



*HTML, What's This? and More*

Cover Art By: Tom McKeith

## ON THE COVER



### 5 HTML Help — Ron Loewy

Mr Loewy introduces HTML Help — a collection of software tools, technologies, and specifications defined by Microsoft as a replacement for Windows' aging online help system. He then goes one step further, and demonstrates its use from Delphi.



### 10 What's This? — David Hemphill

You know what it is — that little question-mark button. You click on it first, then click on what you're interested in to get help. You never gave it much thought, however, until it showed up as a client's specification. Fortunately, when asked "What's this?" Mr Hemphill replies: "Delphi!"

## FEATURES



### 15 Visual Programming

#### Setting Limits: Part II — Gary Warren King

Continuing his discussion of form-size control, Mr King shares a component that intercepts messages intended for the form, and modifies them to alter the behavior of the form.



### 21 OP Tech

#### Delphi Import/Export: Part I — Bill Todd

Mr Todd begins a two-article series. This month's topic is getting data into (or out of) delimited or fixed-length ASCII text files, and out of (or into) a database — or was it the other way around?



### 28 Columns & Rows

#### AS/400 Shortcut — Bradley MacDonald

As it turns out, there are several ways to run commands or programs on an AS/400 from Delphi, without using an RPC call. Mr MacDonald examines one — the Query component.



### 31 In Development

#### Any Port in a Storm — Alan C. Moore, Ph.D.

Do you need to move an application from 16-bit Windows to 32-bit Windows? Or are you developing a program that needs to compile into a 16- or 32-bit version? Dr Moore explains.



### 34 Algorithms

#### Linked Lists — Rod Stephens

Supplying Delphi implementations of the extremely flexible data structures known as linked lists, Mr Stephens demonstrates what to do when the data is too dynamic for arrays.

## REVIEWS



### 38 HelpScribble

Product Review by Alan C. Moore, Ph.D.



### 41 DotHLP

Product Review by Cary Jensen, Ph.D.



### 43 WinHelp Office 5.0

Product Review by Gary Entsminger

## DEPARTMENTS

### 2 Delphi Tools

### 4 Newline

### 46 From the Trenches by Dan Miser

### 47 File | New by Alan C. Moore, Ph.D.





## Innoview Releases Multilanguage for VCL 3.0

Innoview Data Technologies Ltd. announced

*Multilanguage for VCL 3.0*, a collection of components and tools for Delphi and C++Builder that globalizes applications by making them multilingual. A multilingual application can handle any number of different languages, and can switch from one language to another on-the-fly.

Multilanguage for VCL 3.0 is a localization and globalization solution for corporate developers, ISVs, VARs, and system integrators that delivers large-scale applications internationally. The enhanced support for three different types of character sets, including bi-directional (Arabic and Hebrew) and double byte (Far East) character sets, ensures compatibility with all Windows language editions.

Version 3.0 offers enhancements in performance, compatibility, and support for standards, while keeping the core technology intact. Multilanguage for VCL 3.0 provides Fast-and-Easy Translation Technology and the Dictionary Editor utility, which works together with the software components compatible with Delphi and C++Builder environments.

At press time, Multilanguage for VCL 3.0 was scheduled for release in March, 1998.

**Innoview Data Technologies Ltd.**

**Price:** US\$290 and US\$580 for Standard Edition without and with source code, respectively; US\$790 and US\$1,580 for Professional Edition without and with source code, respectively.

**Phone:** +358-9-4762 0550

**Web Site:** <http://www.innoview-data.com>



## Digital Metaphors Ships Piparti 3.0

Digital Metaphors announced the release of *Piparti 3.0*, an update to the company's native Delphi reporting tool. Piparti 3.0 provides an extensible platform for advanced reporting requirements in four areas: report layout and design, data access, output device support, and dialog customization.

This new release offers new features for designing complex reports without code, including free-form sub-reports for handling multiple master/detail relationships, multiple section reports, and side-by-side presentation of data; regions, containers for grouping report components (similar to Delphi's *TPanel*); and precise layout positioning capabilities.

In addition, data access and report output are open and extensible via *DataPipeline* and *Device*.

Piparti 3.0 ships with a *BDEPipeline* that supports standard Delphi data access and a *TextPipeline* for accessing ASCII data.

Piparti 3.0 also includes a forms API, which allows for the replacement of any of Piparti's built-in dialogs without changing the source.

Piparti 3.0 and Piparti Pro 3.0 include versions for Delphi 1, 2, and 3, and full source code.

**Digital Metaphors**

**Price:** Piparti 3.0, US\$249; Piparti Pro 3.0, US\$495.

**Phone:** (214) 800-8760

**Web Site:** <http://www.digital-metaphors.com>

## Torry's Delphi Pages CD-ROM Edition Released

The second edition of *Torry's Delphi Pages*, a collection of components and tools for Delphi developers, was released and is available on CD-ROM. The new release offers more than 1,300 components for Delphi 1.x, 2.x, 3.0, and C++Builder, including *ToolBar97*, *XToolBar*, *TStatusLine*, and *TExplorerButton*. It includes code samples, Delphi and C++Builder programming tools, such as

Quick Glyph and HelpScribble, and freeware and shareware applications, such as eAuthor/Site, EditPad, and HTML Buddy.

The CD-ROM also contains *Delphi Knowledge Base* by Marko Tietz, *Delphi-News* by N. Hartkamp, and *The Unofficial Newsletter of Delphi Users* by Robert Vivrette.

**Torry's Delphi Pages**

**Price:** US\$36

**Web Site:** <http://torry.magnitka.ru>



## Dunstan Thomas Releases IB\*Doc 1.1

Dunstan Thomas Ltd. released IB\*Doc 1.1, a utility that analyzes and generates reports from InterBase databases. The product creates a number of different reports that can be used to provide a hard copy of the database for disaster recovery or to document an existing database for further analysis or development. These reports include Domain Report, Exception Report, Generator Report, Stored Procedures (both list and detail), Tables (both list and detail), Triggers (both list and detail), UDFs, Users, and Views (both list and detail). IB\*Doc 1.1 is available free of charge and can be downloaded from the Dunstan Thomas InterBase site at <http://www.interbase.dthomas.co.uk/files/ibdoc.zip>.

## PowerBBS Computing Releases Delphi2Java/VB2Java Toolkit

PowerBBS Computing released the *Delphi2Java/VB2Java Professional Toolkit*, which includes the tools to convert Delphi and Visual Basic applications to Java.

The professional version adds support for JDK 1.1, allowing developers to choose from JDK 1.02 or 1.1 Java output. The Professional Toolkit also includes complete Java source code.

The Professional Toolkit offers the JavaSizer, which improves a program's interface by shrinking or stretching the controls and fonts on Java screens according to the client's resolution capabilities.

The Professional Toolkit can also center the form on

initialization.

The Professional Toolkit provides customizable Java comments to facilitate future Java development. Additionally, it will transfer comments in original source code to the Java source file.

In addition, the Professional Toolkit includes

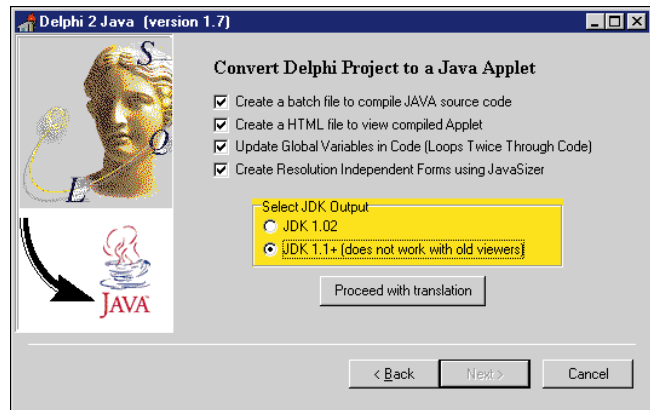
five free consulting hours, which may be used to assist developers during the conversion process, including customizing the actual conversion software.

### PowerBBS Computing

**Price:** US\$799

**Phone:** (516) 938-0506

**Web Site:** <http://www.javadelphi.com>



## InterBase Releases InterBase 5.0

InterBase Software Corp. announced the availability of *InterBase 5.0*, an embedded database management system that integrates with a variety of development tools, including Delphi, C++Builder, IntraBuilder, and Visual dBASE.

InterBase 5.0 offers a multi-client, multi-threaded architecture that eliminates bottlenecks and reduces the overhead required for multi-process tasks.

InterBase 5.0 includes InterClient, an all-Java JDBC driver that allows high-performance connectivity and easier deployment, reducing deployment and life cycle costs to Java client/server developers.

InterBase allows any client platform to communicate transparently with any server platform that the product supports. It also offers "Year 2000" correctness, stored and

select procedures, and server-side triggers.

### InterBase Software Corp.

**Price:** US\$40 per copy (units of

## devSoft Announces New IP\*Works! Package

devSoft Inc. announced a version upgrade of the *IP\*Works!* package, a TCP/IP developer's toolkit for Microsoft Windows and Windows 95/NT.

The toolkit provides Internet and intranet development components for Delphi, C++Builder, Visual Basic, Visual C++, and other environments.

The new version offers more flexibility, more components, and more sample applications.

New features include a new NetDial control for managing dialup connections, the Multicast component, support for custom commands in the FTP/SMTP/POP/NNTP

100); US\$200 for server; US\$150 per additional user. Volume discounts are available.

**Phone:** (888) 345-2015

**Web Site:** <http://www.interbase.com>

components, better firewall support for the FTP control, and support for resuming interrupted FTP downloads/uploads.

The version upgrades are offered for all editions of IP\*Works!, including IP\*Works! Delphi Edition, IP\*Works! ActiveX Edition, IP\*Works! C++ Edition, and IP\*Works! C++Builder Edition, which are now at Version 2.0. IP\*Works! Visual Basic Edition is now at Version 3.0.

### devSoft Inc.

**Price:** US\$195; owners of previous versions of IP\*Works! may upgrade for US\$95.

**Phone:** (919) 493-5805

**Web Site:** <http://www.dev-soft.com>



May 1998



## CNET Names JBuilder One of Ten Best Computer Products of 1997

Scotts Valley, CA — Borland's JBuilder family of Java development tools has been named one of the ten best new computer products of 1997 by CNET. In a survey in *ComputerWorld*, JBuilder was named the second-best corporate application development tool (behind Borland's Delphi Client/Server Suite) in terms of value and return on investment.

JBuilder also received other

awards and accolades, including the *Java Developers Journal* World Class Award, *developer.com's* Users' Choice

Award, 1997 Best of Comdex finalist, *Windows* magazine's Recommended WinList, and others.

## Borland Spins Off US Channel Sales and Marketing

Scotts Valley, CA — Borland announced that it has spun off its US channel sales and marketing organization as an independent company. The new company, Frontline Now!, will represent and market Borland's

development tools, including Delphi, C++Builder, and JBuilder, to distributors and resellers in Borland's US sales channel.

Over the past nine months, Borland has been transitioning its sales organization to focus on direct selling of client/server and enterprise products to large corporations. As a result, Borland generates approximately 50 percent of its revenue from these products.

Frontline Now! will establish relationships with PC software and hardware vendors offering business applications, productivity, and utility products, and will provide a complete turn-key channel marketing and sales solution. Services include distributor, retail, VAR, catalog and corporate reseller sales and account management, and development and implementation of comprehensive custom channel marketing programs.

## Borland Reports Fiscal Results

Scotts Valley, CA — Borland announced results for its fiscal year 1998 third quarter and nine months ending December 31, 1997. For the quarter, net revenues were US\$43,015,000 (compared with US\$36,756,000 for the third quarter of the previous fiscal year), an increase of 17 percent. Net revenues for the nine months ending December 31, 1997 were US\$127,485,000 (compared with net revenues of US\$114,208,000 for the nine months ending December 31, 1996), a 12 percent increase.

Net income of US\$2,826,000 for the third quarter resulted in basic and diluted earnings per share of US\$0.07 and US\$0.06, respectively. Borland reported a net loss of US\$29,371,000, or US\$0.81 per share, for the same quarter a year ago.

For the nine months ending December 31, 1997, net income was US\$4,423,000, resulting in basic and diluted earnings per share of US\$0.10. For the nine months ending December 31, 1996, net loss was US\$65,490,000, or US\$1.80 per share.

On January 1, 1998, Borland went to a calendar fiscal year, versus a fiscal year beginning April 1, making 1998 a three-quarter fiscal year.

During the third quarter of fiscal year 1998, Borland released JBuilder Client/Server Suite, Delphi Enterprise, Visual dBASE 7 for Windows 95 & NT, Delphi/400 Client/Server Suite for IBM AS/400 developers, and InterBase 5.0. In this period, revenues from client/server, enterprise, and Internet products made up 56 percent of total net revenues (compared with 37 percent during the same period in fiscal year 1997). For the first nine months of fiscal year 1998, these products made up 53 percent of total net revenues (compared with 36 percent during the same period in fiscal year 1997).

## Borland Unveils Business Solutions Program

Scotts Valley, CA — As part of its initiative to help corporations build, deploy, and manage its distributed corporate Information Network applications, Borland unveiled the Business Solutions Program (BSP), a comprehensive program for developers/system integrators, commercial application developers, training partners, and tool and component builders. The program applies to US partners only, with the exception of the program for tool and component builders, which is available to

partners worldwide.

BSP puts all of Borland's US partner activities under one program and is intended to support the full range of Borland's products, including Delphi, JBuilder, C++Builder, IntraBuilder, InterBase, MIDAS, and Entera. BSP's specialized programs are designed for partners of various sizes. Detailed information on the benefits and specific qualifications of each of the programs can be found at <http://www.borland.com/programs/bsp/>.





## ON THE COVER

Delphi 3 / HTML Help

*By Ron Loewy*



# HTML Help

## The New Online Help Standard

**H**TML Help is a collection of software tools, technologies, and specifications defined by Microsoft as a replacement for Windows' aging online help system. HTML Help is based on HTML — the undisputed standard for hypertext document delivery. It's built on top of Microsoft's HTML layout technology, and provides most of WinHelp's desired features — without many of its shortcomings.

Windows Help (WinHelp) became a popular online help and hypertext delivery format with the introduction of Windows 3.0. WinHelp offered a uniform operating-system-supported method of delivering large amounts of content with rich text layout and graphic support. The move to Windows 3.1, Windows 95, and Windows NT was accompanied by enhancements to the WinHelp viewer and compiler. New features such as table of contents and enhanced search capabilities were introduced. These features provided authors and developers with ways to create better online help with more functionality and an easier-to-use interface.

Despite these advancements, WinHelp's roots in the 16-bit world of Windows 3.0, its reliance on a proprietary, undocumented file format (.HLP files), and Microsoft Word-specific source format (.RTF files) were a constant cause of frustration to online help authors, developers, and, therefore, users.

In February 1996, Microsoft announced an initiative to redesign the standard WinHelp delivery mechanism around the increasingly popular HTML format. This allowed the Microsoft online help development team to start the online engine design from scratch,

and deliver the many enhancements frequently requested by the authoring and development communities. The developers' goals were to offer WinHelp's traditional strong points — compressed files, multimedia support, rich text layout, and fast information retrieval and display — with the latest HTML technology available from Microsoft.

### HTML Help Architecture

HTML Help is based on a collection of components built into Microsoft's Internet Explorer Web browser (and future Microsoft operating systems, such as Windows 98 and NT 5.0). The Viewer is based on the HTML layout engine used by Internet Explorer (IE). This allows HTML Help to offer support for the latest HTML technology, including scripting, Java, Internet connectivity, Dynamic HTML, and ActiveX support.

HTML Help offers several extensions to the HTML layout engine, including support for an interactive table of contents, keyword index, and full text search. The extensions are provided as an ActiveX component. Some of the extensions (e.g. table of contents and keywords) are also accessible using a Java applet.



HTML Help introduces a new file format: compile HTML (or CHM) files. These files use structured storage, accessible via standard COM interfaces optimized for the retrieval of multimedia information from large collections of files. A large online help file that compresses

many HTML files into one CHM file can be orders of magnitude smaller in size than the individual files. When you add the fact that individual files take a lot of space on the hard disk under the standard FAT system — even if their physical size is small — the use of CHM files becomes more attractive.

### What HTML Help Offers Developers

HTML Help offers an interface similar to the old *WinHelp* API function. This makes the conversion process from using WinHelp as the application's online help delivery mechanism to the new HTML Help format easy. The new *HtmlHelp* function takes the same number of arguments, and offers commands similar to those used by *WinHelp*.

In addition to the old method of a popup help window that appears over the application's window, HTML Help offers the ability to display multiple help windows by different applications simultaneously, and the ability to embed the help window in the application's window. In the near future, I believe we'll see help windows embedded in an application's own windows and dialog boxes that offer up-to-the-minute help based on the user's actions.

Windows Help offered a poorly documented way to create extensions to the WinHelp display engine. These extensions were written in proprietary DLLs, and suffered from limited access to the layout engine and the event model of the online help. HTML Help — being an Internet Explorer layout engine client — can be expanded using ActiveX controls, Java applets, and other standard methods popularized on the Web. Developers' capabilities of tailoring the help display and delivery by offering custom functionality has been expanded significantly.

### HTML Help vs. WinHelp: A Quick Look at the Differences

While the *HtmlHelp* API function call is modeled after the *WinHelp* function call, HTML Help offers several important differences. One important difference is that HTML Help windows are owned by the calling application, and are not a separate process (see Figure 1). Because of this difference, multiple help windows from different applications can be active at the same time. Help windows are also automatically destroyed when the calling application terminates.

Another major difference is that HTML Help windows can now be hosted in the calling application's windows. These windows can also notify the owner window of events, allow-

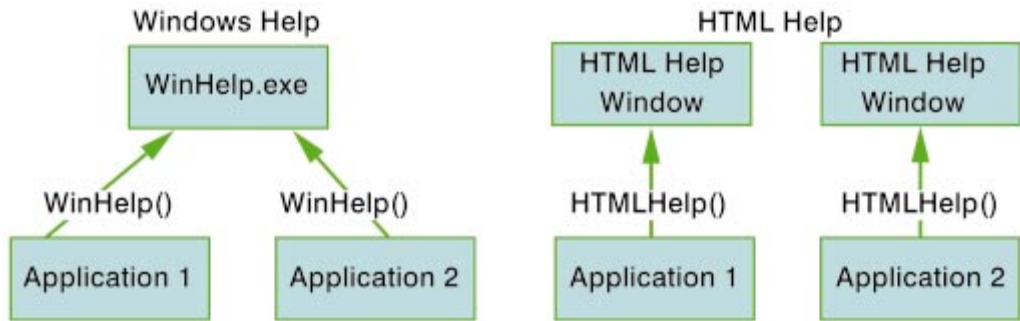


Figure 1: The architecture: HTML Help vs. WinHelp.

ing for a new breed of smart, integrated help in applications. Applications using HTML Help have complete control of the HTML Help windows, including the ability to obtain their handle and even create them in code, on-the-fly.

HTML Help doesn't offer the WinHelp popup windows. If you need fast popup windows that can include graphics and rich text, you might be better off with WinHelp. HTML Help does, however, offer fast text-based popups most suitable for "What's This?" help.

### Authoring HTML Help Files

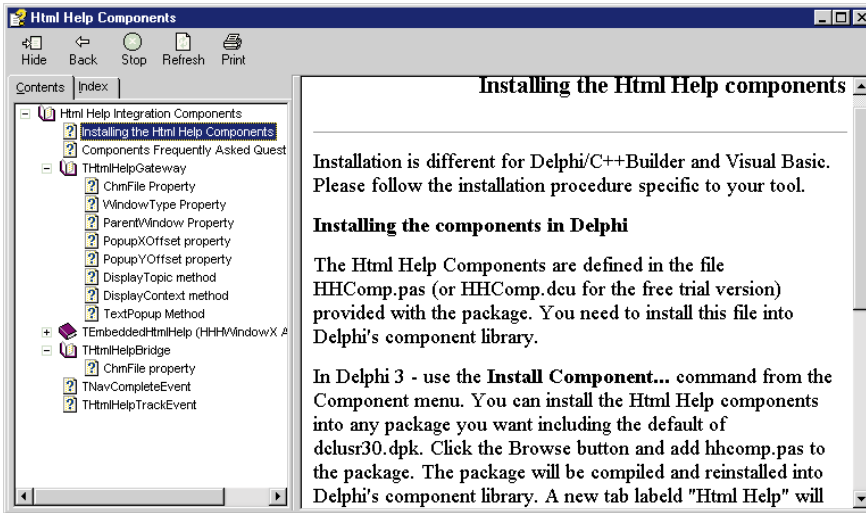
What goes into an HTML Help file? An HTML Help file is a collection of compiled, compressed files stored in a file that uses the CHM extension. The CHM file includes the compressed files in an internal directory structure. You can think of a CHM file as a file system in a file.

The obvious content of the CHM file are the individual HTML files that make up the help document. Every topic that's referenced in the help document needs to be a part of the CHM file. Topics referenced via a complete URL — instead of a relative link — are the exception to this rule.

Many help topics use graphics and multimedia elements (audio, video, etc.). These elements should be part of the CHM file (unless they're referenced via a complete URL). You might be afraid the task of "compiling" all the media and graphic files used by the help topics into the CHM file is complicated and time consuming. Fortunately, the HTML Help compiler is "smart" enough to parse the HTML files as they're being compressed, and to automatically pull the referenced media and graphic files into the CHM archive.

A good help system exposes the information enclosed in it, and allows the user to access this information easily. Unfortunately, standard HTML doesn't offer the table of contents and index that are taken for granted by book readers, and that have been available in help files since WinHelp 4.0. HTML Help offers a standard (based on the proposed Web Collections specification) way to define table of contents (TOC) and index files. These files need to be included in the CHM file as well.

Online help is frequently used by applications to provide context sensitive help about the application. The application designer might want specific windows that will host the help



**Figure 2:** An example of tri-pane HTML Help.

content. HTML Help offers a way to define “help windows,” and the definition needs to be part of the CHM file as well. In addition to the modules mentioned, other files — such as style sheet files, and other support media and Web-related files — can be incorporated into the CHM archive.

