensions selected, as well as the number of summary statistics calculated. Because the number of data points increases exponentially as you add dimensions, the amount of data stored by a DecisionCube can easily get out of hand. You can control this by being selective about the number and size of dimensions, as well as the number of summary statistics calculated for each dimension. If this isn't possible, limit the number of dimensions and summaries stored at any given time.

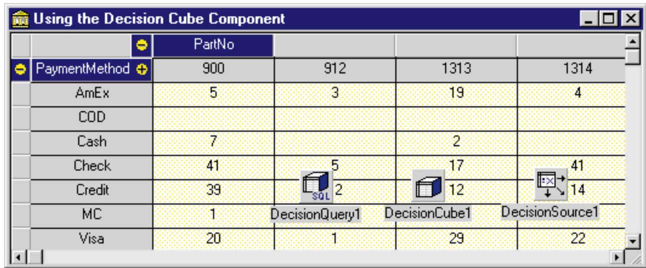


Figure 6: A DecisionGrid displays data at design time when attached to a DecisionSource that points to an active DecisionCube.

These DecisionCube memory issues also affect its performance at design time; you may find there isn't enough memory to work with the DecisionCube at design time. If so, use the Design Data Options section of the Memory Control page in the Decision Cube Editor to limit what is loaded at design time (see Figure 5). Note the DecisionCube also has an *OnLowCapacity* event property. Create an event handler for this property if you need to respond to low-memory conditions at run time. If your DecisionCube uses a DecisionQuery, and low memory isn't a problem, you'll typically use the Decision Cube Editor only to control display labels and summary data formatting.

Using the Data-Aware Decision Controls

There are three data-aware decision controls you can use with a DecisionCube: DecisionPivot, DecisionGrid, and DecisionGraph. Placing any one of these requires you first add a DecisionSource, and link its *DecisionCube* property to a DecisionCube.

Once that's completed, add one or more of the following components: DecisionPivot, DecisionGrid, and DecisionGraph. While it can be done, there is rarely any justification for placing a DecisionPivot on a form without a corresponding DecisionGrid or DecisionGraph. Furthermore, typically you won't place a DecisionGraph without a corresponding DecisionPivot. A DecisionGrid, in contrast, can appear by itself on a form, or with a DecisionGraph. To demonstrate the relative roles of these components, start by placing only a DecisionGrid.

Using the DecisionGrid

As long as the non-visual DecisionCube components are in place and active, data is available as soon as you set the DecisionGrid's *DecisionSource* property to a valid DecisionSource (see Figure 6). This may be affected by design-time memory-control settings you've made.

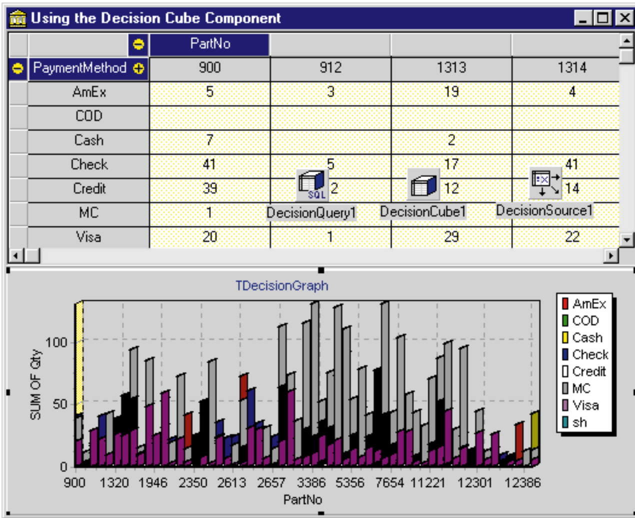


Figure 7: Placing a DecisionGraph on a form and pointing it to a DecisionSource creates a chart.

If the DataSet associated with the DecisionCube defines more than one dimension, the DecisionGrid will display a 2D grid — one dimension identifying row membership, and another defining column membership. The corresponding summary value will be displayed for each row and column combination. For example, the rows identify payment method, the columns identify part number, and a sum of amount paid is the summary value. A single cell within the grid will contain the sum of all payments for one particular part number made using one particular payment method. For instance, as seen in Figure 6, a given cell may display the total payments made using American Express for part number 900.

Both at design time (if the properties permit it) and during run time, you can right-click on various parts of the DecisionGrid to change the dimensions represented, as well as the summary values displayed. These changes to the selected summary values and dimensions affect the DecisionCube's properties directly. Furthermore, you can drag a column to a row, or a row to a column, to change the orientation of the DecisionGrid. Finally, you can right-click the DecisionGrid to change additional properties of the DecisionCube. However, in most cases, you will also provide a DecisionPivot, giving the user a more friendly way to affect DecisionCube properties. (Using the DecisionPivot is described later in this article.)

Using the DecisionGraph

The DecisionGraph is different from the Chart and DBChart components because it has a *DecisionSource* property, ultimately getting its data from a DecisionCube. To demonstrate this, place a DecisionGraph on your form. As soon as you set its *DecisionSource* property to the DecisionSource, a chart appears in the DecisionGraph (see Figure 7). Furthermore, if you change summary values of dimensions using the DecisionGrid on the form, the data in the chart changes as well.

While you can control the dimensions and summaries displayed in a DecisionGraph using a DecisionGrid, in most cases you'll want to place a DecisionPivot on your form,

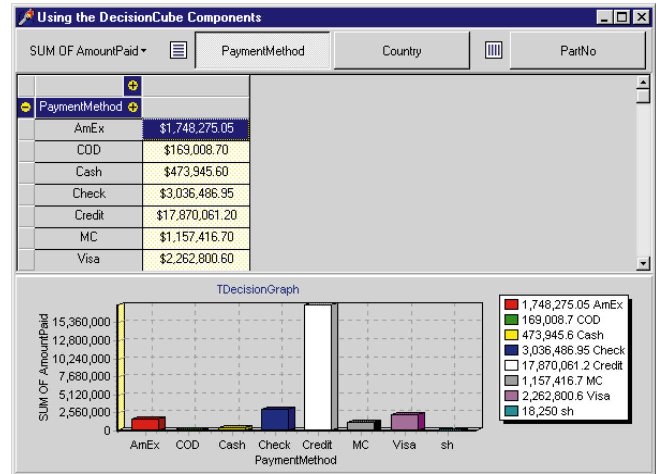


Figure 8: The main form of the example DECCUBE project.

and control the DecisionCube through it. Changes you make to the DecisionCube properties using a DecisionPivot are reflected in any DecisionGraphs that also point to the DecisionCube. For information on controlling the chart type and individual characteristics of a DecisionGraph, such as line colors and axis dimensions, see October's installment of DBNavigator.

Using a DecisionPivot

The DecisionPivot is a visual control that provides a run-time interface to certain essential properties of a DecisionCube. Its purpose is to permit users to select dimensions to expand or collapse, and the summary values to display.

To demonstrate the use of a DecisionPivot, place one on the form. Align it to *alTop*, and set its *DecisionSource* property to *DecisionSource1*. To finalize the decision form at this point, you may also want to align the DecisionGrid to *alTop*. Next, add a Splitter component, align it to *alTop* and set its *Cursor* property to *crVSplit*. Finally, align the DecisionGraph to *alClient*. An example of these components in this orientation can be found in the project named DECCUBE (see Figure 8).

To change the summary value displayed, select the left-most button on the DecisionPivot and choose the summary value to display. To collapse or expand a dimension, click on the button associated with the dimension. To change a dimension from a row to a column, or a column to a row, drag the button associated with the dimension from its current position to a new position. Buttons that appear to the right of the small table icon with horizontal stripes in the DecisionPivot define the row dimensions. The buttons appearing to the right of the small table icon with vertical stripes define the column dimensions. To change the order of dimensions on a particular axis, drag the buttons associated with the dimensions in question to new positions. The left-most button represents the highest-level dimension on its axis, the second left-most button represents the second dimension along that axis, and so forth.

You can use two or more DecisionPivot components for a single DecisionSource. This is usually done to place only row

dimensions on one DecisionPivot, and only column dimensions on another DecisionPivot. You control which types of dimensions appear on a given DecisionPivot by using its *Groups* property. You can also define a DecisionPivot to only display the drop-down list for available summaries.

When using more than one DecisionPivot, align them differently. For example, align your column-dimension DecisionPivot to *alTop,* and the row DecisionPivot to *alLeft*. You can change the *GroupLayout* property of this control to permit the buttons to appear vertically, rather than horizontally. However, remember that changes to the dimensions of a DecisionCube can take a long time, depending on the number of discrete levels of the dimensions involved. Using the methods of the DecisionSource component, such as *OnBeforePivot* and *OnNewDimensions*, you can ask the user to confirm the changes.

For more information on using DecisionCube components, see Chapter 16, "Using Decision Support Components," in the *Delphi 3 Developer's Guide* that ships with Delphi.

Conclusion

The Decision Cube components constitute a powerful collection for manipulating and displaying data for decision-support applications. It's one more reason to consider investing in a copy of Delphi 3 Client/Server edition. ∆

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

Delphi Reports

Delphi 1 / Delphi 2 / Delphi 3

By Warren Rachele

Avoid Report-Engine Overkill

Return to Basics with the *TPrinter* Object

A user clicks the Print button of the shiny new version of SUPER APP, expecting a near-instant reaction from the printer. Instead, he is greeted by the sound of a grinding disk drive struggling to load the report engine. This test of patience is amplified by the brevity of the two-line report that eventually appears in the printer tray.

We developers must recognize that this kind of small annoyance leads to a greater overall dissatisfaction with our products. Fortunately, this situation is easily preventable by adding low-level understanding of the WIN32 API and Delphi print functions to your quiver of techniques.

The developer's task is to measure the quantity and style of the required output against the effort and overhead of programmatic solutions. Simple text and formatted database reports are sometimes better handled by manually manipulating the Windows printer functions, rather than loading a print engine such as ReportSmith, with all its associated overhead. As with most development tasks, Delphi provides a number of approaches.

A Printing Primer

To lay a foundation for Delphi techniques, a quick primer on Windows printing is in order. Printing in both Windows 3.x and Windows 95 is handled by the graphics-device interface (GDI). The procedure for outputting text and graphics on a printer is very much the same as for displaying text and graphics on a video-output device. The program will retrieve a handle to a device context, then direct the output accordingly. This is Windows' method for ensuring device independence. Delphi doesn't need to know the specific commands required by a particular printer. Instead, it calls high-level functions from the GDI, which converts them into low-level device commands specific to the output device. When the application sends output to a printer device context, Windows activates the print spooler to manage the print request. The print spooler provides six functions for controlling the print job (see Figure 1).

Function	Role
AbortDoc	Terminates a print job.
EndDoc	Ends a print job.
EndPage	Ends a page.
SetAbortProc	Sets the abort function for a print job.
StartDoc	Starts a print job.
StartPage	Prepares the printer driver to receive data.

Figure 1: Windows print-spooler functions.

TPrinter is the Delphi class that encapsulates this printer interface. Within the Printers unit, the variable *Printer* is declared as an instance of *TPrinter* ready to be called. (Within 16-bit Delphi, *Printer* is the name of a global object of the *TPrinter* class. 32-bit Delphi still instantiates a global object of class *TPrinter*, but is local to the Printers unit. This relates to the addition of the function *SetPrinter*, which can change the global object.) Figure 2 lists the properties of the *TPrinter* object.

