

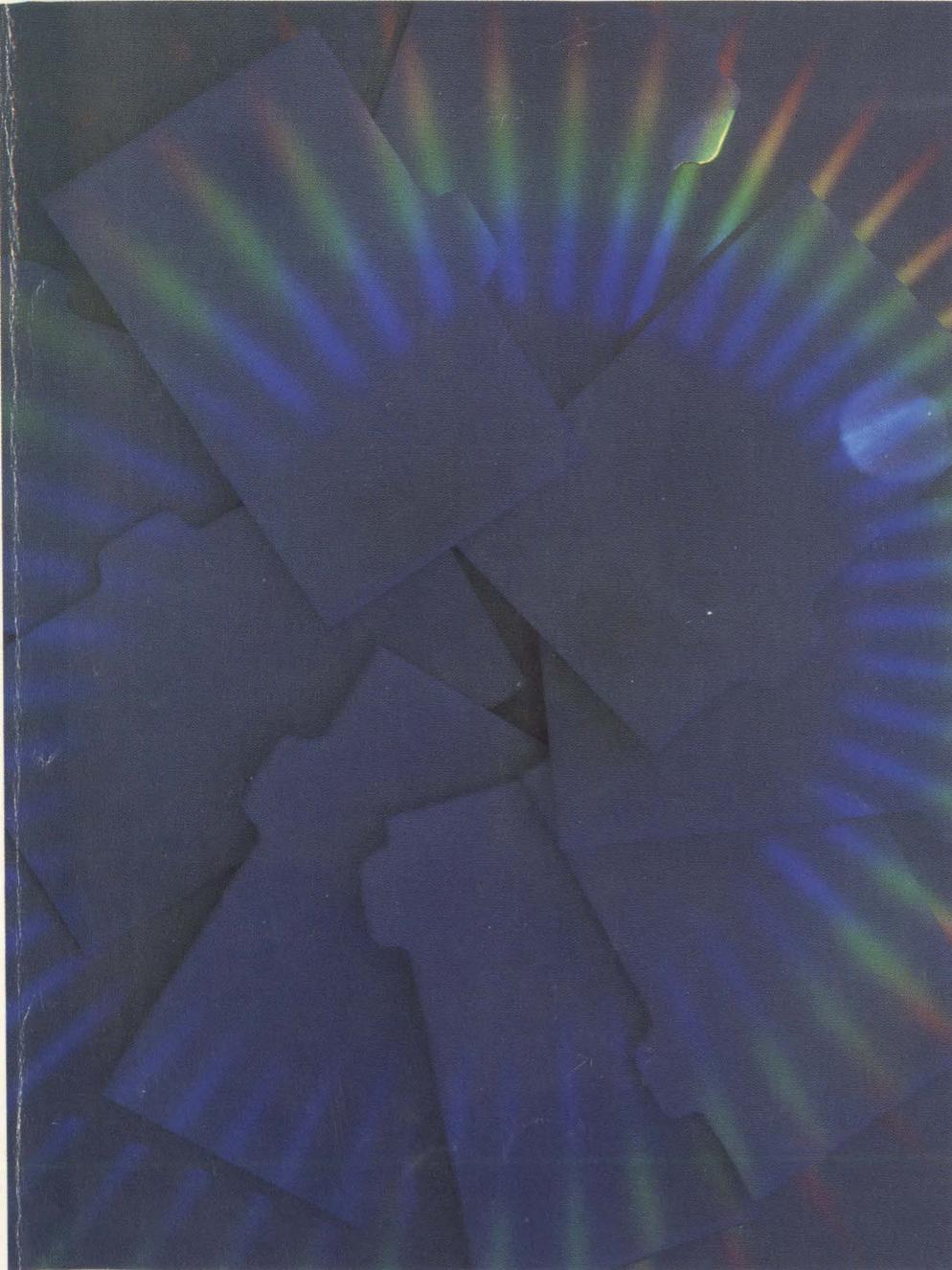
NEW!

SPRINT & TURBO PROLOG 2.0

**SEE MONEY-SAVING
OFFERS INSIDE!**

TURBO TECHNIX

THE BORLAND LANGUAGE JOURNAL • JULY/AUGUST 1988 • VOLUME ONE NUMBER FIVE • \$10.00



ONE DATABASE, MANY INDEXES

**Creating and using
multiple indexes
with Turbo Access**

**Turbo Pascal's custom
text file device drivers**

**Sprint macros
for time and date
formatting**

**Creating Turbo Prolog
applications
with mouse support**

Get your work done before 1991.

The future of personal computing is clear. More powerful PCs. Easier to use PCs. With graphics and character-based programs working side by side. Talking to each other. Multitasking. Windowing. Menuing. Mousing. Getting your work done easier and faster.

labels while you're writing a report in **Word Perfect**, or laying out a newsletter in **Ventura Publisher**, or designing a building in **AutoCAD**. DESQview even lets you transfer text, numbers, and fields of information between programs.

Have it all now.

DESQview™ is the operating environment that gives DOS the capabilities of OS/2.™ And it lets you, with your trusty 8088, 8086, 80286, or 80386 PC, leap to the productivity of the next generation. For not much money. And without throwing out your favorite software.

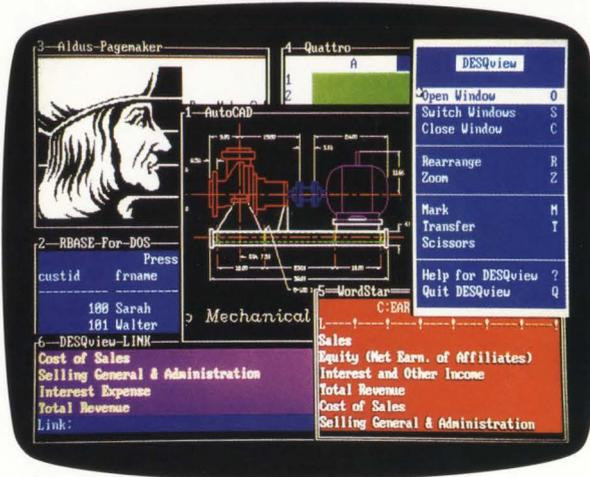
Add DESQview to your PC and it quickly finds your programs and lists them on menus. So you can just point to the program, using keyboard or mouse, to start it up. DESQview knows where that program lives. And what command loads it.

For those who have trouble remembering DOS commands, it adds menus to DOS. It even lets you sort your files and mark specific files to be copied, backed-up, or deleted—all without having to leave the program you're in.

Best of all, DESQview accomplishes all this with a substantial speed advantage over any alternative environment.

Multitask beyond 640K.

When you want to use several programs together, you don't have to leave your current program. Just open the next program. View your programs in windows or



For programmers, DESQview's API, with its strengths in inter-task communications and multitasking, brings a quick and easy way to adapt to the future. With the API's mailboxes and shared programs, programmers are able to design programs running on DOS with capabilities like those of OS/2.

full screen. Open more programs than you have memory for. And multitask them. In 640K. Or if you own a special EMS 4.0 or EEMS memory board, or a 386 PC, DESQview lets you break through the DOS 640K barrier for multitasking. If you have other non-EMS memory expansion products like AST's Advantage or the IBM® Memory Expansion Option, we have a solution for you, too. The ALL CHARGE-CARD™ 'unifies' all your memory to provide up to 16 megabytes of continuous workspace. DESQview lets you use this memory to enhance your productivity. You can start 1-2-3 calculating and tell Paradox to print mailing

Fulfill the 386 promise.

For 80836 PC users, DESQview becomes a 386 control program when used in conjunction with Quarterdeck's Expanded Memory Manager (QEMM)-386—giving faster multitasking as well as virtual windowing support.

And when you use DESQview on an IBM PS/2™ Model 50 or 60 with QEMM-50/60 and the IBM Memory Expansion Option, DESQview gives you multitasking beyond 640K.

Experts are voting for DESQview. And over a million users, too.

If all of this sounds like promises you've been hearing for future systems, then you can understand why over a million users have chosen DESQview. And why *PC Magazine* gave DESQview its Editor's Choice Award for "The Best Alternative to OS/2," why readers of *InfoWorld* twice voted DESQview "Product of the Year" why, by popular vote at Comdex Fall for two years in a row, DESQview was voted "Best PC Environment" in *PC Tech Journal's* Systems Builder Contest.



DESQview lets you have it all now.

DESQVIEW SYSTEM REQUIREMENTS:
IBM Personal Computer and 100% compatibles (with 8086, 8088, 80286, or 80386 processors) with monochrome or color display; IBM Personal System/2* Memory: 640K recommended; for DESQview itself 0-145K* Expanded Memory (Optional): expanded memory boards compatible with the Intel AboveBoard; enhanced expanded memory boards compatible with the AST RAMpage; EMS 4.0 expanded memory boards* Disk: two diskette drives or one diskette drive and a hard disk* Graphics Card (Optional): Hercules, IBM Color/ Graphics (CGA), IBM Enhanced Graphics (EGA), IBM Personal System/2 Advanced Graphics (VGA)* Mouse (Optional): Mouse Systems, Microsoft and compatibles* Modem for Auto-Dialer (Optional): Hayes or compatible* Operating System: PC-DOS 2.0-3.3; MS-DOS 2.0-3.2* Software: Most PC-DOS and MS-DOS application programs; programs specific to Microsoft Windows 1.03-2.03, GEM 1.1-3.0, IBM TopView 1.1* Media: DESQview 2.0 is available on either 5-1/4" or 3-1/2" floppy diskette.

YES!
I need increased productivity now!

Name _____
Address _____
City _____ State _____ Zip _____
Payment Method Visa MasterCard Expiration ____/____
Account # _____

Qty	Product	Format	Price Each	Totals
	DESQview 2.0	<input type="checkbox"/> 5-1/4 <input type="checkbox"/> 3-1/2	\$129.95	
	QEMM-386	<input type="checkbox"/> 5-1/4 <input type="checkbox"/> 3-1/2	\$59.95	
	QEMM-50/60	<input type="checkbox"/> 5-1/4 <input type="checkbox"/> 3-1/2	\$59.95	
	ALL CHARGE CARD (Special for DESQview owners)		\$200.00*	

Shipping & Handling \$5 in USA / \$10 outside USA
Calif Residents add 6.5%

Grand Total

Quarterdeck
150 Pico Boulevard, Santa Monica, CA 90405
(213) 392-9851

*This ALL CHARGE CARD is designed for the IBM PC AT and PS/2 50 and 60. If you have another type of 80286-based PC, there's a version for you, too. Please call 1-800-387-2744 for special ordering information. Offer expires August 31, 1988. Trademarks are property of their respective holders: IBM, OS/2, PS/2, 1-2-3, Paradox, Word Perfect, Ventura Publisher, AutoCAD, Intel, Above Board, AST, RAMpage, Advantage, Hercules, Mouse Systems, Hayes, Microsoft, Windows, TopView.

Make your programs millions of times smarter

More and more, programmers and workstation builders are using DESQview 2.0 as a development tool. The reason is simple. They can create powerful, multitasking solutions today for the millions of DOS PCs in use today. Solutions comparable to those promised for tomorrow by OS/2.

The API Advantage

Programmers who take advantage of DESQview's API (Application Program Interface) get access to the powerful capabilities built into DESQview—multitasking, windowing, intertask communications, mailboxes, shared programs, memory management, mousing, data transfer, menu-building and context sensitive help.

Bells and Whistles

A program taking advantage of the DESQview 2.0 API can spawn subtasks for performing background operations or new processes for loading and running other programs concurrently. It can schedule processing after an interval or at a certain time. It can use DESQview's intertask communications to rapidly exchange data between programs, share common code and data; or interrupt at critical events. It can use DESQview's menuing and mousing capabilities to create menus. And there's lots more it can do.

Some of the applications under development right now using DESQview 2.0 API Tools: CAD, Medical systems, insurance, 3270 mainframe communications, network management, real estate, typesetting, point of sale, education, commodity trading, stock trading and online voting.

80386 Power

80386 programmers can take advantage of the 80386's protected mode for large programs, yet run on DOS and multitask in DESQview—side by side with other 80386 and DOS programs. The breakthroughs that make this possible: DOS Extenders from PharLap Software and AI Architects and DESQview support of these DOS extenders.

DESQview Developer Conference

So if you are a developer, looking to create programs with mainframe capabilities, but wanting to sell into the existing base of millions of DOS PCs, come to Quarterdeck's first DESQview API Developers Conference, August 16-18, 1988 at the Marina Beach Hotel, in Marina del Rey, California. For more information call or write us.

Come learn about the DESQview 2.0 API and 80386 DOS Extenders. Meet 80386 experts as well as those smart people who are creating DESQview 2.0 API workstations solutions.

And if you want to get a leg up before the conference, ask us about the DESQview API Tools for assembler or C programmers.

Bringing New Power to DOS. DESQview 2.0 API Toolkit.

The logo for Quarterdeck, featuring the word "Quarterdeck" in a bold, red, sans-serif font. To the left of the text is a stylized graphic consisting of several parallel, slanted lines of varying lengths, creating a sense of motion or a wing-like shape.