### Topic Files

Topics in the help document are defined in standard HTML files. Generally speaking, everything you can put in an HTML file prepared for the Web can go into an HTML file prepared for your help document. The following points should be considered:

- Relative links will be resolved from the file system internal to the CHM file. Make sure that you include all the necessary files in the CHM archive.
- Images and media files should also be referenced using relative links. Those that do will be included as part of the CHM archive.
- Scripts (JavaScript, VBScript, etc.) can be used in help topics, but be sure to test your pages after they’ve been compiled. My experience indicates that some scripts cause problems when they’re executed from within a CHM file. In other words, the fact that a script works on a Web site does *not* ensure that it will work in a CHM file.

### Table of Contents

HTML Help includes a table of contents control capable of displaying sources based on the proposed Web Collections standard. The standard HTML Help documentation includes information about the format of HTML Help Contents files (HHC). Most HTML Help authoring tools offer visual tools that will help you construct these files without learning the details of the format.

When you create HTML Help documents, you have to decide if you want the help to appear in the standard browser window, or in the tri-pane window supplied by HTML Help. The tri-pane window displays a window divided into three panes: the toolbar at the top, the navigation pane on the left, and the help topic content on the right (see [Figure 2](#)). When

you use the tri-pane window, all you must do to create a table of contents is to create the HHC file and include it with the CHM archive. If the browser is your target user interface, you will need to create a Frameset file and an HTML page that hosts the table of contents control. Microsoft provides an ActiveX component and a Java applet; both can be used to display the table of contents.

### Keyword Index

HTML Help includes an index control capable of displaying sources based on the proposed Web Collections standard. The standard HTML Help documentation includes information about the format of HTML Help index files (HHK).

format of HTML Help index files (HHK).

Like table of contents files (HHC), index files can be used in an HTML Help document displayed in the browser, by using a Frameset file, or the standard tri-pane window supplied by HTML Help.

### HTML Help “Windows”

A help system can use different windows to display the help document. HTML Help allows the creation of window types that can be used when the help is displayed. The windows are defined in the HTML Help project file (HHP file), and you can define parameters, such as window style, placement, toolbar format, and more. An application can use the different window types to display the different kinds of help topics.

Most HTML Help authoring tools will allow you to create window definitions visually without learning the syntax used in the HHP file. If you need to learn this syntax, however, it’s covered in the standard HTML Help documentation. In addition to window definitions in the authored file, HTML Help windows can be defined in code.

### Context Sensitive Help

Context Sensitive help is the ability to call help topics from a running application using help topic identifiers, i.e. numeric values assigned in the application to refer to a specific topic. HTML Help CHM files can include the mapping between a topic’s URL to its numeric identifier, thus allowing an application to call help display using the numeric ID.

HTML Help numeric mapping is done using the “alias” section of the HHP file, where a URL is “aliased” to an identifier. Although the HTML Help documentation states that numeric values can be used in the alias section, presently the HTML Help compiler seems to ignore this option, so it’s important you create a string alias to the URL. For example:

```
IDH_MAIN=homepage.html
```

will set the alias IDH\_MAIN to point to the URL homepage.html. After the alias has been set, you need to add a standard C header file to the map section of the HHP file. Use the syntax:

```
#include myheader.h
```

to use the header file named myheader.h. The header file should include the mapping between the alias values and the numeric values. Assuming that IDH\_MAIN is mapped to the numeric value 251, the header file will include a line like the following:

```
#define IDH_MAIN 250
```

## How to Author HTML Help

The standard HTML Help distribution from Microsoft is available as a free download from <http://www.microsoft.com/workshop/author/htmlhelp/>. The download includes the HTML Help Workshop — a bare-bones HTML editor and the HTML Help project editor. HTML Help Workshop comes with a good image capture and conversion utility, and the original htmlhelp.h header file.

Most of the authoring tool vendors offer some help for the help authors. Some offer conversion of old WinHelp projects to the new HTML Help format (Microsoft's HTML Help Workshop does it too), and some offer native HTML Help authoring tools.

## Using HTML Help in Your Apps

HTML Help is accessed from applications using the new *HtmlHelp* API function. This function is modeled after the *WinHelp* API function call and offers similar parameters and command sets. The original C header definition is:

```
HWND HtmlHelp(HWND hwndCaller, LPCSTR pszFile,
              UINT uCommand, DWORD dwData);
```

The standard HTML Help documentation provides detailed information about the *HtmlHelp* call. The standard distribution includes the htmlhelp.h header file that defines the HTML Help constants and calls. For Delphi programmers, a file named HtmlHelp.pas is available with this article (see end of article for download details).

If you want to link to the HTML Help DLL dynamically, you need to use the *LoadLibrary* function call to load HHCtrl.OCX, and use *GetProcAddress* on HtmlHelpA to get the function address. This will give you the pointer to the *HtmlHelp* function. Because of Delphi's naming restrictions, this function is named *HH* in the converted HtmlHelp.pas file (you cannot have a function that shares the name of the unit). This translates to the following Object Pascal definition:

```
function HH(hwndCaller: THandle; pszFile: PChar;
            uCommand: Cardinal; dwData: Longint): THandle;
```

## Calling Context Sensitive HTML Help

HTML Help offers context sensitivity through the HH\_HELP\_CONTEXT command of the *HtmlHelp* API function. Assuming you created a CHM file and defined topic aliases and mapping as described in this article, all you need to do is call the *HtmlHelp* function with the HH\_HELP\_CONTEXT constant, and pass the topic's numeric value as the data parameter of the function.

If you are developing an application using Delphi or C++Builder, and want to use a standard control's *HelpContext* property to supply the topic IDs, Delphi's VCL will try to execute WinHelp to deliver the context sensitive help topics. To fix this problem you need to hook Delphi's application object's *OnHelp* event.

In the code of your application's main form unit, add a function with the following signature:

```
function HelpHook(Command: Word; Data: Longint;
                 var CallHelp: Boolean): Boolean;
```

Then write the code to the function:

```
function TmainForm.HelpHook(Command: Word; Data: Longint;
                             var CallHelp: Boolean): Boolean;
begin
  if (Command in [Help_Context, Help_ContextPopup]) then
    begin
      HH(Application.MainForm.handle,
          'd:\path-to\chmfile.chm', HH_Help_Context, Data);
      CallHelp := False;
    end
  else
    // Trap context calls; ignore all other calls.
    CallHelp := True;
    Result := True;
end;
```

The last step is to set the application's *OnHelp* event to *HelpHook* in the *OnCreate* event of the main form:

```
procedure TMainForm.FormCreate;
begin
  ...
  Application.OnHelp := HelpHook;
  ...
end;
```

You can now set the *HelpContext* property of any control that requires context sensitive help to the topic ID from the CHM file. When the user presses **F1** when the control has focus, context sensitive help will be provided by HTML Help. Note that you must ensure the application's help file does not point to any HLP (or other) file. If this happens, the VCL's default behavior overrides the hook we've been discussing.

## Embedding HTML Help Windows

To embed an HTML Help window in your application, you need to create a new HTML Help window type that



uses the `WS_CHILD` style instead of the default `WS_OVERLAPPED` style.

The easiest way to embed an HTML Help window in your application's window is to create a new HTML Help window definition in code using the `HH_SET_WIN_TYPE` constant of the *HtmlHelp* API function. You will need to create an `HH_WinType` structure and set it to the window properties you want. It will be important to set the *fValidMembers* member of this record to include `HHWIN_PARAM_PROPERTIES`, `HHWIN_PARAM_RECT`, and `HHWIN_PARAM_STYLES`. You can also choose to set other members of the record as valid.

One of the parameters you must define for *fWinProperties* is `HHWIN_PROP_NOTITLEBAR`, and `HHWIN_PROP_NODEF_STYLES`. You will need to trap size changes, update the *rcWindowPos* member of the structure, and update the window definition by calling *HtmlHelp* again with `WW_SET_WIN_TYPE`.

HTML Help windows can notify the parent window of navigation changes and track events. You can use these notification messages to update your application's user interface based on the user's navigation of the help file, thus creating true-to-life tutorial sections that help the user see the application's dialog boxes and controls to learn a complicated procedure.

Creating embedded HTML Help windows is beyond the scope of this article, but if you're interested in a free component that encapsulates the embedded HTML Help creation and usage procedures, point your browser to <http://www.hyperact.com/-DelphiStuff.html> and download one that I wrote.

### Deploying Applications with HTML Help

HTML Help is a standard part of Microsoft IE 4 and future Microsoft operating systems (Windows NT 5.0 and Windows 98). If you know your customers have one of these environments, you don't need to worry about deployment.

HTML Help can also work on NT 4.0x and Windows 95 if the client machine includes a version of IE 3.02 or later. The easiest way to ensure your application will work on the target machine is to ask your customer to install IE 4; it doesn't need to be the default browser, but it's necessary because HTML Help uses it as its layout engine.


If the customer is limited to IE 3.02, you will need to distribute and install the following files:

- Place `hhctrl.ocx`, `itss.dll`, and `itircl.dll` in the Windows \System directory.
- Place `hh.exe` in the \Windows directory.

You'll also need to execute the following commands to register HTML Help with Windows:

- `hh /register`
- `Regsvr32 hhctrl.ocx`

### Conclusion

This should get you started using HTML Help with Delphi. A sample Delphi application that implements HTML Help is available for download (see end of article for details). And should you need it, documentation for the HTML Help API is available online at <http://www.microsoft.com/msdn/sdk/inetsdk/-help/htmlhelp/hhwapi.htm>. 

*The files referenced in this article are available on the Delphi Informant Works CD located in `INFORM\98\MAY\DI9805RL`.*

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





By David Hemphill



## What's This?

### Implementing a "What's This?" Help Toolbar Button

Windows 95 introduced a new form of context-sensitive help known as "What's This?" However, the built-in functionality available from Windows 95 and Delphi cover only two of the methods for invoking "What's This?" help mentioned in *Windows Interface Guidelines*: **F1** and the "?" border icon, as shown in the upper-right corner of **Figure 1**. Using only these two methods assumes all controls in the application can acquire focus (so the user can press **F1** while on the focused control) or reside in a dialog box, which is the only place the "?" border icon is available.

Most Microsoft applications, such as Microsoft Word, provide a "?" toolbar button (see **Figure 2**) that, when clicked, puts the application into a help mode just like the border icon available within dialog boxes. This extends the "What's This?" help mode to the entire application, allowing

users to access help for any control. While this method of invoking help is mentioned in *Windows Interface Guidelines*, developers are on their own if they want this functionality in their applications. This article demonstrates how to add a "What's This?" help toolbar button to a main form, a technique that can be used for new or existing applications simply by implementing the following:

- a message filter to suspend normal message processing and put the application in a help-mode state; and
- a speed button with a "?" icon to invoke the help mode.

Because this help button can be programmed entirely in the main form, you may want to consider creating a form from which all your main forms can descend, thus providing "What's This?" help buttons to all applications. For the purposes of this article, we'll create a new application (start by selecting **File | New Application**).

#### Create the Message Filter

To put an application into help mode, a method must be plugged into *Application.OnMessage*. It's important to note that all messages handled by Delphi applica-

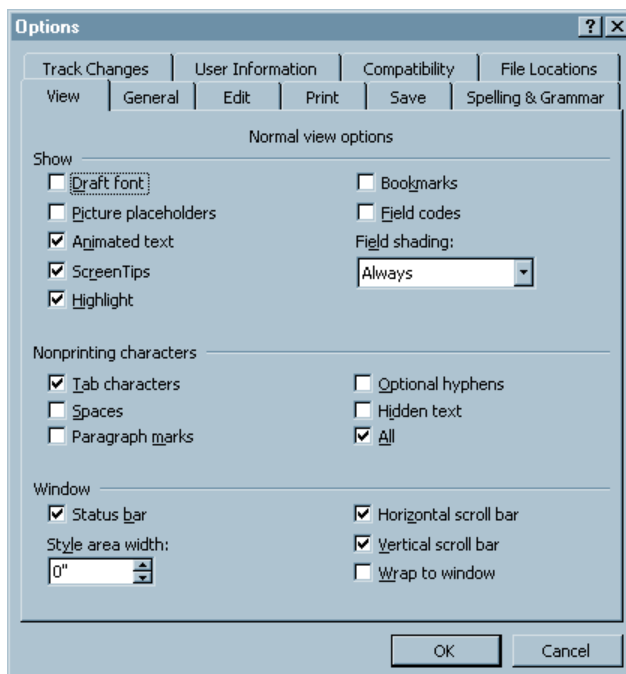


Figure 1: The "?" border icon.

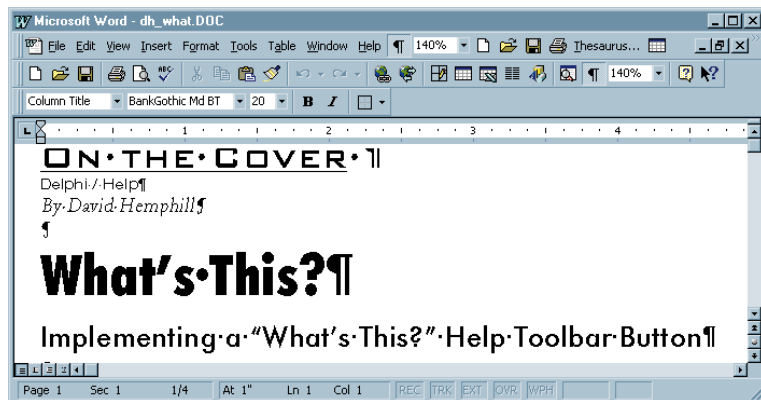


Figure 2: The “What’s This?” toolbar button in Microsoft Word.

tions come through this event. And that’s a lot of messages! Adding even the simplest processing within this event could potentially cause Delphi applications to run noticeably slower. Because we only need to filter messages during a user-invoked help mode, the message filter will be connected only during the time the application is in help mode.

To implement the help-mode message filter, we are interested in the following subset of Windows messages:

- WM\_KEYDOWN
- WM\_LBUTTONDOWN
- WM\_COMMAND
- WM\_APPSYSTEMCOMMAND
- WM\_SYSCOMMAND
- WM\_SETCURSOR

To sustain the help mode, all other messages will be stopped short of their destination. The code for implementing the message filter is shown in [Listing One](#) (beginning on page 13).

The WM\_KEYDOWN message is sent whenever the user presses a key on the keyboard. The message filter is only interested in one key during the help-mode state: `[ESC]`. If the user presses `[ESC]`, help mode is terminated.

While in help mode, if the user presses the primary mouse button (the left mouse button for right-handed mouse users), we must look for a control and its associated *HelpContext* ID that the user just clicked over, assuming the user clicked over a control. The message WM\_LBUTTONDOWN is sent when this occurs. The Delphi *FindControl* function is then used to return a valid reference to the control based on the Windows handle passed in *Msg.wParam*. If *FindControl* returns a non-`nil` result, all we must do is get the help context value. However, if the control the user clicked on has a *HelpContext* value of zero, we must traverse our way up the parent tree until we either find a parent control with a non-zero *HelpContext* value, or we reach the form (where *Parent* = `nil`). If we can’t find a non-zero *HelpContext* value, the message is ignored and help mode is sustained. (Note: Passing a zero value for *HelpContext* is possible. However, because this is the *HelpContext* default value, it’s best to avoid passing zero; this ensures all help links are explicitly defined.)

The WM\_COMMAND message is employed to handle menus. This message is sent when the user selects a menu item capable of invoking a command, i.e. the item doesn’t simply provide access to a submenu, as is the case with the File menu item. For example, if the user clicks on File, we don’t want to display help just yet (because there would be no way for users to access the menu items under File, such as New or Open).

Because WM\_COMMAND is used for a variety of menu functions, there are a few more steps involved in getting the *HelpContext* value. First, we need a reference to the menu item we’re dealing with. *TMenu.FindItem* can be used to obtain this reference using the value *LoWord(Msg.wParam)* passed by Windows. However, at this point, we aren’t even sure within which menu the item resides: We could be dealing with the application’s main menu, a form’s popup menu, or even a control’s popup menu. To handle this situation, we first assume the item is within the main menu. If *FindItem* returns `nil`, we know the item is within a popup menu.

The popup case gets a bit messy for two reasons. First, Windows doesn’t pass a reference to the control associated with the popup. This must be obtained by calling the *FindVCLWindow* procedure and passing the parameter *Msg.Pt*, which is the *x, y* location of the mouse at the time the event occurred. *FindVCLWindow* returns a reference to the control at that location. But we’re still not out of the woods. What if the user clicked over an Edit component that doesn’t have a popup menu specified, but resides within a GroupBox component that does? If this is the case, we must traverse the parent tree until we find our menu. To make things worse, the *PopupMenu* property is protected at the *TWinControl* level, which means we must perform a type check and type-cast to specific controls to obtain access to this property.

If your application doesn’t use popup menus, or your application uses popups only on certain controls (e.g. GroupBoxes or Panels), the code within the *GetPopupMenuItem* function can be modified to check only for those specific types (or eliminated altogether). As with the mouse-down event, if a non-zero *HelpContext* cannot be found for the menu item, this message is ignored and help mode is sustained.

To deal with the system menu (available by clicking on the application icon displayed in the upper-left corner) and border icons, the WM\_APPSYSTEMCOMMAND and WM\_SYSCOMMAND messages must be handled. A set of predefined constants can be compared to the parameter *Msg.wParam* passed by Windows. Because there are no *HelpContext* IDs associated with these menu items, they must be provided directly from within Delphi code.

At this point, almost all the message processing is in place. However, support for the WM\_SETCURSOR and

WM\_SYSCOMMAND messages must be added directly in the main form. To do this, declare two private methods in the form itself using the `message` directive:

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

The implementation of these two methods is shown in [Figure 3](#). The WM\_SETCURSOR message is not processed by Delphi, and thus, isn't dispatched through *TApplication.OnMessage*. However, to keep the mouse cursor set to the “?” icon during help mode (the cursor changes back to the default when clicking or hovering over menus), we need to handle this message when it is sent directly to the form to keep *Screen.Cursor* set properly. The WM\_SYSCOMMAND event must be handled from both the message filter and the main form. This is because Delphi handles the clicks for the border icons before *TApplication.OnMessage* has a chance to respond. Therefore, we need to check if the user clicked over the Minimize, Maximize, or Close border icon, and translate the message parameters so this situation can be handled in the same manner as the system menu items of the same name.

Now it's time to deal with invoking help, should a non-zero *HelpContext* ID be found. This is accomplished by canceling help mode and posting a message to the Windows message queue that will tell our application to invoke help (see the *HlpMessageFilter* method used in Listing One). It's critical that help mode be terminated before calling *PostMessage*. Otherwise, the message filter will intercept the message we just posted and help won't be displayed. Furthermore, the user will still be in help mode. The information passed to *PostMessage* tells Windows to send the CM\_INVOKEHELP message to our application. HELP\_CONTEXTPOPUP is the type of help *TApplication* will invoke (in this case, “What's This?”-style help), as well as the *HelpContext* ID identifying the actual help item in the help file.

## Connecting and Disconnecting the Message Filter

Everything is now in place for the help mode to be invoked; the only thing left to do is program a speed button to invoke the help-mode state. Place a SpeedButton component on the main form (don't worry about the “?” graphic for now), and double-click on the control to modify its *OnClick* event method as shown in [Figure 4](#).

First, set the value of *FHelpMode* to *True*. Then, assign the *HlpMessageFilter* method to *TApplication.OnMessage*. As soon as the method is connected, the message filter will replace all normal *Application* processing with the help-mode processing. Although this seems simple, exercise caution whenever hooking up to *TApplication.OnMessage*. To ensure normal processing will resume if an exception is

```
procedure TMainFM.WMSetCursor(var Msg: TWMSysCommand);
begin
  if FHelpMode then
    begin
      Screen.Cursor := crHelp;
      Msg.Result := 1; // Message handled.
    end
  else
    inherited;
end;

procedure TMainFM.WMSysCommand(var Msg: TWMSysCommand);
begin
  if FHelpMode and
    ((Msg.CmdType and $FFF0 = SC_MINIMIZE) or
     (Msg.CmdType and $FFF0 = SC_MAXIMIZE) or
     (Msg.CmdType and $FFF0 = SC_CLOSE)) then
    PostMessage(Application.Handle, WM_SYSCOMMAND,
                $FFF0 and Msg.CmdType, 0)
  else
    inherited;
end;
```

Figure 3: Cursor and border icon message handling.

```
procedure TMainFM.HelpBtnClick(Sender: TObject);
var
  OnMessageBackup: TMessageEvent;
  FOrigCursor: TCursor;
begin
  // Save the current cursor value.
  FOrigCursor := Screen.Cursor;

  // Suspend current message handling, if applicable.
  OnMessageBackup := Application.OnMessage;
  Application.OnMessage := HlpMessageFilter;

  try
    FHelpMode := True;
    Screen.Cursor := crHelp;
    // Localize message processing to ensure
    // turning off FHelpMode.
    while FHelpMode do
      Application.ProcessMessages;
  finally
    // Ensure help mode is turned off.
    FHelpMode := False;
    // Resume previous message handling, if applicable.
    Application.OnMessage := OnMessageBackup;
    Screen.Cursor := FOrigCursor; // Restore the cursor.
  end;
end;
```

Figure 4: An *OnClick* event to invoke help mode.

raised while the application is in the help-mode state, we can localize the message processing within the *OnClick* event. To do this, a `while` loop is implemented that does nothing more than execute *Application.ProcessMessages* while the application is in the help-mode state. This allows the message processing to execute within the context of a `try..finally` block. The `finally` clause ensures the help-mode state will be cleaned up whether help mode terminates normally, or not.

## Run the Application

Before testing the help-mode button, a few controls and a help file will be needed to know if the help filter is successful in displaying “What's This?” help. For testing purposes,



Wordpad.hlp (located in the \Windows directory by default) can be specified as the help file on the Application tab of the Project Options dialog box (accessed by selecting **Project | Options**). The help text displayed will be meaningless within the context of our application, but this will allow us to test the message filter without having to build a Windows Help file.