Property	Role
Aborted	Checks whether the user aborted the print job.
Canvas	Serves as the drawing surface of the page.
Capabilities (32-bit only)	Encapsulates the capabilities of the selected printer, indicating the orientation, number of copies, and whether to collate.
Copies (32-bit only)	Returns information from the device mode to select the number of copies printed.
Fonts	A list of fonts supported by the printer.
Handle	A handle to the device context.
Orientation	The paper orientation.
PageHeight	The height of canvas in pixels.
PageNumber	The page number.
PageWidth	The width of the canvas in pixels.
PrinterIndex	The index of the selected printer from the *Printers* property.
Printers	A string list of the printers recognized by Windows.
Printing	The property has a value of *True* when the *BeginDoc* method has been called, *False* when *EndDoc* has been called.
Title	The job title that will appear in the print manager.

Figure 2: The properties of *TPrinter*.

```
implementation

uses Printers;

proc PrintSomething;
begin
  Printer.BeginDoc;
  Printer.Canvas.Textout(10,10,'This is some text');
  Printer.EndDoc;
end;
```

Figure 3: A call to *TextOut*.

TPrinter Properties

The key property of the *TPrinter* object is *Canvas*, which is used the same way for the printer as for a form. Text and graphical output is directed to the printer's canvas, from which it is converted to printer commands and sent to the device. To use the canvas and begin a print job, the developer calls the printer's *BeginDoc* method. This will open the canvas surface to receive output. All *Canvas* methods are then available to produce the desired format. When a page of output has been laid out on the canvas, the printer's *EndDoc* method is used to send the output to the printer. The *Abort* method can be called to discard a print job. The *NewPage* method, alternately, will send output to the printer, then start working on a new canvas page.

The methods used to format the output on the canvas differ in their abilities. I'll present them in order of declining precision, with regard to placing text on the canvas.

Canvas Differences

Before continuing, let's note some important differences in canvas usage. Coordinates for printed output lie along a plane that is much more dynamic than that of a video-output device. Changing the resolution for the application's carefully designed output can happen as quickly as the user can switch from the network laser printer to the local line printer or fax software. *X* and *Y* on the laser-printer plane will not fall in the same spot on lower-resolution print devices such as the line printer. Hard-coded coordinates will produce incomplete output, and scroll bars — obviously — will not be on hand to display the missing data.

By providing device information through its device context, Windows eases the task of managing this dynamic environment. Specifics such as pixel measurements can be retrieved through a call to the API function *GetDeviceCaps*. A specific example of calling the function with a device context and an enumerated constant will appear later in this article.

Centering Text

The first example will print text lines to the printer canvas, using the *TextOut* method. *TextOut* requires three parameters when called: the *X* and *Y* pixel co-ordinates for placing the text string, and the text string itself. A simple call is shown in Figure 3.

The power of the *TextOut* method lies in its ability to precisely place the output at a desired location on the document, through the *X* and *Y* coordinates. Consider the placement of a centered report title. To supply the proper starting location to the *TextOut* method, the developer must do some simple calculating within — or before — the call. Figure 4 shows a snippet for centering text on the applicable printer.

To determine the center of the canvas as defined by the printer interface, the *PageWidth* property is queried. This property contains the width of the page measured in pixels. This value is divided in half to determine the approximate center line of the page. A pair of subcalculations provides the center of the text string. The *TextWidth* method of the canvas provides the width, in pixels, of the string data passed to it.

In the example, the function is passed a single character, and returns a base-width value per character to use as a basis for computing the length of a full line of text. The base value is then multiplied by half the number of characters in the text line to be printed. This equation returns a factor in pixels which, when subtracted from the center point of the canvas, gives a starting point to the *X* parameter of the *TextOut* method.

Computing the Line Height

Utilizing the *TextOut* method with dynamic parameters — to specify the text placement on multiple planes — takes

```
Printer.BeginDoc;

OutLine   := 'This is the line to be centered.';
AlignText := (Printer.PageWidth div 2) -
             (Printer.Canvas.TextWidth('A') *
             (Length(OutLine) div 2));
Printer.Canvas.Textout(AlignText, 10, OutLine);
Printer.EndDoc;
```

Figure 4: Centering text.

```
{ Get vertical pixels per inch }
VPixelsPerInch := GetDeviceCaps(Printer.Handle,LOGPIXELSY);
{ Set line spacing 1/10 of vertical pixels per inch }
VLineSpacing := VPixelsPerInch div 10;
{ Set the Line Height }
LineHeight := Printer.Canvas.TextHeight('A') +
              VLineSpacing;
```

Figure 5: Determining the height of the text lines.

still more work on the part of the developer. To print multiple lines along the *Y* axis of the page, the developer must compute the line height of the applicable font, add some spacing between the printed lines, and maintain a sum of printed lines so the end of the page is not overwritten and output is not lost. The first step in this process is to retrieve a measure of the resolution in pixels per inch, using the following API call:

```
GetDeviceCaps(Printer.Handle,LOGPIXELSY);
```

LOGPIXELSY will return an integer value that represents the number of pixels along an inch of the display height. This value is then used in an algorithmic computation of the line placement. The code snippet in Figure 5 shows the sequence of calls needed to determine the height of the text lines.

The total line height is measured as the sum of the number of pixels in *VLineSpacing*, plus the number of pixels returned from the *TextHeight* call. Note that the divisor value used in the computation of *VLineSpacing* (10) is arbitrary, and must be adjusted as necessary for the specific application. As with the *TextWidth* call used earlier, the *Canvas* method, *TextHeight,* considers the metrics of the font currently in use — an important factor in outputting to various printers.

Determining Page Breaks

Querying the *PageHeight* property of the canvas provides the total pixel height of the page. The number returned is the total against which the accumulated printed lines are measured, to determine when a page break is required. The *TPrinter* object provides a method called *NewPage*, which forces the canvas to send its current output to the printer and begin printing on a new canvas. This method will handle the canvas itself, resetting its *Pen* position to 0,0, and incrementing the page number. The developer must reset all the print variables, such as *LinesPrinted*, within the application.

Printing from Tables

Simply formatted data-table listings and other columnar reports can be quickly handled through manual means, as well. Aside from the controlling loop that steps through the items in the table, the only additional computations necessary for placing the output on the canvas are those that determine the starting point for each column on an output line. Listing One, on page 23, shows the basic code needed for printing data items from a table.

Placement of the columns is by pixel across the *X* plane, much as the line is printed on the *Y* axis of the canvas. The columns in the example are placed by adding each field's *DisplayWidth* property to the accumulated pixels used on the line. The *X* parameter is constrained by the *PageWidth* property of the *Printer* object.

```
var
  OutFile : TextFile;

begin
  AssignPrn(OutFile);
  Rewrite(OutFile);

  Writeln(OutFile,'This is a print line');

  CloseFile(OutFile);
end;
```

Figure 6: The *Write* and *Writeln* methods handle many printing details (e.g. streaming and wrapping) for you.

Printer Peculiarities

While most methods from the *Form* object canvas will work on the *Printer* canvas, there are some important differences. The printer is a mechanical entity with a unique set of exceptions that must be handled by the developer. The items most likely to trip up the programmer are mechanical errors generated by the printer. The program calling the *Printer* object directly will be responsible for handling "out of paper" or "printer not online" errors gracefully. Calls to the *Printer* methods should be enclosed within a resource-protection (**try..finally**) block, so that all resources are released, and all exceptions are handled.

Exception handling aside, the output capabilities of the two devices must be considered by the developer. Because the user will be running Windows, the developer is guaranteed that graphics can be displayed on the screen. There is no guarantee however, that the printer selected is capable of reproducing them. Text and graphics sent to the screen can be erased; output to the printer canvas is sent to the printer, and cannot be recalled. Therefore, the user should be given the option of printing graphics or leaving them out, based on the capabilities of the chosen printer.

Finally, drawing to the screen canvas is instantaneous, while drawing to the printer canvas is not. Sending output to the printer is slow, so the developer should always provide the ability to abort a print process. For this reason, the *TPrinter* object provides the *Abort* method.

Tools for the Job

Using the *TextOut* method gives the developer precise control over the placement of text on the canvas, right down to the pixel. This power takes a heavy toll in programming overhead and exception handling, and might be overkill if the output is, for instance, a couple of lines on an exception report. Simplifying the print process, Delphi allows the developer to assign a print file to the printer, then use the *Write* and *Writeln* functions to stream output to the file. All the line sizing is handled by the system, and lines that extend beyond the limits of the printer's page width are automatically wrapped. Figure 6 shows an example of this use.

The *AssignPrn* procedure assigns a text-file variable — in this case, *OutFile* — to the printer. The Pascal procedure *Rewrite*

must be called to create a new output file. Once the output file has been created, the *Write* and *Writeln* procedures are used to send the output text to the file name.

Our discussion ends with the least-powerful technique. By calling the *Print* method of a *Form* object, the entire form canvas is sent to the currently selected Windows printer. The layout is then in control of the form's canvas; the amount of data that can be output is constrained by the size of the form.

Conclusion

Each of the techniques presented has trade-offs that must be weighed by the developer when considering the best way to produce printed output for the user. The most basic consideration regarding any of these is the amount of programming needed to provide the user with the fastest method of receiving output. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\NOV\DI9711WR.

Warren Rachele is Chief Architect of The Hunter Group, an Evergreen, CO software development company specializing in database-management software. The company has served its customers since 1987. Warren also teaches programming, hardware architecture, and database management at the college level. He can be reached by e-mail at wrachele@earthlink.net, or by telephone at (303) 674-8095.

Begin Listing One — Printing from a table

```
begin
  Printer.BeginDoc;

  Table1.First;
  while not Table1.EOF do begin
    if LinesPrinted < Printer.PageHeight then
      begin
        NextCol := 10;
        Printer.Canvas.Textout(NextCol,LinesPrinted,
          Table1.FieldByName('CUSTNO').AsString);

        NextCol := NextCol + (
          Table1.FieldByName('CUSTNO').DisplayWidth *
            Printer.Canvas.TextWidth('A'));
        Printer.Canvas.Textout(NextCol, LinesPrinted,
          Table1.FieldByName('COMPANY').AsString);

        NextCol := NextCol +
          (Table1.FieldByName('COMPANY').DisplayWidth *
          Printer.Canvas.TextWidth('A'));
        Printer.Canvas.Textout( NextCol,LinesPrinted,
          Table1.FieldByName('COUNTRY').AsString);

        Table1.Next;
        LinesPrinted := LinesPrinted + LineHeight;
      end
    else
      begin
        Printer.NewPage;
        LinesPrinted := 0;
        PrintTimeStamp;
        PrintHeading;
      end;
    end;

  Printer.EndDoc;

end;
```

End Listing One

By Bill Todd

Using the BDE API

Part I: Mastering the Direct Approach

Although much of the Borland Database Engine's API is surfaced in Delphi, some useful tools still are available only by calling the API directly. In this two-part series, I'll examine some of these and their table-related features.

There are two ways to call a BDE API function: You can use the BDE session created automatically by any Delphi program that uses data-access components, or you can create your own session using BDE API calls. Because the first option is easier, the examples presented here will use that technique; the next installment will address the second option.

BDE Error Messages

One thing all BDE API functions have in common is that they return a numeric result code of type *DbiResult*, which indicates whether the function was successful. If the value DBIERR_NONE is returned, no error occurred; any other returned value is an error code. You can obtain the message that corresponds to any error code by calling *DbiGetErrorString*, as shown in Figure 1.

```
{ Given a BDE error code, return the corresponding
  error message as a Pascal string. }
function dgGetEngineErrorMessage(
  ErrCode: DbiResult): string;
var
  EngineMsg: array[1..255] of Char;
  EngineMsgPtr: PChar;
begin
  EngineMsgPtr := @EngineMsg;
  DbiGetErrorString(ErrCode, EngineMsgPtr);
  Result := StrPas(EngineMsgPtr);
end;
```

Figure 1: Getting BDE error messages.

```
{ Given a BDE error code, returns the corresponding
  error message as a Pascal string.}
function dgGetEngineErrorMessage(
  ErrCode: DbiResult): string;
var
  EngineMsg: string;
begin
  DbiGetErrorString(ErrCode, PChar(EngineMsg));
  Result := EngineMsg;
end;
```

Figure 2: Casting to PChar.