Quarterdeck Office Systems 150 Pico Blvd., Santa Monica, CA 90405
(213) 392 9851

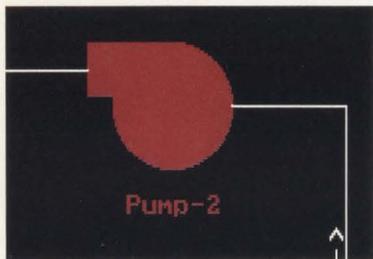
TURBO TECHNIX

The Borland Language Journal
July/August 1988
Volume 1 Number 5

FEATURES

TURBO PASCAL

- 12 Multiple Indexes with Turbo Access
William Meacham
- 27 Catch and Throw with Turbo Pascal
Jon Shemitz
- 30 Recursing Without Cursing
Jeff Duntemann
- 34 Custom Text File Device Drivers
Neil Rubenking



86 Turbo Prolog 2.0 takes full advantage of the Borland Graphics Interface (BGI) for device-independent graphics displays. Expert systems can now show as well as tell what they know.

TURBO C

- 42 Mouse Mysteries, Part II: Graphics
Kent Porter
- 54 Formatting Output in Turbo C
Peter Aitken
- 60 Allocating Full 64K Blocks in Turbo C
Michael Abrash
- 61 Worth the Wait
Jonathan Sachs

TURBO PROLOG

- 70 Certainty Factors in Turbo Prolog
Tom Castle
- 76 Failing with Grace
Edward B. Flowers
- 86 In Graphic Harmony
Alex Lane
- 92 Logic and Turbo Prolog
Alex Lane
- 98 Cat and Mouse in Turbo Prolog, Part II
Safaa H. Hashim

42

Mice can work well with text applications, but mice were *created* to steer a graphics cursor around your screen. The cursor can be a miniature icon reflecting the work currently being done, with one single pixel in the cursor (called the "hot spot") empowered to say, *You are here.*



110

Moving a binary file across a 7-bit communication channel can be done by encoding the binary file as a series of printable **DATA** statements within an equally printable Turbo Basic program. The program, when run on a computer at its destination, re-creates the original binary file.

TURBO BASIC

- 110 Binary to Text for Communications
Robert E. Stearns, Jr.
- 114 Viewports in Turbo Basic
Peter Aitken
- 120 Calling BIOS Services from Turbo Basic
Ethan Winer

BUSINESS LANGUAGES

- 122 Date Formatting with Sprint
Neil Rubenking
- 130 Bounce and Choose in PAL
Alan Zenreich

TURBO TECHNIX makes reasonable efforts to assure the accuracy of articles and information published in the magazine. *TURBO TECHNIX* assumes no responsibility, however, for damages due to errors or omissions, and specifically disclaims any implied warranty of merchantability or fitness for a particular purpose. The liability, if any, of Borland, *TURBO TECHNIX*, or any of the contributing authors of *TURBO TECHNIX*, for damages relating to any error or omission shall be limited to the price of a one-year subscription to the magazine and shall in no event include incidental, special, or consequential damages of any kind, even if Borland or a contributing author has been advised of the likelihood of such damages occurring.

Trademarks: *Turbo Pascal*, *Turbo Basic*, *Turbo C*, *Turbo Prolog*, *Turbo Toolbox*, *Turbo Tutor*, *Turbo GameWorks*, *Turbo Lightning*, *Lightning Word Wizard*, *SideKick*, *SuperKey*, *Eureka*, *Reflex*, *Quattro*, *Sprint*, *Paradox*, and *Borland* are trademarks or registered trademarks of Borland International, Inc. or its subsidiaries.



Sprint, the Professional Word Processor, is now shipping. In keeping with Borland's evolving open architecture philosophy, the engine at the heart of the product is available to programmers through a powerful text-processing programming language. Sprint's macro language is far more than recorded keystrokes—the language can make DOS and BIOS calls, create custom pop-up menus, treat entire documents as variables, and read or write any memory location or I/O port, all within program structures familiar from C, Pascal, and BASIC. See page 122 for an example of Sprint's macro power, and page 125 for a special offer that could add Sprint to your programming arsenal for less than you think.

COLUMNS

- 4 BEGIN: Naming the Animals
Jeff Duntemann
- 132 Binary Engineering: Designing Data Structures, Part I
Bruce F. Webster
- 138 Language Connections: Turbo Prolog to Turbo C is Now a Two-Way Bridge
Gary Entsminger
- 145 Tales from the Runtime: Organization and Optimization
Mark L. Van Name and Bill Catchings
- 160 Philippe's Turbo Talk

DEPARTMENTS

- 6 Dialog
- 150 Archimedes' Notebook: Rocketry With Eureka
David Eagle
- 153 Critique: Desktop for Paradox
Alan Zenreich
- 154 Critique: Peabody 1.02
Peter Aitken
- 155 BookCase: *Turbo C: The Art of Advanced Program Design, Optimization, and Debugging*
Reviewed by Marty Franz
- 156 BookCase: *Turbo Basic Programs for Scientists & Engineers*
Reviewed by Peter Aitken
- 157 Turbo Resources

Cover: *Sorting a database can be done on only one field at a time, and puts your data files at risk. Indexing a database can be done on any number of fields, and involves little or no risk to the database files themselves. Turbo Access (from the Turbo Pascal Database Toolbox) provides a fast interface to your database files through as many indexes as you care to create. Photography by Bradley Ream.*

TURBO TECHNIX

Publisher
John Hemsath
Editor in Chief
Jeff Duntemann

EDITORIAL

Managing Editor
Michael Tighe
Technical Editor
Michael A. Floyd
Copy Editor
Pamela Dillehay
Editorial Assistant
Sheriann Glass

Technical Consultants
Brad Silverberg
David Intersimone
Adam Bosworth
Paul Chui
Lee Cantey
David Golden
Peter Iaria

DESIGN & PRODUCTION

Art Director
Karen Miner
Production Assistant
Annette Fullerton
Typesetting Manager
Walter Stauss
Typesetter/System Supervisor
Jeffrey Schwerley
Typesetters
Ron Foster
Jeanie Maceri
Typesetting Traffic
Charlene McCormick
Photographer
Bradley Ream

ADMINISTRATION

Purchasing
Brad Asmus

ADVERTISING SALES OFFICES

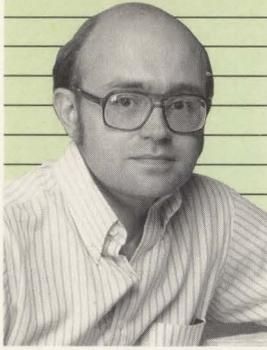
Home Office
(408) 438-9321
Western Office
Janet Zamucen
(714) 858-0408
New England Office
Mid-Atlantic Office
Merrie Lynch
Nancy Wood
(617) 848-9306
Southeastern Office
Megan Patti
(813) 394-4963

TURBO TECHNIX (ISSN-0893-827X) is published bimonthly by Borland Communications, a division of Borland International, Inc., 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001. *TURBO TECHNIX* is a trademark of Borland International, Inc. Entire contents Copyright ©1988 Borland International, Inc. All rights reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the publisher. For a statement of our permission policy for use of listings appearing in the magazine, send a self-addressed stamped envelope to Permissions, *TURBO TECHNIX*, 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001. Editorial and business offices: *TURBO TECHNIX*, 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001. Subscription rate is \$49.95 per year; rate in Canada \$60.00 per year, payable in U.S. funds. Single copy price is \$10.00. For subscription service write to Subscriber Services, *TURBO TECHNIX*, 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001.

BEGIN

Naming the animals

Jeff Duntemann



The story goes that God created the animals, and Adam named them—not to call them, as in Ralph or Chewy or Spike, but to *classify* them. It was a process to separate the eaters from the eaten, the pet from the pot. A name implies a method of dealing with the named entity. If you get the name wrong, your assumptions may never catch up with reality, and you may end up (at best) hunting mosquitoes with a howitzer, or (at worst) hunting tigers with a flyswatter.

One very fine example of this kind of trap involves laser printers versus the more traditional dot-matrix and daisy-wheel printers. We call them both “printers,” but the two groups have a fundamental difference: Traditional printers move a piece of paper past a printhead, printing as the paper goes by. Laser printers build a virtual image in their buffers, in a sense passing a virtual printhead over the entire sheet at random before finally “developing” the image by sending the paper through the xerographic engine. This difference intrudes when you try to mask variations among printers behind a suite of generalized printer routines. A **PrinterXY** routine that moves the printhead to a given X,Y position on the sheet is nothing more than an escape sequence to a laser printer. On the other hand, the paper in a matrix printer can't be backed up to position 0,0 once it's been indexed downward. Implementing **Print-**

erXY on a matrix printer means allocating and writing to a “virtual page” in memory somewhere, and not actually moving the physical printhead over the paper until the virtual image is complete and a form feed is sent to the driver. Quite apart from finding enough memory for graphics imaging—brother, can you spare me a megabyte?—this is a lot like gluing fangs on a hamster and calling it a lion. The hamster people will come to expect too much, and the lion people will *not* be fooled.

The correct way to deal programmatically with laser printers, as far as I'm concerned, is to call them what they really are: plotters. Plotters, like laser printers, can write to any location on the sheet at random, creating both text and graphics interchangeably on command. Laser printers are primarily character devices, but I have yet to see one that couldn't generate a full page of graphics at some resolution, even if not at 300 dpi. Most plotter command sets (such as the ubiquitous HPGL) include commands for printing text at any arbitrary size. Creating a common API for laser printers and plotters requires little more than a translate table for each device, with no need for jury-rigging virtual pages in scarce system memory.

It's far too late to get people to call laser printers “plotters” out in the real world. Within your programs, however, you can call anything by any name you want. Back in the software realm, text files are character-oriented sequential devices, and binary files are block-oriented random-access devices. Calling both of them

“files” is confusing and limiting. For clarity's sake, it might make sense to build a suite of block-oriented storage routines that do not rely on file jargon—with the added benefit that storage could be accessed identically whether it were located on a disk, on the heap, or on some I/O mapped peripheral. Just because it acts like a file doesn't mean it has to be called a file.

Consider this an opportunity to rethink the nature of every element of your program design, along with every machine component your program must interact with. The Game Control Adapter is a joystick socket—says so right on the label—yet it monitors four switches and measures four resistance values simultaneously. With sensors to monitor temperature, humidity, and solar flux, and a four-bit shaft encoder attached to a wind vane, your joystick port becomes a weather station interface.

All too often, we accept a name at face value without truly understanding what sort of creature lives beneath the name. Names evolve haphazardly, but the logic and sense in a program cannot. Consider each programming project a new Eden, where *you* get to name the animals. If it walks like a duck, it may be a duck—or it may be a hamster trying to live up to the name Donald. ■

Opinions expressed in this column are those of the editor and do not necessarily reflect the views of Borland International, Inc.

Blaise Passes the Screen Test.

POWER SCREEN

Best performance in a supporting role.

Because your time is more valuable than ever, Blaise Computing presents POWER SCREEN™, the new high performance screen management system designed to support your own creative programming efforts.

POWER SCREEN provides reliable, lightning fast data entry screens and menus to create your own sophisticated window oriented applications. It allows you to design screens exactly as you want them to appear in your final application. Screens are efficiently stored in a file so they can be used by your application or later modified without program code changes.

PAINT, the screen painter included with POWER SCREEN, has the appearance and performance of the popular integrated programming language environments. It lets you design and modify screens, and define and format fields. All VGA, EGA and monochrome text modes, attributes and colors are supported.

The POWER SCREEN Runtime Library allows you to construct screens in memory, display screens in windows and read and write data to fields within the screen. All screens and menus are window-oriented, so they can be stacked, removed or moved on the physical screen. You can access screens field-by-field or a whole screen at a time. POWER SCREEN takes care of field input editing, data and range checking, and data formatting.

POWER SCREEN out-performs the runners-up with a dazzling display of capabilities FEATURING:

- ◆ **Virtual screens.** Screens that can be larger than the physical screen, with just a portion of the screen displayed within a window. Write to any screen any time, even if it is not visible. Automatic physical screen update.
- ◆ **Context sensitive help.** Create help text on a field-by-field basis or for the entire screen with a window-oriented help facility.
- ◆ **Intervention routines.** Install them so your application gains control when a field is entered, exited and between keystrokes.
- ◆ **Range checking.** Supported for all standard data types.
- ◆ **Unlimited screens.** Subject only to the amount of available memory.
- ◆ **Definable keys.** Fully configurable field editing keys.

POWER SCREEN includes PAINT, the POWER SCREEN Runtime Library, as well as other utilities for creating help files and maintaining and documenting your screen database files. Language interfaces with source code are included

for C, Turbo Pascal 4.0 and QuickBASIC.

The package is accompanied by a fully-indexed comprehensive User Reference describing POWER SCREEN procedures and utilities. Complete example programs are supplied on the diskettes.

POWER SCREEN requires an IBM PC, XT, AT, PS/2 or close compatible and DOS 2.00 or later. To write POWER SCREEN applications, you need one of the supported compilers: Turbo C, Microsoft C (4.00 or later), QuickC, Turbo Pascal (4.0 or later), QuickBASIC (4.0 or later). Interfaces for all supported compilers are included with POWER SCREEN.

Blaise Computing: We've passed the screen test so you won't have to.