Next, place a GroupBox component containing two Edit components on the form. Give the GroupBox and one of the Edit components a *HelpContext* ID, but leave the other Edit component's *HelpContext* set to zero. Now, run the application and click on the help speed button to invoke the message filter. The mouse cursor should change to the "?" icon. Click on one of the edits to invoke "What's This?" help. Note that the Edit component with the zero *HelpContext* value displays help attached to the GroupBox, but the Edit component containing a *HelpContext* value displays its own help.

## The "?" Glyph

To polish the look of the "What's This?" help speed button, you'll want to set the *Glyph* property of the button to the "?" graphic. This graphic isn't readily available from Windows or Delphi; it has been included with the source code accompanying this article for your convenience. Otherwise, this becomes a do-it-yourself project.

## Conclusion

Providing a "What's This?" toolbar button on the main form of your applications allows users to access context-sensitive help on all controls, not just controls that can acquire focus. In this example, the message filter calls *PostMessage* specifying the style of help to be `HELP_CONTEXTPOPUP` (which displays the help in the "What's This?" format). For some applications, however, it may make sense to display more information in a full-blown Help window. This can easily be accomplished by passing `HELP_CONTEXT` instead of `HELP_CONTEXTPOPUP` in the *PostMessage* call. One of the benefits of "What's This?"-style help, however, is that the entire help application doesn't need to be loaded into memory to display the help text — whereas using `HELP_CONTEXT`, the help application is invoked.  $\Delta$

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM98\MAY\DI9805DH.*

David Hemphill is a consultant for Compuware Corporation in Minneapolis, MN, specializing in Delphi and object-oriented design and development. David can be contacted by e-mail at [hemptava@pdlinc.com](mailto:hemptava@pdlinc.com).

## Begin Listing One — Message Filter

```

procedure TMainFM.CancelHelpMode;
begin
    FHelpMode := False;
    FHandled := True;
end;

function TMainFM.HandleClick(Msg: TMsg): THelpContext;
var
    Control: TWinControl;
begin
    Result := 0;

    // Find the control at position user clicked.
    Control := FindControl(Msg.Hwnd);
    while (Result = 0) and (Control <> nil) do begin
        // Get Control's HelpContext.
        Result := Control.HelpContext;
        // If HelpContext = 0, recursively traverse the child/-
        // parent tree until a HelpContext ID or Form is found.
        if Result = 0 then
            Control := Control.Parent;
    end;
    FHandled := True;
end;

function TMainFM.HandleMenuItem(Msg: TMsg): THelpContext;
var
    AMenu: TMenuItem;
    MenuItem: TMenuItem;
    ID, MenuFlag: Integer;

    function GetPopupMenuItem: TMenuItem;
    var
        AControl: TWinControl;
    begin
        AMenu := nil;
        AControl := FindVCLWindow(Msg.Pt);

        repeat
            if AControl is TForm then // TForm
                AMenu := TForm(AControl).PopupMenu
            else if AControl is TGroupBox then // TGroupBox
                AMenu := TGroupBox(AControl).PopupMenu
            else if AControl is TPanel then // TPanel
                AMenu := TPanel(AControl).PopupMenu
            else if AControl is TRadioGroup then // TRadioGroup
                AMenu := TRadioGroup(AControl).PopupMenu
            else if AControl is TDrawGrid then // Grids
                AMenu := TDrawGrid(AControl).PopupMenu;
            if AMenu <> nil then
                // Does parent have the popup menu?
                AControl := AControl.Parent;
            until (AMenu <> nil) or (AControl = nil);

            result := AMenu.FindItem(ID, fkCommand);
        end;

    begin
        Result := 0;
        MenuItem := nil;
        ID := LoWord(Msg.wParam);
        MenuFlag := LoWord(Msg.hWnd);

        if (MenuFlag <> $FFFF) or (ID <> 0) then
            begin
                // Look for item in form's main menu.
                AMenu := Screen.ActiveForm.Menu;
                if AMenu <> nil then
                    MenuItem := AMenu.FindItem(ID, fkCommand);
                // Try to find item in popup menu.
                if MenuItem = nil then
                    MenuItem := GetPopupMenuItem;
                // Get menu item's HelpContext.
                if MenuItem <> nil then

```

```

        Result := MenuItem.Helpcontext;
        FHandled := True;
    end;
end;

function TMainFM.HandleSystemMenu(Msg: TMsg): THelpContext;
const
    // Assign values to use for system menu and border icon
    // HelpContext IDs. These can be any number. Negative or
    // positive numbers are valid in WinHelp files.
    cSysMenuRestore = -100;
    cSysMenuMove    = -110;
    cSysMenuSize    = -120;
    cSysMenuMinimize = -130;
    cSysMenuMaximize = -140;
    cSysMenuClose   = -150;
begin
    Result := 0;

    case Msg.wParam of
        SC_RESTORE: Result := cSysMenuRestore;
        SC_MOVE:   Result := cSysMenuMove;
        SC_SIZE:   Result := cSysMenuSize;
        SC_MINIMIZE: Result := cSysMenuMinimize;
        SC_MAXIMIZE: Result := cSysMenuMaximize;
        SC_CLOSE:  Result := cSysMenuClose;
    else
        CancelHelpMode;
    end;

    // Keep menu command from executing if
    // HelpContext ID is found.
    FHandled := (Result <> 0);
end;

procedure TMainFM.HlpMessageFilter(var Msg: TMsg;
                                   var Handled: Boolean);
var
    HContext : integer;
begin
    FHandled := Handled;
    HContext := 0;

    case Msg.Message of
        WM_KEYDOWN:
            if (Msg.wParam = VK_ESCAPE) then
                // User pressed <esc> key.
                CancelHelpMode;
        WM_LBUTTONDOWN:
            // User clicked while in help mode.
            HContext := HandleClick(Msg);
        WM_COMMAND:
            // User clicked on a menu item while in help mode.
            HContext := HandleMenuItem(Msg);
        CM_APPSYSCOMMAND, WM_SYSCOMMAND:
            // Border icon or System Menu item.
            HContext := HandleSystemMenu(Msg);
    end;

    Handled := FHandled; // Message handled above?
    if HContext <> 0 then // HelpContext ID found?
    begin
        CancelHelpMode;
        // Tell Delphi to invoke help.
        PostMessage(Application.Handle, CM_INVOKEHELP,
                    HELP_CONTEXTPOPUP, HContext);
    end;
end;
end;

```

## End Listing One





## Setting Limits: Part II

### Hijacking a Form's Message Loop to Enforce Form Size Limitations

This is the second installment of a two-part series on controlling the minimum and maximum sizes of your application's forms using the Windows `WM_GETMINMAXINFO` message. The [first article](#) set the stage and discussed two ways to catch the message:

- Writing the code in each form that requires it.
- Using form inheritance in Delphi 1 and visual form inheritance in Delphi 2 and 3.

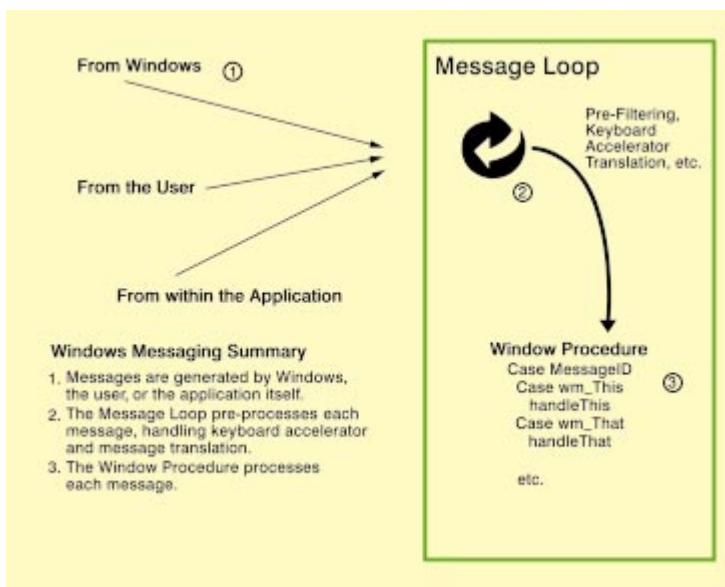
We also looked at the problems with these approaches, and wondered if there was a better way of doing things. To that end, this article discusses how to build a component that adds size control to the form that owns it. This component, and others like it, lets us build flexible forms without the liabilities imposed by static inheritance.

#### ***WndProc* and All That**

As you probably know, Microsoft Windows works by sending messages back and forth

between all the processes running on your computer. Some of these messages are created by your activities (e.g. moving the mouse or typing on the keyboard); many more are created by Windows itself as it goes about its work of managing memory and user input, and handling networking and the file system. Each window and process (actually each thread, but we don't need to get that complicated here) has its own message queue where the messages targeted for that process line up and wait to be seen.

The messages in the message queue are processed one-by-one by a message loop — a chunk of code that does some preliminary filtering — then sends the message on to the window procedure (*WndProc* for short) of the window for which the message is intended (see [Figure 1](#)). In essence, a window procedure is a glorified `case` statement; it takes a particular message ID and some parameters, and does something in response. Every window in Windows has an associated window procedure that gives that window its particular behavior. Because many windows need to handle the same messages in the same way, the Windows operating system provides two mechanisms to share *WndProcs*: window classes and window subclasses. Note that these are *not* the same as object-oriented classes and object-oriented subclasses. The concepts are similar, but different. It's unfor-



**Figure 1:** The message loop filters, then sends messages to *WndProc*.

tunate (and confusing) that Windows overloads these words, but there isn't much that can be done about it now.

A window class, sometimes called a *WndClass* because of the structure associated with it, is a way of associating each "physical" window in the system with a particular category, e.g. an edit box, a button, etc. The class of a window is specified when it is created via the *CreateWindow* or *CreateWindowEx* API call. Delphi takes care of all the Windows-level class registration and window creation automatically, so we won't cover the details here. Note that each window class implies a standard set of attributes: the class style, the default icon, the default background color, etc. The window class also specifies the window procedure that all the windows in this class use in common. This class mechanism makes it easy for many of the standard windows to share their window procedures, but in and of itself, does not allow windows that are almost like an edit box or button. To handle these windows and prevent the proliferation of window classes, Windows provides a window subclassing mechanism.

Window subclassing provides a way to link window procedures in a chain so *WndProcs* higher up the chain can call the ones lower down the chain. This means we can base a new window on a standard *WndClass*, then write a window procedure that handles only the new functionality while building on the standard behavior of the window procedure of the original *WndClass*. In practice, the first window procedure in the chain is given the message to process. It can process it completely or partially, or it can ignore it (see [Figure 2](#)). If processed completely, there will be no more processing for that message. In the two other cases, the message is passed on to the next window procedure in the chain. Eventually, the message is either consumed, or is sent to *DefWndProc*, the standard Windows window procedure.

### Windows Messaging and Delphi

As we saw [last month](#), most of the complexity of windows messaging is hidden very nicely by Delphi. In relation to the previous discussion, Delphi automatically handles all the details of

associating each form (window) in your application with the correct class. Delphi also makes it easy to customize message handling for particular messages by using the standard event-handling system, or via the message keyword when the regular event handlers are not flexible or specific enough. When this isn't enough, Delphi also exposes everything we need to do real window subclassing and to customize the message handling.

### Hijacking a Form's Message Loop

Our goal is to have a form that responds to the *WM\_GETMINMAXINFO* message in a custom fashion. Last month's article showed how to do this by altering the object-oriented class of the form (directly or by Delphi object subclassing). The discussion of window subclassing in the previous section shows there is another way; if we can intercept messages intended for the form and pre-process them, we can alter the behavior of the form (with respect to *WM\_GETMINMAXINFO* or any other message) without altering the form itself. Thus, our plan is to create a component that can take over part of its form's window procedure. This component will intercept the messages intended for its form and pre-process them before passing them on to the form for the regular processing. Let's see what we need to do.

### Changing a Form's Window Procedure

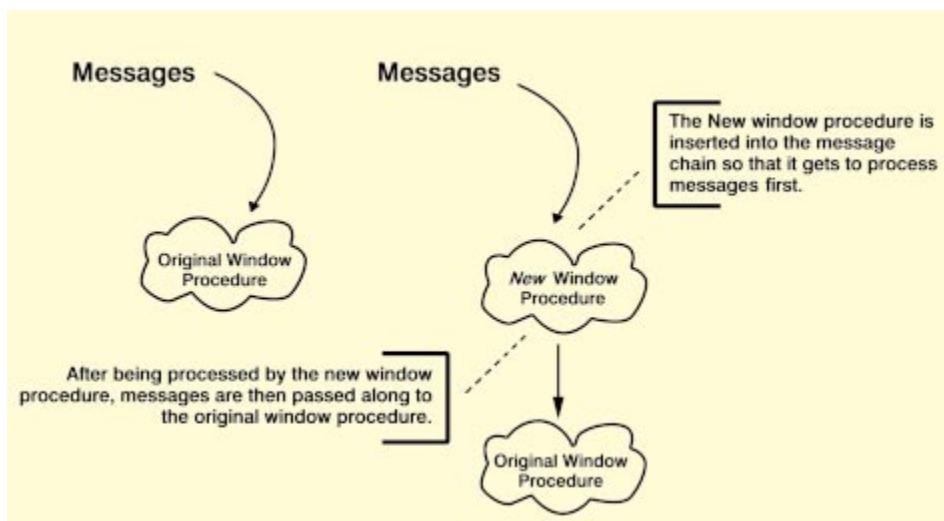
Generically speaking, a window procedure takes the form:

```
function WindowProc(Wnd: HWND; Msg, wParam: Word;
                    lParam: Longint): Longint;
```

or, in 32-bit land:

```
function WindowProc(Wnd: HWND; Msg, wParam: Longint;
                    lParam: Longint): Longint;
```

where *Wnd* is the handle of the window currently using this window procedure, *Msg* is the identifier for the Windows message being processed, and *wParam* and *lParam* contain information whose meaning varies for each message.



**Figure 2:** The first window procedure in a chain can process a message completely or partially, or it can ignore it altogether.

A window procedure is just a function, and a function is just an address in memory that your application (and the operating system) treats in a special way. We can change a window's window procedure and alter its behavior by manipulating this address. Every window in Windows stores the address of its window procedure as a pointer, which can be accessed and changed via the *GetWindowLong* and *SetWindowLong* Windows API calls. As their names imply, *GetWindowLong* retrieves one of the pieces of window information and *SetWindowLong* sets



```

type
  { Defined by Delphi in the WinTypes or Windows unit. }
  TFarProc = Pointer;

var
  FOldWndProc: TFarProc;
  FNewWndProc: TFarProc;
  ...
  FNewWndProc := // A windows procedure that is
                 // one of my methods.

  FOldWndProc := TFarProc(GetWindowLong(AForm.Handle,
                                       gw1_WndProc));
  SetWindowLong(AForm.Handle, gw1_WndProc,
               Longint(FNewWndProc));

```

**Figure 3:** Changing the window procedure of a form.

one of the pieces of window information. In both API calls, the value being manipulated is a 32-bit long integer.

*GetWindowLong* takes a window handle and a code that specifies exactly what piece of information to return. *SetWindowLong* takes the handle, the code, and the new value to which this information should be set. To make Delphi and Windows happy, everything passed in and out of these API calls must be a *Longint*, or it must be typecast into a *Longint*. Let's see how these two calls can allow us to change a window's window procedure.

If we assume that *FNewWndProc* is a pointer to a method that has the correct signature, we can change the window procedure of a form, as shown in [Figure 3](#). Because *SetWindowLong* returns the original value of the information being set, we can code it more succinctly as:

```

FOldWndProc :=
  TFarProc(SetWindowLong(AForm.Handle, gw1_WndProc,
                        Longint(FNewWndProc)));

```

where *gw1\_WndProc* is one of the many constants defined by Windows that makes the code somewhat easier to read. In this case, the "gw1" prefix specifies that this constant is used in the *GetWindowLong* API call. Note that we need to typecast the result of *GetWindowLong* and the input parameter of *SetWindowLong* to keep the Delphi compiler and Windows happy with the pointers we're passing about.

Mucking about with function pointers and addresses is a little confusing (and a good way to generate a GPF when you make a mistake), but the basic idea is simple:

- Functions are merely addresses in memory.
- Variables can point to functions (in which case we call them function pointers).
- We can use function pointers to call the functions to which they point.
- Changing the value of a function pointer (making it point to a new function) changes the function we call.

Using a function pointer is just like consulting your directions when you are driving; you read the directions to see

where to go. If you are given different directions (change the function pointer), you'll go somewhere else.

In our example, once the call to *SetWindowLong* returns, every message that would have gone to the regular window procedure for *AForm* will come to whatever *FNewWndProc* is pointing to. This procedure can then do anything it likes with the incoming messages; it can respond to them itself, alter them and pass them along, or leave them unchanged. After it does its work, our new window procedure should use the *CallWindowProc* API function to pass the message on to the original window procedure. We'll see this in detail when we create our component.

One problem remains with the previous code; it will work wonderfully *if* we have a window procedure to pass to *SetWindowLong*. However, since we're writing a component, we need to tell Windows to call a particular method in a particular object. Sadly, all Windows knows how to do is call a global method not attached to an object or class. Fortunately, Delphi comes to the rescue with the *MakeObjectInstance* function and its partner, the *FreeObjectInstance* function. Though nearly undocumented, this function takes a Delphi method pointer and uses some low-level assembly magic to return a pointer to a window procedure function we can use. This window procedure that Delphi creates will call the method we specified with the method pointer. The only important difference here is that the Delphi window procedure looks like this:

```

procedure TSomeClass.AWindowProcedure(var msg: TMessage);

```

where *TMessage* is defined as:

```

type { In Messages.pas (32-bit version) }
  TMessage = record
    Msg: Cardinal;
    Wparam: Longint;
    LParam: Longint;
    Result: Longint;

```

The actual definition is slightly more complex, but this is the important part. As you can see, the separate parameters in the original procedure are mapped into a single record, and the window handle disappears altogether. The assumption is that the class maintaining this window procedure will also maintain the identity of the window handle.

*MakeObjectInstance* and *FreeObjectInstance* should be used in pairs because *FreeObjectInstance* releases any resources that *MakeObjectInstance* allocates. These functions are defined in *Forms.pas* and are exactly what Delphi uses to connect instances of the VCL classes (like *TForm*) with their Windows counterparts. Though undocumented, the code will show that using them isn't difficult.

## Putting It Together

To make all this clear, we'll build our new *TDSMinMax* component (see [Figure 4](#)). Much of this code is similar to the code

```
TDSMinMax = class(TComponent)
  Private
    FOldWndProcForm : TFarProc;
    FOwnerOnShow : TNotifyEvent;
    FParentForm : TForm;
    FWindowProcedureForm : TFarProc;

    procedure HookForm;
    procedure UnhookForm;
    procedure WndProcForm(var msg: TMessage);
```

**Figure 4:** Building the new *TDSMinMax* component, a component that gives its form complete control over its size by using window subclassing.

```
procedure TDSMinMax.HookForm;
begin
  { Hook the window procedure of my owner only if I have
  an owner and its window handle has been created. }
  if assigned(FParentForm) and
    FParentForm.HandleAllocated then
    begin
      { Create the window procedure from one of my methods;
      be sure to pair this with a call to
      FreeObjectInstance. }
      FWindowProcedureForm :=
        MakeObjectInstance(WndProcForm);
      { Subclass my form's window by inserting my window
      procedure into the message chain. }
      FOldWndProcForm := TFarProc(SetWindowLong(
        FParentForm.Handle, gw1_WndProc,
        Longint(FWindowProcedureForm)));
    end;
end;
```

**Figure 5:** *HookForm* assigns the new window procedure (*WndProcForm*) to the form.

in last month's article, and deals with correctly handling the *WM\_GETMINMAXINFO* message. The new code involves subclassing the component's owner (the form it lives on). First, we need several new attributes to store various window procedures and other important information. The snippets shown in this article only include the code required for the subclassing; the rest of the code deals with tracking the values needed to provide size control, and are much like the code from last month's article.

Note: *MaxHeight*, *MaxWidth*, *MaxLeft*, and *MaxTop* are values used to determine the size of a form when you maximize it. The *ResizeMaxHeight* and *ResizeMaxWidth* properties are used to define the maximum allowable height a form can achieve when being resized. The *ResizeMinHeight* and *ResizeMinWidth* properties are conversely used to define the minimum allowable height and width the form can achieve when being resized.

As you can see, we store the form's original window procedure, and a pointer to the one we'll create, with *MakeObjectInstance*. We also store a pointer to the form's *OnShow* event (for reasons that will become evident) and an object reference to our parent form (mostly to make the code a little easier to read).

The actual job of starting and stopping the interception of the form's messages is taken care of in the *HookForm* and

```
procedure TDSMinMax.WndProcForm(var msg: TMessage);
begin
  if msg.Msg = wm_GetMinMaxInfo then
    { We pay special attention to the WM_GETMINMAXINFO
    message. The WM_GETMINMAXINFO message is sent to a
    window when Windows needs the maximized position or
    dimensions of the window or needs the maximum or
    minimum tracking size of the window. The maximized
    size of a window is the size of the window when its
    borders are fully extended. The maximum tracking size
    of a window is the largest window size that can be
    achieved by using the borders to size the window. The
    minimum tracking size is the smallest window size
    that can be achieved by using the borders to size
    the window. }
    with PMinMaxInfo(msg.lParam)^ do begin
      if FMaxSizeAssigned then
        ptMaxSize := FMaxSize;
      if FMaxPositionAssigned then
        ptMaxPosition := FMaxPosition;
      if FMinTrackSizeAssigned then
        ptMinTrackSize := FMinTrackSize;
      if FMaxTrackSizeAssigned then
        ptMaxTrackSize := FMaxTrackSize;
    end;

    { Call the original window procedure. }
    msg.Result :=
      CallWindowProc(FOldWndProcForm, ParentForm.Handle,
        msg.Msg, msg.wParam, msg.lParam);
  end;
```

**Figure 6:** The *WndProcForm* method.

*UnhookForm* member functions. *WndProcForm* is the method that will become the new window procedure. **Figure 5** shows how we assign our new window procedure to the form.