You can code this function more simply in Delphi 3 by taking advantage of the compatibility between null-terminated strings and the new long string type, as shown in Figure 2. (The Figure 1 version works in Delphi 1, 2, and 3.) To take advantage of Delphi's exception-handling mechanism, call BDE API functions by passing them as parameters to the *Check* procedure; for example:

```
Check(DbiGetErrorString(ErrCode,
  PChar(EngineMsg)));
```

The *Check* procedure raises an exception if the API function returns an error.

Packing dBASE Tables

dBASE tables need to be packed periodically because deleting a record from a dBASE table doesn't really delete it; it merely flags the record *for* deletion. The record is not physically deleted from the table, and the space it occupies cannot be reused until the table is packed.

The following procedure will pack a dBASE table. To use this procedure, you must have a Database component and a Table component on the form. Set the Database component to point to the directory that contains the table you want to pack by setting its *AliasName* property to an alias that points to the directory, or by setting the *Path* parameter in the *Params* property to the directory's path. Do this by adding an entry to the *Params* property, using the String List Editor:

```
Path=c:\mytables
```

```
{ Pack a dBASE table by calling DbiPackTable. The table
  passed as a parameter will be opened if it isn't open. }
function dgPackDbaseTable(Tbl: TTable): DBIResult;
begin
  Result := DBIERR_NA;
  if Tbl.Active = False then
    Tbl.Open;
  Result := DbiPackTable(Tbl.DBHandle, Tbl.Handle,
                         nil, nil, True);
end;
```

Figure 3: Packing a dBASE table.

```
function dgPackParadoxTable(Tbl: TTable;
  Db: TDatabase):DBIResult;
{ Packs a Paradox table by calling the BDE DbiDoRestructure
  function. The TTable passed as the first parameter must
  be closed. The TDatabase passed as the second parameter
  must be connected. }
var
  TblDesc: CRTblDesc;
begin
  Result := DBIERR_NA;
  FillChar(TblDesc, SizeOf(CRTblDesc), 0);
  StrPCopy(TblDesc.szTblName, Tbl.TableName);
  TblDesc.bPack := True;
  Result := DbiDoRestructure(Db.Handle, 1, @TblDesc,
                             nil, nil, nil, False);
end;
```

Figure 4: Packing a Paradox table.

Set the Database component's *DatabaseName* property to the value you want to use for the temporary alias that the Database creates. Next, set the Table component's *DatabaseName* to the same value you used for the Database component's *DatabaseName*. Set the Table's *Exclusive* property to *True*, and its *TableName* to the name of the table you want to pack. If the Table is connected to a DataSource, call its *DisableControls* method.

To pack the table, call the *dgPackDbaseTable* function shown in Figure 3. Pass as the parameter the Table component for the table you want to pack. In Delphi 1, the DbiProcs, DbiTypes, and DbiErrs units must be included in the **uses** clause of the unit that contains the *dgPackDbaseTable* function. In Delphi 3, the BDE unit must be included in the **uses** clause. The function opens the table if it isn't already, then packs it by calling the BDE API function **DbiPackTable**.

The first parameter of **DbiPackTable** is a BDE handle to the database that contains the table. This is surfaced as the *DBHandle* property of *TTable*. The second parameter is the BDE handle to the table itself, which is surfaced in Delphi as the *Handle* property of *TTable*.

The third and fourth parameters — table name and driver type, respectively — provide an alternate way to identify the table if it isn't open. The table name can include a full path. If not, the location is assumed to be the database that *DBHandle* (the first parameter) points to. The only valid driver type is *szDbase*, and you can omit the driver type if the table name includes the .DBF file extension.

The fifth and final parameter specifies whether the table's indices should be rebuilt; it should be set to *True*.

Packing Paradox Tables

Paradox tables do not normally need to be packed because deleting a record from a Paradox table makes the space available for any new record. However, there still may be cases when you want to pack a Paradox table — when a large fraction of the records in the table are deleted, for instance. This might occur in a table where most of the records are moved to an archive at the end of each year. In this case, packing will reduce the size of the table on disk, and compress and rebalance the indices for faster access.

The function in Figure 4 requires two parameters. The first is the Table component that contains the table's name, and whose *Exclusive* property must be set to *True*. The second parameter is the Database component connected to the database that contains the table you want to pack.

When you use BDE API calls to work with Paradox tables, a table-descriptor structure is used to pass information to the BDE about the table and what you want to do to it. The call to *FillChar* initializes the table descriptor by filling it with binary zeros. The next line copies the name of the table from the Table's *TableName* property to the table descriptor's *szTblName* field, as a null-terminated string. The following statement sets the table descriptor's *bPack* field to *True* to inform the BDE to pack the table.

Finally, **DbiDoRestructure** is called to pack the table. The first parameter to **DbiDoRestructure** is the BDE database handle provided by the *Handle* property of the Database component. The second parameter is the number of table descriptors, and must always be set to 1. The third parameter is a pointer to the table descriptor. Note the use of the @ sign to specify the address of the table descriptor. The fourth parameter is the new name for the restructured table. If you provide a name, the restructured table is saved with it; the original table is not changed.

The fifth and sixth parameters are the names of the key-violation and problems tables, respectively. These tables are not used when the only operation to perform is packing the table. The sixth parameter isn't used for anything in the current version of the BDE; it's a Boolean parameter, and should be set to *False*.

Table Pathfinding

If a table uses a permanent BDE alias, you can obtain its path easily using the *GetAliasParams* method of *TSession* (see Figure 5). However, *GetAliasParams* will not work if the alias is a temporary one created by a *TDatabase*, or if it's hard-coded in the *DatabaseName* property. The function in Figure 6 uses BDE API calls, and works in both these situations, as well as with a table whose BDE alias is permanent.

The call to *AnsiToNative* translates the table's *TableName* property to the local BDE character set, and converts it to a null-terminated string. **DbiGetCursorProps** fills the *CURProps* structure with a variety of properties about the cursor. For a complete list of the properties returned, see the **DbiGetCursorProps** function in the BDE API online Help

(C:\Program Files\Borland\Common Files\BDE\Bde32.hlp by default). The call to **DbiFormFullName** gets the full path to the table, including the table name. The last statement converts the full table name to a Pascal string, and extracts the path portion returned by the function.

```
{ Returns the path for the alias passed in AliasName.
  This must be an alias that points to a subdirectory.
  This function does not verify that the alias points
  to a subdirectory. }
function dgGetAliasPath(var Session: TSession;
                        const AliasName: string): string;
var
  AliasParams: TStringList;
begin
  Result := '';
  AliasParams := TStringList.Create;

  with Session do
    try
      GetAliasParams(AliasName, AliasParams);
      Result := UpperCase(AliasParams.Values['PATH'])+'\';
    finally
      AliasParams.Free;
    end;
end;
```

Figure 5: Getting the alias path.

```
function dgGetDatabasePath(ATable: TTable): string;
var
  TblProps: CURProps;
  pTblName, pFullName: DBITblName;
begin
  with ATable do begin
    AnsiToNative(Locale, TableName, pTblName, 255);
    Check(DbiGetCursorProps(Handle, TblProps));
    Check(DbiFormFullName(
            DBHandle, pTblName, TblProps.szTableType,
            pFullName));
    Result := ExtractFilePath(StrPas(pFullName));
  end;
end;
```

Figure 6: Getting the path for any alias.

```
{ Copy a table to the same or different directory. Local
  table names can include a path, so you can copy tables
  from one directory to another. Server tables can be
  copied only within a database.

  Parameters:
    Database:      TDatabase connected to source db.
    SourceTblName: Source table name.
    DestTblName:   Destination table name.
    Overwrite:     Overwrite destination if True. }
procedure dgCopyTable(Database: TDatabase;
  SourceTblName, DestTblName: string; Overwrite: Boolean);
var
  pSrcTblName,
  pDestTblName: DBITBLNAME;
begin
  AnsiToNative(Database.Locale, SourceTblName, pSrcTblName,
               DBIMAXPATHLEN);
  AnsiToNative(Database.Locale, DestTblName,
               pDestTblName, DBIMAXPATHLEN);
  Check(DbiCopyTable(
          Database.handle, { Database handle. }
          Overwrite,       { Overwrite destination table. }
          pSrcTblName,     { Source table path & name. }
          nil,             { Table type. }
          pDestTblName));  { Destination table path & name. }
end;
```

Figure 7: Copying a table.

Copying Tables

Although you can copy local tables using the Delphi BatchMove component, this technique has a major disadvantage in that only the data is copied. Indices are not copied, and in the case of Paradox tables, neither is the .VAL file that contains the validity checks and referential-integrity definitions. The *dgCopyTable* procedure in Figure 7 will copy a local table, including its "family" (indices and .VAL file), to the same or a different directory. This procedure will also copy server tables, but only within a single database.

This procedure begins by converting the source and destination table names to null-terminated strings, and also to the BDE's character set, by calling *AnsiToNative*. Next, **DbiCopyTable** is called to actually copy the table. In this call, the table's type parameter is set to **nil**; therefore, the names of local tables must include file extensions.

Network Usernames

If you need the user's network username, use the BDE API function **DbiGetNetUserName**, as shown in Figure 8.

Regenerating Indices

Regenerating a non-maintained index brings it up to date, so it includes all the data in the table. Regenerating both maintained and non-maintained indices compacts and balances them for best performance. You regenerate all of the indices associated with a Paradox or dBASE table calling **DbiRegenIndexes**:

```
DbiRegenIndexes(Table.Handle)
```

The parameter is the *Handle* property of the Table component. The table must be opened for exclusive use. For dBASE tables, all open indices are regenerated. For Paradox tables, both maintained and non-maintained indices are regenerated. You can also regenerate a single index using **DbiRegenIndex,** as described in BDE online Help.

Group Record Deletion in Paradox Tables

Although this section's subhead refers to deleting records, the techniques shown here also apply to changing existing records. To understand how to work with groups of records safely, you need to understand a little about the Paradox record-locking mechanism. The BDE uses the information stored in separate lock files on disk (files with the .LCK extension) to control which records are locked in Paradox tables; however, it uses the file-locking mechanism built into the operating system to lock the appropriate section of the

```
{ Return the network username. }
function dgGetUserName: string;
var
  UserNameBuff: array[0..255] of Char;
  pUserName: PChar;
begin
  Result := '';
  pUserName := @UserNameBuff;
  Check(DbiGetNetUserName(pUserName));
  Result := StrPas(pUserName);
end;
```

Figure 8: Getting the network username.

```
procedure dgLoadBdeBatchFunctions;
begin
  Idapi := LoadLibrary('IDAPI32.DLL');
  if Idapi = 0 then
    ShowMessage('IDAPI32.DLL Load Error' + IntToStr(Idapi))
  else
    begin
      @DbiBeginBatch :=
        GetProcAddress(Idapi, 'DbiBeginBatch');
      @DbiEndBatch := GetProcAddress(Idapi, 'DbiEndBatch');
    end;
end;
```

Figure 9: Getting the function handles.

lock file when a record lock must be placed or released. Thus, the sequence of events to lock a record is:
1) lock the lock file
2) write the lock information to the lock file
3) unlock the lock file

Processing many records involves a lot of overhead; two operating-system calls are required to place each record lock, and two more are required for release. Clearly, it would be more efficient to lock the lock file once, place and release all record locks, then unlock the lock file. Fortunately, you can do exactly that, but you must do it with great care. No other user can place or release a lock while you have the lock file locked. If another user tries, his or her machine will appear to "hang" while it waits for access to the lock file. This can cause network timeout errors, or — worse — prompt impatient users to reboot their machines. If you use this technique, make sure you hold the lock on the lock file for only a short time; a good maximum is two seconds.