Complete price: \$129.

Blaise Computing has a full line of support products for both Pascal and C. Call today for your free information packet.

BLAISE COMPUTING INC.

2560 Ninth Street, Suite 316 Berkeley, CA 94710 (415) 540-5441

THE BLAISE M E N U

Turbo POWER TOOLS PLUS \$129.00
Screen, window, and menu management including EGA and VGA support; DOS memory control; ISRs; scheduled intervention code; and much more. For Turbo Pascal.

Turbo ASYNCH PLUS \$129.00
Interrupt driven support for up to four COM ports. I/O buffers up to 64K; XON/XOFF; hardware handshaking; up to 19.2K baud; modem control and XMODEM file transfer. For Turbo Pascal.

C TOOLS PLUS \$129.00
Windows; menus; ISRs; intervention code; screen handling and EGA 43-line text mode support; direct screen access; DOS file handling and more. Specifically designed for Microsoft C 5.0 and QuickC.

C ASYNCH MANAGER \$175.00
Full featured interrupt driven support for up to four COM ports. I/O buffers up to 64K; XON/XOFF; hardware handshaking; up to 19.2K baud; modem control and XMODEM file transfer. For Microsoft C and Turbo C.

Turbo C TOOLS \$129.00
Full spectrum of general service utility functions including: windows; menus; memory resident applications; interrupt service routines; intervention code; and direct video access for fast screen handling. For Turbo C.

KeyPilot \$49.95
"Super-batch" program. Create batch files which can invoke programs and provide input to them; run any program unattended; create demonstration programs; analyze keyboard usage.

EXEC \$95.00
Program chaining executive. Chain one program from another in different languages; specify common data areas; less than 2K of overhead.

RUNOFF \$49.95
Text formatter for all programmers. Written in Turbo Pascal; flexible printer control; user-defined variables; index generation; and a general macro facility.

**TO ORDER CALL TOLL FREE
800-333-8087**

NOW

THE SEARCH IS OVER! Peter Norton's Online Guides Instant Access Program eliminates most manual searches with a few simple keystrokes. Now, when you order our featured product, we'll send you its Online Database along with Peter Norton's Instant Access Program, absolutely free!

Microsoft C, QuickC and QuickBASIC are registered trademarks of Microsoft Corporation. Turbo C and Turbo Pascal are registered trademarks of Borland International.

DIALOG

**GOTO jail; do not pass the ROM BIOS;
do not collect $2^{(n-1)}$ multiplies.**

Are we glowing in the dark, or is the smoke pouring out of your ears? Errata or accolade? Bug or feature? Let us and your fellow readers know what's on your mind, and our editorial staff and authors will respond as best they can.

Address letters to:

DIALOG
TURBO TECHNIX Magazine
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066

Letters become the property of TURBO TECHNIX and cannot be returned. We cannot answer all letters individually, but we will try to print a representative sampling of mail received.

SAVE SOME MULTIPLIES

I was reading over the Bezier curve routines in Kent Porter's article, "Curves, Bezier-Style" (March/April, 1988), and thought you might be interested in a slightly more efficient binomial coefficient function. Your method, although more straightforward, requires $2^{(n-1)}$ multiplies and 1 divide. If you recognize that

$$\frac{n!}{i!(n-i)!} = \frac{n(n-1)(n-2)\dots(n-i+1)}{i!}$$

then you can rewrite the C function given in the article as:

```
FUNCTION C(n,i : Integer):Integer;
```

```
VAR  
  num,den,j : Integer;
```

```
BEGIN  
  num := 1;  
  den := 1;  
  FOR j := 2 TO i DO  
    BEGIN  
      num := num * (n-j+1);  
      den := den * j  
    END;  
  c := num DIV den  
END;
```

This method only requires $2^{(i-1)}$ multiplies and 1 divide, thus saving $2^{(n-i)}$ multiplies. It would certainly be difficult to find fault with such a well-written, informative article, but in graphics, speed is everything!

—Paul Cifarelli
Forest Hills, NY

Hey, I'll save a multiply anywhere I can, and I haven't had the opportunity to save $2^{(n-i)}$ of them in quite a while. Thanks for the tip—every cycle, however spent, makes a difference in plotting curves of any stripe.

—Jeff Duntemann

GOTO, GOTO, GONE

I enjoyed Bruce Webster's Binary Engineering column "Go to, go to" (January/February, 1988), but I must challenge his claim that one cannot prematurely exit a **FOR** loop in Turbo Pascal without use of a **GOTO**. His example was:

```
FOR I := 1 TO 50 DO  
  BEGIN  
    . . .  
    IF PanicButton THEN GOTO 100;  
    . . .  
  END;  
100: Writeln('Loop finished!');
```

This can be revised to avoid the use of **GOTO** by placing the **FOR** loop in its own procedure:

```
PROCEDURE Loop;
```

```
VAR I : Integer;
```

```
BEGIN  
  FOR I := 1 TO 50 DO  
    BEGIN  
      . . .  
      IF PanicButton THEN Exit;  
      . . .  
    END  
  END;
```

```
. . .  
Loop;  
Writeln('Loop finished!');
```

While I consider this to be another example of the "Tastes Great/Less Filling" controversy, the fact remains that the **GOTO** was not really necessary!

—P. Kenneth Morse
Augusta, GA

Your example is accurate but is one of those cases of the cure possibly being worse than the disease. Yes, there are times when it would be valid to transform a given **FOR**
continued on page 8

Interlocking Pieces: Blaise and Turbo Pascal.

Whether you're a Turbo Pascal expert or a novice, you can benefit from using professional tools to enhance your programs. With Turbo POWER TOOLS PLUS™ and Turbo ASYNCH PLUS™, Blaise Computing offers you all the right pieces to solve your 4.0 development puzzle.

Compiled units (TPU files) are provided so each package is ready to use with Turbo Pascal 4.0. Both POWER TOOLS PLUS and ASYNCH PLUS use units in a clear, consistent and effective way. If you are familiar with units, you will appreciate the organization. If you are just getting started, you will find the approach an illustration of how to construct and use units.

◆ **POWER TOOLS PLUS** is a library of over 180 powerful functions and procedures like fast direct video access, general screen handling including multiple monitors, VGA and EGA 50-line and 43-line text mode, and full keyboard support, including the 101/102-key keyboard. Stackable and removable windows with optional borders, titles and cursor memory provide complete windowing capabilities. Horizontal, vertical, grid and Lotus-style menus can be easily incorporated into your programs using the menu management routines. You can create the same kind of moving pull down menus that Turbo Pascal 4.0 uses.

Control DOS memory allocation. Alter the Turbo Pascal heap size when your program executes. Execute any program from within your program and POWER TOOLS PLUS automatically compresses your heap memory if necessary. You can even force the output of the program into a window!

Write general interrupt service routines for either hardware or software interrupts. Blaise Computing's unique intervention code lets you develop memory resident (TSRs) applications that take full advantage of DOS capabilities. With simple procedure calls, "schedule" a Turbo Pascal procedure to execute either when pressing a "hot key" or at a specified time.

◆ **ASYNCH PLUS** provides the crucial core of hardware interrupts needed to support asynchronous data communications. This package offers simultaneous buffered input and output to both COM ports, and up to four ports on PS/2 systems. Speeds to 19.2K baud, XON/XOFF protocol, hardware handshaking, XMODEM (with CRC) file transfer and modem control are all supported. ASYNCH PLUS provides text file device drivers so you can use standard "Readln" and "Writeln" calls and still exploit interrupt-driven communication.

The underlying functions of ASYNCH PLUS are carefully crafted in assembler and drive the hardware directly. Link these functions directly to your application or install them as memory resident.

Blaise Computing products include all source code that is efficiently crafted, readable and easy to modify. Accompanying each package is an indexed manual describing each procedure and function in detail with example code fragments. Many complete examples and useful utilities are included on the diskettes. The documentation, examples and source code reflect the attention to detail and commitment to technical support that have distinguished Blaise Computing over the years.

Designed explicitly for Turbo Pascal 4.0, Turbo POWER TOOLS PLUS and Turbo ASYNCH PLUS provide reliable, fast, professional routines—the right combination of pieces to put your Turbo Pascal puzzle together. **Complete price is \$129.00 each.**

BLAISE COMPUTING INC.

2560 Ninth Street, Suite 316 Berkeley, CA 94710 (415) 540-5441

Now, for
Turbo Pascal 4.0!

THE BLAISE
M E N U

Turbo POWER SCREEN \$129.00
NEW! General screen management; paint screens; block mode data entry or field-by-field control with instant screen access. Now for Turbo Pascal 4.0, soon for C and BASIC.

Turbo C TOOLS \$129.00
Full spectrum of general service utility functions including: windows; menus; memory resident applications; interrupt service routines; intervention code; and direct video access for fast screen handling. For Turbo C.

C TOOLS PLUS \$129.00
Windows; menus; ISRs; intervention code; screen handling and EGA 43-line text mode support; direct screen access; DOS file handling and more. Specifically designed for Microsoft C 5.0 and QuickC.

ASYNCH MANAGER \$175.00
Full featured interrupt driven support for the COM ports. I/O buffers up to 64K; XON/XOFF; up to 9600 baud; modem control and XMODEM file transfer. For Microsoft C and Turbo C or MS Pascal.

PASCAL TOOLS/TOOLS 2 \$175.00
Expanded string and screen handling; graphics routines; memory management; general program control; DOS file support and more. For MS-Pascal.

KeyPilot \$49.95
"Super-batch" program. Create batch files which can invoke programs and provide input to them; run any program unattended; create demonstration programs; analyze keyboard usage.

EXEC \$95.00
NEW VERSION! Program chaining executive. Chain one program from another in different languages; specify common data areas; less than 2K of overhead.

RUNOFF \$49.95
Text formatter for all programmers. Written in Turbo Pascal; flexible printer control; user-defined variables; index generation; and a general macro facility.

TO ORDER CALL TOLL FREE
800-333-8087

TELEX NUMBER - 338139

NOW

THE SEARCH IS OVER! Peter Norton's Online Guides Instant Access Program eliminates most manual searches with a few simple keystrokes. Now, when you order our featured product, we'll send you its Online Database along with Peter Norton's Instant Access Program, absolutely free!

Microsoft
and QuickC are
registered trademarks of
Microsoft Corporation. Turbo Pascal is a reg-
istered trademark of Borland International.

DIALOG

continued from page 6

loop into a separate procedure (particularly if it were a large **FOR** loop with several exit points), but to do so merely to avoid using a **GOTO** statement in any **FOR** loop could make the program less clear, not more.

—Bruce Webster

BIOS, THE MISUNDERSTOOD

Bearing in mind your January/February editorial comment that “we” know “exactly” what DOS can do, I am quite uneasy about the technique used in the Turbo Pascal article “Replacing the Keyboard Interrupt,” by Neil Rubenking. The keyboard interrupt is a *hardware* interrupt, and so may occur at any time, including on an internal stack, on top of, and/or underneath multiple other hardware interrupts on that same stack.

Since DOS has limited-size internal stacks, and the processor has no hardware stack-limit checks, it is not hard to imagine that, under possibly rare but still normal circumstances, the Turbo Pascal 4.0 interrupt procedure overhead (that of pushing all registers onto the current stack) may exceed the bounds of a DOS internal stack. This will not cause an immediate crash, but rather a probable crash waiting to happen, which may occur when the overwritten code or data are used. Turbo Pascal interrupt procedures are somewhat more appropriate with *software* interrupts, since an application stack will then be in control. The timer (INT 1CH) is also a hardware interrupt, of course.

As far as I know, this sort of problem has been well understood for years, and indeed, has been properly handled in various examples of public domain **INLINE** code for Turbo Pascal 3.0; it is thus disappointing to see such an outdated and dangerous technique appear on your pages. Nevertheless, your editorial made an interesting point, even with our limited understanding of DOS.

—Terry Ritter
Austin, TX

There's some truth in the caution that a replacement ISR should not put anything onto the stack that the ROM BIOS ISR doesn't put there, and Neil's routines do push a couple of extra registers onto the stack, which, as you say, could be a DOS stack. However, as Lane Ferris reminded me, there can never be more than one INT 9 stack frame on the stack at any one time, because INT 9 disables interrupts while it executes, and doesn't enable them until it goes home. The extra burden is thus never more than a handful of bytes, and is unlikely to cause any problems. INT \$1C is, in fact, a software interrupt, called from the INT 8 hardware interrupt ISR. However, because it's called from a hardware ISR, the INT \$1C stack frame can indeed be placed on the DOS stack. We'll be having some articles from Lane on the problems of reentrancy, DOS stacks, and TSRs in the future; we hope this material will shed some light on this very messy issue.

—Jeff Duntemann

TP OR TC, IT'S ALL THE SAME TO ME

I thought your readers might be interested in this curiosity—a program that compiles and runs identically under both Turbo Pascal and Turbo C. (A warning is generated by Turbo C, but can be ignored.)

```
const (* Zelkop ) = 100;
main ()
{
  printf ( "Hello, world.\n" );
}
/*) Zelkop = 100;
begin
  writeln ( 'Hello, world.' );
end.
*/
```

The trick is to mix the comment delimiters. Turbo Pascal ignores everything between (* and *), while Turbo C ignores everything between /* and */. The **const** may seem to serve no purpose, but it is essential, since **const** is the only token that can legally begin both a Turbo Pascal and Turbo C program.