If we have a parent form to subclass, and this parent form has created a handle for its physical window (see the next section for a discussion on the *HandleAllocated* function and form window handles in general), then we create a window procedure out of one of our member functions (*WndProcForm*) and hand it to our parent form to use as its new window procedure. This means that all the messages intended for the form will first come to us by calling the *WndProcForm* method, as shown in **Figure 6**.

As you can see, the new window procedure passes every message but one to the original window procedure unchanged. However, any *WM\_GETMINMAXINFO* messages will be massaged with the other information stored and managed by the component (details of this massaging can be found in last month's article). All of the messages are passed to the original procedure using the *CallWindowProc* API call. Notice how we need to split the various pieces of the *TMessage* record apart to call *CallWindowProc*.

This message massaging will continue until *UnhookForm* is called to break the connection, as shown in **Figure 7**. *UnhookForm* simply undoes the actions that *HookForm* started; it resets the window procedure of the parent form and calls *FreeObjectInstance* to restore the memory claimed by *MakeObjectInstance*.

## Getting It Started

So far, we've seen how to grab a form's messages (*HookForm*), how to process them once we've grabbed them

```

procedure TDSMinMax.UnhookForm;
begin
  { Undo what HookForm did; reset the window procedure
    and FreeObjectInstance. }
  if not (FOldWndProcForm = nil) then
    begin
      if (FParentForm <> nil) and
        (FParentForm.HandleAllocated) then
        SetWindowLong(FParentForm.Handle, gw1_WndProc,
          Longint(FOldWndProcForm))
      FOldWndProcForm := nil;
      FParentForm := nil;
    end;

  if not (FWindowProcedureForm = nil) then
    begin
      FreeObjectInstance(FWindowProcedureForm);
      FWindowProcedureForm := nil;
    end;
end;

```

Figure 7: The *UnhookForm* procedure.

(*WndProcForm*), and how to stop grabbing them (*UnhookForm*). The only missing piece in our component's puzzle is the timing. When do we start and stop subclassing? The important point here is that we can't subclass a form unless it actually has a window handle on which to hang our hat. Delphi carefully separates the creation of a form object and the creation of the window associated with that object. This makes good sense because creating a *TForm* instance requires a small amount of memory (and little else), but creating a window requires allocating a window handle and making a series of potentially time- and resource-expensive API calls. Delphi hides this distinction and its complexity by hiding the "physical" window handle behind the form's handle property. The form will create the handle the first time this property is referenced, which means that you don't want to use the handle property unless you must. To give the developer greater control over this process, Delphi also provides the *HandleAllocated* method, to tell you if the handle has been created, and the *HandleNeeded* method, to tell Delphi that you want the handle now.

When Delphi creates a subclass of *TForm* that contains a *TDSMinMax* component, it will eventually call the *TDSMinMax* constructor. The previous handle discussion means it's very unlikely that the form's window will have been created at this time. Furthermore, we don't want our constructor to force the creation of the form's window (which would happen, for example, if we referenced the form's handle property). Because we can't guarantee the existence of the window handle, we can't call *HookForm* in our component's constructor. To get around this problem, we cheat a bit and play a game with our owner's *OnShow* event. The component's constructor is shown in Figure 8.

First, we make sure our application is running by testing the *ComponentState*. If we aren't designing, we save a reference to our owner (with a cast to *TForm*) to make the code easier to follow. Then, we give it a new *OnShow* event (after being sure to save the original). Everything proceeds normally until the owner form is actually shown.

```

constructor TDSMinMax.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);

  if not (csDesigning in ComponentState) then
    begin
      FParentForm := (Owner as TForm);
      { Catch form show. }
      FOwnerOnShow := FParentForm.OnShow;
      FParentForm.OnShow := OwnerShow;
    end;
end;

```

Figure 8: The *TDSMinMax* constructor.

```

procedure TDSMinMax.OwnerShow(Sender: TObject);
begin
  { Create our hook. }
  HookForm;
  { Call the original OnShow event (if any). }
  if assigned(FOwnerOnShow) then
    FOwnerOnShow(Sender);
  { Restore things to normal. }
  (Owner as TForm).OnShow := FOwnerOnShow;
  FOwnerOnShow := nil;

  UpdateParentSize;
end;

```

Figure 9: The *OwnerShow* method.

When this happens, the *OnShow* event will fire, and our *OwnerShow* method will execute, as shown in Figure 9. In this event handler, we first hook the form. Then, we call the original *OnShow* event (if there was one) and restore the *OnShow* event to normal. Lastly, we call *UpdateParentSize* to force the generation of an initial *WM\_GETMINMAXINFO*.

Generally speaking, it's not a great idea to play games like this with another component's events. Swapping events in and out can confuse other clients of the component and lead to code that's hard to follow and failures that are difficult to fix. In this case, the chance of error is slight and the benefits are manifest; it's not common to change a form's event assignments between its creation and the time it's shown.

## Good News and Bad News

The good news is that the *TDSMinMax* component is complete. Dropping this component on a form adds size control without altering the form's class directly or through inheritance. We have created a component that enhances a form with a new feature that's transparent to the form and to any other clients of the form. This is what object-orientation is supposed to be — objects interacting dynamically in collaborations that produce wholes that can be greater than the sum of their parts. Components such as *TDSMinMax* add features and tools for the developer that can be wired together flexibly without code. They also allow us to build complex assemblies without creating dozens of classes.

However, there are at least a few flies in the ointment. Our component does its magic by carefully taking control of

another component's Windows message stream. To be fully robust, there are a few more wrinkles we need to consider. The first is that ours might not be the only component trying to alter the form's behavior in this manner. If multiple objects are subclassing the form, we need to ensure things are put back together properly as objects hook and unhook themselves. As presented, *TDSMinMax* ignores this complexity.

Furthermore, it's quite possible in Delphi for a form's physical window to be destroyed and re-created. This happens, for example, whenever the *BorderStyle* or *BorderIcon* properties are changed; both of these properties affect the form's window style and necessitate the creation of an entirely new window. Delphi handles the destruction and recreation so transparently and seamlessly that it seems as if nothing has happened. When the window is destroyed and re-created, the window handle and window procedure that *TDSMinMax* owns becomes null and void, and *TDSMinMax* is left out in the cold.

It's possible to make components such as *TDSMinMax* that correctly notice and respond to the multiple subclassing and the dynamic window recreation that Delphi sometimes performs. Doing so requires creating some standards and playing some tricks, so we'll leave it all for a future article.

### **The Thrilling Conclusion**

Hopefully, I've convinced you that it's both possible and desirable to create components that build new features into a system without requiring changes to the system's class structure — components that play with aggregation and association rather than inheritance. We've implemented these ideas in Delphi via the magic of window subclassing. In the process, we've created a component that gives any form complete control over its maximum and minimum sizes. Components like this make it easy for developers to add useful enhancements to their applications, making building better and more useable applications easier and more fun. ▲

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM98\MAY\DI9805GK*

Gary King is the principal of DesignSystems (<http://www.dsgnsystems.com>), makers of DSAppLock, Remember This, and other fine products for Delphi. He can be reached at [gwking@dsgnsystems.com](mailto:gwking@dsgnsystems.com).







## Delphi Import/Export

### Part I: Getting Data into, and out of, ASCII Text Files

**D**atabase applications frequently need to import and export data to and from the database, for use in other applications. This two-part series explores the tools that Delphi and the Borland Database Engine (BDE) provide to get data in and out of your database.

#### Using the BDE ASCII Driver

If you need to import data from a delimited or fixed-length ASCII text file, you may be able to use the ASCII driver built into the BDE. You can also use the ASCII driver to export to a fixed-length or delimited ASCII text file. Note also that the ASCII driver only reads and writes text files; that is, files where each record ends with a carriage return character, followed by a line feed character. Reading and writing fixed-length, undelimited files used by some mainframe and minicomputer applications are discussed later in this article.

#### Importing ASCII Data

Delphi's implementation of the BDE ASCII driver lets you treat an ASCII text file as a table in some respects, and access the file using a Table component. However, before you can use an ASCII file, you must create a schema file that tells the BDE about the format of the ASCII file. The BDE expects your ASCII file to have a .TXT file extension. The schema file must have the same name as the ASCII file with an extension of .SCH. The schema file is also an ASCII text file with a

```
[CNS]
FileType=Fixed
CharSet=ascii
Field1=CNS_ID,Longint,3,0,0
Field2=Lab_Id,Char,10,0,5
Field3=Sample_Number,Char,15,0,18
Field4=Material,Char,15,0,36
Field5=Method,Char,8,0,54
Field6=Weight,Float,6,4,64
```

**Figure 1:** An example of a schema file.

format very similar to a Windows .INI file. None of the entries in the schema file are case sensitive. [Figure 1](#) shows an actual schema file from an application.

The first line contains the name of the file enclosed in brackets. The second line contains the FileType, which must be Fixed for a fixed-length file or Varying for a delimited file. The third line contains the CharSet parameter, which specifies the language driver you're using. If you are importing a delimited file, you'll need two more entries in your schema file, as shown here:

```
Delimiter = "
Separator = ,
```

The Delimiter specifies the character used to enclose alphanumeric fields within the file, while the Separator identifies the character used to separate fields within each record of the delimited file. Following these global parameters is an entry for each field in a record. The fields are identified as Field1, Field2, Field3, etc. You must provide five values for each field, separated by commas. These values are:

- 1) Field Name: The field name is limited to 25 characters and must follow the Paradox field naming conventions. It cannot start with a space, but it can contain spaces. The characters { } ( ) , . [ ] ! | > - are invalid. The easiest way to stay out of trouble is to restrict field names to letters, numbers, and the underscore character.

```
[CustFix]
FileType=Fixed
CharSet=ascii
Field1=Customer_No,Float,20,0,00
Field2=Name,Char,30,00,20
Field3=Phone,Char,15,00,50
Field4=First Contact,Date,08,00,65
```

**Figure 2:** A schema file for the fixed-length import, from the ASCII DRV project.

- 2) Field Type: The type of data contained in this field. The following types are allowed:
  - Char*: alphanumeric characters
  - Float*: floating point number
  - Number*: 16-bit integer
  - Longint*: 32-bit integer
  - Bool*: Boolean (T or F)
  - Date*: a date formatted according to the date setting in the BDE configuration file
  - Time*: a time formatted according to the time setting in the BDE configuration file
  - Timestamp*: a date and time formatted according to the BDE configuration file settings
- 3) Number of Characters: This must be less than or equal to 20 for numeric data types, and must be the maximum number of characters for *Date*, *Time*, and *Timestamp* data types.
- 4) Number of Digits After the Decimal: While this is supposed to specify the number of digits after the decimal point for floating point numbers, it doesn't appear to work. No matter what this value is set to, the number of digits to the right of the decimal is determined by the location of the decimal point in the text file. If there is no decimal point in the text file, the number is imported as an integer, i.e. with no digits to the right of the decimal point.
- 5) Offset: The number of characters from the beginning of the record where this field begins. This value applies to fixed-length records only, and should be set to zero for delimited files.

To import an ASCII file, add two Table components and one BatchMove component to a form in your project. Set the following properties for one of the Table components:

- *DatabaseName* = the location of the ASCII text file you are going to import
- *Exclusive* = True
- *TableType* = *ttASCII*
- *TableName* = the name of the ASCII text file

Set the *DatabaseName* and *TableName* properties of the second Table component to connect it to the table into which you will import the ASCII file. This table doesn't have to exist. Finally, set the following properties of the BatchMove component:

- *Destination* = the Table component connected to the database table that will receive the imported data
- *Mode* = *batCopy* to create a new table, or *batAppend* to append to an existing table

```
[CustVar]
FileType=Varying
Delimiter="
Separator=,
CharSet=ascii
Field1=Customer No,Float,20,04,00
Field2=Name,Char,30,00,20
Field3=Phone,Char,15,00,145
Field4=First Contact,Date,11,00,160
```

**Figure 3:** A schema file for the delimited import, from the ASCII DRV project.

- *Source* = the Table component connected to the ASCII file

When you call the BatchMove component's *Execute* method, the ASCII file will be imported into the destination table, using the schema file you've created for it. The ASCII DRV project that accompanies this article demonstrates importing both fixed-length and delimited files (see end of article for download details). **Figure 2** shows the schema file for the fixed-length import; **Figure 3** shows the schema file for the delimited import.

You can export a database table to a fixed-length ASCII file in exactly the same way. The only difference is that the *Source* property of the BatchMove component will be the database table, and the *Destination* property will be the Table component that points to the ASCII file. When you export to an ASCII file, the BDE ASCII driver creates a schema file for the exported ASCII file automatically. You can use this schema file to import the file again using Delphi, or to provide documentation to the recipient of the file that shows the record layout.

**Editing an ASCII File**

With some restrictions, you can treat a fixed-length ASCII file as a table and edit the data in the file. The restrictions that apply are:

- ASCII files do not support indexes or any functions that use indexes.
- You must open the ASCII file for exclusive use.
- You cannot use a Query component to access an ASCII file.
- You cannot delete records.
- You cannot insert records in the middle of the file. Inserting a record appends the record to the end of the file.
- Although you cannot view a subset of records in an ASCII file using indices or queries, you can use *TTable's Filter* property to apply a filter to the file, just as you would to a database table. This provides a convenient way to import or view only those records that meet the filter criteria you impose.

The ASCEDIT project that accompanies this article demonstrates editing a fixed-length ASCII file. It uses Table, Datasource, DBGrid, and DBNavigator components, configured as if to edit a database table, with the exception of the Table component's *TableType* and *Exclusive* properties, as previously described.

**Using Delphi's Text-File I/O Routines**

Delphi's System unit contains the Pascal text-file input/output

```

procedure TForm1.OpenBtnClick(Sender: TObject);
var
    txtFile: System.Text;
    txtLine: string;
begin
    { Open the file. }
    AssignFile(txtFile, FileListBox1.FileName);
    Reset(txtFile);
    { Empty the memo and read the file. }
    Memo1.Lines.Clear;
    while not EOF(txtFile) do begin
        Readln(txtFile, txtLine);
        Memo1.Lines.Add(txtLine);
    end;
end;

```

**Figure 4:** Using the *AssignFile* procedure to open the file, then calling *Reset* from *TextFile*.

routines. The first step in using Delphi's text-file routines is to declare a file variable whose type is *Text*, as shown here:

```

var
    txtFile: System.Text;
    txtLine: string;

```

Line two declares the text-file variable, and line three declares a **string** variable that will hold each line as it's read from the file. Note that the file variable's type is declared as *System.Text*. The name of the System unit must be included with the type (*Text*) to avoid confusion with the *Text* property of many of the components in Delphi.

The next step in reading a text file is opening the file by calling the *AssignFile* procedure, and moving to the beginning of the file by calling *Reset* (see [Figure 4](#)) from the sample project, *TextFile*. *TextFile* reads the contents of a text file, and displays it in a memo component on a form.

The *AssignFile* call takes two parameters. The first is the text-file variable and the second is the name of the file. The call to *Reset* positions the file pointer to the beginning of the file. If you want to create a new file and write to it, call *Rewrite* instead of *Reset*. To append to an existing text file, call *Append* instead of *Reset*. The **while** loop reads the lines from the file one at a time and loads them into the memo component using the *Readln* procedure to read each line. *Readln* takes two parameters; the first is the file variable, and the second is a *String* variable that receives the line of text. You don't have to use a *String* variable with *Readln*. You can use any valid type. For example, if you have a text file that contains an integer on each line, you can use:

```
Readln(TxtFile, I);
```

to read the numbers, where *I* is an integer variable. *Readln* will ignore any white space in front of the integer and ignore any text that follows the integer. The code in [Figure 5](#) is the *OnClick* event handler for the **Copy** button in the *TextFile* sample program; it illustrates copying the lines from the Memo component to a new text file. This code is virtually identical to the code used to read the text file, except for the call to *Rewrite*, which creates a new file for output, and the

```

procedure TForm1.CopyBtnClick(Sender: TObject);
var
    txtFile: System.Text;
    i: Word;
begin
    { Open the output file. Overwrite if it exists. }
    AssignFile(txtFile, 'copy.txt');
    Rewrite(txtFile);
    { Write all of the lines in the Memo component. }
    for i := 0 to Memo1.Lines.Count - 1 do
        Writeln(txtFile, Memo1.Lines[i]);
    { Close the output file to save what was written. }
    System.Close(txtFile);
end;

```

**Figure 5:** This *OnClick* event handler copies lines from the Memo component to a new text file.

call to *Writeln*, which writes a line from the Memo component to the file.

### Exporting to a Delimited File

There are two ways to export the contents of a table to a delimited file. The first is using the BDE ASCII Text Driver. However, there is a trick: The text file must exist. You can also write a general-purpose, delimited-text export routine using the text file run-time library routines described in the previous section. The procedure in [Figure 6](#) takes a file name and a DataSet component as parameters, and exports the contents of the DataSet to the file.

This routine starts by making sure that the file name and DataSet parameters are valid. Next, it opens the text file for output and saves a zero-based number of fields in the DataSet component. The real work takes place within the **for** loop. This loop writes each field from the current record in the DataSet to the text file, inserting the necessary field delimiters and separators based on the type of each field.

An **if** statement checks the field type and skips any BLOB fields except for memos, because there is no way to store a BLOB that doesn't contain text in a text file. The next **if** statement writes the opening delimiter if the field is a string or memo (i.e. *ftString* or *ftMemo*). If the field is a memo, a custom procedure, *dgWriteMemo*, is called to write it. Otherwise, the standard Pascal *Write* procedure is called. The *WriteMemo* procedure is described later in this section.

Notice that if the field is not a memo, the *Write* procedure is used to write it instead of *Writeln*. This is because *Writeln* writes an entire line to the text file, including the carriage return/line feed at the end. *Write*, however, does not append a carriage return/line feed, so it can be used to write the fields one at a time.

Next, the closing delimiter character is written if this field is a string. Finally, the separator character is written if this is not the last field in the record. The *Writeln* procedure call writes the carriage return/line feed at the end of the record. The call to *FDataSet.Next* moves to the next record in the DataSet, and the process repeats until the end of the DataSet is reached.

```

procedure TdgDelimitedExport(DelimitedFile: string;
  SourceDataSet: TDataSet);
var
  AsciiFile: System.Text;
  I: Integer;
  LastField: Integer;
  RecCount: Longint;
begin
  { Make sure the user specified a file name. }
  if DelimitedFile = '' then
    raise Exception.Create('No text file name.');
```

{ Make sure the DataSet property is set. }

```

  if SourceDataSet = nil then
    raise Exception.Create('DataSet is nil.');
```

{ Open the delimited file. }

```

  AssignFile(AsciiFile, FDelimitedFile);
  Rewrite(AsciiFile);
  LastField := SourceDataSet.FieldCount - 1;
  RecCount := 0;
  while (not FDataSet.EOF) and
    (RecCount < MaxRecords) do begin
    Inc(RecCount);
    for I := 0 to LastField do begin
      { If the field is a BLOB other than a memo skip it. }
      if SourceDataSet.Fields[I].DataType in
        [ftBlob, ftGraphic] then
        Continue;
      { If the field is not numeric write the opening
        delimiter character. }
      if (SourceDataSet.Fields[I].DataType in
        [ftString, ftMemo]) then
        Write(AsciiFile, Delimiter); { Write field value. }
      if SourceDataSet.Fields[I].DataType = ftMemo then
        dgWriteMemo(SourceDataSet, I, AsciiFile)
      else
        Write(AsciiFile, SourceDataSet.Fields[I].AsString);
      { If the field type is not numeric write the closing
        delimiter character. }
      if (FDataSet.Fields[I].DataType in
        [ftString, ftMemo]) then
        Write(AsciiFile, Delimiter);
      { If this is not the last field write the separator
        character. }
      if I < LastField then
        Write(AsciiFile, Separator);
    end; { for }
    { Write carriage return/line feed at end of record. }
    WriteLn(AsciiFile, '');
    FDataSet.Next;
  end; { while }
  System.Close(AsciiFile);
end;
```

**Figure 6:** This procedure takes a file name and a DataSet component as parameters, then exports the DataSet contents to the file.

Figure 7 shows the *dgWriteMemo* procedure, which writes the contents of a memo field to the ASCII file. Because memo fields can be very large, it may not be possible to fit the entire contents of a memo in memory; therefore, memo fields must be accessed using a *BlobStream* object. This code begins by creating a *BlobStream*. The constructor for *TBlobStream* takes two parameters; the first is the field object for the memo field and the second is the mode — in this case *bmRead*, because this routine only needs to read the memo. The **repeat** loop reads the memo 1,024 bytes at a time, and writes the data to the ASCII file. When the *BlobStream*'s *Read* method returns zero for the number of bytes read, the loop terminates and the *BlobStream* is freed. Although it is not likely you would export really large memos to an ASCII file, you can improve performance with

```

procedure dgWriteMemo(SourceDataSet: TDataSet;
  FieldNum: Integer; var AsciiFile: System.Text);
var
  ReadCount: Longint;
  StringBuffer: string;
  BlobStream: TBlobStream;
begin
  { Create the blob stream for this field. }
  BlobStream := TBlobStream.Create(TBlobField(
    FDataSet.Fields[FieldNum]), bmRead);
  SetLength(StringBuff, 1024);
  try
    repeat
      { Read some bytes from the memo. }
      ReadCount := BlobStream.Read(StringBuff[1], 1024);
      { Write the characters to the delimited file. }
      Write(AsciiFile, StringBuffer);
    until ReadCount = 0;
  finally
    BlobStream.Free;
  end;
end;
```

**Figure 7:** The *dgWriteMemo* procedure writes the contents of a memo field to an ASCII file.

```

{ Copy the source file to the destination file. }
procedure CopyFile(srcName, DestName: string);
var
  buff: array[1..8192] of Char;
  srcFile, destFile: File;
  readCount, writeCount: Integer;
begin
  { Open the source file. }
  AssignFile(srcFile, srcName);
  Reset(srcFile, 1);
  { Open the destination file. }
  AssignFile(destFile, destName);
  Rewrite(destFile, 1);
  { Copy a buffer-full at a time until the end of the file
    is reached, or a write error occurs. }
  repeat
    BlockRead(srcFile, buff, SizeOf(buff), readCount);
    BlockWrite(destFile, buff, readCount, writeCount);
  until (readCount = 0) or (writeCount < readCount);
  { Close the files. }
  System.Close(srcFile);
  System.Close(destFile);
end;
```

**Figure 8:** The *CopyFile* procedure.

large memos by reading and writing the data in much larger blocks (for example 64KB instead of 1KB).