Now that we've covered the benefits and dangers of locking the lock file, let's look at how to do it. Controlling access to the lock file is done with two undocumented BDE API calls: **DbiBeginBatch** and **DbiEndBatch**. Normally, using undocumented calls is dangerous because they may not be supported in future versions, or may have bugs. However, because the BDE itself makes extensive use of these calls, and because they've been surfaced for developer use in Paradox, they should be safe to use. Both these functions take *TTable*'s *Handle* property as their only parameter.

You must take one extra step to use these calls; because they're undocumented in Delphi, they're not included in the BDE interface unit. Therefore, you must dynamically load the IDAPI32.DLL, and get the address of these functions. The first step in this process is to declare the global variables in the unit that will call these functions:

```
var
  Idapi: THandle;
  DbiBeginBatch: function(hCur: hDBICur):
    DbiResult stdcall;
  DbiEndBatch: function(hCur: hDBICur): DbiResult stdcall;
```

The first declaration, *Idapi*, declares a handle for IDAPI32.DLL. The next two statements declare procedural variables that will hold the addresses of the two functions.

```
with Table do begin
  { Lock the lock file and initialize the flag variable. }
  NoLocks := True;
  DbiBeginBatch(Handle);

  { See if any records are locked. }
  while not EOF do begin
    try
      Edit;
    except
      begin
        Ok := False;
        Break;
      end;
    end;
    Next;
  end;

  { If no records are locked, delete the records. }
  if NoLocks then
    begin
      First;
      while not EOF do
        Delete;
    end;

  { Unlock the lock file. }
  DbiEndBatch(Handle);

end;
```

Figure 10: Deleting a group of records.

Next, you need a procedure to load the DLL and get the function addresses via the Windows API **LoadLibrary** and **GetProcAddress** functions (see Figure 9). This procedure begins by getting a handle to the DLL, then gets the addresses of the functions by calling **GetProcAddress**. You should call this function once when your main form, or the form that uses these functions, is created.

When the form is destroyed, call:

```
FreeLibrary(Idapi);
```

to release the DLL. Because the IDAPI32.DLL was loaded into memory when your program's default BDE session was initialized, the call to **LoadLibrary** simply obtains a handle to the DLL and increments its use count, rather than reloading it. Likewise, the call to **FreeLibrary** simply decrements the DLL's use count. Windows takes care of unloading the DLL from memory when its use count reaches zero, which happens when all programs that are using the DLL have terminated.

Suppose you need to delete a group of records from a table. You must delete all the records, or none of them. That means you must make sure none of the records are locked before you attempt to delete them, and you must prevent other users from locking any of them until you are finished deleting. First, use *SetRange* to restrict your view of the table to just those records that you want to delete. The code in Figure 10 will delete all or none of the records. This code begins by setting the Boolean variable *NoLocks* to *True*, then calling **DbiBeginBatch** to lock the lock file. The **while** loop attempts to lock each record inside a **try..except** block. If another user has the record locked, Delphi will raise an

```
initialization

begin
  { Initialize the BDE. }
  Check(DbiInit(nil));
  try
    { Open; get a handle to the alias list. }
    Check(DbiOpenDatabaseList(TmpCursor));
    AliasFound := False;
    repeat
      { Get a DBDesc record for the next alias. }
      rslt:= DbiGetNextRecord(TmpCursor, dbiNOLOCK,
                               @Database, nil);
      if (rslt <> DBIERR_EOF) then
        if StrPas(Database.szName) = 'DWTAlias' then
          begin
            { The alias DWTAlias already exists. }
            AliasFound := True;
            Break
          end;
    until rslt <> DBIERR_NONE;

    { Release the handler. }
    Check(DbiCloseCursor(TmpCursor));
    if not AliasFound then
      { If the alias was not found, add it to IDAPI.CFG. }
      Check(DbiAddAlias(nil, PChar('DWTAlias'), nil, PChar(
        'PATH:' + ExtractFilePath(Application.ExeName)),
        True))
  finally
    DbiExit;

end;
```

Figure 11: Creating a permanent alias.

exception; the code in the **try..except** block will be executed to set *NoLocks* to *False*, and break out of the **while** loop. If all the records can be locked, the code in the **if NoLocks...** statement deletes them. Finally, the call to **DbiEndBatch** releases the lock on the lock file.

If you use this technique to change records instead of delete them, make sure you post the last record to release its lock before calling **DbiEndBatch**. This technique can be used to lock any group of records in a table for a short time.

Creating Permanent Aliases

In Delphi 2 and 3, you can create a new permanent alias by using *TSession*'s *AddAlias* and *SaveConfigFile* methods (these are not available in Delphi 1). When you create an alias using BDE API calls, you may also encounter a timing problem because the BDE reads its configuration file once when a session is opened. Therefore, for an alias to be used by the program that creates it, you must create the alias before the program's default BDE session is opened.

When a Delphi program begins execution, the first code that runs is the **initialization** section of the main form. Any code you place in the main form's **initialization** section will run before the program's default BDE session is opened so it's the perfect place to create an alias. However, creating an alias before the default BDE session is opened means you must create your own BDE session, then create the alias and close your session. The code from the **initialization** section of the main form's unit (see Figure 11) does just that. The variable declarations that follow are included in the main form's unit, and are used by the code that creates the alias:

```
var
  Database:    DBDesc;
  rslt:        DbiResult;
  TmpCursor:   hDbiCur;
  i:           Integer;
  AliasFound: Boolean;
```

The code in Figure 11 begins with a call to **DbiInit**, the function that opens a BDE session. It takes a single optional parameter: a pointer to a structure that lets you specify the working directory path to the BDE configuration file and language driver. Defaults exist for all these values, and the parameter is optional; so in this code, **nil** is passed.

Next you need to ensure that the alias you want to add does not exist. The call to **DbiOpenDatabaseList** opens a cursor to the alias list in the BDE configuration file. Next comes a **repeat** loop that iterates through the alias list looking for the alias name you want to add — "DWTAlias" in this case. The key code is the call to **DbiGetNextRecord**. This reads the record for the next alias into the *Database* variable, and saves the return code in the variable *rslt*. If *rslt* is equal to DBIERR_EOF, you've reached the end of the alias list. Next, the alias name is extracted from the Database structure, converted to a Pascal string, and compared to the alias to be added. If the alias is found, the flag variable *AliasFound* is set to *True*, and the **repeat** loop is exited. Once the **repeat** loop has finished, **DbiCloseCursor** is called to close the cursor to the alias list.

The call to **DbiAddAlias** adds the new alias. The first parameter identifies the BDE configuration file to be used, and must be null. Only the configuration file for the current BDE session can be modified. The second parameter is the name of the new alias. The third parameter is the driver type. If this parameter is **nil**, the standard drive is assumed.

The fourth parameter is a string containing all the parameters required for the driver type specified in the third parameter. For the standard driver, the only parameter is the path to the database. In this example, the path is set to the directory that contains the program's .EXE file. If you plan to add an alias to a database server, see the examples in the BDE online Help file for the structure of the parameter's string.

The fifth and final parameter is a Boolean that indicates whether the new alias should be saved in the BDE configuration file for use in future sessions, or whether it's a temporary alias for this session only. Set this parameter to *True* to save the alias. Finally, **DbiExit** is called to close the BDE session.

More to Come

There's much left to cover, but we can absorb only so much at one sitting. Next month's installment will explore

the use of Paradox tables with read-only media, using persistent locks as semaphores, sorting local tables, and much more. Δ

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\NOV\DI9711BT.

Bill Todd is President of The Database Group, Inc., a Phoenix-area consulting and development company. He is co-author of *Delphi: A Developer's Guide* [M&T Books, 1995], *Creating Paradox for Windows Applications* [New Riders Publishing, 1994], and *Paradox for Windows Power Programming* [QUE, 1995]; a contributing editor of *Delphi Informant*; a member of Team Borland; and a speaker at every Borland Developers Conference. He can be reached at (602) 802-0178, or on CompuServe at 71333,2146.

By Robert Vivrette

Disassembly View Revealed

Plus, Outsmarting Screen Savers, Registry Tricks, and Other Useful Tips

When Delphi 2 was released, Borland had a little secret. Those who knew were threatened 50 lashes with a wet noodle if they told. The secret was an Easter egg in the form of a window that showed disassembled code (i.e. the assembly-language code generated by the Delphi compiler). Borland didn't make the Disassembly View window common knowledge, because it was a bit experimental and had a few small visual bugs.

Delphi 3 contained an even fancier multi-pane version, but Borland was still maintaining secrecy. Rats! The pressure had been killing me for over a year now; that is, until I saw a reference on Borland's Web site about how to turn on this viewer. With the cat apparently out of the bag and the wet-noodle threat removed, I bring you … the Disassembly View! Simply open RegEdit and go to the following key:

```
HKEY_CURRENT_USER\Software\Borland\Delphi\
  3.0\Debugging
```

Add the name `EnableCPU` at that location, and set its value to `1` (a single-character string just like the other settings in that key). When you next launch Delphi, a **CPU Window** option will have been added to the **View** menu. (This also works in Delphi 2, with a "...\2.0\Debugging" subkey.)

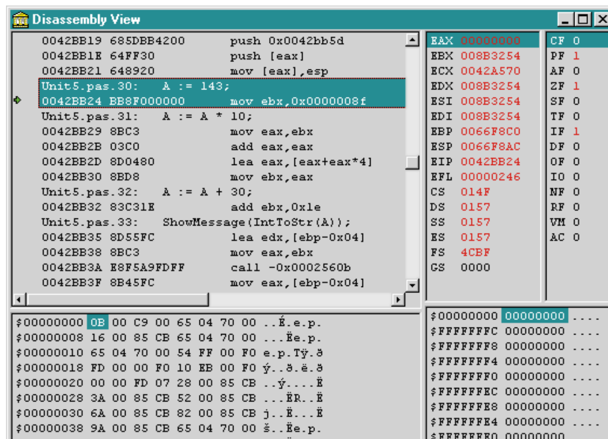


Figure 1: The undocumented Delphi 3 Disassembly View.

The Disassembly View shows valid information only at run time. When you stop your program at a break point, it will show you the current statement, and the assembly code associated with that line. It will also show the contents of the registers, and other cool things. Figure 1 shows the Disassembly View in Delphi 3.

Be forewarned: The Disassembly View still has a few minor visual bugs, though these don't diminish its value.

Nix on the Screen Saver

Screen savers have established their place in the Windows world. Ostensibly developed for protecting against screen burn-in, they've become vehicles for amusement and self-expression. Over the last year or two, more elaborate, resource-hungry screen savers, such as PointCast's SmartScreen, have come onto the scene. Instead of dancing graphics, they display custom news stories, stock prices, weather updates, and other information obtained from the Internet. Unfortunately, if a screen saver activates during certain processor- or time-intensive operations, your application could be robbed of precious resources. And equipping an application against such interruptions can be a programming challenge.