I don't believe such a program can be written in standard Pascal as defined by Wirth, or standard C as defined by Kernighan and Ritchie. To compile with Standard Pascal, it would have to begin with either the word **program**, or an opening comment delimiter, (*, but I can't see any way to write a C program that fits these criteria.

My thanks to Neil Rubenking for originally setting me this challenge.

—David Dubois
Halifax, Nova Scotia
CANADA

At last, something for the Turbo hacker who can't make up his mind.

—Jeff Duntemann

RENDER UNTO THE WIZARDS ...

I have enjoyed my first issues of *TURBO TECHNIX* immensely. I also find them very instructive and appreciate the three-tiered approach to writing articles. Not everyone is equally proficient at different languages. I, for example, am a BASIC programmer from the bad old days of eight-bit machines, and am able to appreciate and utilize articles at the Wizard level. However, I am a much more recent C user and the Programmer level is about as much as I can absorb without cerebral overload. Keep up the good work.

There is a slight correction to be made to Bruce Tonkin's article “Converting .COM Files to \$INCLUDE files” (January/February, 1988). The error only occurs if the last byte of the file corresponds to the start of a new line. When this happens, the last byte is just attached to the end of the previous line, like so:

```
$INLINE &H1, &H2, &H3, &H4, &$H5&H6
```

The generated code should, in fact, look like this:

```
$INLINE &H1, &H2, &H3, &H4, &H5
$INLINE &H6
```

To correct this problem, replace these lines in COM2INC.BAS:

continued on page 10

ORDERING INFORMATION

FREE SHIPPING. Orders within the USA (including Alaska & Hawaii) are shipped FREE via UPS. Call for express shipping rates.

NO CREDIT CARD CHARGE. VISA, MasterCard and Discover Card are accepted at no extra cost. Your card is charged when your order is shipped. Mail orders please include expiration date and authorized signature.

NO COD OR PO FEE. CODs and Purchase Orders are accepted at no extra cost. No personal checks are accepted on COD orders. POs with net 30-day terms (with initial minimum order of \$100) are available to qualified US accounts only.

NO SALES TAX. Orders outside of Ohio are not charged sales tax. Ohio customers please add 5% Ohio tax or provide proof of tax-exemption.

30-DAY GUARANTEE. Most of our products come with a 30-day documentation evaluation period or a 30-day return guarantee. Please note that some manufacturers restrict us from offering guarantees on their products. Call for more information.

SOUND ADVICE. Our knowledgeable technical staff can answer technical questions, assist in comparing products and send you detailed product information tailored to your needs.

INTERNATIONAL ORDERS. Shipping charges for International and Canadian orders are based on the shipping carrier's standard rate. Since rates vary between carriers, please call or write for the exact cost. International orders (except Canada), please include an additional \$20 for export preparation. All payments must be made with US funds drawn on a US bank. Please include your telephone number when ordering by mail. Due to government regulations, we cannot ship to all countries.

MAIL ORDERS. Please include your telephone number on all mail orders. Be sure to specify computer, operating system, diskette size, and any applicable compiler or hardware interface(s). Send mail orders to:

**Programmer's Connection
Order Processing Department
7249 Whipple Ave NW
North Canton, OH 44720**

Norton Guides Specify Language	100	65
For OS/2	New	150
Norton Utilities	150	109
Norton Utilities	150	59
quinn-curtis products		
DOS/BIOS & Mouse Tools for Turbo Pascal	75	69
MetaByte Data Acquisition Tools	100	89
Science & Engineering Tools	75	69

C-terp for Turbo C by Gimpel Software

List \$139 Ours \$119

C-terp is an interpreter/semi-compiler that serves as a powerful, professional C debugging and development environment. It features: full K&R C support with ANSI extensions; a full-screen, built-in, reconfigurable editor; fast semi-compilation and linking; complete multiple module support; 8087 support; full graphics support including dual displays; and much more.

software bottling products

Flash-up	89	79
Flash-up Developer's Toolbox	49	47
Screen Sculptor Supports Turbo Pascal	125	109
SoftCode Supports Borland Languages	New	195
Speed Screen	35	34

turbo pascal utilities

Btrieve ISAM File Mgr by Novell	245	184
Overlay Manager by TurboPower Software	New	45
TDEBUG 4.0 by TurboPower Software	45	43
Turbo Analyst by TurboPower Software	New	75
Turbo Professional 4.0 TurboPower	New Version	99
TurboHALD by IMSI, Specify Turbo C or Pascal	95	75
TurboRef by Gracon Services	50	45

other products

Brief by Solution Systems	195	CALL
Dan Bricklin's Demo II by Software Garden	195	179
Dan Bricklin's Demo Pgm by Software Garden	75	57
Dan Bricklin's Demo Tutorial by Software Garden	50	47
OPT-Tech Sort by Opt-Tech Data Proc.	149	99
risC Assembly Language by IMSI	80	65

CALL for Products Not Listed Here

KNOWLEDGE IS POWER

We're Programmer's Connection, the leading independent dealer of quality programmer's development tools for IBM personal computers and compatibles. We can give you the knowledge to help you make the best software buying decisions possible.

Informative Buyer's Guide. The CONNECTION, our comprehensive buyers guide and catalog, contains prices and up-to-date descriptions of over 750 programmer's development tools by over 250 manufacturers. Each description covers major product features as well as special requirements, version numbers, diskette sizes, guarantees, and more. In addition, the CONNECTION features interesting articles by leaders in the programming industry.

How to Get Your FREE Copy: 1) Mail us a card or letter with your name and address; or 2) Call one of our convenient toll free telephone numbers.

If you haven't yet received your copy of the Programmer's Connection Buyer's Guide, act now. Increasing your knowledge about these products could be one of the most powerful things you'll ever do.

USA 800-336-1166

Canada 800-225-1166
Ohio & Alaska (Collect) 216-494-3781
International 216-494-3781
TELEX 9102406879
FAX 216-494-5260

Business Hours: 8:30 AM to 8:00 PM EST Monday through Friday
Prices, Terms and Conditions are subject to change.
Copyright 1988 Programmer's Connection Incorporated



Established 1984

386 products

	List	Ours
386 AMS/386 LINK by Phar Lap Software	New	495 389
386 DEBUGGER by Phar Lap Software	New	195 145
FoxBASE + /386 by Fox Software	New	595 399
Microsoft Windows 386 by Microsoft		195 129
NDP C-386 by Microway	New	595 529
NDP FORTRAN-386 by Microway	New	595 529
Paradox 386 by Ansa/Borland	New	895 639

blaise products

ASYNCH MANAGER Supports Turbo C	175	135
C TOOLS PLUS/5.0	129	99
Turbo ASYNCH PLUS/4.0	129	99
Turbo C TOOLS	129	99
Turbo POWER SCREEN	New	129 99
Turbo POWER TOOLS PLUS/4.0	129	99

Peabody Pop-Up Reference Utility by Copia International

List \$100 Ours \$89

Peabody is a fast and flexible on-line reference utility with databases available for Turbo Pascal v3 & 4, Turbo C, Microsoft C v5, MS Assembler, or MS DOS. It provides instant, accurate and complete language information in pop-up frames at the touch of a key. With Peabody, you can select general topics from a structured subject menu, or use Peabody's hyperkey to get instant help for the keyword closest to the cursor. Specify database desired. Additional databases are available for \$45.

borland products

EUREKA Equation Solver	167	115
Paradox 1.1 by Ansa/Borland	495	359
Paradox 2.0 by Ansa/Borland	725	525
Paradox 386 by Ansa/Borland	New	895 639
Paradox Network Pack by Ansa/Borland	995	725
Quattro: The Professional Spreadsheet	247	179
Reflex: The Analyst	150	105
Sidekick Plus	New	200 125
Turbo Basic Compiler	100	68
Turbo Basic Database Toolbox	100	68
Turbo Basic Editor Toolbox	100	68
Turbo Basic Telecom Toolbox	100	68
Turbo C Compiler	100	68
Turbo Lightning	100	68
Turbo Lightning and Lightning Word Wizard	150	105
Turbo Pascal	100	68
Turbo Pascal Database Toolbox	100	68
Turbo Pascal Developer's Toolkit	395	285
Turbo Pascal Editor Toolbox	100	68
Turbo Pascal Gameworks Toolbox	100	68
Turbo Pascal Graphics Toolbox	100	68
Turbo Pascal Numerical Methods Toolbox	100	68
Turbo Pascal Tutor	70	49
Turbo Prolog Compiler	100	68
Turbo Prolog Toolbox	100	68

c language

CBTREE by Peacock Systems	159	129
Essential Software Products All Varieties	CALL	CALL
Greenleaf Products All Varieties	CALL	CALL

creative programming products

Vitamin C Supports Turbo C	225	159
Reference Database for Norton Guides	New	50 47
VC Screen Forms Designer	New Version	150 119

crepuscul software products

GraphPak for Turbo BASIC	69	59
GraphPak Professional for Turbo BASIC	New	99 89
QBase Relational Database for Turbo BASIC	99	89
QBase Report Report Generator for Turbo BASIC	69	59
QuickPak Professional for Turbo BASIC	149	129

database management

Clipper by Nantucket	695	379
dBASE III Plus by Ashton-Tate	695	389
FoxBASE + by Fox Software	395	249
FoxBASE + /386 by Fox Software	New	595 399
Genifer by Bytel	395	249
R:BASE for DOS by Micromim	725	539
R:BASE for OS/2 by Micromim	New	895 649
R:BASE Program Interface by Micromim	595	389

microsoft products

Microsoft C Compiler 5 w/CodeView	New Version	450 299
Microsoft COBOL Compiler with COBOL Tools	700	465
Microsoft FORTRAN Optimizing Comp	New Version	450 299
Microsoft Macro Assembler	New Version	150 105
Microsoft Mouse All Varieties	CALL	CALL
Microsoft OS/2 Programmer's Toolkit	New	350 239
Microsoft Pascal Compiler	New Version	300 199
Microsoft QuickBASIC	99	69
Microsoft QuickC	99	69
Microsoft Windows	99	69
Microsoft Windows 386	195	129
Microsoft Windows Development Kit	500	329
Microsoft Word	450	299
Microsoft Works	195	129

migent products

DATABASE SERVER Multi-user database engine	New	695 629
Developer's Toolkit for C	New	495 449
EAGLE Database Application Language	New	495 449
SUMMIT Database Add-in for Lotus 1-2-3	New	195 175

nostradamus products

Instant Assistant	100	89
Instant Replay III	150	129
Turbo-Plus Supports Turbo Pascal 4.0	100	89

peter norton products

Advanced Norton Utilities	150	89
Norton Commander	75	55
Norton Editor	New Version	75 59

```

/*
 * Fills the color text mode screen with character,
 * Character, displayed with attribute Attribute.
 */
void FillScreen(char Character, char Attribute)
{
    int i;
    unsigned int far *DisplayMemoryPtr;
    unsigned int VideoWord;

    /* Construct a word that contains the character in the low
     byte and the attribute in the high byte */
    VideoWord = (Attribute << 8) | Character;

    /* Build a far pointer to color text mode display memory */
    DisplayMemoryPtr = MK_FP(COLOR_TEXT_SEGMENT, 0);

    /* Set every character on the color text screen to Character,
     displayed with attribute Attribute */
    for ( i = 0; i < FILL_LENGTH; i++ )
        *DisplayMemoryPtr++ = VideoWord;
}

```

DIALOG

continued from page 8

```

GET 1, LASTBYTE
PRINT #2, "&H"; HEX$(ASC(A$))

```

In their place, put these lines:

```

GET 1, LASTBYTE
IF((I-1) MOD 5=0) THEN 'IF PREVIOUS LINE
PRINT #2, "" 'ENDS AT POSITION 5
PRINT #2, "$INLINE &H"; HEX$(ASC(A$))
ELSE
PRINT #2, "&H"; HEX$(ASC(A$))
END IF

```

—Gavin Cole
Guelph, Ontario
CANADA

You've hit on our big secret, Gavin: While other magazines only want to reach wizards, we want to make wizards. In other words, wherever on the ladder you happen to fall in connection with a given language, grab a rung—we'll give you a boost to the next step up. It's called building skills, and we think we do it like nobody else. Thanks for the COM2INC patch, too.

—Jeff Duntemann

SCREAMIN' SCREENS

Michael Abrash's article "Building Far Pointers with MK_FP" (March/April, 1988) has a sample program that illustrates what is probably the most efficient way of writing data directly to screen memory in Turbo C. But if you are crazy about getting the most from your C code, several changes would reduce the statements in the **for** loop in function **FillScreen()** to one statement. These changes are:

```

unsigned int far *DisplayMemoryPtr;
unsigned int VideoWord;

```

```

VideoWord = Attribute << 8;

```

```

for ( i = 0; i < FILL_LENGTH; i++ )
    *DisplayMemoryPtr++ =
        VideoWord | Character;

```

Everything else in the function can stay the same. And, by the way, keep up the work on what appears to be an excellent magazine. I'm looking forward to the next issue.