### Working with Fixed-Length Files

Delphi's untyped files can be opened in read, write, or read-write mode, and can be randomly positioned to any record in the file. What makes untyped files unique is that you can read or write more than one fixed-length record at a time, and reading and writing is done with special low-level, high-performance procedures. The sample project COPY.DPR uses untyped files to allow you to copy any file, regardless of its type. The *CopyFile* procedure from this example is shown in Figure 8.

The first thing you'll notice about untyped files is that the *Reset* procedure takes an optional second parameter. This parameter



```

type
  TAddress = record
    name: array[1..35] of Char;
    addr: array[1..35] of Char;
    city: array[1..18] of Char;
    state: array[1..2] of Char;
    zip: array[1..10] of Char;
end;

```

**Figure 9:** The sample project FAST.DPR uses this record structure to process a file of multi-field, fixed-length records.

```

procedure TForm1.CreateBtnClick(Sender: TObject);
const
  MaxRecs = 100;
var
  buff: array[1..MaxRecs] of TAddress;
  addrFile: File;
  i, count: Word;
begin
  Assign(addrFile, 'addr.dat');
  Rewrite(addrFile, SizeOf(TAddress));
  { Put 100 records into the buffer. }
  for i := 1 to MaxRecs do
    with buff[i] do begin
      PasToArray('John Doe', name);
      PasToArray('123 East Main Street', addr);
      PasToArray('New York', city);
      PasToArray('NY', state);
      PasToArray('55555-5555', zip);
    end;
  { Write 100 buffers (10,000 records). }
  for i := 1 to 100 do
    BlockWrite(addrFile, buff, MaxRecs, count);
  System.Close(addrFile);
end;

```

**Figure 10:** The *OnClick* method for the **Create** button.

is the record size in bytes. If you omit the record size, it defaults to 128. The purpose of this program is to copy any file of any type and size, so the record size is set to one byte, indicating that the size of the copy will match the size of the original file.

The actual copying occurs in the **repeat** loop. You must use *BlockRead* to read untyped files and *BlockWrite* to write to them. *BlockRead* takes four parameters. The first is the file variable. The second is the name of the variable or structure to use as a buffer for the data. In this example, *buff* is an 8KB array of *Char*. The third parameter is the number of records to read. The number of records times the record size must be less than or equal to the size of the buffer. The final parameter, *readCount*, is set to the number of records actually read. This will always equal the number of records to read (the third parameter), except when you reach the end of the file, when it might be less.

In the call to *BlockRead*, the number of records to write is given as *SizeOf(buff)*. The *SizeOf* function will return the size in bytes of any simple or structured variable. Since the record size is one byte in this example, the size of the buffer array is equal to the number of records to read. Always use *SizeOf* anywhere that your Pascal code requires the size of a variable or structure, so the code will still be correct if you change the size of the variable.

*BlockWrite* uses the same parameters as *BlockRead*, except that the last parameter is set to the number of records actually written. This will always equal the number of characters to write (the third parameter) unless an error

```

for i := 1 to MaxRecs do
  with buff[i] do begin
    PasToArray('John Doe', name);
    PasToArray('123 East Main Street', addr);
    PasToArray('New York', city);
    PasToArray('NY', state);
    PasToArray('55555-5555', zip);
  end;

```

**Figure 11:** This **for** loop loads the buffer with 100 identical records.

occurs. In this example, the number of records to write is set to *readCount*, which is the number of records read.

The **repeat** loop continues until *readCount* equals zero, indicating the end of a file, or until all the records cannot be written, which means an error (e.g. a full disk) has occurred. The last two statements close the source and destination files.

Using *BlockRead* and *BlockWrite* allows you to transfer many records at one time, which also improves performance. To see how fast untyped files are, try the sample project FAST.DPR. This program lets you create and read a 10,000-record, 1MB file. You'll be astounded at the speed. After you write the file, get out of Windows and reboot your computer to make sure the file is not in your disk cache before you click the **Read** button, so you will get an accurate demonstration of how long it takes to read the file from disk.

This program also demonstrates how to process a file of multi-field, fixed-length records, using the record structure shown in **Figure 9** as an example. This defines the *TAddress* type with a total size of 100 bytes. The code in **Figure 10** is from the **Create** button's *OnClick* method. This procedure begins by declaring an array of 100 *TAddress* records to use as the buffer. The *AssignFile* and *Rewrite* statements assign the file name to the file variable and open the file with a record size of *SizeOf(TAddress)*. (Again, using *SizeOf* ensures that if you change the structure of *TAddress*, the record size in the *Rewrite* call will adjust automatically.) Notice the use of the constant *MaxRecs* throughout this code to define the number of records in the buffer. This also makes changes easier. If you want to change the number of records in the buffer, all you have to change is the constant declaration, and the rest of the code takes care of itself.

The **for** loop shown in **Figure 11** loads the buffer with 100 identical records. *PasToArray* is a custom procedure that assigns the value of a Pascal string to a type *Char* array, using the code in **Figure 12**. Notice the use of an open array parameter for the array. This lets you pass an array of any size. The first **for** loop copies the contents of the string to the array. The subscript of the array is computed as  $[i - 1]$  because the subscript of an open array formal parameter is always zero-based, regardless of how the actual parameter is defined. The second **for** loop fills the remainder of the array elements with spaces. The following **for** loop:

```

for i := 1 to 100 do
  BlockWrite(addrFile, buff, MaxRecs, count);

```

```

{ Copy a Pascal String to a Char array. The array is padded
with blanks. }
procedure PasToArray(const str: string;
var arr: array of Char);
var
i: Word;
begin
{ Copy the string to the array. }
for i := 1 to Length(str) do
arr[i - 1] := str[i];
{ Fill the array with spaces in case the string is
shorter than the array. }
j := i + 1;
for i := j to High(arr) do
arr[i] := ' ';
end;

```

Figure 12: The *PasToArray* procedure.

```

procedure TForm1.ReadBtnClick(Sender: TObject);
const
MaxRecs = 500;
type
Tbuff = array[1..MaxRecs] of TAddress;
var
buff: ^Tbuff;
addrFile: File;
total, count: Word;
begin
try
New(buff);
except
on EOutOfMemory do Exit;
end;

try
Assign(addrFile, 'addr.dat');
Reset(addrFile, SizeOf(TAddress));
{ Read the file MaxRecs records at a time. }
total := 0;
repeat
count := 0;
BlockRead(addrFile, buff^, MaxRecs, count);
total := total + count;
until count = 0;
System.Close(addrFile);
ReadCount.Caption := IntToStr(total);
finally
Dispose(buff);
end;
end;

```

Figure 13: The *Read* button's *OnClick* method.

writes the buffer to disk 100 times to create the 10,000-record test file. The *Read* button's click method, shown in Figure 13, is similar, except that it dynamically allocates a 500-record buffer on the heap. Start by looking at the declarations in this procedure, particularly the following statements:

```

const
MaxRecs = 500;
type
Tbuff = array[1..MaxRecs] of TAddress;
var
buff: ^Tbuff;

```

Here, the constant *MaxRecs* is set to 500, and a type, *Tbuff*, is declared as an array of 500 *TAddress* records. The pointer variable, *buff*, is declared as a pointer to type *Tbuff*. Because each

```

type
TAddress = record
name: array[1..35] of Char;
addr: array[1..35] of Char;
city: array[1..18] of Char;
state: array[1..2] of Char;
zip: array[1..10] of Char;
delimiter: array[1..2] of Char;
end;

```

Figure 14: The revised *type* declaration for the record.

*TAddress* record is 100 bytes, this array will consume 50KB of memory. You need to allocate this large array on the heap by calling the *New* procedure. The remaining code is enclosed in a *try..finally* block to ensure the *Dispose* procedure will be called to release the memory allocated by *New*, if an exception (i.e. run-time error) occurs.

The next two statements open the file and the *repeat* loop reads 500 records at a time, until it reaches the end of the file. The *repeat* loop also counts the number of records read in the variable *total*. After the file has been read, the statement:

```
ReadCount.Caption := IntToStr(total);
```

displays the number of records in the Label component, *ReadCount*.

Another approach to reading and writing fixed-length files is to use the Windows API calls *OpenFile*, *ReadFile*, and *WriteFile*. For an example of these routines, see the *CopyFile* procedure in the *FilManEx.dpr* sample program that ships with Delphi 3.

### Working with Fixed-Length Text Files

While fixed-length files on mainframes and minicomputers don't have record delimiters, this is not the case in the PC world. Many PC programs that produce fixed-length files append a carriage return/line feed pair to the end of each record, so the files are actually text files, even though each record has a fixed length.

However, you can still read or write the file as an untyped file, and you will probably want to because both typed and untyped file processing is faster than text-file processing. The *FIXTEXT.DPR* project is identical to *FAST.DPR*, except that it creates a text file.

The revised *type* declaration for the record is shown in Figure 14. The only change is to add the two-character array-named delimiter to the end of the record to hold the carriage return and line feed.

The only other change is in the *Create* button's *OnClick* method, shown in Figure 15. Here, the lines:

```

delimiter[1] := #13;
delimiter[2] := #10;

```

have been added to assign the carriage return character (ASCII 13), and the line feed character (ASCII 10), to the

```

procedure TForm1.CreateBtnClick(Sender: TObject);
const
  MaxRecs = 100;
var
  buff: array[1..MaxRecs] of TAddress;
  addrFile: File;
  i, count: Word;
begin
  Assign(addrFile, 'addr.dat');
  Rewrite(addrFile, SizeOf(TAddress));
  { Put 100 records into the buffer. }
  for i := 1 to MaxRecs do
    with buff[i] do begin
      PasToArray('John Doe', name);
      PasToArray('123 East Main Street', addr);
      PasToArray('New York', city);
      PasToArray('NY', state);
      PasToArray('55555-5555', zip);
      delimiter[1] := #13;
      delimiter[2] := #10;
    end;
  { Write 100 buffers (10,000 records). }
  for i := 1 to 100 do
    BlockWrite(addrFile, buff, MaxRecs, count);
  System.Close(addrFile);
end;

```

**Figure 15:** The **Create** button's **OnClick** method, revised.

two elements of the delimiter array at the end of each record. If you have an editor that can read large files, take a look at ADDRESS.DAT after you run this program, and you'll see that each record appears as a separate line.

## Conclusion

The BDE and Object Pascal have a variety of tools you can use to get data into, or out of, your programs in the format you need. Next month, we'll discuss more options — including the use of *FileStream* objects and bitwise operators — as we continue to explore the features that make it possible to import and export data in any format. ▲

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAY\DI9805BT.*

Bill Todd is President of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is co-author of four database programming books, including *Delphi: A Developer's Guide* and *Creating Paradox for Windows Applications*. He is a Contributing Editor of *Delphi Informant*, a member of Team Borland providing technical support on the Borland newsgroups, and has been a speaker at every Borland Developers Conference. He can be reached at [BillTodd@compuserve.com](mailto:BillTodd@compuserve.com) or (602) 802-0178.





# COLUMNS & ROWS

Delphi / AS/400

By *Bradley MacDonald*



## AS/400 Shortcut

Executing AS/400 Commands/Programs with *TQuery*

**T**here are many ways to execute commands or programs on an AS/400 from Delphi. The most common is to use a Remote Procedure Call (RPC) — the method used by Delphi/400 and Light Lib/400. This usually requires software on both the AS/400 and the client machine. However, if you're using an ODBC driver, you don't always have access to an RPC API (Note: ClientAccess/400 has both an ODBC driver and an RPC API).

Fortunately, there are many ways to run commands or programs on the AS/400 without using an RPC call. These include database triggers, File Transfer Protocol (FTP), and the Delphi Query component. We'll briefly discuss database triggers and FTP; however, the focus of this article is on running commands and programs on the AS/400 using only the standard Delphi Query component.

**Database triggers.** On the AS/400, it's possible to set up a database trigger to call a high-level language program instead of SQL. A file could be set up with one column that will hold the command to be run. A trigger would then be set up for the file that would run after update or insert. The trigger would call a program that would read the new value of the field and run that new value as a command on the AS/400 via the *QCMD* API. Delphi would have to perform an update or insert on the trigger table using a SQL statement to have the command in the field run on the AS/400.

**File Transfer Protocol.** The AS/400 server supports the FTP sub-command *RCMD*.

Using *RCMD*, you can run just about any command on the AS/400, providing the AS/400 is running the FTP server. You could use the FTP component that ships with Delphi to connect to the AS/400 and run the command. The format of the sub-command is:

```
quote rcmd <AS/400 command>
```

The *quote* sub-command sends the command to the server for processing, and is used when your client doesn't support a sub-command the server supports. Here's an example FTP script:

```
open AS400IPNAME
user USERID PASSWORD
quote rcmd SNDMSG MSG(HELLO!) TOUSR(QSYSOPR)
quit
```

### Let's Get to It

The AS/400 has a utility for executing AS/400 commands from inside high-level languages, such as COBOL and RPG. The utility is an API procedure named *QCMD* that takes two parameters: the first is the command or program to be run

## COLUMNS & ROWS

on the AS/400 as a string; the second is the length, in characters, of the first parameter. The second parameter has a unique requirement: It must be in the format of “10.5” with full-zero fill. For example, a command that was 26 characters in length would have a second parameter of 0000000026.00000. So, if you wanted to send a “Hello!” message to the system operator, the full command string would look like:

```
CALL QSYS.QCMDEXC('SNMSG MSG(Hello!) TOUSR(QSYSOPR) ',  
0000000033.00000)
```

This is the procedure we'll use to run commands on the AS/400. Note the use of the period to separate the *QSYS* library from the *QCMDEXC* object. This isn't standard AS/400 syntax; standard syntax would use a forward slash instead of a period. The reason for the change is the ClientAccess/400 ODBC driver. It allows the developer to choose whether they will follow the standard AS/400 syntax and use a slash, or follow SQL syntax and use a period. The ODBC driver defaults to the SQL syntax. This can become an issue, so be sure all your client machines have the same ODBC driver configuration.

### The Query Component

The next step is to show how the Query component can be used to execute *QCMDEXC* on the AS/400. The *TQuery* class has a method called *ExecSQL* that issues the *SQL* property to the server and doesn't expect a result. To use this method to issue the AS/400 command, you must assign the API string to the *SQL* property, then call the *ExecSQL* method. Because *QCMDEXC* is part of the operating system API, it will run from almost any environment on the AS/400. When the *ExecSQL* method is called, it sends the *SQL* property to the AS/400 SQL environment. There, the API is recognized and executed, not as a SQL statement, but as an AS/400 command. By using this technique, you should be able to issue commands on the AS/400 from Delphi without having to use an RPC call or stored procedure. (Using a stored procedure would require the full SQL product to be installed on the AS/400.)

The main drawback of this method is that it isn't possible to receive results from the AS/400. Also, *QCMDEXC* is sensitive to AS/400 commands with quotes in them. During my tests, I couldn't get an AS/400 command with quotes to work. For example:

```
SNMSG MSG('Hello to the AS/400') TOUSR(QSYSOPR)
```

wouldn't work because of the quotes in the *MSG* parameter.

This means the developer won't receive clear error messages when a command doesn't run correctly on the AS/400. The developer must dig into AS/400 job logs which don't seem to provide the same level of detail as when the command is run natively on the AS/400. If the command fails on the AS/400, it will generate a *DBEngineError* exception in Delphi, which allows the

## Accessing Multiple-Member AS/400 Files

One issue facing developers creating client/server programs against existing AS/400 files is that of accessing multiple-member files. It's possible for an AS/400 file to contain *multiple members*, where each member is almost a file unto itself. Each member has the same structure (columns) as well as other attributes, but contains different records and/or data.

When you access a multiple-member file without specifying which member, it accesses the member with the same name as the file by default. This causes a problem with SQL, which doesn't support the AS/400 member specification syntax. For a client/server program to access a different member from the default, it must issue an AS/400 command named *OVRDBF* (Override Database File). This command is used to redirect the I/O request from the intended file/member to a different file/member.

The *OVRDBF* command affects only the AS/400 job under which it is run. Depending on the method you use to execute the command on the AS/400, it may or may not have an effect on your SQL query. For example, you might connect to the AS/400 database using ODBC, and use a third-party product for issuing remote commands. The ODBC database connection would be one AS/400 job, and the RPC command would be another, possibly requiring its own logon ID and password. In this example, the *OVRDBF* command would have no effect on the SQL query because it's being run in a different AS/400 job. While this isn't the case for all RPC connections, it's something to be aware of.

An interesting solution to this problem is to use another Query component to submit the *OVRDBF* command to the AS/400. As long as both Query components are using the same Database component, they'll both run in the same job on the AS/400.

The syntax for *OVRDBF* that allows you to access a different member of the same file is:

```
OVRDBF FILE(X) TOFILE(Library/X) MBR(Y)
```

Using a Query component, the *SQL* property would look like:

```
CALL QSYS.QCMDEXC('OVRDBF FILE(X) TOFILE(Library/X) MBR(Y) ',  
0000000039.00000)
```

Where *Library* is the library that contains the file *X*. Then, whenever you issue a SQL statement that references file *X*, you'll actually access the particular member *Y* of file *X*. This will be in effect until you change it, or close and reopen your database connection. You could even use the same Query component to run the *OVRDBF* command, then run the SQL query.

— Bradley MacDonald



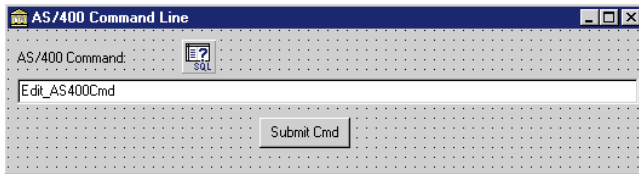


Figure 1: The demonstration form at design time.

```

procedure TForm_AS400Command.Button_SubCmdClick(
    Sender: TObject);
const
    QUOTE = '''';
var
    AS400Cmd, CmdLen, ParamTwo: string;
    ZeroFill, I: Integer;
begin
    { Close the Query }
    Query_AS400CMD.Close;

    { Create the Command to send to the AS/400 }
    CmdLen := IntToStr(Length(Edit_AS400Cmd.Text));
    ZeroFill := 10 - Length(CmdLen);
    ParamTwo := '';
    for i := 1 to ZeroFill do
        ParamTwo := ParamTwo + '0';
    ParamTwo := ParamTwo + CmdLen + '.00000';
    AS400Cmd := 'CALL QSYS.QCMDEXC(' + QUOTE +
        Trim(Edit_AS400Cmd.Text) + QUOTE + ',' +
        ParamTwo + ')';
    Query_AS400Cmd.SQL.Clear;
    Query_AS400CMD.SQL.Add(AS400Cmd);

    { Execute the command on the AS/400 }
    try
        Query_AS400CMD.ExecSQL;
    except
        ShowMessage('AS/400 Command Failed');
    end;
end;
    
```

Figure 2: This procedure sends a command to the AS/400.

developer to check to see if the command was run successfully. When the command is submitted to the AS/400, the Delphi program waits until the AS/400 command has finished before continuing.

While this may be fine for small jobs, the developer should encase the command to be run inside a *SBMJOB* command. The *SBMJOB* will cause the command to be run in the Batch subsystem on the AS/400 (by default), and return control to the Delphi program immediately. The only drawback to this technique is that the Delphi program will not know when the AS/400 program has completed — or if it has completed successfully.

### Putting It to Use


There are a variety of ways to put this technique to use. An obvious one is to run reports and batch-oriented jobs on the AS/400. It becomes fairly easy to set up a situation where a user clicks a button to have Delphi submit a job to the AS/400, e.g. print a report to a LAN printer close to the user's desk. While the report is being generated and printed, the user is able to continue with other work.

One of the more beneficial ways of using this method would be to access multi-member files. You could also use this technique to

run the AS/400 *OVRDBF* command before running the SQL SELECT against a file, both using the same Query component.

I've created a sample program (see Figures 1 and 2) that implements a command-line interface to the AS/400 using a Query component and the ClientAccess/400 ODBC driver. The user can enter the AS/400 command into the Edit component, and hit the **Submit Cmd** button to send it to the AS/400. See the sidebar "Accessing Multiple-Member AS/400 Files" for details.

### Conclusion

This article is based on various discussion threads on Borland newsgroups. The news server name is forums.borland.com, specifically the borland.public.delphi.as400 newsgroup. If you are using Delphi against the AS/400 server, I recommend visiting this newsgroup. 

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAYDI9805BM*

Bradley MacDonald is a Technical Planner at the British Columbia Liquor Distribution Branch, where he supports Delphi, AS/400, and Lotus Notes. He can be reached on the Internet at [Bradley\\_MacDonald@LDB.GOV.BC.CA](mailto:Bradley_MacDonald@LDB.GOV.BC.CA) or [Bradley\\_MacDonald@BC.Sympatico.CA](mailto:Bradley_MacDonald@BC.Sympatico.CA).





## IN DEVELOPMENT

Delphi 1, 2, 3 / Windows / DOS

By Alan C. Moore, Ph.D.

# Any Port in a Storm

## Moving Applications to New Platforms

**F**ew developers are moving applications from DOS to Windows these days. Porting applications from 16-bit to 32-bit Windows, however, is commonplace. Recently, I began the large task of porting a major application from DOS to 16- and 32-bit Windows. In this article, I'll share some of my experiences and tips. I'll concentrate on the problems and solutions of writing for both versions of Windows. I'll also discuss some of the more general issues of writing a large-scale application for any platform.