Fortunately, there is a little-known technique for holding a screen saver at bay. When Windows tries to activate a screen saver, it first broadcasts an SC_ScreenSave message to all running applications. If any of these set this message's *Result* field to -1, the screen

```
procedure TForm1.WMEndSession(var Message: TWMEndSession);
var
  Reg : TRegistry;
begin
  Reg := TRegistry.Create;
  try
    Reg.RootKey := HKEY_CURRENT_USER;
    if Reg.OpenKey('\Software\Microsoft\Windows\' +
                   'CurrentVersion\RunOnce',True) then
      Reg.WriteString('MyApp','"' + ParamStr(0) + '"');
  finally
    Reg.CloseKey;
    Reg.Free;
    inherited;
  end;
end;
```

Figure 2: The handler for application relaunch.

saver will not activate. Therefore, all you need to do is add a declaration for *WMSysCommand* in the **private** section of your main form:

```
private
  procedure WMSysCommand(var Msg: TWMSysCommand);
    message WM_SYSCOMMAND;
```

Then, in the **implementation** section, add the following handler:

```
procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);
begin
  if Msg.cmdType = SC_ScreenSave then
    Msg.Result := -1
  else
    inherited;
end;
```

Now, when the screen saver attempts to blast off, your main form can catch the message — and scrub the launch.

Relaunching an Application at Start Up

Have you ever noticed that when you shut down Win95 and leave an Explorer window open, it reappears the next time you start up? Explorer is really just a program running under Windows, so what tells it to relaunch?

The answer is a simple technique that can be applied, not just to Explorer, but to any application. Win95 monitors a registry key named `RunOnce`, which it checks at start-up for any applications that must be launched. It simply looks through all name/value pairs under that key, then launches each application it finds. After it's done, it deletes all entries from the `RunOnce` key. The complete location of this key is:

```
HKEY_CURRENT_USER\Software\Microsoft\
  Windows\CurrentVersion\RunOnce
```

You need only pick a good time to write a value. Because this key relates to shutting down Windows, Microsoft recommends adding this value when the application detects a WM_ENDSESSION message. Simply add this message-handler declaration to the **private** section of your main form:

```
private
  procedure WMEndSession(var Msg: TWMEndSession);
    message WM_ENDSESSION;
```

Then, in the **implementation** section, add the handler shown in Figure 2. The name/value pair written to the registry key defines the application to launch. The "name" portion of that pair is just a unique identifier for that application (in this case, `MyApp`). It can be anything, as long as it's unique among the keys. (Otherwise you'd write over them, wouldn't you?) The value portion is `ParamStr(0)` with quotes added on both sides. The *ParamStr* function returns individual parameters from the command line when an application is launched. `ParamStr(0)` is the name and path of the application itself, i.e. `C:\Borland\Delphi 3\Project1.exe`.

When Windows starts, it recognizes this entry and launches your application. Of course, what your application does at that point is up to you. If it's a document-based application (such as a word processor), you could provide the name of the file after the name of the application, then have your program extract that value and retrieve the document when it starts. It could also read saved values from the registry.

You can also make the application run each time Windows starts by writing the name/value pair into the `Run` key instead of the `RunOnce` key. The only difference is that Windows does not delete entries from the `Run` key, as it does for `RunOnce`. You can also use the `RootKey` of HKEY_LOCAL_MACHINE, so the same `Run` and `RunOnce` items will execute regardless of who logs on.

Controlling Form Resizing

Many Delphi programmers have learned you can control the minimum width and height of a form by adding a message handler that intercepts the WM_GETMINMAXINFO message. Whenever a user attempts to resize a form, Windows sends this message. The form then has the opportunity to provide data about sizing limitations.

A lesser-known trick is to control where a form resizes when you maximize it. This technique is used in the Delphi IDE. When you maximize Delphi, the panel with the menus, Component palette, and speed buttons is snugged up to the top of the screen.

You can do three things with WM_GETMINMAXINFO:

- Specify the minimum width and height of the form when resizing by setting the *ptMinTrackSize* field of the message.

```
procedure TForm1.WMGetMinMaxInfo(
  var Message: TWMGetMinMaxInfo);
begin
  with Message.MinMaxInfo^ do begin
    // Minimum dimensions of form (width/height).
    ptMinTrackSize := Point(200,100);
    // Position (top/left) of form when maximized.
    ptMaxPosition  := Point(50,50);
    // Dimensions of form when maximized (width/height).
    ptMaxSize      := Point(400,200);
  end;
  Message.Result := 0;
  inherited;
end;
```

Figure 3: The handler for form resizing.

- Specify the top/left position of a form when it's maximized by setting the *ptMaxPosition* field.
- Specify the maximum width and height of a form when it's maximized by setting the *ptMaxSize* field.

To intercept this message, put a message handler in the **private** section of your form declaration, like this:

```
private
  procedure WMGetMinMaxInfo(var Message: TWMGetMinMaxInfo);
    message WM_GETMINMAXINFO;
```

Then add a message handler such as that shown in Figure 3. You wouldn't often set all three of these at once, but you can; I used all three to demonstrate their formats. ∆

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

OP TECH

Delphi 3 / COM

By Ron Mashrouteh

Multi-Tier Development

Building a Remote Server Application with Delphi 3

In Delphi 1 and 2, developing three-tier applications isn't especially easy. In Delphi 3, however, Borland has done a fabulous job of enabling applications to communicate with each other through COM and DCOM. Delphi 3 features a new type, *interface*, which is specified with the new **interface** keyword. Interfaces are the building blocks of COM. They are groups of semantically related routines through which COM objects communicate.

This article will provide an overview of multi-tier development, and demonstrate how to develop a simple database application using this technology. We'll focus on developing a server application and a client application that can communicate with each other through interfaces. We will also look into parameterized queries and stored procedures. (A discussion of COM and DCOM is outside the scope of this article. For an introduction to the topic, refer to Danny Thorpe's article, "Delphi 3: The ActiveX Foundry," in the May 1997 *Delphi Informant*.)

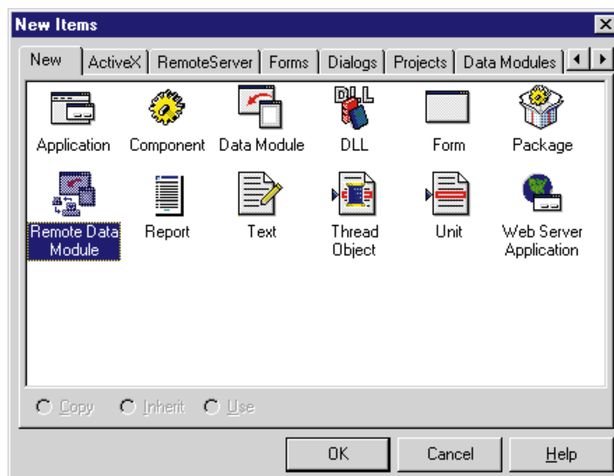


Figure 1: The New Items dialog box.

One of the key features in developing multi-tier applications is a remote data module (RDM); you'll create one in this article. RDM is new to Delphi 3 Client/Server Suite, and is an integral part of remote server applications. The RDM is a repository that maintains all database components and process requests from client machines.

The Server Application

First, let's develop our server application: Launch Delphi 3 and open a new application if one isn't opened automatically. Save Unit1 as uServer.pas, and the project as RemoteServer.dpr. Using the Object Inspector, change the form's *Name* property to `fmServer`. Select **File | New** to display the New Items dialog box. Select the **New** tab, click on the **Remote Data Module** icon, and press **OK** (see Figure 1). When the Remote Dataset Wizard appears, enter `MyServer` in the **Class Name** edit box. By default, **Instancing** is set to Multiple Instance; this allows more than one client application to talk to the server simultaneously. Leave this setting as is, and press **OK** (see Figure 2). Select **File | Save** and save this unit as uRDM.pas. From the Data Access page on the Component palette, select a Query component and a Provider component, and place

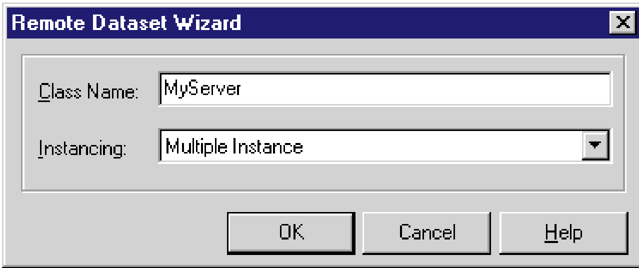


Figure 2: The Remote Dataset Wizard.

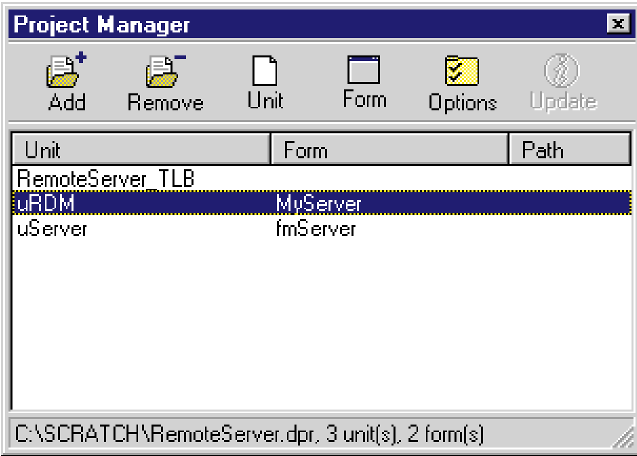


Figure 3: The Project Manager.

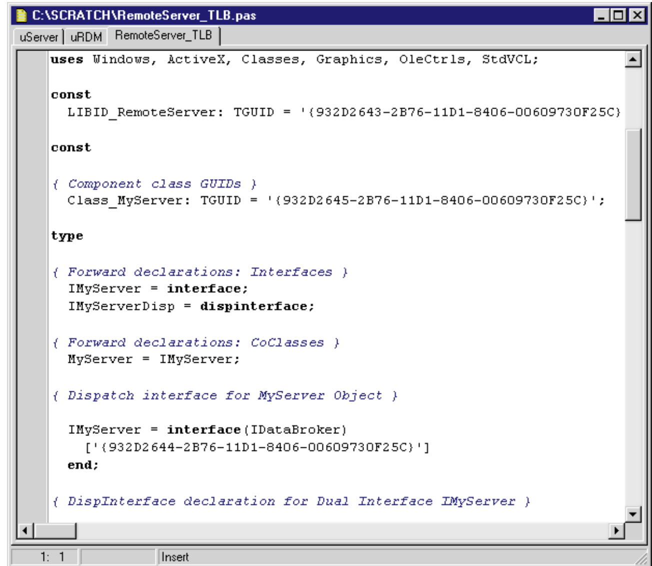


Figure 4: A portion of the RemoteServer_TLB unit.

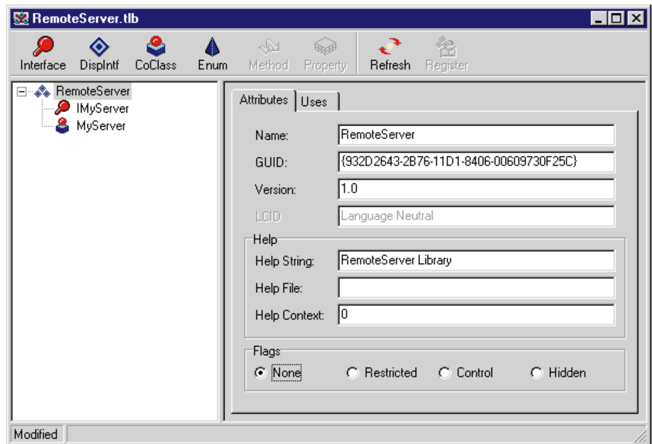


Figure 5: The Type Library window.

them on the *MyServer* form. Select the *Query1* component and change its *DatabaseName* property to DBDEMOS (a sample database that ships with Delphi). Select *Provider1* and set its *DataSet* property to *Query1*.