—Ramon Rivas
Miami, FL

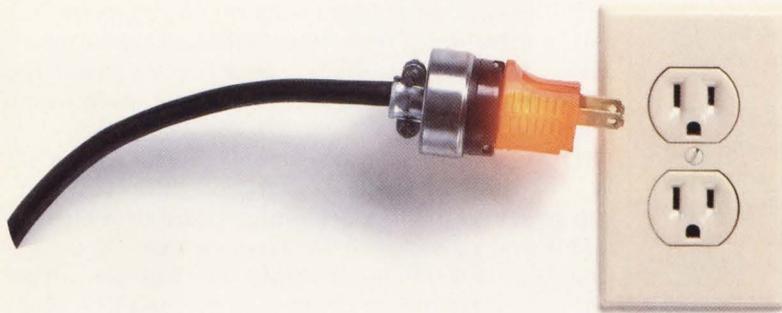
*Mr. Rivas' changes to **FillScreen()** certainly make the function more efficient, both in code size and in speed. Still, there's one tweak yet to be made to Mr. Rivas's code: The ORing together of the character and the attribute should be moved out of the loop. After all, why perform this operation once every time through the loop, when we actually only need to do it once, before the loop? I've written a version of **FillScreen()** that incorporates both Mr. Rivas's suggestions and my final tweak. It's shown in Listing 1.*

—Michael Abrash

Listings may be downloaded from CompuServe as
DILOG5.ARC.

Turbo Expert.

Now it doesn't take a genius to plug into Expert Systems.



For only \$99.95, you can incorporate the power of a full-fledged Expert System into your TURBO PASCAL programs. Seamlessly. Affordably. Finally. Actual Expert Systems, developed for simple use by any Turbo Pascal 4.0 programmer.

Take a look at all the features you suddenly have available with this single Turbo Pascal 4.0 Unit: The ability to create large Expert Systems, or even link multiple Expert Systems together. A powerful backward-chaining inference engine. Easy flow of both data and program control between Turbo Expert and the other parts of your program, to provide Expert system analysis of any database, spreadsheet, file or data structure. The ability to add new rules in the middle of a consultation, so your Expert Systems can learn—really *learn*—and become even more intelligent.

You also have the ability to create large rule bases and still have plenty of room left for your program, thanks to conservative memory use. You can link multiple rule bases, you'll be compatible with our Turbo Toolkit units, and you'll be able to do mathematical calculations, confidence factors, windowing, and more.

Imagine a single "EXE" file containing your user interface and data handling, *and* a full Expert system. Call for more information or to order, (317) 876-3042. Software Artistry Inc., 3500 Depauw Blvd., Suite 2021, Indianapolis, IN 46268



MULTIPLE INDEXES WITH TURBO ACCESS

Use multiple indexes with the Turbo Pascal Database Toolbox to sort your data—without physically sorting it at all.

William Meacham



PROGRAMMER

One of the utilities provided with the Turbo Pascal Database Toolbox is Turbo Access, a set of functions and procedures to create and manipulate data files and their associated indexes.

An *index* shows the location of something. The index in the back of a book shows the reader where various topics appear in that book. The card catalog in a library is a huge index that shows (if you know the codes) where to find each book. In fact, a card catalog is actually a *multiple index* because it shows, in more than one way, where a book is located. You can look up a book by its title, by its author, or by its subject.

As used in computer databases, an index is a file of pointers to records in another file. (A *pointer* is the address of some data item—in this case, a pointer is the address of a record in the other file. That file is called the *data file* to distinguish it from the index file.) Typically, the pointers in an index file are ordered in a way that makes them easy to search. When you construct an index file, each record contains both a *key*, which identifies the corresponding record in the data file, and a pointer to the record in the data file. Data file records need not be in order. As long as the *index* is in order, it's easily searched for the pointer to the desired data file record.

As an example, let's say that a data file contains name and address records, and the key is the last name. The data file might contain the following records in this random order (each record also contains additional data, not shown here for clarity's sake):

MARTINEZ
ALPHONSE
SMITH
BRYKER

If this data file doesn't have an index, you have to search the entire file to find Bryker. If the file contains a large number of records, the search process could get quite tedious. An index, however, speeds up the search. In this example, the index file would contain the records in the following order:

Key	Record in data file
ALPHONSE	2
BRYKER	4
MARTINEZ	1
SMITH	3

The alphabetical order of the records makes it easy to perform a binary search on the index to find the desired last name. Once the name is found, we go directly to that record in the data file without searching the records before it.

In this case, a multiple index would simply be two or more index files. One could be keyed on the phone number, and another keyed on the last name. Multiple indexes let you access information in more than one way.

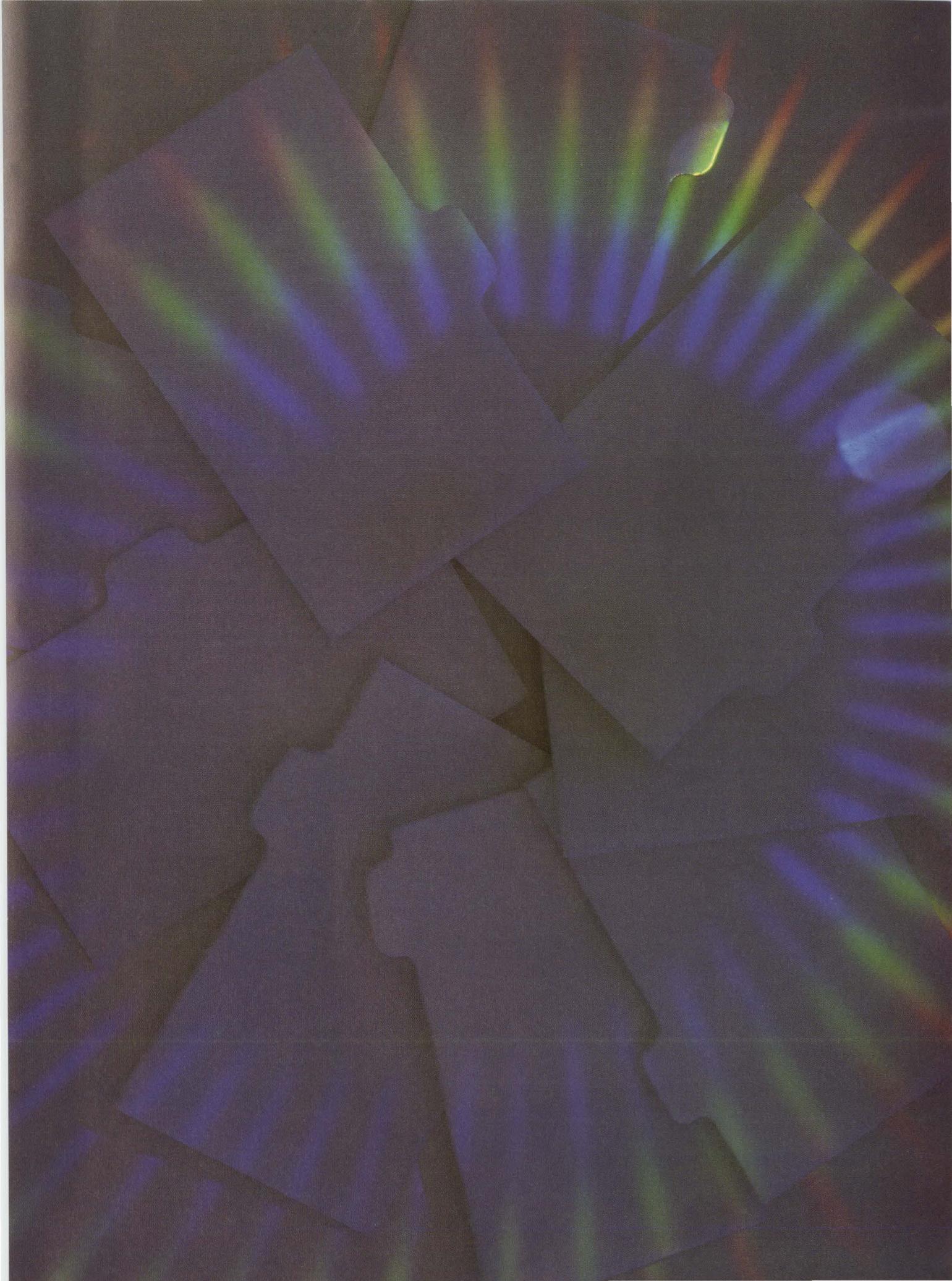
This is a very simplistic example. Turbo Access 4.0 is a collection of functions and procedures that you can plug into your application to create data files and indexes that far exceed this example—to the tune of over two billion records! The indexes are sophisticated B+ trees, not just alphabetized lists, and the search algorithm is faster than a binary search. In this article, I'll show you how to use the Turbo Access routines. I'll also provide concrete examples from a shareware application I've written, called the Reliance Mailing List.

DATABASE FUNCTIONS

There are basically seven things you can do with any database:

1. Create the database;
2. Open the database;
3. Add or insert records;
4. Retrieve records (either randomly or in order) to be displayed, printed, etc.;
5. Change or update records after retrieving them;
6. Delete records; and
7. Close the database.

continued on page 14



INDEXES

continued from page 12

The rich set of procedures in Turbo Access lets you do all of these things with simple procedure calls. In addition, you can have more than one data file, and more than one index per data file.

Turbo Access provides two different ways to handle database functions: "high-level" calls and "low-level" calls. If you've used Turbo Access 1.x with Turbo Pascal 3.0, you'll feel right at home with the low-level calls. With Turbo Access 4.0 (which is used with Turbo Pascal 4.0), the low-level calls are the same as in version 1.x, with a couple of additions. The low-level calls in 4.0

manipulate the data files and the index files separately. As a result, you can index a non-Turbo Access data file, or even use a Turbo Access data file without any index (although I don't know why anyone would want to do that). The high-level calls treat the data file and the index file as one entity, called a *data set*. This makes it easier to add, retrieve, update, and delete records, but it has certain limitations. You can have only one index per data set, and the index cannot contain duplicate keys. I'll say nothing more about high-level calls in this article; they're easy, but low-level calls are much more interesting.

GETTING STARTED

First, create a file that defines the data file record type(s) and the index file record type(s). Next, run the TABUILD program to compile the Turbo Access unit and configure it for your application. This is an important step because the Turbo Access routines are generic—they work with any size data record and any size index record (within certain limits). Turbo Access 1.x required that some global variables be defined before the Turbo Access source code was included. The most important of these global variables were **MaxDataRecSize** (the size of the largest data record) and **MaxKeyLen** (the size of the largest string to be used as a key). Turbo Access 4.0 resides in a unit that is compiled separately from your program.

Since this unit needs to know the values for these global variables, the values are built in by TABUILD when the unit is compiled.

Incidentally, this means that you'll need to compile a separate version of the TACCESS unit for each of your applications. I keep a separate subdirectory on my hard disk for each application that uses Turbo Access, and then compile a separate version of TACCESS in each subdirectory. For instance, I used this command line to compile TACCESS for my mailing list program:

```
TABUILD \TP4\MAIL\MAIL.TYP
```

This command line was executed from the TACCESS subdirectory. During execution, it created TACCESS.TPU in the MAIL subdirectory. The important file here is MAIL.TYP—you'll need to understand this file in order to follow the examples presented later. MAIL.TYP is listed in Figure 1.

TABUILD uses **MaxDataType** to set aside enough space in TACCESS.TPU for the largest data record you'll be using. Likewise, **MaxKeyType** reserves enough space for the largest index record. You can have numerous data files and indexes—TACCESS works fine with all of them as long as no data file record exceeds the size of **MaxDataType**, and no index file record exceeds the size of **MaxIndexType**.

Figure 2 contains a few of the constants and variables that are

continued on page 16

Write Better Turbo 4.0 Programs... Or Your Money Back

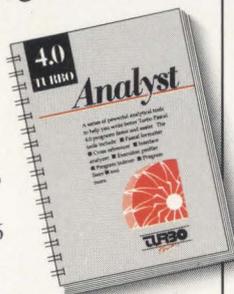
You'll write better Turbo Pascal 4.0 programs easier and faster using the powerful analytical tools of **Turbo Analyst 4.0**. You get • Pascal Formatter • Cross Referencer • Program Indexer • Program Lister • Execution Profiler, and more. Includes complete source code.

Turbo Analyst 4.0 is the successor to the acclaimed TurboPower Utilities:

"If you own Turbo Pascal you should own the Turbo Power Programmers Utilities, that's all there is to it."

Bruce Webster, BYTE Magazine, Feb. 1986

Turbo Analyst 4.0 is only \$75.



A Library of Essential Routines

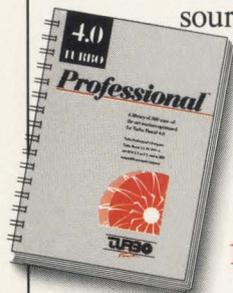
Turbo Professional 4.0 is a library of more than 400 state-of-the-art routines optimized for Turbo Pascal 4.0. It includes complete source code, comprehensive documentation, and demo programs that are powerful and useful. Includes • TSR management • Menu, window, and data entry routines • BCD • Large arrays, and more.

Turbo Professional 4.0 is only \$99.