My preferred method for writing the same application for both Windows 3.x and Windows 95/NT is "back porting." This involves two steps: Step one is writing all the code in Delphi 3, since I have access to Raptor and its powerful keyboard templates (see my review in the [October, 1997 Delphi Informant](#)). Step two is simply copying the .DPR, .RES, .PAS, and .DCL files to a Delphi 1 subdirectory, and making the changes necessary for the application to run in that version.

What are those changes? Let's take a look.

### The IfDef Shuffle

One major change you'll need to make in every .PAS file written in Delphi 3 is substituting Windows with WinTypes and WinProcs in the `uses` clause. Of course, if we're porting from Delphi 1 up to Delphi 2 or 3, we don't need to be concerned about this; aliases in the newer version would take care of this detail. In this case, however, it gives us an opportunity to learn about a useful technique for dual 16- and 32-bit development: a particular use of conditional defines.

If you use the following series of statements in your `uses` clause, it will compile in any version of Delphi:

```
uses
{ $IfDef VER80 }
  WinTypes, WinProcs,
{ $Else }
  Windows,
{ $EndIf }
```

A somewhat comparable statement would be:

```
uses
{ $IfDef Win32 }
  Windows,
{ $Else }
  WinTypes, WinProcs,
{ $EndIf }
```

The first is based on different versions of Delphi; the latter is based on different versions of Windows. Why version VER80 and not VER10? This is for backward compatibility with Delphi's predecessor, Turbo/Borland Pascal. The last DOS version of Turbo/Borland Pascal was version 7; the first version of Delphi is considered version 8. Delphi 2 is VER90 and Delphi 3 is VER100.

This particular conditional define is vital for developing dual 16- and 32-bit code. For example, if you want to take advantage of 32-bit capabilities, and at the same time allow the code to compile under Delphi 1, you would place the 16-bit-specific code in its own block, and the 32-bit code after the `{ $Else }` statement.

What about when you must deal with code specific to Delphi 1, 2, and 3? In that case, you could use nested conditionals like this:

```
{ $IfDef VER80 }
  Delphi 1 code
{ $Else }
  { $IfDef VER90 }
    Delphi 2 code
  { $Else }
    Delphi 3 code
  { $EndIf }
{ $EndIf }
```

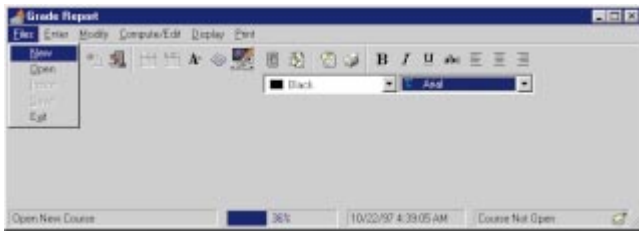


Figure 1: Several menu choices and toolbits are disabled.

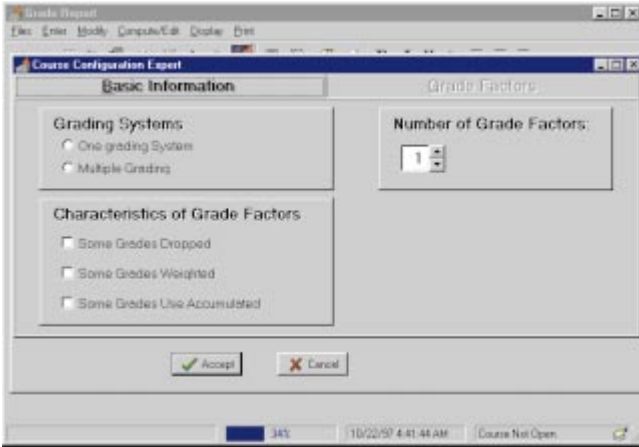


Figure 2: Creating a new course file automatically brings up the configuration dialog box.

The other problem with back porting is with the form definition (.DFM) file. Some properties such as the new *CharSet* aren't found in Delphi 1. You can safely delete the properties by opening the .DFM file in text mode, but usually you just need to open the project in Delphi 1, close the project, and save the changes. This lets Delphi delete any unsupported properties.

### The Contemporary Look

Whether you're developing in 16-bit, 32-bit, or both, you generally want your applications to include the latest Windows features — and the latest look. It's important to consider the components you're going to use. If you choose new Delphi 2/3 components, then try to back port, you'll have problems. Fortunately, there's another way.

Often you can find the functionality — the look and feel — you want by using third-party component libraries that support all versions of Delphi. Two such libraries I relied upon heavily in this application were TurboPower's venerable Orpheus collection of data-entry components and the Raize Components. Both provide tremendous functionality and modern visual qualities, and both work just as well in Delphi 1 as they do in Delphi 3.

### How to Proceed

Okay. We know the basic approach, and we've gathered our tools. Where do we start, and how do we proceed? I prefer to use a modified, top-down approach: Start with the most basic design, working out the larger details first, and saving the minute details for last. The approach is modified because often, as I'm working with an interface object, such as a data-entry screen, I work simultaneously with the data structure(s) that screen is manipulating.

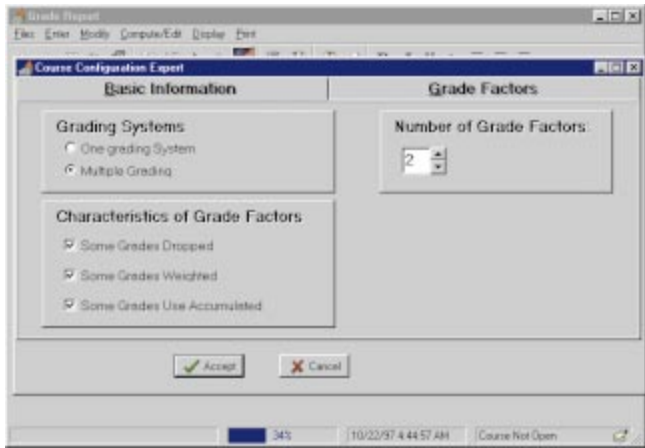


Figure 3: The second page is based on the data entered on the first page.

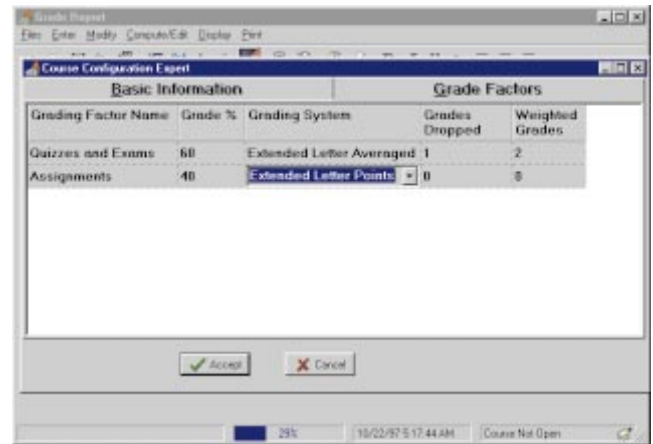


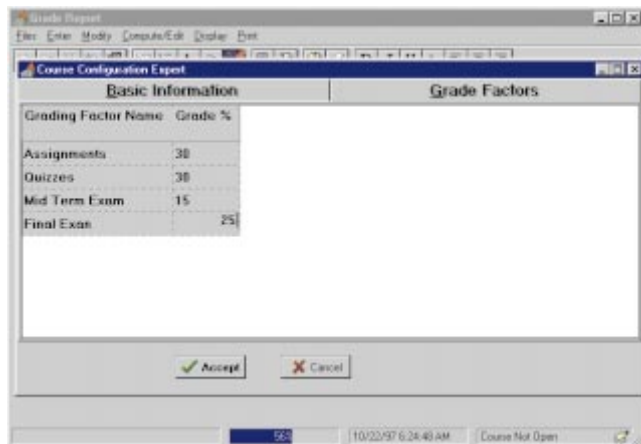
Figure 4: The two rows correspond to the two grading factors; there are five columns because three special grading factors were selected.

It's often been said that a good design at the beginning can prevent problems in the long run; that well-conceived design will lead directly to a well-designed menu system and/or toolbar. Having created this basic interface structure, you will have a good sense of what tasks lie ahead. Most of the menu items or toolbits will activate dialog boxes, which in turn will trigger program events. However, as we add menu items and toolbits, there's one thing we should take into consideration: communicating properly with the user.

### Making It Foolproof

This point is obvious, and I've seen it written more than once: Regardless of where we're porting to or from, if selecting an option is inappropriate, it should be disabled. That's why controls have an enable property. Let's look at two examples.

Take a look at Figure 1. Note that when the application starts, several menu choices and toolbits are disabled. It makes no sense to save a file, or import data, if a file hasn't been opened. When you click on **New**, the configuration dialog box comes up. Again, it makes sense to force the user to add this configuration information immediately, because nothing further can occur until they make certain decisions. Figure 2 shows the configuration dialog box. When first dis-



**Figure 5:** A different look on the second page.

played, the second page is disabled; until the user decides between a single grading system or multiple grading systems, it makes no sense to move to the second page. An alternative would be to pre-select the single grading system and give the user the option to change it. In [Figure 3](#), everything on the first page has been selected, and the second page is now operational. The second page is constructed based on the data on the first page. Let's see how.

As [Figure 4](#) shows, there are two rows because there are two grading factors; but there are five columns, because all three special grading factors have been selected. What if we didn't select any of the special options, but increased the grade factors to four? Look at [Figure 5](#). Here, compiled under Delphi 1, we see a different look on the second page. If you're wondering what kind of table control I'm using, it's *TOcTable* from TurboPower's Orpheus library. I like this control because it provides flexibility, in terms of data entry fields (note the combo box field shown in [Figure 4](#)), and the ability to easily add or hide columns and rows.

We've seen some of the beginning and middle stages of developing a 16/32-bit application. What occurs at the end?

## Wrapping Up

In the final stage, you're concerned with enabling code, such as file input/output, calculations, etc. During this stage, you might want to take advantage of 32-bit features to optimize code. For example, you may want to use threads to isolate certain segments of code, taking advantage of multi-tasking. Again, as we saw at the beginning of this article, you can make excellent use of conditional defines, so you can compile the same code in any Delphi version.

Before you turn your application over to beta testers, you'll want it to look as professional as possible. So, you may want to develop a complete help system. There are differences in help files from Windows 3.x to Windows 95/NT. As with choosing component libraries, if you're involved in dual 16/32-bit development, you'll want a help-file-generating tool that supports 16- and 32-bit Windows help file systems.

In this article, we've taken a look at some of the basic considerations of writing for 16- and 32-bit Windows. We saw that the initial design is crucial; that there are differences from one Delphi version to another that we need to be aware of; and that we can help ourselves greatly by shopping wisely for third-party components that support both versions. ▲

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







## Linked Lists

### When the Data Is Too Dynamic for Arrays

Delphi arrays are ideal for storing simple lists of objects. If the list is small, the size of the list is known, and the items in the list don't move around, an array is a perfect solution. However, if items must be frequently added and removed — possibly from different positions in the list — using an array can be difficult. This article describes an alternative: linked lists. These extremely flexible data structures allow a program to manage lists of unknown size quickly and easily.

#### Cells

The items in a linked list are stored in data structures called *cells*. Each cell is a record that contains whatever data is needed by the program. The cell also contains a pointer, or link, to the next cell in the list. For example, the following code defines a *TCell* data type

that contains a 20-character string. This code also defines a *PCell* data type representing pointers to cells:

```
type
  PCell = ^TCell;
  TCell = record
    Value: string[20]; // The data.
    NextCell: PCell;  // The next cell in
                      // the list.
  end;
```

To connect one cell to another, a program sets the *NextCell* pointer in the first cell to point to the second. The program can connect any number of cells to make a linked list as long as it needs.

By convention, a program should set the *NextCell* pointer in the last cell in a list to *nil*. This allows the program to detect the end of the list. For example, the following code displays the values stored in a linked list. The variable *top\_cell* points to the first cell in the list:

```
var
  cell: PCell;
begin
  cell := top_cell;
  while (cell <> nil) do begin
    ShowMessage(cell^.Value);
    cell := cell^.NextCell;
  end;
```





## ALGORITHMS

Linked lists are usually very dynamic, with new cells added and removed as the program runs. For that reason, most cells aren't allocated statically using the *TCell* type. Instead, a pointer to a cell is allocated, then the new cell is created using the *New* procedure. When a cell is no longer needed, its memory is released by the *Dispose* procedure.

The following code creates and then destroys a new cell:

```
var
  new_cell: PCell;
begin
  // Allocate a new cell.
  New(new_cell);
  // Free the new cell.
  Dispose(new_cell);
```

It's extremely important that a program dispose of a cell when it's removed from its linked list. Otherwise, if the program has no variable pointing to the cell, its memory will no longer be usable. The memory will be wasted until the program ends.

A program must also be certain it does not dispose of any cells it's still using. A program that accesses the fields in a cell that has been disposed will crash.

### Managing Linked Lists

Adding a cell at the top of a linked list is simple. The program sets the new cell's *NextCell* field to point to the current top cell. It then updates the top cell pointer so it points to the new cell.

For example, the following procedure adds a new cell to the top of a linked list. The process is shown graphically in [Figure 1](#). The small box with an X in it represents the *nil* pointer ending the list:

```
procedure AddToTop(var top_cell: PCell; value: string);
var
  new_cell: PCell;
begin
  // Create the new cell.
  New(new_cell);
  new_cell.Value := value;
  // Add the new cell to the list.
  new_cell.NextCell := top_cell;
  top_cell := new_cell;
end;
```

Adding an item in the middle of a linked list is almost as easy (see [Figure 2](#)). The following procedure adds a new cell after the cell pointed to by *after\_me*:

```
procedure AddAfter(after_me: PCell; value: string);
var
  new_cell: PCell;
begin
  // Create the new cell.
  New(new_cell);
  new_cell.Value := value;
  // Add the new cell to the list.
  new_cell.NextCell := after_me.NextCell;
  after_me.NextCell := new_cell;
end;
```

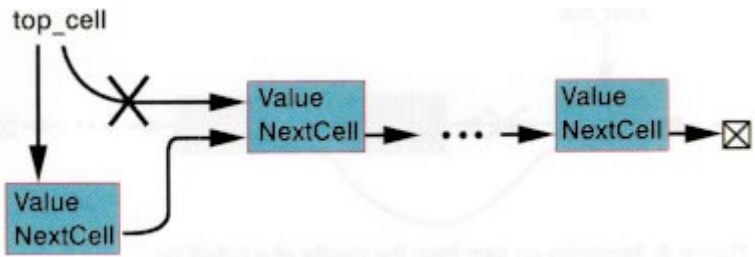


Figure 1: Adding an item to the top of a linked list.

Removing a cell from the top of a linked list is also easy (see [Figure 3](#)). The only interesting detail is that the program should dispose of the removed cell if it will no longer be needed:

```
procedure RemoveTop(var top_cell: PCell);
var
  target: PCell;
begin
  // Save a pointer to the cell.
  target := top_cell;
  // Remove the cell from the list.
  top_cell := top_cell.NextCell;
  // Dispose of the removed cell.
  Dispose(target);
end;
```

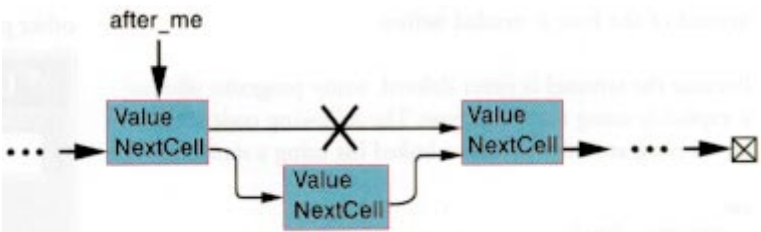


Figure 2: Adding an item in the middle of a linked list.

Finally, it is also easy to remove a cell from the middle of a linked list. This operation is shown in the following code and in [Figure 4](#):

```
procedure RemoveAfter(after_me: PCell);
var
  target: PCell;
begin
  // Save a pointer to the cell.
  target := after_me.NextCell;
  // Remove the cell from the list.
  after_me.NextCell := target.NextCell;
  // Dispose of the removed cell.
  Dispose(target);
end;
```

### Sentinels

You may have noticed that the code is almost the same for adding an item at the beginning of a list or adding it in the

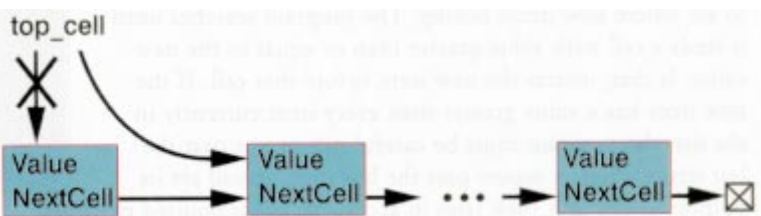
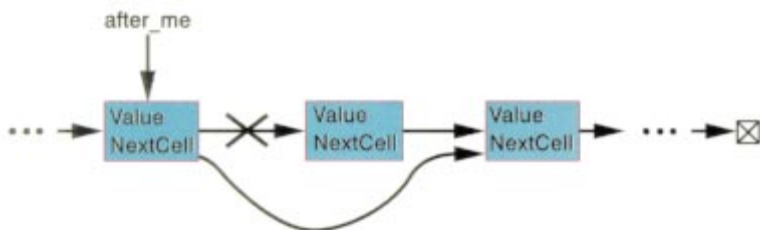


Figure 3: Removing the top item from a linked list.



**Figure 4:** Removing an item from the middle of a linked list.

middle. Similarly, the code that removes an item from the top or middle of the list is almost the same.

To allow a program to treat these cases identically, a program can add a *sentinel* to the top of the list. The sentinel is a cell that has its *NextCell* field pointing to the first actual cell in the linked list. The sentinel is never released and it never contains any real data. Its only purpose is to allow the program to treat the first cell on the list (the one after the sentinel) just like it treats cells in the middle of the list.

Now to add a cell to the top of a list, the program adds the item after the sentinel. To remove the first cell from the list, the program removes the cell after the sentinel. The program only needs the two routines *AddAfter* and *RemoveAfter* instead of the four it needed before.

Because the sentinel is never deleted, many programs allocate it explicitly using the *TCell* type. The following code shows how a program might create a linked list using a sentinel:

```
var
  sentinel: TCell;
...
// Initialize the empty list.
sentinel.NextCell := nil;
// Add some cells to the top of the list.
AddAfter(@sentinel, 'Cell 1');
AddAfter(@sentinel, 'Cell 2');
AddAfter(@sentinel, 'Cell 3');
```

The example program *LListP*, available for download (see end of article for details) and shown in **Figure 5**, demonstrates a linked list. Enter a string and click on an item in the list or the sentinel. Then click the **Insert After** button to add the new item after the one you selected. Pick an item or the sentinel and click the **Remove After** button to remove the following item from the list.

**Sorted Lists**

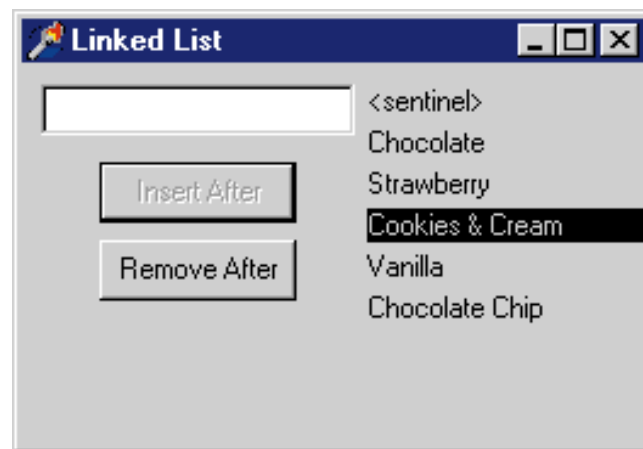
With just a little more work, a program can turn a linked list into a sorted list. Instead of adding items at specified positions in the list, the program searches through the list to see where new items belong. The program searches until it finds a cell with value greater than or equal to the new value. It then inserts the new item before that cell. If the new item has a value greater than every item currently in the list, the program must be careful not to run past the last item. When it moves past the last item, it will set its pointer to *nil*. If it then tries to access the fields pointed to by the *nil* pointer, it will crash.

Constantly checking to see if it has reached the end of the list slows the program down and makes the code more complicated. The process can be simplified by adding another sentinel at the bottom of the list. The program should give that sentinel a value greater than any that could be added to the list. For text strings, the sentinel could be given the value #255 because all normal strings come alphabetically before the character #255. Now when the program searches the list, it will definitely find a cell with value greater than the new item's value. If there's no such item in the list, the bottom sentinel will stop the search. The program no longer needs to check at each step to see if it has run off the end of the list.

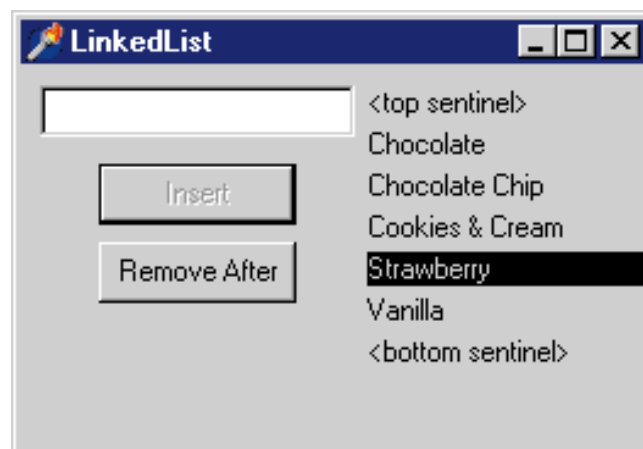
The example program *SortListP* (see **Figure 6**) manages a sorted linked list. Enter a string and click the **Insert** button to insert the item at its proper position in the list. Click on an item or the top sentinel, and click the **Remove After** button to remove the following item from the list.