If you select **View | Project Manager**, you'll notice that Delphi has automatically created a Type Library file, RemoteServer_TLB.pas (see Figure 3). This unit declares all interfaces for our server application. It contains types and numbering schemes new in Delphi 3 (see Figure 4); e.g. the long numbers are the interface identifiers automatically generated by Delphi, and are of the new type GUID (globally unique identifier). The GUID for our server application allows only applications that refer to it specifically to establish communication. These GUIDs are also automatically registered in the Windows registry. So, as usual, Delphi 3 takes care of many programming chores behind the scenes that would otherwise take a significant amount of developers' time.

As mentioned, Delphi 3 has a new interface type. As its name implies, this type provides us with a mechanism for communicating with other applications — in this case, the client application with the server. Interface types defined in a type library are "automation safe"; i.e. derived from *IDispatch* and containing only OLE automation-compatible types. OLE types are declared as OleVariant, WideString, WordBool, etc. We'll use the OleVariant type as we develop our client application. Now we need to implement the interface types of our server application so our client application can see our data-aware component, *Query1*, in the *MyServer* remote data module.

The Type Library Window

Delphi 3 ships with a new utility that allows you to define and manage the interfaces and their members. Select **View | Type Library** to display the Type Library window (see Figure 5). Notice that the title of the window is our Remote Server type library, RemoteServer.tlb. The left pane of the window displays a tree view of the interface type and its members. In this application, our interface type is IMyServer; at this point, no members are defined for it. Click on **IMyServer**, select the Members page in the right pane, and enter:

```
Property Query1: IProvider;
```

Then press **Refresh** from the toolbar to synchronize the RemoteServer_TLB.pas unit with this change. Notice the **dispid** keyword followed by the members we define. You can enter the dispid, or leave it to Delphi to assign it (as we did in this case). The dispid is Delphi's way of keeping the members in the order they're defined, so the compiler can identify the dispids and their order in the Type Library unit.

We still need to define one more member, but before doing so, let's see why. As you know, we didn't write any

SQL statements in the *Query1.SQL* property. We need to provide an interface so our client can send a dynamic SQL statement to *Query1,* and let it process the query on the server.

In the *MyServer* RDM, select the *Provider1* component, right-click to display the properties menu, and select the first menu choice, **Export Provider1 from data module** (see Figure 6). Note that the second member is added to the Type Library window on the Members page. Again, press **Refresh** from the toolbar to synchronize the unit with this change. Now we have to write a few lines of code for the server application to process the request from the client application. We need to add the following code to the *OnDataRequest* event of *Provider1*:

```
function TMyServer.Provider1DataRequest(Sender: TObject;
  Input: OleVariant): OleVariant;
begin
  Provider1.DataSet.Close;
  // Since DataSet in TProvider doesn't support SQL,
  // Provider1 must be cast to type TQuery.
  TQuery(Provider1.DataSet).SQL.Add(Input);
  Provider1.DataSet.Open;
  Result := Provider1.Data;
end;
```

Save and run the program; that's it for our Server application.

The Client Application

Open a new project by selecting **File | New**. On the New page, select **Data Module** and press **OK**. Save Unit1.pas as uClient.pas, Unit2.pas as uDM.pas, and Project1.dpr as Client.dpr. From the Data Access page in the Component palette, select a RemoteServer, a ClientDataSet, and a DataSource component, and drop them on the *DataModule1* form.

We must also include some components on our client form and set their properties. Place a Label, Button, Edit, and DBGrid on the form and arrange them as shown in Figure 7. Then set the Label's *Caption* property to `SQL Statement`, and the Button's *Caption* to `&Run Query1`; delete the Edit's *Text* property; and set the DBGrid's *DataSource* to *DataModule1.DataSource1*. Change *Form1*'s *Name* property to `fmClient`.

Select the *RemoteServer1* component on the *DataModule1* form. Select the *ServerName* property drop-down list in the Object Inspector, and select **RemoteServer.MyServer** (the server application we just created). Once you select it, the *ServerGUID* property displays the GUID for the server. If you check the RemoteServer_TLB.pas unit of the RemoteServer project, you'll see that the ServerGUID matches that of the Class ID.

(Note: If you don't see the server name, select **Project | Build All** and try again.)

Now comes the interesting part. Select *ClientDataSet1* on the *DataModule1* form, and set its *RemoteServer* property to *RemoteServer1*. Next, we need to set the *ProviderName* property.

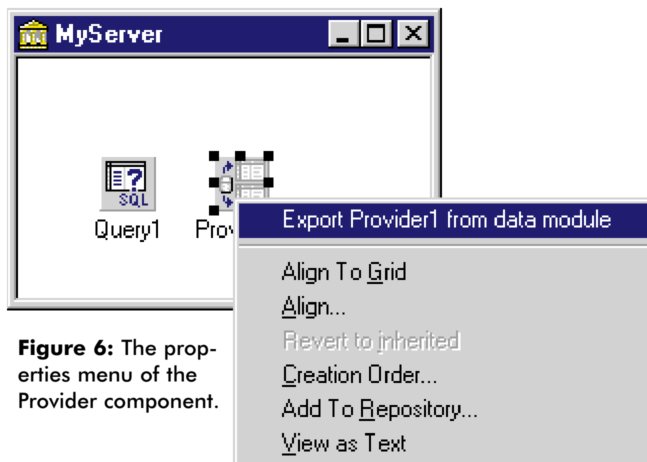


Figure 6: The properties menu of the Provider component.

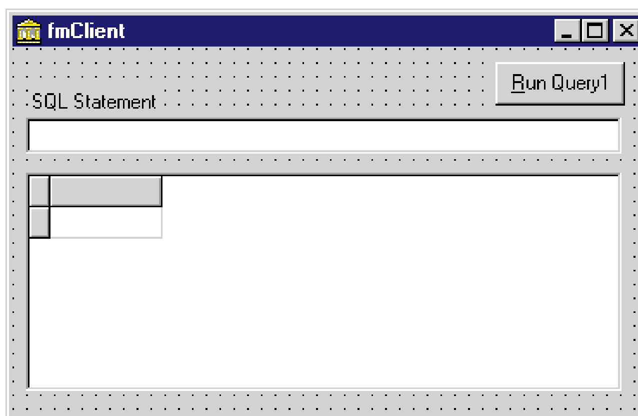


Figure 7: The client form.

Notice that as soon as you click on the drop-down list in the Property Inspector, the RemoteServer starts and displays the interface members we defined in our server project's Type Library; select *Provider1*. Set *DataSource1*'s *DataSet* property to *ClientDataSet1*.

We need to write a few lines of code in the *OnClick* event of *Button1*:

```
procedure TfmClient.Button1Click(Sender: TObject);
var
  StrSql : OleVariant;
begin
  StrSql := Edit1.Text;

  with DataModule1.ClientDataSet1 do
    Data := Provider.DataRequest(StrSql);
end;
```

Before running the client application, make sure the remote server application isn't running. Save and run the client application; you'll see the server application automatically launches without the user's interaction.

The remote database application server could — and probably would — be sitting on another machine, in which case you'd need to assign a value to the *RemoteServer1*'s *ComputerName* property. Once you launch the server application on a different machine, it automatically registers itself with the Windows registry.

Running the Application

When the client application is up and running, enter:

```
SELECT * FROM ANIMALS
```

in the Edit component, and press the **Run Query1** button.

As mentioned earlier, using a parameterized query or stored procedure requires a different function call to pass parameter(s) from the client to the server. For example, if you want to query for animals of a certain size and weight, you'll need to provide a mechanism for two parameters. To do this, place code such as the following in the *OnDataRequest* event of the *Provider1* component on the server:

```
function TMyServer.Provider1DataRequest(Sender: TObject;
  Input: OleVariant): OleVariant;
var
  P : OLEVariant;
begin
  ClientDataSet.Close;

  P := VarArrayCreate([0,1], VarVariant);
  P[0] := VarArrayof([<Param1>, <Value1>]);
  P[1] := VarArrayof([<Param2>, <Value2>]);

  ClientDataSet.Provider.SetParams(P);
  ClientDataSet.Open;
  Result := Provider1.Data;
end;
```

This assumes that *Query1* has a SQL statement in its *SQL* property, such as:

```
"SELECT * FROM ANIMALS WHERE SIZE = :size AND WEIGHT = :weight"
```

where `:size` and `:weight` are parameters that need to receive their values from the client application.

Conclusion

Two advantages of developing *n*-tier applications are that they're easier to maintain and deploy. Such applications can become complex depending upon the number and type of triggers, stored procedures, business rules, etc; however, once the server application is configured, no individual BDE set-up or configuration is required on the client machines. ∆

The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\97\NOV\DI9711RM.

Ron Mashrouteh is an application-development consultant with XySoft Systems in Houston. He has worked with C/C++ in UNIX, and the Windows environment and Delphi since Version 1.0. He also has done consulting for Dresser Industries and Texas Commerce Bank, helping them in developing critical real-time data collection applications using Delphi. He can be reached on MSN at xysoft@msn.com or at (713) 789-1094.

By Peter Hyde

WebHub

HREF Tools' Extensible, OOP-Based Framework for Web Site Creation

You want to be on the bleeding edge? Try submitting your largest-ever contract bid, in a technology area new to almost everyone, based on two unreleased development products. *That's* bleeding-edge development!

In early 1996, our company did just that when bidding for the production of a high-performance Web site and search engine. After much research, we based our successful bid on Delphi 2 and HREF Tools' WebHub product. According to plan, Delphi 2 was released in plenty of time for site deployment. In contrast, WebHub remained in its "Early Experience Program" for another year. However, that didn't prevent us from finishing that job on schedule and deploying several other highly automated sites in the months that followed.

In that time, we've had several opportunities to examine other Web automation tools for ourselves and on behalf of clients — approaches ranging from other Delphi component sets to suites such as Microsoft FrontPage, Active Server Pages, Borland IntraBuilder, Sapphire/Web, and OneWave. Our conclusion has been consistent throughout: For the kind of sites we tend to create — dynamic, data-driven, remotely maintainable, and surfer-friendly — it's hard to find a tool set that comes close to matching WebHub.

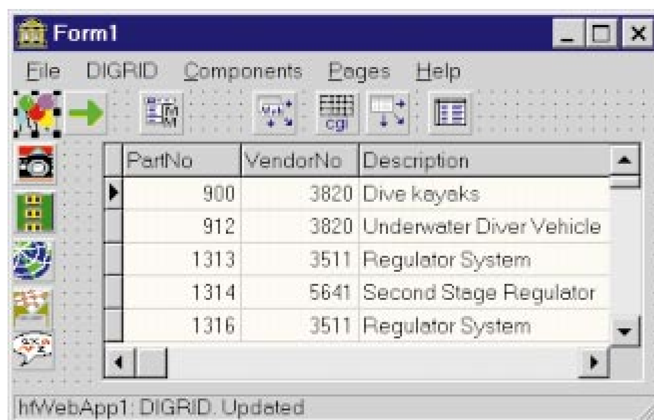


Figure 1: A zero-code WebHub application in Delphi's IDE.