Call toll-free for credit card orders.

1-800-538-8157 ext. 830 (1-800-672-3470 ext. 830 in CA)

Satisfaction guaranteed or your money back within 30 days.



Fast Response Series:

- T-DebugPLUS 4.0—Symbolic run-time debugger for Turbo 4.0, only \$45. (\$90 with source code)
- Overlay Manager 4.0—Use overlays and chain in Turbo 4.0, only \$45. Call for upgrade information.

Turbo Pascal 4.0 is required. Owners of TurboPower Utilities w/o source may upgrade for \$40, w/source, \$25. Include your serial number. For other information call 408-438-8608. Shipping & taxes prepaid in U.S. & Canada. Elsewhere add \$12 per item.



TurboPower Software
P. O. Box 66747
Scotts Valley, CA 95066-0747



Finally. A programming tool for people who hate manual labor.

Nobody ever said programming PCs was supposed to be easy.

But does it have to be tedious and time-consuming, too?

Not any more.

Not since the arrival of the remarkable new program you see here.

Which is designed to save you most of the time you're currently spending digging through the books and manuals on the shelf above.

drawing characters, error messages, memory usage maps, important data structures and more.

How much more?

Well, the Guides to BASIC, C and Pascal contain detailed listings of all built-in and library functions.

The Guide to BIOS/DOS/Assembly delivers a complete collection of DOS service calls, interrupts and ROM BIOS routines.

While the Guide to OS/2 API packs a handy DOS-to-OS/2 conversion table.

You can, of course, find most of this information in the books and manuals on our shelf.

But Peter Norton—who's written quite a few books himself—figured you'd rather have it on your screen.

Instantly.

In either full-screen or moveable half-screen mode.

Popping up right next to your work. Right where you need it.

This, you're probably thinking, is precisely the kind of thinking that produced the classic Norton Utilities.

And you're right.

But even Peter Norton can't think of everything.

Which is why each version of the Norton Guides comes equipped with a built-in compiler—the same compiler used to develop the databases contained in the Guides.

So you can create new databases of your own, complete with electronic indexing and cross-referencing.

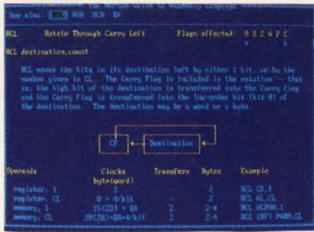
No wonder *PC WEEK* refers to the Guides as "a set of programs that will delight programmers."

Your dealer will be delighted to give you more information. All you have to do is call. Or call Peter Norton Computing.

And ask for some guidance.



A Guides reference summary screen (shown in blue) pops up on top of the program you're working on (shown in green).

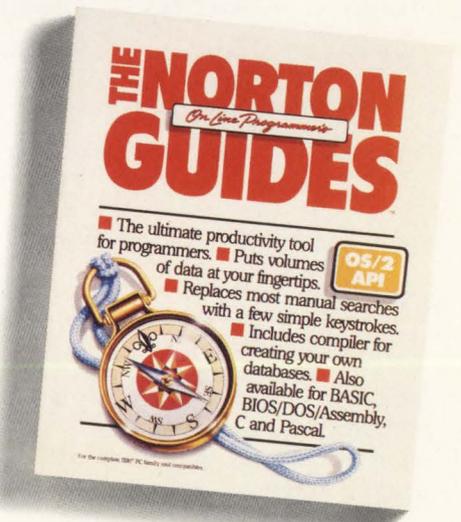


Summary data expands on command into extensive detail. And you can select from a wide variety of information.

It's one of a quintet of pop-up reference packages, called the Norton On-Line Programmer's Guides, that search for information automatically—in DOS or in OS/2 protected mode.

Each package comes complete with a comprehensive, cross-referenced database crammed with just about everything you need to know to write applications.

Everything from facts about language syntax to a variety of tables, including ASCII characters, line



Peter Norton
COMPUTING

```

type
  date      = record
    yr,
    mo,
    dy : integer
  end ;

key1_typ = string[5] ;
  { The key is the first five characters of the last name, stripped of
  blanks and capitalized. }

key2_typ = string[14] ; { zip code + key1_typ }

mf_rec = record      { Master File record }
  status      : longint ; { 0 = active, else deleted } { 4}
  last_name   : string[30] ; {31}
  frst_name   : string[18] ; {19}
  title       : string[9] ; { eg, Dr., Mr., Ms., etc } {10}
  salutation  : string[11] ; { Dear ... } {12}
  addr1       : string[25] ; {26}
  addr2       : string[25] ; {26}
  city        : string[23] ; {24}
  state       : string[2] ; { 3}
  zip         : string[9] ; {10}
  home_phon   : string[14] ; {15}
  work_phon   : string[14] ; {15}
  precinct    : string[3] ; { 4}
  last_amt    : real ; { last contribution amount } { 6}
  last_date   : date ; { last contribution date } { 6}
  tot_amt     : real ; { total contribution amount } { 6}
  flags       : byte ; { 8 booleans, user-defined } { 1}
end ; { total 218}

MaxDataType = mf_rec ;
MaxKeyType = key2_typ ;

```

Figure 1. The master file record format for a typical database application.

```

const
  mf_fname   : string[14] = 'MASTER.RML' ; { master file }
  ix1_fname  : string[14] = 'INDEX1.RML' ; { index file--last name }
  ix2_fname  : string[14] = 'INDEX2.RML' ; { " " zip+last name }
  no_dups = 0 ; { parameter for proc OpenIndex }
  dups_ok = 1 ; { parameter for proc OpenIndex }

var
  master      : mf_rec ; { master record }
  key1       : key1_typ ; { key1 work area }
  key2       : key2_typ ; { key2 work area }

  mf_file    : datafile ; { master file -- type def. in TACCESS }
  ix1_file   : indexfile ; { index file -- type def. in TACCESS }
  ix2_file   : indexfile ; { index file -- type def. in TACCESS }

  rec_num    : longint ; { relative record number of master rec }
                { called DataRef in the manual }

```

Figure 2. File and record declarations for a typical database application.

INDEXES

continued from page 14

used later in the examples. My application has one data file of names, addresses, and other information; and two index files.

The important items here are the variable declarations for the data file and the index files. The file definitions, such as **DataFile**

and **IndexFile**, are predefined in the TACCESS unit; all you have to do is define the file variables. (Do not declare **mf_file** as a file of **mf_rec**; instead, use the TACCESS predeclared type **DataFile**.) Turbo Access declares record types or record variables for the indexes.

DATA FILE STRUCTURE

Basically, a Turbo Access data file is a file of records that is just like

a normal Turbo Pascal file, except that the first record is reserved for system information and doesn't hold data. This first record contains the length of each data record, and a pointer to a list of deleted records. Records are retrieved according to their physical location in the file; I call this location the *relative record number*. The first record is record 0, the second is record 1, and so on.

Record 0 also contains pointers to a free list that is kept within the file. A *free list* is a list of records that have been logically deleted. When the physical records are deleted, they're not removed from the file; instead, the deleted records are marked with a long integer value that is stored in the record's first four bytes. Each value points to (i.e., contains the relative record number of) the next logically deleted record in the list. The long integer value stored in the last record in the chain is set to -1. In accordance with the manual's recommendation, I've set up a long integer status field in the first four bytes (see the **status** field in the **mf_rec** definition in Figure 1). **status** indicates whether the record is active or deleted. When adding a record, set **status** to 0, because a deleted record will never have a status value of 0. You can use the **status** field to rebuild the indexes by searching the data file sequentially. I'll give an example of this process later on.

CREATING A DATABASE

Now let's look at the seven basic database functions, beginning with the process of creating data files and index files. When creating your files, you should follow a certain sequence:

1. Create the data file;
2. If that is successful, create the first index;
3. If that is successful, create the second index; and
4. Repeat the index-creation step for as many indexes as you want to have.

If any step along the way fails, halt the program immediately, because you can't continue and still generate an intact database.

continued on page 18

For Anyone Who Considers Code A Four Letter Word.

If you think writing program code is a dirty business, we have something to help you clean up your act.

It's called Matrix Layout. Layout lets you create programs that do exactly what you want, quickly and easily — without writing a single line of code. Layout does it for you automatically, in your choice of Turbo Pascal, Turbo C, Microsoft C, Quick-Basic or Lattice C. And if you're not a programmer, you can even create programs that are ready-to-run.

As the first true CASE (Computer Aided Software Engineering) development tool for the PC, Layout lets you write your programs simply by drawing an icon-based flow chart. They'll have windows, icons, menus, buttons, dialog boxes, and beautiful graphics and text. Like the Macintosh and the OS/2 Presentation Manager.

And because Layout is so efficient, everything you create will work incredibly fast, even on standard PC's with 256K and only one disk drive. To top it off, all your programs will feature Layout's automatic mouse support, sophisticated Hypertext functions, and decision handling.

The full Layout package also

comes with three additional programs:

Matrix Paint is a professional paint program that comes with a full palette of high-powered graphics tools, plus scanner support. And any picture or symbol that you draw or

scan into Paint can be included in your program.

Matrix Helpmaker allows you to include an electronic manual in all your programs. Context-sensitive help windows, a table of contents, indexing, and the convenience of Hypertext functionality can now become a part of everything you create.

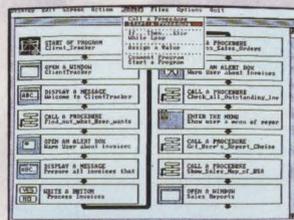
Finally, Matrix Desktop gives you the ability to organize your files and disks in a very Macintosh-like easy to see, easy to use way.

What's the cost? At just \$149.95 for the entire package, Layout speaks in a language you'll love to hear. Especially with our free customer support, no copy protection, and a 30-day, money-back guarantee.

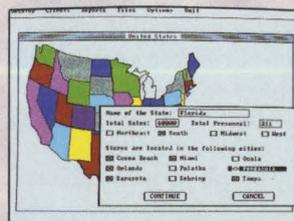
Video Tape Offer

Our new demonstration videotape graphically illustrates how the many features of Matrix Layout will make a difference in your life. Call **1-800-533-5644** and order your VHS copy now (just \$9.95 for shipping and handling, credited against your purchase). In Massachusetts, call (617) 567-0037.

Do it today. Because once you see what Layout can do for you, we think you'll swear by it.



1. Draw a flow-chart.
2. Matrix Layout creates the program code.
3. Your program is complete.



MATRIX
LAYOUT

Matrix Software Technology Corporation • One Massachusetts Technology Center • Harborside Drive • Boston, MA 02128 • (617) 567-0037

Matrix Software/UK • Plymouth, England • 796-363 • Matrix Software/Belgium • Geldenaaksebaan 476 • 3030 Leuven • 016202064

The following are registered and unregistered trademarks of the companies listed: Matrix Layout, Matrix Paint, Matrix Helpmaker, Matrix Desktop, Matrix Software Technology Corporation; Macintosh, Apple Computer, Inc.; OS/2 Presentation Manager, International Business Machines Corporation.

```

procedure create_files ;
{ This is called if files are not found on the selected drive.
  It creates the master and index files. }

{ global type
  str14 = string[14] ; }

procedure bomb (filename : str14) ;
begin
  show_msg (concat('CANNOT CREATE ',filename, '!')) ;
  halt
end ;

begin
  makefile (mf_file,mf_fname,sizeof(master)) ;
  if not OK then bomb (mf_fname) ;
  makeindex (ix1_file,ix1_fname,sizeof(key1)-1,dups_ok) ;
  if not OK then bomb (ix1_fname) ;
  makeindex (ix2_file,ix2_fname,sizeof(key2)-1,dups_ok) ;
  if not OK then bomb (ix2_fname) ;

  clear_master ;                               { set values to zero and blank }
  addrec (mf_file,rec_num,master) ;
  deleterec (mf_file,rec_num) ;
  close_database
end ; { --- Procedure create_files --- }

```

Figure 3. Creating a database.

```

procedure close_database ;
{ Close master file and index files }

begin
  closefile (mf_file) ;
  closeindex (ix1_file) ;
  closeindex (ix2_file)
end ; { proc close_database }

```

Figure 4. Closing a database.

INDEXES

continued from page 16

Figure 3 shows a portion of my code for creating a database. The Turbo Access routines called by procedure `create_files` are **MakeFile**, **MakeIndex**, **AddRec**, and **DeleteRec**.

MakeFile takes three parameters: a file of type **DataFile**, the filename, and the length of the record. Notice the use of Turbo Pascal's **SizeOf** function. You probably know better than to code a numeric constant into a call like this, because if you change the record size, you risk forgetting to change the numeric constant somewhere.

MakeIndex takes four parameters. The first three correspond closely to those of **MakeFile**. The fourth parameter, **Status**, is a flag that indicates whether duplicate keys are allowed. **Status** is passed a value of 0 if duplicate keys are forbidden, and a value of 1 if duplicates are allowed. Notice that the size of the key variable passed to **MakeIndex** in the **KeyLen**

parameter is one less than the actual size of the variable. This is because all index keys are strings. If a key is of another type (such as a real or an integer), that key must first be converted to a string in order to be used as a key. **MakeIndex** wants the maximum number of bytes of string data containable in the string, not the physical length of the entire variable. The length byte must *not* be counted in the value passed in **KeyLen**.