**Other Linked Structures**

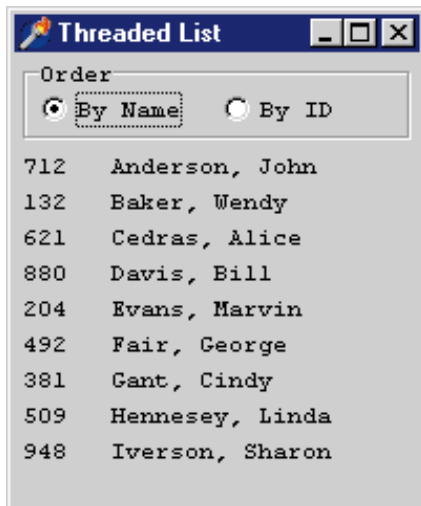
The *TCell* structure described earlier contains a single *NextCell* pointer. There's no reason it could not contain other pointers as well.



**Figure 5:** The example program *LListP* managing a linked list.



**Figure 6:** The example program *SortListP* managing a sorted linked list.



**Figure 7:** The example program ThreadP managing a threaded linked list.

For example, the cells in a doubly-linked list contain an additional *PrevCell* pointer indicating the previous cell in the list. Using the *NextCell* and *PrevCell* pointers, a program can move forward and backward through the list. This makes operations like adding or removing a cell before another cell easy.

The example program ThreadP (see [Figure 7](#)) manages a small employee list. If the user clicks on the **By Name** radio button, the program lists the employees ordered by last name. Likewise, if the user selects **By ID**, the program lists the employees ordered by ID.

A program can add other pointers to cells to create even more elaborate data structures. Using cells with *LeftChild* and *RightChild* pointers, a program can build binary trees. Using still more pointers, a program can build trees of higher degree, graphs, and networks. Using pointers makes it easy to add, remove, and rearrange cells in all these structures.

## Conclusion

Arrays are fine for storing small lists of fixed size that are seldom rearranged. But when the data changes frequently, or must often be reorganized, linked lists and other linked structures offer much greater flexibility.  $\Delta$

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM\99\MAY\DI9805RS.*

This idea can be extended to support other orderings. For example, a list of employees might contain sorted links ordering employees by name or by employee ID. The employee cells might be declared like this:

```
type
  PEmpCell = ^TEmpCell;
  TEmpCell = record
    LastName: string[20];
    FirstName: string[20];
    ID: Longint;
    NextName: PEmpCell; // Next cell in name order.
    NextID: PEmpCell;   // Next cell in ID order.
  end;
```

Each sequence of links that gives the cells a particular ordering is called a *thread*. A list made up of this kind of cell is called a *threaded linked list*. When a program adds a new employee to a threaded linked list, it must place the record in the proper position in each thread.

Rod Stephens is the author of several books, including *Custom Controls Library* [John Wiley & Sons, 1998] and *Visual Basic Algorithms* [John Wiley & Sons, 1998]. He also writes algorithm columns in *Visual Basic Developer* and *Microsoft Office & Visual Basic for Applications Developer*. You can reach him at [RodStephens@compuserve.com](mailto:RodStephens@compuserve.com), or see what else he's up to at <http://www.vb-helper.com>.





## NEW & USED

By Alan C. Moore, Ph.D.

# HelpScribble

As I am sure you're aware, there's no shortage of Help-file tools for Windows developers. Several, which include WYSIWYG and all of the "bells and whistles," come with an equally impressive price tag — in some cases two hundred dollars or more. Interestingly enough, you can spend that much money and still not get support for context-sensitive help in Delphi components. The product I'll be describing in this review, HelpScribble from JG Software, may not be as fancy as some of the products you've heard so much about. But it does get the job done, and it has excellent support for Delphi and its components.

Figure 1 shows the main HelpScribble window in an educational application. Some might call this user interface "busy." Once you get used to it, however, I think you'll find all the functionality you need is right at your fingertips. It includes five toolbars (ProjectBar, TopicBar, FootnoteBar, TextBar, Help Toolbar), a Ruler, a Topic Grid, and an Error Log window. Most of the options are available both from the toolbars and from the menu. Best of all, you can hide or show any of the toolbars, the Ruler, or Error Log window. For contrast, take a look at Figure 2, which shows the Help file (from the same educational application) being edited, but without the secondary windows or toolbars.

In addition, there are four separate editors for browse sequences, contents files (32-bit Windows only), Segmented HyperGraphics (SHG), and windows (main and secondary).

### HelpScribble at Work

As in Delphi, the central metaphor in HelpScribble is the Project. From the Project menu you can create new Help projects, as well as save, save as, open, or re-open them. You can import or export Help files between HelpScribble's native format and the traditional Windows source types (.RTF and .HPJ). You can compile or test a Help file you're working on, and set various program options. If you're producing a manual, and want to include the same language as in the Help file, you can produce a Flat Manual (a text file in which all the special footnotes, links and Help file-specific information has been removed). If needed, you can even change the numbering of topics.

Creating a new topic couldn't be more simple: Just select Topic | New, click the green "plus" tool, or hit [F4]. If you type the title of the topic in the New Topic combo box, it will automatically appear in the main edit window. (I like to copy the topic's name immediately to the Keywords edit box, directly below and to the left, right above the font name combo box.) Anything you type in the Keywords edit box will appear immediately in the Topic Grid, shown in the left column.

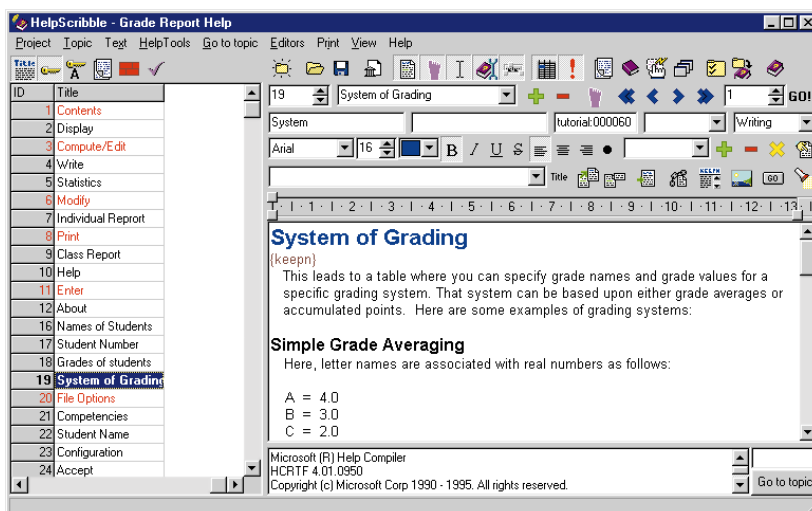


Figure 1: HelpScribble's IDE with all toolbars present.

The next step is to separate the scrolling from the non-scrolling (Title) region. Simply move the cursor to the line underneath the title in the edit window and click on the KEEP tool and HelpScribble inserts the divider for you. Now you're ready to write your Help text. Often you'll use bullets to list topics (to which you can jump), features, or similar groups of information. If you click on the bullet tool, a bullet will be inserted whenever you hit **[Enter]** to begin a new line.

**Footnotes and Links**

Writing and formatting text is part of creating an effective Help file. As with other hypertext systems, Windows Help files rely on links to enable the user to quickly find information, jumping from one page to another. It's extremely simple to add links to a Help file with HelpScribble. Simply select the word or phrase that will identify the link, then click on one of the link tools. If you select one of the tools without selecting text, HelpScribble will insert the title of the topic you want to link to into the Help text and use that as the hotspot text. You can create regular links, popup links, or target links to special areas of the Help file, such as a glossary. You can also create special links to other

Help files. If you're working in 32-bit Windows, you can create links to Internet pages or e-mail applications.

The term "footnote" is a carryover from the old days of writing Help source files as .RTF files when you placed a topic's ID, title, and other special information in actual footnotes that would be interpreted by the Help file compiler. A more appropriate term for this information, and one familiar to Delphi developers, is *property*. As we've seen, you can add just about all of the information about a topic from HelpScribble's IDE. However, you can also edit a topic's properties in the Topic Footnotes dialog box. HelpScribble automatically assigns the context string to a default value so it can display topic titles instead of context strings when you pick a topic to which to link. Some of the properties listed here, such as browse sequences, are not always used; others, such as the "A" footnote, are specific to WinHelp 4.0 and will be discussed later.

**Special Edit Windows**

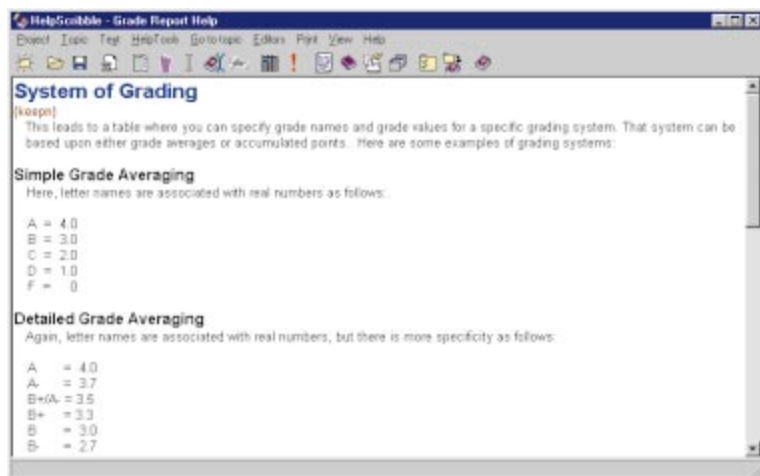
As previously mentioned, HelpScribble has four specialized edit windows, a Browse Sequence Editor, a Contents Editor, a Segmented HyperGraphics (SHG) Editor, and a Window Editor. First, let's start with the Window Editor.

HelpScribble's Window Editor allows you to set various properties for the main window, or for any secondary windows you may create. You can set the dimensions of the window, its caption, its behavior, and its colors (non-scrolling title region and main help area). If you want to add graphics with hotspots to your application, the SHG Editor is particularly useful. Typically, you take a screenshot of a dialog box and then associate various regions with Help tips concerning the data entered there. **Figure 3** shows an example from the previous education application. There are menu items and tools for creating, opening, and saving .SHG files. You can create hotspots; for each hot-spot on the graphic, you can define its type (link, popup link, or macro), links, and location. However, I recommend setting the location visually, rather than with these spin controls.

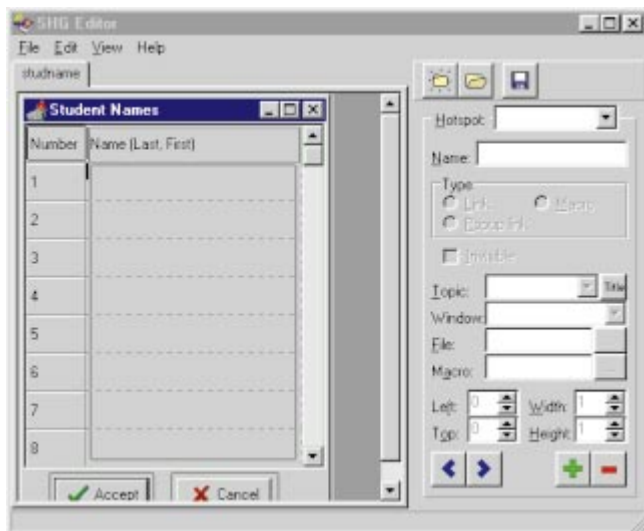
Sometimes you want to create a browse sequence in your Help file, arranging many of the screens in a certain order to lead your users by the hand in a tutorial. With HelpScribble's Browse Sequence Editor, you simply move topics from an available topics list to a browse-sequence list.

**Macro Magic and More**

With HelpScribble you have easy access to WinHelp macros and editing macros. WinHelp macros are used for various operations, including jumps to other Help files and executing applications. With HelpScribble's Macro Editor, you can associate WinHelp macros with buttons that you place in your file or hotspots on an SHG page. On the other hand, TextMacros are a special feature of



**Figure 2:** HelpScribble's simplified user interface.



**Figure 3:** HelpScribble's SHG Editor.



**INFORMANT**  
**FACT FILE**

HelpScribble is a powerful tool for creating Windows Help files. Although it is a 32-bit application, it can create Windows 3.1 or Windows 95 Help files. It includes several editors for working with SHG files, WinHelp macros, contents (.CNT) files, and windows. Its power, ease-of-use, and integration into Delphi make it a logical choice for Delphi developers.

**JG Software**  
Jan Goyvaerts  
Lerrekensstraat 5  
2220 Heist-op-den-Berg  
Belgium

**Web Site:** <http://www.tornado.be/~johnfg/helpscr.html>  
**E-Mail:** [johnfg@tornado.be](mailto:johnfg@tornado.be)  
**Price:** US\$79. Complete registration details can be found in HelpScribble's online Help.

HelpScribble you can use to store text properties. This makes it easy to enforce a consistent text style throughout your Help file. And the good news doesn't stop here: There are other tools that make creating a Help file much easier.

Bookmarking topics makes it much easier to work with a large Help file. Topics are easily bookmarked by right-clicking on them and selecting "bookmark." After that, they're shown in red in the Topic Grid (again, see **Figure 1**). After you set bookmarks, you have two

ways of navigating the topics in your Help file: sequentially by ID number or by bookmarks.

### Support for Delphi and WinHelp 4.0

HelpScribble's support for Delphi is marvelous. If you write custom components, you'll find its ability to scan a component unit and create a context Help file from the comments in that file extremely helpful. (I wish it were possible to do this with application units as well.) Also, the HelpContext Editor (see **Figure 4**) that comes with HelpScribble makes it a snap to add context-sensitive Help to your application. Once you've installed it, just double-click on any HelpContext property in the Object Inspector and the editor automatically comes up, establishing a link with HelpScribble. Notice in the figure that all of the available topics are in the listbox at the right. All you have to do is select one of these, or create a new one, to integrate the Help topic with the proper control in Delphi.

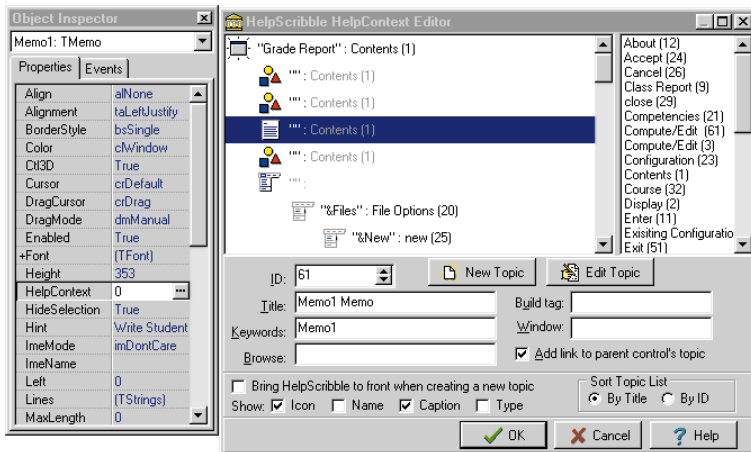
HelpScribble also includes full support for the new capabilities of WinHelp 4.0. For example, it includes all the tools you need to create a contents file, including a

Contents Editor (see **Figure 5**). There's also support for the new "A" footnotes (keywords) that allows you to create lists of related topics. You can also add buttons (associated with macros) to your Help file.

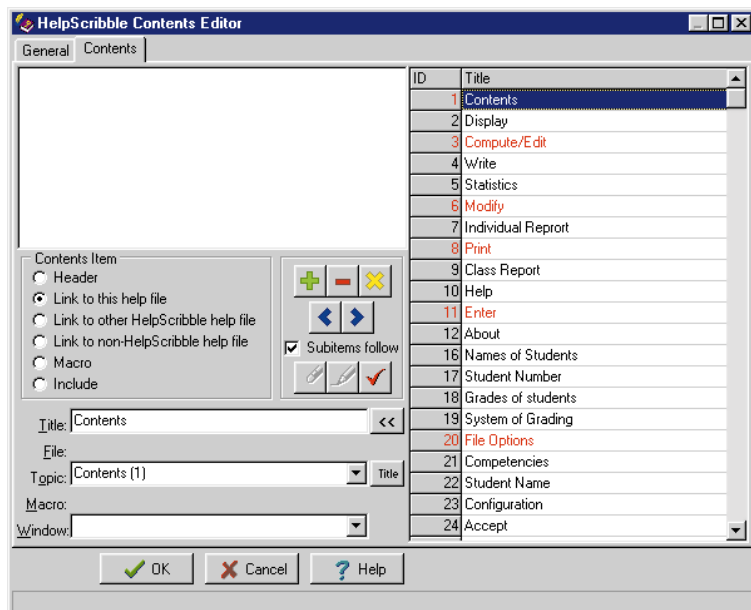
### An Ideal Help Tool for Delphi

I'm satisfied and have no plans to look any further for a Help-file tool. Once I became familiar with the interface, I was able to quickly build the Help system for the application I'm developing. As I learned more about the subtle aspects, I was able to go back and make enhancements. For example, while HelpScribble may not be WYSIWYG, its ruler and font tools allow a great deal of formatting that approach WYSIWYG. Needless to say, I recommend this product highly. Best of all, it's shareware, so you can download the shareware version (which just adds a little note to all your Help topics) and try it out before you buy it. ▲

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



**Figure 4:** The HelpScribble HelpContext Editor.



**Figure 5:** The HelpScribble Contents Editor.





## NEW & USED

*By Cary Jensen, Ph.D.*

# DotHLP

A Little Help with Your Help from Auric Visions Ltd.

**A**dding online Help to your application is a necessary step in delivering a professional, final product. As you've learned from some of the articles in this issue, using the contents of Windows Help files (.HLP) from within a Delphi application is almost trivially easy. It's the *creation* of these files that is often tedious.

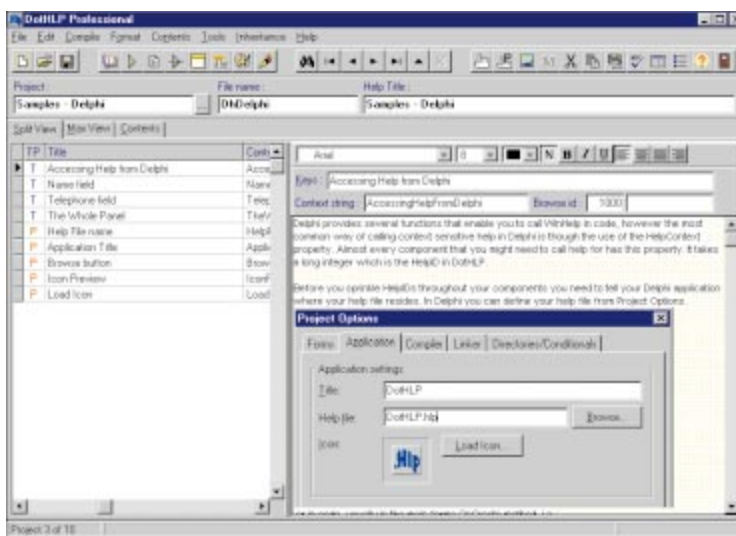
The actual task of writing the text of a useful Help system aside, the process of creating .HLP files is one that most developers would care to avoid. Just ask anyone who's ever had to resort to creating .RTF (rich text format) files with the necessary footnotes for manual compilation using HC31.EXE or HCW.EXE.

### Help Is Here

Fortunately, there are a number of third-party tools available that simplify the process of creating .HLP files. One of these, DotHLP from Auric Visions Ltd., is notable for its ease-of-use, wealth of features, and reasonable price — a combination that makes this utility worthy of consideration by anyone faced with the prospect of writing a Help system.

The features of DotHLP are accessed through the easy-to-use interface shown in [Figure 1](#). The individual pages of the Help system are written using DotHLP's integrated editor, and are stored in Paradox tables. When you generate a .HLP file, DotHLP reads the information it has stored, generates the necessary .RTF, .HPJ (help project), and .CNT (content) files, then compiles them using the Windows Help compiler.

DotHLP permits you to create Help systems as sophisticated as you need. For example, your Help pages can include numerous fonts (DotHLP includes a tool for managing fonts), font sizes and colors, page-background



**Figure 1:** All features of DotHLP are available through an integrated development environment.

colors, embedded graphics, hyperlink jumps and hypertext popups, browse-sequence definitions, embedded multimedia (Windows 95 and Windows NT only), macros, and more. In fact, DotHLP simplifies the process of adding almost any feature supported by the Windows Help compiler.

Each .HLP file in DotHLP is defined by a project that corresponds to the Paradox table in which the Help pages are stored. Within a project, each page is stored as a single record. That DotHLP stores your Help projects in a database affords you a level of flexibility that would be difficult or impossible to achieve if the Help were stored as straight .RTF. For example, you can easily copy an entire project, then add or remove individual pages. Likewise, pages can effortlessly be copied from one project to another.

### DotHLP Features

There are many other features of DotHLP; here are some of its highlights:

- Creates both 16- and 32-bit Help files, with the look of either Windows 3.1 or Windows 95, from a single project source.
- Has a built-in spell checker.
- In addition to generating .HLP files, projects can be output in .RTF format. You can use this output as the basis for written documentation.
- Existing .TXT and .RTF files can be inserted into a project.
- Includes a New Project Wizard to help you start a new project.
- Includes a number of demonstration projects so that you can see how to implement its various features.
- Online Help is extensive and informative.
- Registered users can download product updates from the Web. Auric Visions Ltd. has been very good about adding features and correcting problems identified by users.
- The integrated environment permits you to test hypertext jumps and popups without compiling the project.
- Existing projects can be used as templates for new projects, so you can easily duplicate desirable project options.
- Includes Microsoft's SHED program, which permits you to define hypertext popup regions on a 16-color bitmap you want to embed in your Help file.

There are two versions of DotHLP: Standard and Professional. Aside from the features shared by both, the Professional edition permits you to inherit the pages of one project from two or more other projects, generate HTML files from your projects, de-compile an existing .HLP file, and insert the results into a DotHLP project.