Don't get me wrong; the alternatives have their place. The best tool for allowing people on an intranet to do simple jobs (e.g. publish a phone-list database) is probably FrontPage or IntraBuilder. These types of tools have friendlier learning curves, and are adept at producing simple pages quickly. But the minute any part of a site becomes a "real" application — one needing high performance, complex search-and-response pages, or features such as e-commerce — the limitations of these tools become alarmingly evident.

WebHub won us over initially because it provides an extensible, OOP-based framework for creating Web sites. Out of the box, it handles the issues critical for dynamic Web site management: surfer tracking, security, query cloning, quick page turnaround, server independence, upscaling, and maintainability. It also includes highly configurable components for presenting database or other information as customizable, automatically paged HTML tables. Over time, its repertoire has grown to include features such as credit-card validation, zero-code form mail/mail-merge support, live graphic creation, and "instant forms" for database editing and posting from a browser.

Down to Cases

To illustrate WebHub in action, let's look at how easy it is to build a full-fledged, database-aware Web page from scratch. Figure 1 shows the application at design

Parts Table

#	PartNo	VendorNo	Description	OnHand	OnOrder	Cost	ListPrice
1	900	3820	Dive kayaks	24	16	$1,356.75	$3,999.95
2	912	3820	Underwater Diver Vehicle	5	3	$504.00	$1,680.00
3	1313	3511	Regulator System	165	216	$117.50	$250.00
4	1314	5641	Second Stage Regulator	98	88	$124.10	$365.00
5	1316	3511	Regulator System	75	70	$119.35	$341.00

|| First Page || Prior Page || Next Page || Last Page ||

Rows:

Figure 2: The application shown in Figure 1 as it appears in a browser.

time; Figure 2 shows it running in a browser. Here's an overview of the steps to create it:

- Place four WebHub components on an empty Delphi form. The menu bar is generated automatically when the WebMenu component is placed on the form. The menu provides a user interface with direct run-time access to WebHub components, their properties, and external files they manage (e.g. HTML pages).
- Add standard DataSource, DataSet, and DBGrid components. (The DBGrid component is optional; it simply gives the Webmaster a non-Web way of viewing and editing the table.)
- Set two WebHub component properties, and five new WebHub components are automatically created and linked at design time because the other components "knew" they needed them.
- Tweak one system-wide control file to tell the WebHub system where the application is located.
- Create an HTML file containing code such as this:

```
<h1>-Page:homepage=,,,A Simple Scrolling Table</h1>
<h2>Parts Table</h2>
<!-- call the WebDataGrid component to do its stuff -->
%=WebDataGrid1=%
```

The resulting application allows any number of surfers to page through the Parts table independently and efficiently. It can be deployed on a wide range of Web servers — even simultaneously via secure and insecure servers on the same system — without recompilation. Modifying a few properties can alter the table fields, rows-per-page, column headers, and index order, as well as the location and appearance of the paging buttons.

Other property changes will make fields editable and add a "post" button, making it easy to implement database updates via the Web. To round out this feature, you must add some Delphi code, but only enough to locate and update the altered record, just as you would do for a conventional database application.

But Wait, There's More

If this were the whole story — WebHub as a tool box of fully-functional components for easily creating Web applications — it would be interesting. But remember, these are *Delphi* components we are talking about; WebHub is an application framework designed from the ground up to be extensible.

The best example of this is the *TWebAction* class — components designed to be dropped on your form, customized with the logic specific to your application, then invoked directly from the HTML, just as the *TWebDataGrid* was earlier. For logic that's likely to be reused in different applications, you simply inherit a new component from *TWebAction* and make it fully reusable — which is how *TWebDataGrid* came into being.

The *TWebAction* component properties are also available to HTML, as is the case with the *TWebCreditCard* component. *TWebCreditCard* automatically validates credit-card number and date information entered by a user, and publishes properties that can be used to display (or act on) the results of the validation.

Church and State

WebHub's design, and the extensible macro system that supports it, result in a true separation of duties: HTML specialists can concentrate on the layout while Delphi programmers implement specific business rules. Therefore, as long as the relevant properties of the custom *TWebAction* components are documented, the HTML writer can "hook into" the required logic at any time, without knowing a whit about how it works internally. Similarly, the programmer needs to know only enough HTML to test the component logic.

This separation of HTML from program logic also has major implications for a deployed site. With many Web development tools, changes to a live site require program-level tweaks and rebuilds — which often involve taking the server down to deploy them. This is because the HTML is embedded within the program code. In contrast, WebHub sites can usually be updated (remotely, if desired) without touching the running application. The benefits, in terms of minimal site "down" time and lower costs for long-term maintenance, are significant.

Once a dynamic Web site extends beyond the realm of the trivial, three considerations become vital: surfer tracking, resource management, and handling peak traffic loads without breaking.

Surfer tracking. WebHub supports surfer tracking (i.e. "saving state") in a way that is dramatically simpler for the programmer than other approaches. Effectively, by the time your custom code is called, the current user's information (including the data gleaned from them during their stay at the Web site) is in memory and available via various component properties. This includes automatic "cloning" of any queries the application has made for that user, ensuring that one visitor doesn't end up looking at another's query results.

This shows the importance of designing your application to deal with *one* user at a time — the current one. WebHub developers shouldn't be concerned about issues such as multi-threading, or managing system resources in complex ISAPI DLLs.

Resource management. Resources such as system memory or client/server database licenses can require careful management

in a Web site that is subject to widely varying loads. WebHub's query-cloning feature can be preset to manage a finite pool of open queries, ensuring an optimal balance between quick response (because cloned queries need not be re-run as the surfer pages through a DataSet), and the cost of back-end licenses.

Handling traffic. When a site is experiencing high traffic, this must be managed properly by queuing requests. If the site is really busy, it should return a message saying "we're busy; try again later," after a specified time. WebHub handles these elements in a bulletproof fashion, leaving the application developer free to concentrate on their custom logic.

Key WebHub Components

Component	Description
TWebApp	Represents and manages the Web application as a whole. It provides direct pointers to the essential components and properties needed while writing a Web application. Variants of this component are provided that have cut down or extended HTML macro processing, including easy database-access macros.
TWebSession	Does for surfers what *TWebApp* does for the application as a whole. It tracks each surfer's data from arrival at the site, including data entered on HTML forms, to application-specific surfer data and component-state data.
TWebServer	Encapsulates the properties of the Web server being used. It presents all the CGI and system environment data passed from the Web server to the application as accessed string and list properties.
TWebAppOutput	Sends output from the application to the Web server. Typically, the output consists of HTML, but it can be a different MIME type (e.g. binary). Furthermore, this component can perform WebHub macro expansion during transmission, allowing a programmer to send HTML "chunks" and macros that have been defined externally by the HTML specialist.
TWebCommandLine	Connects the application to the Runner and the Hub, and brings each page request into the application. Best regarded as the "input" part of the equation, while *TWebAppOutput* is the "output."
TWebInfo	Keeps track of other components — it monitors open applications, open pages and sessions, and assists in saving state. It is the closest thing to a "control component" in WebHub.
TWebMenu	Automatically checks other WebHub components and builds a menu of their management functions, providing an instant user interface.
TWebAction	A component class that can be directly called from HTML to perform a given task, enabling access to pieces of reusable custom Delphi logic.
TWebDataGrid	Presents tabular data such as static database tables or the results of dynamic queries. It automatically manages display of different data types, with surfer-selectable index order, display sets, and page sizes (if desired). It provides paging controls, but these can be overridden by the developer. Hot links can be made within any field — these links can be jumps to detail pages or links to resources such as audio or image files. Other property changes can turn a *TWebDataGrid* table into a grid of editable fields, allowing for database display and update.
TWebDataScan	Provides the core functionality of *TWebDataGrid*, but enables the developer to output each row of the data, providing a flexible free-form output and table format. (Related components include *TWebDropDown*, *TWebOutline*, *TWebStringGrid*, and *TWebListGrid*.)
TWebScanList	Handles paged display of non-database lists and information, useful for result sets from a source other than a database query (i.e. an indexing engine).
TWebDataForm	Supports "instant form" presentation and provides display-only and editable views of database fields. A simple macro parameter determines which view will appear, and a *TWebDataPoster* component can be added to handle table updating.
TWebCreditCard	Used to verify credit-card number and expiration by applying standard number-validation techniques.
TWebMailForm	Takes surfer form input and automatically e-mails data to a preset destination. Fully configurable, allowing one Web application to support many sites, forms, e-mail templates, and target addresses. For developers creating mail applications, such as bulk e-mailing from a site, the *TWebMail* component provides lower-level functionality.
TWebHTTP	Makes Web-page requests from within a Web application — useful for sites that need to "watch" or draw information from other sites. Like *TWebMail* and *TWebTelNet*, this component depends on the *TWebSock* component that can also be used to implement direct TCP/IP communications with external services, or even with applications on the surfer's machine, such as a Java applet.
TWebPicture	Creates on-the-fly, custom images for display in Web pages. Similar logic can be used to convert images from database BLOBs into files that can be viewed in a browser.

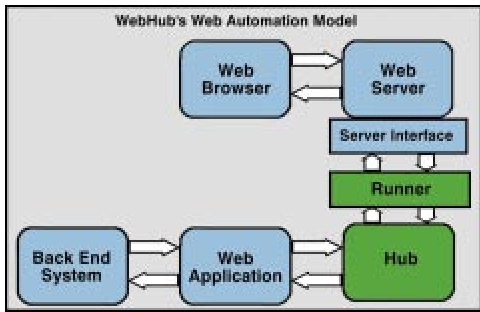


Figure 3: The architecture: WebHub's Runner and Hub model allow you to create simpler Web applications.

Better yet, WebHub's architecture makes it exceptionally easy to upscale the site and implement load-sharing. This may include moving "media" files (i.e. images, video, and sound) to another server. It may also run additional copies of your application on the original server, and set up a cluster of servers handling the same site in a round-robin fashion. WebHub's design permits any, and all, of these options without needing to change a line of code, leading to near- linear scalability of site throughput without redesigning your application.

WebHub's Architecture

A quick look at WebHub's architecture will clarify how this is possible (see Figure 3). The blocks shaded in blue represent features common to all automated Web sites. However, they also hide many areas of complexity, such as the extent to which the Web application needs to be customized to match different server interfaces, or extended to handle some of the queuing and resource features noted earlier.

The green blocks are unique to WebHub. The Runner is a small executable or DLL (typically 100KB) that manages all server-specific logic. Select a different Runner from the set supplied with WebHub, and you can support a different server-interface protocol (e.g. ISAPI, CGI-WIN, or CGI-BIN). In the case of the CGI protocols, the fact that the Runner is small is a major bonus. As CGI loads and unloads the target application for each request, enormous overhead is avoided because Runner is tiny.

The Hub provides bulletproof request queuing and also acts as a manager for Web applications deployed on the site. With the Runner, the Hub implements the basic surfer identification and tracking mechanisms that make saving-state and several other advanced features possible. Significantly, the Web application in this model now becomes a "lite" version. It no longer needs to be concerned about server, queuing, or resource issues, and instead can use the custom page-generation and back-end system interaction logic on a "one request for one surfer at a time" basis.

The full potential of this architecture is seen early on in the development process. For example, instead of trying to debug complex, thread-safe DLLs that can crash the entire Web server when an error occurs, WebHub developers can do their development and live debugging in the Delphi IDE. Application bugs don't affect the Hub, Runner, or Web server, and new versions can be built, deployed, and tested in seconds without taking the server down.

Scalability derives from the Hub's ability to manage various applications, each of which provides different services required by the site. A truly modular solution is possible because the same surfer information can be seamlessly shared across and among applications. Apart from the improvements in program maintainability, this is important when some requests take significantly longer to service than others. They can be routed to a Web application built specifically for that purpose, while "regular" requests from other surfers are handled by the usual application. The result is that other surfers can have their requests serviced, although one surfer may have made a request that would normally tie up the site for several seconds.