Boolean variable **OK** is predefined in the TACCESS unit, and its value is updated after many of the Turbo Access procedure calls. According to the documentation, if a call to **MakeFile** or **MakeIndex** fails, **OK** is set to **False**. (I have not tested every possible way to make these procedures fail. I have found, however, that if a disk is full, the program crashes with an I/O error. Although it looks pretty, my **bomb** procedure has

never been executed; I leave it in just in case.)

Now look at what happens after the files are created—a record is added and then deleted. Why? To protect data file integrity, I prefer to keep files closed except when they're actually being accessed. But if you create a file and then close it before writing any records, the file length is zero. However, since no record 0 exists, the necessary system information is not contained anywhere. As a result, the data file crashes the application when the data file is opened later on. The solution is to add a record and then delete the record immediately. This creates record 0 (and record 1, which, as you may recall, is logically but not physically removed from the file).

Procedure **Close_Database** is shown in Figure 4. This procedure is an easy way to close the data file and the index files at the same time.

OPENING A DATABASE

The steps for opening the database are very similar to those for creating the database:

1. Open the data file;
2. If that is successful, open the first index;
3. If that is successful, open the second index; and
4. Repeat for as many indexes as you want to have.

Figure 5 contains my **open_database** procedure. The parameters for **OpenFile** and **OpenIndex** are exactly the same as for **MakeFile** and **MakeIndex**. Notice that in both sets of procedures, actual records are not passed. Instead, only the length of the data or index records contained in the file is passed.

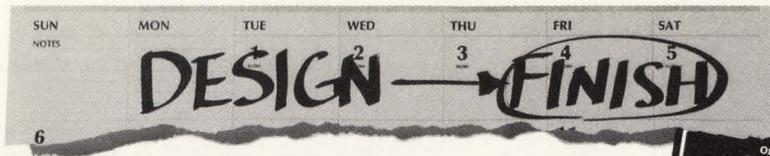
ADDING RECORDS

Now we're ready to add records. The procedure for adding a record mirrors that for creating and opening databases, with a few additional steps. Assuming that the user has filled out a data entry screen for the record to be added, take the following steps:

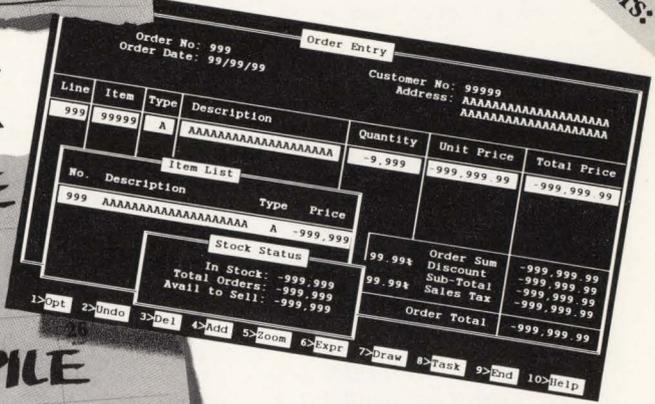
1. Construct a key for the first index. Each key should be related

continued on page 20

Attn
Btrieve®
programmers:



Cut to the Quick



MAGIC PC ELIMINATES CODING . . . CUTS MONTHS OF DATABASE DEVELOPMENT!

Time is money. And coding a DBMS application like Accounting or Order Entry takes a lot of both. Simply because hacking out mountains of code with your RDBMS or 4GL is too slow. Not to mention the time to re-write if you make a mistake or change the design.

EXECUTION TABLES ELIMINATE CODE!

Magic PC cuts months of your application development time because it eliminates coding. You program with the state-of-the-art Execution Tables in place of conventional programming.

HOW DOES IT WORK?

Magic PC turns your database design scheme directly into executable applications without any coding. Use Execution Tables to describe only what your programs do with compact design spec's, free from lengthy how to programming details. Each table entry is a powerful non-procedural design instruction which is executed at compiled-like speed by a runtime engine. Yet the tables can be modified "on the fly" without any maintenance. Develop full-featured multi-user turn-key systems with custom screens, windows, menus, reports and much more in days — not months! No more low-level programming, no time wasted . . .

MAGIC PC™

The Visual Database Language



"Magic PC's database engine delivers powerful applications in a fraction of the time... there is no competitive product."



"Overall, Magic PC is one of the most powerful DBMS packages available."

- Quick Application Generator
- BTRIEVE® — based multi-user RDBMS
- Visual design language eliminates coding
- Maintenance-free program modifications
- Easy-to-use Visual Query-By-Example
- Multi-file Zoom window look-ups
- Low-cost distribution Runtimes
- OEM versions available

ATTENTION BTRIEVE® USERS

Now you can quickly enhance your BTRIEVE®-based applications beyond the capabilities of XTRIEVE® and RTRIEVE®. Use Magic PC as a turn-key BTRIEVE® Application Generator to customize your applications without even changing your existing code.

DATABASE PROGRAMMERS

Join the thousands of professional database programmers and vertical market developers who switched to Magic PC from dBase®, R:BASE®, Paradox®, Clipper®, Dataflex®, Revelation®, Basic, C, Pascal, etc.

TRY BEFORE YOU PAY

We're so sure you'll love Magic PC — we'll let you try the complete package first. Only a limited quantity is available, so call us today to reserve your copy. Pay for Magic PC only after 30 days of working with it. * To cancel . . . don't call . . . simply return in 30 days for a \$19.95 restocking fee.

OR PAY NOW AT NO RISK

Pay when you order and we'll wave the \$19.95 restocking fee so you have absolutely no risk.

SPECIAL OFFER \$695

\$199



Magic LAN multi-user — \$399
Magic RUN — call for price

Order Now Call: 800-345-MAGIC

In CA 714-250-1718

TT

Add \$10 P&H, tax in CA. International orders add \$30.
*Secured with credit card or open P.O. Valid in US.
Dealers welcomed



19782 MacArthur Boulevard, Suite 305
Irvine, California 92715

TLX: 493-1184

FAX: 714-955-0199

```

procedure open_database ;
{ Open master file and index files }

procedure bomb (filename : str14) ;
begin
  show_msg (concat('CANNOT OPEN ',filename,'!')) ;
  halt
end ; { proc bomb }

begin
  openfile (mf_file,mf_fname,sizeof(master)) ;
  if not OK then bomb (mf_fname) ;
  openindex (ix1_file,ix1_fname,sizeof(key1)-1,dups_ok) ;
  if not OK then bomb (ix1_fname) ;
  openindex (ix2_file,ix2_fname,sizeof(key2)-1,dups_ok) ;
  if not OK then bomb (ix2_fname)
end ; { proc open_database }

```

Figure 5. Opening a database.

```

{ global type
  str30 = string[30] ;
  str_type = string[80] ; }

{ ----- }

function purgech (instr : str_type ; inchar : char) : str_type ;
{ Purges all instances of the character from the string }
var
  n : integer ; { Loop counter }
  outstr : str_type ; { Result string }

begin
  outstr := '' ;
  for n := 1 to length (instr) do
    if not (instr[n] = inchar) then
      outstr := concat (outstr, instr[n]) ;
  purgech := outstr
end ;

{ ----- }

function build_key1 (name : str30) : key1_typ ;
{ Construct key for index file 1, last name }
var
  work_area : key1_typ ; { only five characters }
  i : integer ;

begin
  work_area := purgech(name, ' ') ; { Get rid of blanks and }
  { truncate to 5 characters }
  for i := 1 to length(work_area) do { Make upper case }
    work_area[i] := upcase(work_area[i]) ;
  build_key1 := work_area
end ; { function build_key1 }

{ ----- }

function build_key2 (name : str30 ; zip : str9) : key2_typ ;
{ Construct key for index file 2, zip plus last name }
begin
  build_key2 := concat(purgech(zip, ' '),build_key1(name))
end ; { function build_key2 }

{ ----- }

{ Code fragment -- Save_It is a boolean to capture the user's
  response to the question whether to save the data just entered. }

  if save_it then { User says to save the record }
  begin
    key1 := build_key1 (master.last_name) ;
    key2 := build_key2 (master.last_name,master.zip) ;
    addrec (mf_file,rec_num,master) ;
    addkey (ix1_file,rec_num,key1) ;
    addkey (ix2_file,rec_num,key2)
  end ;

```

Figure 6. Adding a record to a database.

INDEXES

continued from page 18

- to the data record in some unambiguous way;
- If there is more than one index, construct the key for the second index, and so on for all of the remaining indexes;
 - Add the record to the data file. The **AddRec** procedure returns the relative record number of the new record in the data file;
 - If adding the record was successful, add the key to the first index through the **AddKey** procedure, using the first key value and the relative record number of the new record. This inserts the index record into the index file in its proper place, so that the index file is always easily searchable; and
 - Repeat the previous step for each of the remaining indexes.

The code in Figure 6 demonstrates this process. Procedure **AddRec** adds the record to the data file (reusing a deleted record slot if available), and returns the relative record number in the global variable **rec_num**. **AddKey** uses this variable and the key value to add an index record to the index. **AddKey** inserts the index record into its proper place in the B+ tree, adjusting the tree as necessary.

The global Boolean flag **OK** is not checked after these calls. **AddRec** doesn't update **OK**; if **AddRec** fails, the program crashes with a Turbo Pascal I/O error. **AddKey** does affect **OK**, but **OK** is relevant *only* if the index files were created and opened with duplicate keys disallowed. In such a case, an attempt to add an index record with a key that was already in the index fails, and **OK** is set to **False**. My particular application allows duplicate keys, so there is no need to check **OK**.

Note how the key is constructed. The primary key consists of the first five characters of the last name, forced to uppercase and purged of blanks. This puts all of the keys into a standard format so that they can be compared easily, but leaves the information in the data file intact. Since the key can

continued on page 22

Sophisticated User Interfaces in Minutes!

Put magic in your programs with



New!
Version 2.0

The World's Best Code Generator!

Windows for data-entry (with full-featured editing), context-sensitive help, Lotus-style menus, pop-up menus, and pull-down menu systems. Overlay them. Scroll within them.

Users and critics say it all!...

"... the best I've used ... The code that it generates is excellent, with every feature you could conceivably desire. ... if you have problems, they give excellent technical advice over the phone. ... It saves time, is flexible and produces screens which are state of the art."

Sally Stott, Software Developer

"... the best screen generator on the market."

George Kwascha, TUG Lines, Nov/Dec 87

"... the Cadillac of prototyping tools for Turbo Pascal. ... Unlike the others, turboMAGIC is extremely flexible. ... [it] clearly offers the greatest variety of options."

Jim Powell, Computer Language, Jun 87

"Fast automatic updating of dependent fields adds flair to your input screens. ... turboMAGIC will be a blessing for programmers who would rather not write the user interface for every program."

Neil Rubenking, PC Magazine, 24 Feb 87

"I was impressed with the turboMAGIC package. ... the procedures created by turboMAGIC are well commented and easy to add to your own code."

Kathleen Williams, Turbo Tech Report, May/Jun 87

"... definitely a recommended program for any Turbo Pascal programmer, novice or expert."

Terry Lovegrove, Library Hi Tech News, Oct 87

ORDER your Magic TODAY! Only \$199.
CALL TOLL FREE 800-225-3165 or 205-342-7026

**sophisticated
software**



6586 Old Shell Road, Mobile, AL 36608

Requires 512K IBM PC compatible and Turbo Pascal 4.0. 30-Day Money Back Guarantee. Foreign orders add \$15.

```
{ Code fragment for searching an index file
and retrieving a data record }
```

```
key1 := build_key1 (name) ;
findkey (ix1_file,rec_num,key1) ;
if OK then
  begin
    getrec (mf_file,rec_num,master) ;
    { display the record on the screen }
  end ;
```

Figure 7. Searching an index file and retrieving a data record.

INDEXES

continued from page 20

always be reconstructed from the information in the data file, indexes can be rebuilt if index files are corrupted or destroyed. The key string is short in order to save disk space. Allowing duplicate keys is essential, and we'll see how that impacts retrieval shortly.

Let's step back and look at what we have at this point. The data file contains a new record, at relative record number *n*. The information in the data record bears no relation to the relative record number. Each index file also contains a new record, which is inserted in order according to the key value. The index record contains the key and the relative record number of the associated data record. Assuming that the user has entered a sufficient number of data records to be useful, we are now in a position to retrieve information from the database.

RETRIEVING RECORDS

In an interactive environment, records can be retrieved either one at a time (perhaps for display on the screen) or as a series (perhaps for printing a list of names, or a series of mailing labels). Retrieving one record at a time is easy. Assuming that the user has entered the name of the person whose record is to be retrieved, take these steps:

1. Build the key from the last name;
2. Search the index for the key; and
3. If successful, retrieve the record pointed to by the index record found.

These steps are demonstrated by the code in Figure 7. Turbo Access's procedure **FindKey** is passed the index file's name and

the key, and returns the relative record number. **FindKey** also updates the global Boolean flag **OK**. If **OK** becomes **True**, the relative record number is that of the first index record containing a key that exactly matches the key passed to **FindKey**. Next, call **GetRec** and pass the data file and the relative record number to that procedure. **GetRec** reads the record at the designated relative record number into the data record variable.