With HTML generation, you can quickly turn a Help project into a series of HTML pages. These can be a quick and easy source for a Web-based Help system for your customers. And the de-compilation feature permits you to enhance or modify a Help system for which you have lost the original .RTF files.

The inheritance feature of DotHLP Professional is particularly useful if you need to create many different .HLP files. This fea-

ture permits you to maintain one project linked to two or more projects that need to share the same set of pages. For example, all your applications may need a set of Help pages that describe how to purchase upgrades and obtain technical support. These can be stored in a separate project, and can be inherited by those projects that need to include this information. If you need to update this shared information, you merely update that one project. Re-compiling the .HLP files that inherit this information automatically inserts these changes.

As a Delphi developer, you'll particularly enjoy the fact that the author of DotHLP is a Delphi developer. (DotHLP was written in Delphi.) The extensive .HLP files that accompany DotHLP contain numerous tips and code examples that you can easily use in your Delphi applications. In addition, the online Help includes details for Visual Basic programmers.

### Conclusion

As you can tell, I like DotHLP a lot, but it does have its share of problems. For example, the existing Help de-compilation didn't work correctly with .HLP files I had previously created the old-fashioned way. It did a good job, but I needed to do some fine-tuning once the .HLP files were imported. In particular, I had to delete and re-insert many of my links. Likewise, both the .RTF and HTML export features correctly generated about 90 percent of the material. The remainder had to be adjusted manually. In fact, the DotHLP documentation warns you of this, pointing out that you'll likely have to make final adjustments following an export. Also, like any sophisticated application, using DotHLP takes a little getting used to. While you can quickly generate a simple .HLP file, more complex tasks have a learning curve. Nonetheless, DotHLP is definitely worth a look.

Now for the best part: DotHLP is very affordable. You can register the Standard edition for US\$99, and the Professional edition for US\$149. At these prices, it's really hard to go wrong. ▲



DotHLP is a full-featured, integrated Help generation system written entirely in Delphi. It's easy to use and remarkably affordable. With the exception of actually having to write the Help text, DotHLP takes the sting out of having to produce an application's Help file.

#### Auric Visions Ltd.

4 All Saints Cottages  
Turners Hill Road  
West Sussex, RH10 4HA  
UK

**Phone:** US distributor,  
ComponentSource: (888) 850-9911

**E-Mail:** support@AuricVisions.com

#### Web Site:

<http://www.AuricVisions.com> or  
<http://www.ComponentSource.com>

**Price:** Standard edition, US\$99;  
Professional edition, US\$149;  
upgrade to Professional edition,  
US\$49. A trial version can be  
downloaded from Auric Visions  
Ltd.'s Web site.

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





## NEW & USED

By Gary Entsminger

# WinHelp Office 5.0

## A Great Way to Create Help Systems

So, you've just built your latest (and greatest) Delphi application, and you're ready to sail it out to the world, where it will compete with another half-dozen, similar-looking packages. Yours is better, swifter, or niftier in some way or other, but how can you make it stand out from the pack? One way, which can save you money and help you organize your documentation, is to develop a first-class online Help system. Developed properly, this system becomes a hard copy manual you can print, and a hypertext document users can navigate online.

We're all familiar with these Help systems. They come with the best software packages (Delphi, for example), but they're often perceived by programmers and developers as an adjunct to the application, rather than as an integral part. In the past, there was something to the argument that Help systems were too difficult and time-consuming to build. Therefore, most of us were forgiven if our commentary was weak. This is no longer the case, however, and forgiveness is rare: Users now expect first-rate documentation.

New tools such as WinHelp Office 5.0 make creating a Help system as convenient as creating an outline or other organized document. When you create a Help system with WinHelp Office, you use a familiar tool — the word processor — to create, edit, and format the text for your online Help system. Most of this text resides in Topics, the basic units of a Help system. Any time you select a specific piece of information to view within a Help system, you're selecting a topic. Each topic resides within one or more document pages. WinHelp Office is integrated with Microsoft Word, so if you already know how to use Word and Windows Explorer, you already know how to use the basic tools in WinHelp Office.

The recently-released WinHelp Office 5.0 consists of RoboHELP 5.0, RoboHTML 1.0, a Help Video Kit, and 15 additional WinHelp and HTML Help-authoring tools. You can use these tools individually, or in combination, to create almost any Help system you can imagine, including ones for your Web site.

### Into the World of RoboHELP

When you fire up RoboHELP, it runs an instance of Word, then integrates itself with Word. RoboHELP accomplishes this by adding menu items for RoboHELP Tools to the Word menus, and creating a RoboHELP Tools Palette and a RoboHELP Explorer Toolbar that are accessible directly from Word. Figure 1 shows the integration of Word and RoboHELP. The RoboHELP Tools Palette is used to add topics and

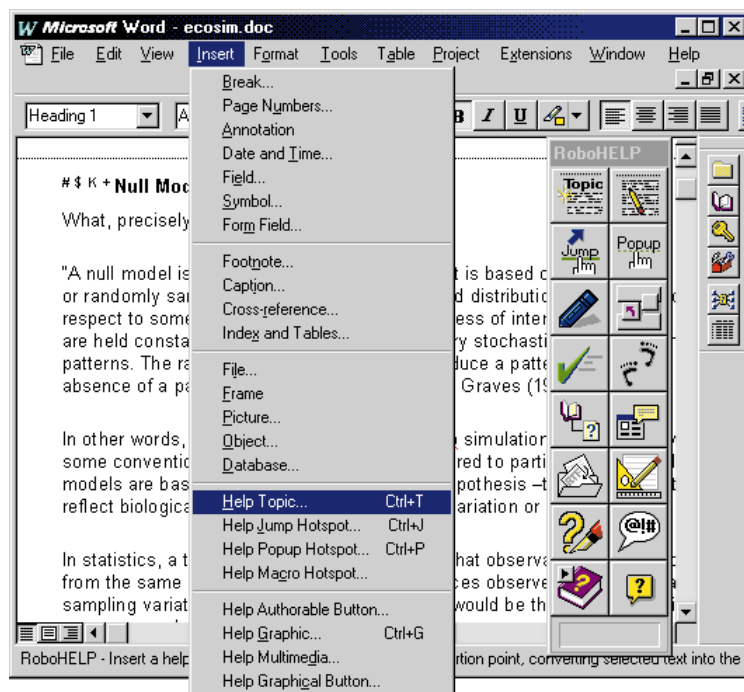
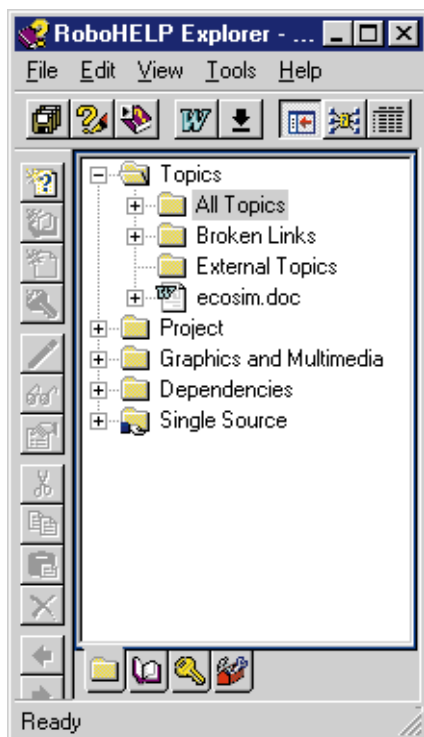


Figure 1: Integrating Microsoft Word and RoboHELP.





**Figure 2:** Navigate to any object via the RoboHELP Explorer.

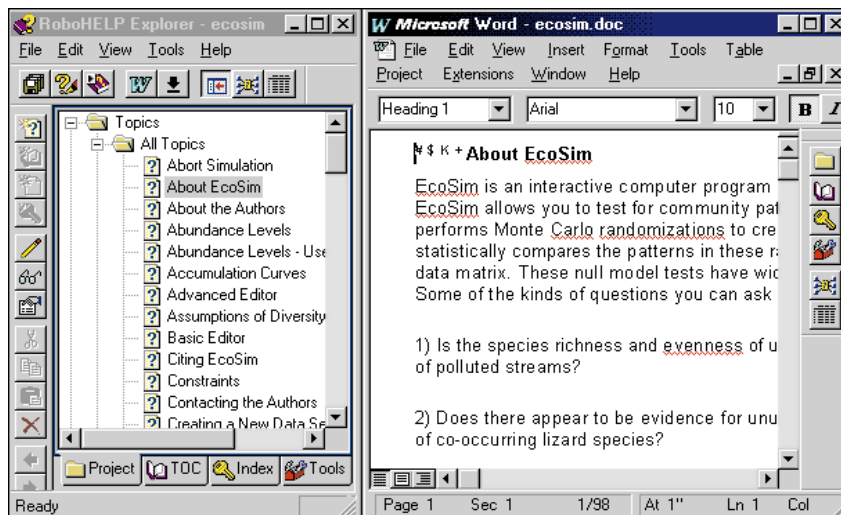
the RoboHELP Explorer. You can turn the Toolbar and Tools Palette off when you don't need them.

After you've created a rough draft of your Help system (one or more topics), you make and compile it. Compiling a RoboHELP system is fast, even by Delphi standards. For example, the 60-page Help system I recently built with RoboHELP for a null-modeling application that I developed in Delphi compiles in seconds. (Note: If you want to peruse my Help system or use the modeling application, called EcoSim, you can download it, free of charge, from <http://www.uvm.edu/~biology/Faculty/EcoSim>. The Help system that's included with EcoSim is the complete hypertext manual for the application, and demonstrates most of the techniques mentioned in this review.)

## The RoboHELP Explorer

The RoboHELP Explorer simplifies the organization of a Help system by allowing you to quickly view any object within the various hierarchies of the Help system. For example, the RoboHELP Explorer in Project View displays **Topics**, **Project**, **Graphics and Multimedia**, **Dependencies**, and **Single Source** hierarchies for the active Help system (see [Figure 2](#)).

The Index View shows the Keyword links for Topics, the TOC View shows the Table of Contents (a hierarchical View of Topics within Books), and so on. To view any object, navigate to it with the RoboHELP Explorer. Or, if you want to find a specific topic in a large system, you can search for it with the Find Topic menu command.



**Figure 3:** RoboHELP launches itself in Word, integrates with Word, then opens the RoboHELP Explorer.

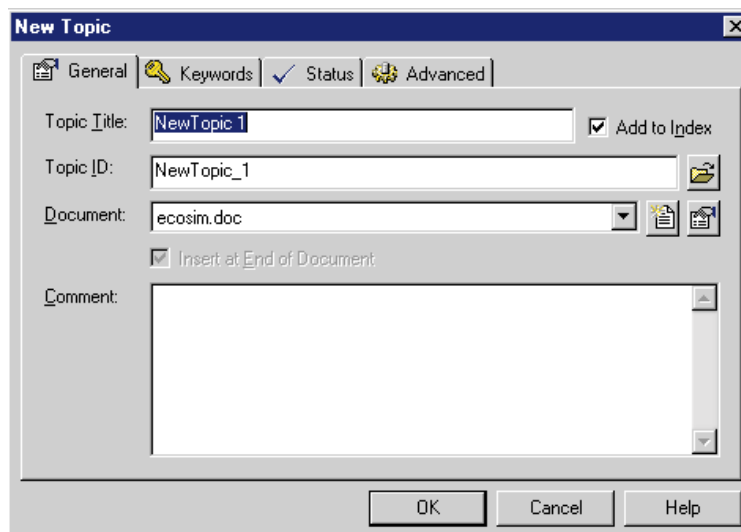
jumps (or links) to topics, and to compile and run your Help system. The RoboHELP Toolbar is used to interact with

Help topics have advantages we now associate with online media. Unlike the pages of hard-copy manuals, Help topics can be created, organized, stored, and accessed in any order. In addition, if you use a sophisticated tool such as WinHelp Office, you can create a Web site directly from a compiled RoboHELP project. The Help to HTML Wizard (one of the WinHelp Office tools) walks you through the process, using your contents file to organize the Web site's contents; it steps you through the creation of Home Page, Index Page, and so on.

## A Quick Step-Through

The following steps will give you a feel for how you might use RoboHELP to develop your own Delphi Help system:

- 1) After you've installed WinHelp Office from a CD-ROM (or disks), click the RoboHELP icon to create or open an existing Help Project. RoboHELP keeps track of the projects you're working on, and always displays a project list when you ask to create or open a new project.
- 2) If you select **New Project**, the New Project dialog box opens, showing you many WinHelp project types (including Delphi Help). After you've selected a type, RoboHELP launches an instance of Word, integrates itself within Word, and opens



**Figure 4:** Creating a new topic in the New Topic dialog box.



the RoboHELP Explorer (see Figure 3). Note that I've reduced the size of the RoboHELP Explorer and Word windows for this article. Typically, you would expand both windows to share your computer screen.

- 3) You now proceed to create and edit topics, format text, and insert graphics and hypertext links using the integrated Word/RoboHELP system. For example, click the **New Topic** button to create a new topic. The New Topic dialog box opens (see Figure 4). You can also click the **New Jump** button to create a hypertext link to another topic. If other topics exist, RoboHELP shows you a list of them from which to choose. If no other topics exist, the Create Hypertext Jump to Help Topic dialog box opens that allows you to create a new topic, which is then linked. The Index or Table of Contents are created similarly. You create and organize a Table of Contents by creating New Books and organizing topics within books by dragging them from a topic list. The Index is created and organized by creating Keywords and associating them to Topics. You create the Table of Contents as you go, then use the Table of Contents to help you organize topics. But you can modify the Table of Contents, the Keyword lists, or Topics at any time. And if your Help system becomes too large for a single Word file, you can separate it into files that RoboHELP helps you manage.
- 4) When you're ready to compile and test your Help system, you compile and run it by clicking the **Run** button.

The WinHelp Office 5.0 package consists of several other useful tools, including an excellent screen capture utility, an image

editor, a graphics converter, an .RTF converter, multimedia support, and support for all Windows 95 and NT 4.0 macros. You can also create context-sensitive Help systems and compile topics one at a time for testing.

### Conclusion

In short, creating every aspect of a Help system with RoboHELP is easy, and version 5.0 is an improvement from previous versions. My only complaint is a surprising one: The printed WinHelp Office documentation is better than the online Help system that Blue Sky Software ships with WinHelp Office. Go figure. Otherwise, thumbs up Blue Sky Software! ▲

**INFORMANT**

**FACT FILE**

WinHelp Office 5.0 makes creating Help systems as convenient as creating an outline, or other organized document. It consists of RoboHELP 5.0, RoboHTML 1.0, a Help Video Kit, and 15 additional WinHelp and HTML Help-authoring tools that can be used individually, or in combination, to create almost any Help system you can imagine, even for a Web site. Installation requirements include Windows 95 or NT 4.0, Word 7 or Word 97, 16MB RAM, and 33MB of free disk space.

**Blue Sky Software**  
7777 Fay Ave., Suite 201  
La Jolla, CA 92037  
**Phone:** (800) 459-2356 or (619) 459-6365  
**Fax:** (619) 459-6366  
**E-Mail:** info@blue-sky.com  
**Web Site:** <http://www.blue-sky.com>  
**Price:** US\$699 as a complete package; components can also be purchased separately. A preview version of RoboHELP 1.0 is free for downloading.

Gary Entsminger is a writer, programmer, and computing consultant. His books include *The Way of Delphi* [Prentice Hall, 1996], *The Way of Java* [Prentice Hall, 1997], and *The Tao of Objects, 2nd Ed.* [M&T Books, 1995]. He can be reached at gentsmin@together.net.



## Learning to Share

**T**he problem of shared code libraries has been around since the dawn of computing. The typical example used to illustrate why a shared code library should be used is the C run-time library. In this scenario, every time a programmer wrote a program in C, they used the C run-time library. Therefore, each application would have its own copy of the library compiled right into the executable.

One solution to this problem was the introduction of DLLs in Windows. No longer was it necessary to have multiple copies of the C run-time library lying around your disk, taking up precious disk space. Furthermore, running multiple applications simultaneously that relied on the run-time library wouldn't require the entire library to be loaded in memory multiple times. This ushered in a new era of computing, where third-party companies sell libraries to you, the programmer, so you don't have to program something that's already been done.

Lately, there has been much discussion about where applications should place shared DLLs. The two most accepted places for third-party DLLs are the `\Windows\System` directory or in a private directory. Due to paranoia in the software industry that a newer version of a DLL may introduce problems with existing applications, some recent publications have even gone so far as to recommend using private directories for all non-system DLLs. Each argument has its merits, but to help protect yourself from latent problems, you should place these files in the `\System` directory.

But what's "third party?" By the very definition of this phrase, we can define our argument. Is third-party supposed to be anything that isn't part of the core Windows operating system? If so, that means Microsoft Office, Visual C++ run-time libraries, Internet Explorer, and even Delphi run-time packages should all be in their own private directories. If we say that certain applications should be exempt from the third-party definition, then who will decide which

applications are considered third-party? On the other hand, if every application should put its DLLs in a private directory, then why use DLLs at all? The same thing could be accomplished by linking the code straight into the EXE.

Imagine the cry of outraged users because they don't have a central repository to ease the burden on their disk space, memory requirements, and system maintenance issues. We already have that feature: It's called the `\System` directory. To blame the strategy because some applications and third-party DLL vendors can't play by the rules seems to not recognize the central need for this strategy to exist. After all, even the good old DOS and UNIX machines presented users with configuration problems. Some of these problems even related to having improper and conflicting versions of software modules (e.g. TSRs and device drivers) loaded at the same time.

Lastly, consider this scenario: Vendor A writes an application that uses a fairly common third-party DLL. They subscribe to the theory that the DLL belongs in a private directory. Now you come along and write an application that uses the same DLL. No matter what strategy you believe in, there's already a very large chance for a defect to surface in your program. How? If the user that purchased both software packages runs Application A and then runs your application, you'll be using the copy of the DLL in Application A's directory! If your application relies on a newer version of the DLL than the one provided by Vendor A, Vendor A can

make your application crash. You might say, "This will never happen to me," but the truth is, it happens all the time.

We live, work, and play in a multi-tasking, multi-application operating system. As developers, we need to ensure the highest level of probability that our applications won't fail. As soon as you try to outsmart the way things are supposed to work, you'll find yourself engaged in an uphill battle that you'll eventually lose. It's far better to fully understand the scope of the problem — and how you might help alleviate it — rather than try to insulate yourself in a corner of the operating system.

If anything, this should be a call to action for:

- third-party developers to be responsible in releasing new versions that are 100 percent compatible with previous versions;
- installation application companies to find an effective work-around;
- Microsoft to devise and implement a new strategy that provides the benefits of DLLs without the current problems; and
- application developers to play by the rules. ▲

— Dan Miser

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



# Working with AnsiStrings



Working with strings is an unglamorous necessity. Fortunately, Delphi's support for strings has grown considerably since Delphi 1. In Delphi 1, you have Pascal-style strings (which are limited to 255 characters) or PChars, pointers to null-terminated strings (à la C). While the latter provide nearly unlimited length possibilities, they force us to be extremely mindful about allocating and de-allocating memory, performing pointer arithmetic, and other such *minutiae*.

Delphi 2 introduced the new long string type, or AnsiString, making our work considerably easier. AnsiStrings combine the best features of the traditional types: Like PChars, they're allocated dynamically, and have no 255-character limit; like Pascal strings, the compiler takes care of the memory-allocation details. You also have the convenience of coding with the familiar Pascal string routines. Still, you'll often find yourself writing quite a bit of code, including low-level routines. Now there's another way. With the HyperString library from EFD Systems, you can avoid having to write general-purpose string routines.

**HyperString library.** HyperString includes over 200 string-manipulation routines designed specifically for working with AnsiStrings. They're very fast because they're written mostly in assembler. The library includes routines for converting to, or from, various mathematical types, verifying string content and format, counting characters and tokens, padding/trimming, and searching (forward, reverse, case insensitive, wildcard, and "fuzzy").

Its editing functions include many you might expect (convert any portion of a string to upper, lower, or proper case) and others you wouldn't. Using the table-oriented functions, you can manipulate AnsiStrings in complex ways, such as replacing a whole series of words (tokens) from one table with corresponding words in another table. With tokens and delimiters, you can create delimited lists and lookup tables.

Combining these functions, you can build complex, string-oriented databases — databases stored as AnsiStrings.

**More than a string library.** Ostensibly, HyperString is a library of string routines. In truth, it's much more. Many of the routines are "string routines" only in that they use a string or two somewhere. The Miscellaneous Group includes routines to perform numeric comparisons and operations such as Min, Max, Mid, Compare (unsigned), Rounding, Bit operations, and more. The Math Group includes routines to convert numbers to, or from, Intel-packed Binary Coded Decimal format, and perform unsigned integer math. This reminds me more of a low-level, general-purpose library such as TurboPower's SysTools, a library with which HyperString compares favorably.

A snapshot of this library shows its versatility: It includes routines in 18 categories, including MIME (base64) encoding, encryption, compression, and Checksum/CRC calculations. There's also an extensive API group for working with the operating system. Included are functions for launching and terminating DOS and Windows applications, as well as using the system "tray," system shell, and more. HyperString also offers a unique implementation of dynamic numeric arrays (*TDynArray*) using dynamic strings as containers with 17 routines to manage them.

To find out for yourself, visit EFD Systems at <http://www.mindspring.com/~efd> (be sure to read the paper

on AnsiStrings). Download the compiled library (HSTR.ZIP — Delphi 3 .DCU format), which is available as "freeware." The full source code is available for US\$30. You can also download three sample programs I wrote to test the product (see end of article for details). They will give you an idea of the power, speed, and versatility of some of these routines. You'll discover a whole new dimension in working with strings. In 32-bit Windows programming, AnsiStrings are the way to go; the HyperString library will help you on that journey considerably. E-mail me if you'd like to read more about working with strings in Delphi, or this library in particular.  $\Delta$

— Alan C. Moore, Ph.D.

*The files referenced in this article are available on the Delphi Informant Works CD located in INFORM99MAYDI9805FN.*

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