INFORMANT FACT FILE

WebHub is a full-featured, extensible component set for building dynamic server-independent Web sites. Its architecture provides significant performance, security, and reliability benefits, allowing developers to build modular Web applications that need only implement the custom features of a given site. The components support Delphi 2 and 3, with C++Builder support pending. Development platforms can be Windows 95 or Windows NT 3.51/4.0, using any of the Web servers commonly available for these platforms. Deployment must be on Windows NT.

HREF Tools Corp.
300 B St., Ste. 215
Santa Rosa, CA 95401
Phone: (707) 542-0844
Fax: (707) 542-0896
E-Mail: webhubsales@href.com
Web Site: http://www.href.com
Price: Entry level system is US$299 (via ZAC Catalog); Professional version is US$595.

The Hub can manage multiple copies of an application called in a "round robin" fashion (including copies hosted on neighboring servers). With modular design, this means a site manager can tune a busy site to strike a precise balance between the number of requests of a given nature, the time taken to handle those requests, and the resources provided (in terms of copies of the application) to make the site run smoothly.

Critically Speaking

If there is any bad news about WebHub, it must center on the fact that it's a sizeable product that is being continually revised and extended. The learning curve for "the basics" is quite good, but with over 30 Web components and 70 other utility components available, it would take a very confident developer to claim in-depth knowledge of the entire product. This is not helped by the fact that (at the time of this writing) many properties for non-core components are not fully documented in the 3MB Help file and 200-page manual that ships with WebHub.

Furthermore, with new revisions and features being released at a steady pace, a sensible eye must be kept on version management, especially when deciding which WebHub version to use for deploying a completed site. On the other hand, the free e-mail list server provides excellent, friendly, and timely technical support. The range of example applications provided with WebHub covers most common — and many uncommon — Web site needs. These include surfer authentication, a shopping cart, form mail, integration with VRML, Java and JavaScript, remote administration, live graphic creation, data-driven queries, database search/detail drill down, database editing, surfer-driven site customization, multilingual pages, and more.

Conclusion

Developing dynamic Web sites is an exciting, challenging, and distinctly non-trivial exercise. All too often, the traps are not obvious until *after* a site has been deployed and starts to experience real traffic. With WebHub as a starting point, a developer can safely ignore many otherwise irksome details, and be assured that if an unexpected need does arise, the development framework is sufficiently flexible and well supported. In other words, this is a tool you can use with confidence for serious applications. ∆

Peter Hyde is the author of TCompress and TCompLHA component sets for Delphi/C++Builder, and Development Director of South Pacific Information Services Ltd., which specializes in dynamic Web site development and is the New Zealand technical support center for WebHub. He can be reached by e-mail at peter@spis.co.nz or via http://www.spis.co.nz.

Steve McConnell Connects Again with *Rapid Development*

Steve McConnell's previous work, *Code Complete* [Microsoft Press, 1993], quickly became the book every software developer serious about the craft had to own and study. His latest book from Microsoft Press, *Rapid Development*, is a worthy follow-up. Subtitled *Taming Wild Software Schedules*, this unusual book seeks to expose those practices and attitudes that contribute to inferior software and schedule delays. Author Steve McConnell offers solutions that enable us to produce the best software in the shortest possible time.

Scheduling, however, hardly begins to encapsulate the topics covered in this ambitious treatise. After briefly defining rapid development, McConnell lays out a basic strategy to avoid classic mistakes: Use development fundamentals; manage risks; and schedule skillfully. He also introduces what he calls the "four dimensions of development speed": people involved, processes used, products to be produced, and technological tools used. These themes are developed throughout the book.

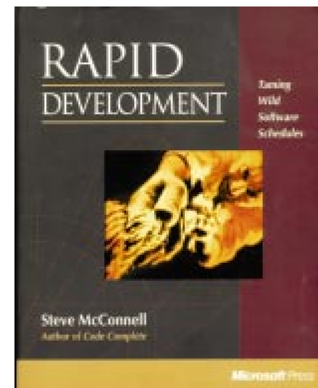
It would be tempting, and certainly easier, to discuss each of the dimensions of software production in isolation. Instead, McConnell takes a holistic approach that considers their relationship to one another. He states "you can focus on all four dimensions at the same time." This becomes particularly evident in the many case studies. Some dramatically demonstrate how to create a programmer's hell. Others provide examples of how to introduce a level of sanity into the process.

McConnell's approach works well — you can hardly miss the point. First, you are presented with a set of principles. Then a case history provides a concrete example. Finally, the principles are expanded so the mistakes or virtues inherent in the case study become clear and applicable.

Rapid Development is divided into three sections: "Efficient Development" introduces the four dimensions and other fundamental principles; "Rapid Development" examines critical issues that must be considered to intelligently plan and complete a software project; and "Best Practices" provides successful examples of approaches involving one or more of the four dimensions.

Two essential topics presented in the opening section are quality assurance and risk management. Lack of attention to quality at the beginning of a project can greatly delay its completion. A bug is easier to find and fix soon after it is created rather than after adding layers of additional code. Likewise, ignoring the risks and pitfalls of a particular approach can create havoc with an otherwise reasonable schedule. With the latter topic, we learn how to identify and analyze risks, then how to prioritize and control them. McConnell points out there is a difference between taking calculated risks and simply gambling.

The heart of the book, the Rapid Development section, contains a wealth of important topics, including scheduling. Some of these, such as motivation and teamwork, concentrate on peopleware. The positive aspects of using rewards and incentives to motivate, and conversely the contributors to low morale, are covered in the section on motivation. These sections alone may inspire many team programmers to consider purchasing Rapid Development as a gift for their bosses! In these pages you'll learn the fine art of planning a software project so that it has a good chance of being completed on schedule and meeting specifications. The section on productivity tools provides a realistic approach to the acquisition and use of new technology to expedite development, and should be



of interest to developers and managers alike.

The final section introduces a series of best practices — approaches to rapid development that have proven successful. McConnell makes it clear that some are mutually exclusive, some are obvious, and some may take getting accustomed to. Each section begins with a summary of its efficacy for each of five considerations, its major risks, and its interactions and tradeoffs. A short description of the practice itself, an outline of how to use it, and a detailed discussion of related issues follows.

Throughout the book, McConnell deals with each topic thoroughly and with a wealth of concrete examples. The large collection of case studies also adds clarity and power. Each subsection concludes with a list of sug-

"Rapid Development" continued on page 43

Rapid Development (cont.)

gested readings, and at book's end there is an exhaustive bibliography of 230 books and articles.

Stylistically, this is a brilliant synthesis of a scholarly treatise, a popularization of more specific and difficult primary sources, and a collection of fascinating case studies. The organization is superb. Therefore, it is very easy to re-locate particular examples after you have read the entire book.

An important question remains: Who will benefit from reading this book? Sections that explain how to work more effectively with people, and, to some extent, processes, will be helpful to software development managers and team leaders. However, many other discussions apply to independent developers. If you feel frustrated with your current approach to development; if you are constantly falling behind schedule; if you need ammunition to convince your managers to adopt a more reasonable approach to the project goals and timelines they impose on you — Steve McConnell's *Rapid Development* is for you.

— *Alan C. Moore, Ph.D.*

Rapid Development: Taming Wild Software Schedules by Steve McConnell, Microsoft Press, One Microsoft Way, Redmond, WA 98052-6399, (206) 936-2810.

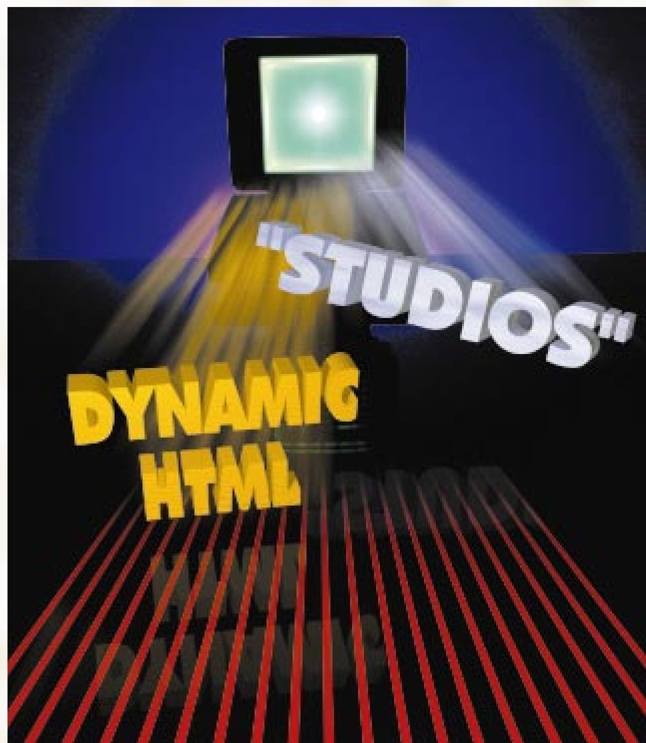
ISBN: 1-55615-900-5
Price: US$35 (647 pages)

Beyond Hype

Two Technologies with Staying Power

In today's climate, determining whether a new software technology is a long-term trend or a quick-hit fad can be a full-time job. Indeed, thumbing through the pages of *PC Week*, *InfoWorld*, etc., one can easily succumb to the seemingly endless dribble of hype that permeates our industry. If we're not on guard, we might actually begin to believe it.

While we may not be able to depend on the validity of a particular technology until the market makes its determination, I think we can come close. This month, I highlight two emerging technologies I believe will be among the software development trends that will last as we move into 1998.



"Studios"

For most developers, the days of working with a single development tool are over. With the diversity of application architectures — desktop, client/server, three-tier, intranets, extranets, and the Internet — you may have to employ several tools for a single application. It's safe to say that, in addition to working primarily with Delphi, many of you work with other development tools along the way.

Until recently, little integration existed across development-tool environments — even from the same vendor. This is changing, and in some important ways. Microsoft's Visual Studio was the first integrated development environment that allowed you to work with multiple development tools from within the same IDE shell. So, whether you use Microsoft Visual InterDev, Visual C++, or Visual J++, you can work within the same environment. Similarly, PowerSoft recently announced its own studio version, coined PowerStudio, to incorporate PowerBuilder, Power ++, and other PowerSoft tools. A developer "studio" offers several advantages over a single-tool, IDE approach. Not only can developers better integrate components from these various tools, they can also learn a new tool or language without being forced to master a new IDE. Will Borland follow suit with a studio that will integrate Delphi, C++Builder, and JBuilder? (I sure hope so, though I'd like to make one humble plea to Borland: Please avoid using the horribly overused words "Builder" and "Studio" in such a product's name.)

Dynamic HTML

The Web industry is filled with hype, but one technology I am convinced will prove to be a key trend is Dynamic HTML. Twelve months ago, many people thought client-side scripting languages — JavaScript and VBScript — would be a fad, superceded by Java and ActiveX for providing dynamic content to the Web. While applets and controls serve a purpose, their component-based nature has proven very specific and contained. However, the power, speed, and flexibility that Dynamic HTML brings to the Web are compelling. While widespread use of Dynamic HTML is surely hindered by the standards war between Netscape and Microsoft, I am convinced that a large subset will become ubiquitous by the end of 1998. ∆

— Richard Wagner

Richard Wagner is Chief Technology Officer of Acadia Software in the Boston, MA area, and Contributing Editor to Delphi Informant. *He welcomes your comments at rwagner@acadians.com.*