There are two interesting possibilities beyond this simple scenario. The first is that the retrieved record may not be the record that the user wants, because more than one record has the same key. The other possibility is that no matching key may be found at all.

It is likely that more than one record will have the same key in this example, since the key is composed of the first five characters of the last name. "Johnson," "Johns," and "Johnston" all have the same key. Therefore, the user is shown the record that was retrieved, and then allowed to browse through the database in either direction from that point. The procedures **PrevKey** and **NextKey** handle the browsing process. Given a key, **PrevKey** finds the previous key, and **NextKey** finds the next one. **FindKey** returns the first matching key and its relative record number. Calls to these procedures are encapsulated in my procedures **get_prev_rec** and **get_next_rec**. If the user chooses to look at the previous record, **get_prev_rec** is called and displays the record. **get_next_rec** works similarly for the next record. Note that these are the previous and next *logical* records, *in key sequence*, not the next or previous *physical* records, whose se-

quence is generally unordered. The code is shown in Figure 8.

PrevKey and **NextKey** are virtually identical (except, of course, that they get different keys). The new key value is returned in the **Key** parameter passed to both procedures. If the previous or next key is found, each procedure sets **OK** to **True**. If **OK** is **False**, you are at the beginning or the end of the index file. I found out through trial and error that when **OK** is **False**, the values of the **rec_num** and **Key** parameters are still updated. That's why I save the initial value and re-establish the index pointer by calling **FindKey** again after a failed **PrevKey** or **NextKey** operation.

The other possible outcome of an attempt to retrieve a record is that the index search may not be successful. In this case, instead of displaying an error message, I ask the user if he or she wants to view the closest record found. If the user agrees, the closest record is found and displayed as shown in Figure 9.

Figure 9 requires a little explanation. When **FindKey** fails and returns **False** in **OK**, it has in fact found a key, which is the first key *greater* than the key searched for. Instead of showing the record associated with this key, the user is shown the record just *before* it. If a call to **PrevKey** to retrieve that record fails (meaning we are at the beginning of the index file) **SearchKey** is called to retrieve the first key in the file. **SearchKey** is similar to **FindKey**, except that **SearchKey** returns the record number of the first key that is equal to or greater than the key requested. **FindKey**, by contrast, only returns success for an exact match. If **PrevKey** fails, the user is at the beginning of the file, so **Searchkey** returns **True** in **OK**.

RETRIEVING MULTIPLE RECORDS

A mailing list program would not amount to much if it could not print a sorted list of its records. The procedure for printing a sorted list is straightforward:

1. Position the index pointer at the beginning of the index file; and

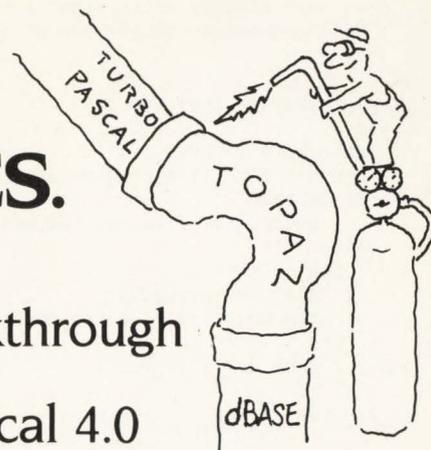
continued on page 24

YOU'LL LOVE THESE UTILITIES.



SAYWHAT?!
The lightning-fast screen generator

TOPAZ.
The breakthrough toolkit for Turbo Pascal 4.0



It doesn't matter which language you program in. With Saywhat, you can build beautiful elaborate, colorful screens in minutes! That's right. Truly *fantastic* screens for menus, data entry, data display, and help-panels that can all be displayed with as little as one line of code in *any* language. Batch files, too.

With Saywhat, what you see is *exactly* what you get. And response time is snappy and crisp, the way you like it. That means screens pop up instantly, whenever and wherever you want them.

Whether you're a novice programmer longing for simplicity, or a seasoned professional searching for higher productivity, you owe it to yourself to check out Saywhat. For starters, it will let you build your own elegant, moving-bar menus into any screen. (They work like magic in any application, with just one line of code!) You can also combine your screens into extremely powerful screen libraries. And Saywhat's remarkable VIDPOP utility gives all languages running under PC/MS-DOS, a whole new set of flexible screen handling commands. Languages like dBASE, Pascal, BASIC, C, Modula-2, FORTRAN, and COBOL. Saywhat works with all the dBASE compilers, too!

With Saywhat we also include a bunch of terrific utilities, sample screens, sample programs, and outstanding technical support, all at no extra cost. (Comprehensive manual included. Not copy protected. No licensing fee, fully guaranteed). **\$49.95**

WE

GUARANTEE IT!

IRON CLAD MONEY-BACK GUARANTEE.
If you aren't completely delighted with Saywhat or Topaz, return them within 30 days for a prompt, friendly refund.



If you'd like to combine the raw power and speed of Turbo Pascal with the simplicity and elegance of dBASE, Topaz is just what you're looking for. You see, Topaz (our brand new collection of

units for Turbo Pascal 4.0) was specially created to let you enjoy the best of *both* worlds. The result? You can create truly dazzling applications in a very short time. And no wonder. Topaz is a comprehensive toolkit of dBASE-like commands and functions, designed to help you create outstanding, polished programs, fast. Think of it. With Topaz you can write Pascal code using SAYs and GETs,

PICTURE and RANGE clauses, then SELECT and USE databases (real dBASE databases!), SKIP through records, APPEND data, and lots more.

In fact, we've emulated nearly one hundred *actual* dBASE commands and functions, and even added *new* commands and functions to enhance the dBASE syntax! All you have to do is declare Topaz's units in your source code and you're up and running!

The bottom line? Topaz makes writing sophisticated Pascal applications a snap. Data entry and data base applications come together with a minimum of code and they'll always be easy to read and maintain.

Topaz comes with a free code generator that automatically writes all the Pascal code you need to maintain a dBASE file with full-screen editing. Plus outstanding technical support, at no extra cost. (Comprehensive manual included. Not copy protected. No licensing fee, fully guaranteed). **\$49.95**

ORDER NOW. YOU RISK NOTHING. Thousands of satisfied users have already ordered from us. Why not call toll-free, right now and put Saywhat and Topaz to the test yourself? They're fully guaranteed. You don't risk a penny.

SPECIAL LIMITED-TIME OFFER! Buy Saywhat?! and Topaz together for just \$85 (plus \$5 shipping & handling). That's a savings of almost \$15.

To order: Call toll-free

800-468-9273

In California: **800-231-7849**
International: 415-571-5019

The Research Group
88 South Linden Ave.
South San Francisco, CA 94080

YES. I want to try:

Saywhat?! your lightning-fast screen generator, so send _____ copies (\$49.95 each, plus \$5 shipping & handling) subject to your iron-clad money-back guarantee.

Topaz, your programmer's toolkit for Turbo Pascal 4.0, so send _____ copies (\$49.95 each, plus \$5 shipping & handling) subject to your iron-clad money-back guarantee.

YES. I want to take advantage of your special offer! Send me _____ copies of both Saywhat?! and Topaz at \$85 per pair (plus \$5 shipping & handling). That's a savings of almost \$15.

NAME _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

Check enclosed Ship C.O.D. Credit card

_____ Exp. date _____ Signature _____

T H E R E S E A R C H G R O U P

```

procedure get_prev_rec ;
  { We have already established a value for key1.
  This procedure returns the previous key and associated record. }
var
  entrykey1 : key1_typ ;
begin
  entrykey1 := key1 ;           { save initial value }
  prevkey (ix1_file,rec_num,key1) ;
  if OK then                   { OK = found previous key }
    getrec (mf_file,rec_num,master)
  else                          { not OK = at first key }
    begin                      { re-establish pointer to key1 }
      key1 := entrykey1 ;
      findkey (ix1_file,rec_num,key1)
    end
end ; { proc get_prev_rec }

{ ----- }

procedure get_next_rec ;
  { We have already established a value for key1.
  This procedure returns the next key and associated record. }
var
  entrykey1 : key1_typ ;
begin
  entrykey1 := key1 ;           { save initial value }
  nextkey (ix1_file,rec_num,key1) ;
  if OK then                   { OK = found next key }
    getrec (mf_file,rec_num,master)
  else                          { not OK = at last key }
    begin                      { re-establish pointer to key1 }
      key1 := entrykey1 ;
      findkey (ix1_file,rec_num,key1)
    end
end ; { proc get_prev_rec }

```

Figure 8. Moving forward and backward from a found key.

```

{ Code fragment. Retrieve the record just before the
one that caused Findkey to fail }

if get_closest then           { user said yes }
  begin
    prevkey (ix1_file,rec_num,key1); { get the key before }
                                       { the key actually }
                                       { found }

    if not OK then
      searchkey (ix1_file,rec_num_key1); { searchkey will always }
                                             { return OK true unless }
                                             { at the end of the }
                                             { index file }

    getrec (mf_file,rec_num,master)
  end ;

```

Figure 9. Retrieving the closest record to a record not found.

INDEXES

continued from page 22

2. Get keys in sequential order, then print the record for each key until the end of the index is reached.

My application easily sorts by last name and zip code because it has two indexes. The code in Figure 10 shows how calls to two different indexes can be incorporated into one procedure.

One Turbo Access procedure that I have not yet discussed is

dex record, and a call to **PrevKey** gives the last record. To print a list in reverse order, substitute **PrevKey** for **NextKey** in the code shown in Figure 10.

UPDATING RECORDS

A database application must let the user update records. The algorithm for updating records is a variant of the algorithm for adding a record. Instead of adding a new record, the program retrieves an existing record, accepts changes from the user, and writes the record back out. Assuming that the user has entered the name of the person whose record is to be retrieved, take these steps:

1. Build the key from the last name;
2. Search the index for the key;
3. If successful, retrieve the record pointed to by the index;
4. Accept changes from the user;
5. Write the changed record to the same position in the data file where the original record was read; and
6. If no key fields were changed, the job is done. If one or more key fields were changed, these actions must be performed:
 - a. Delete the old key or keys, and
 - b. Add the new key or keys.

We've already seen how to perform steps 1 through 3. While step 4 is not difficult, it's outside the scope of this article. Figure 11 contains example code for performing steps 5 and 6 with multiple indexes, and incorporates new Turbo Access procedures. **PutRec**, which looks a lot like **AddRec**, writes data to an existing record in the data file. The parameter **rec_num** is passed to **PutRec**, rather than passed back from it. In Figure 11, **rec_num** has already been established by reading a record from the data file. Since the value of **rec_num** is not altered, it can be used to write the record back out to the file.

The interesting part of Figure 11 is what happens if a field from which a key is derived has changed. First of all, the original value of each of the key fields must be saved for comparison to the values *after* the user changes

continued on page 25

{ Code fragment for sequential retrieval.
The user has already chosen whether to retrieve
in order by last name or by zip code. }

```

open_database ;
if how_to_sort = name then
  clearkey (ix1_file)           { initialize index pointer }
else { how_to_sort = zip code }
  clearkey (ix2_file) ;
repeat
  if how_to_sort = name then
    begin
      nextkey (ix1_file,rec_num,key1) ;
      if OK then
        getrec (mf_file,rec_num,master)
      end
    else { how_to_sort = zip code }
      begin
        nextkey (ix2_file,rec_num,key2) ;
        if OK then
          getrec (mf_file,rec_num,master)
        end ;
      if OK then
        { print the record }
    until not OK ;
  close_database ;

```

Figure 10. Sequential retrieval on one of two indexes.

{ Code fragment for updating a data file and its indexes.
This assumes the user has retrieved a record and made some
changes to it. When the record was retrieved, the values of
the key fields, last_name and zip, were saved in EntryName and
EntryZip. The application has just asked the user if he or
she wishes to save the changes to disk. }

```

if save_it then           { User says to save the record }
  begin
    putrec (mf_file,rec_num,master) ;
                                { change the keys if needed }
    if not (entryzip = master.zip)
    or not (entryname = master.last_name) then
      begin
        key2 := build_key2 (entryname,entryzip) ;
        deletekey (ix2_file,rec_num,key2) ;
        key2 := build_key2 (master.last_name,master.zip) ;
        addkey (ix2_file,rec_num,key2)
      end ;
    if not (entryname = master.last_name) then
      begin
        key1 := build_key1 (entryname) ;
        deletekey (ix1_file,rec_num,key1) ;
        key1 := build_key1 (master.last_name) ;
        addkey (ix1_file,rec_num,key1)
      end
    end ;
end ;

```

Figure 11. Updating a database file and its index files.

{ Code fragment for deleting a data record and its indexes.
User has retrieved a record and displayed it. Program has
asked if the user really wants to delete it. }

```

if delete_it then { user says to delete the record }
  begin
    deletekey (ix1_file,rec_num,key1) ; { we already built key1 }
                                          { to retrieve the record }
    key2 := build_key2 (master.last_name,master.zip) ;
    deletekey (ix2_file,rec_num,key2) ;
    deleterec (mf_file,rec_num)
  end ;

```

Figure 12. Deleting a database record and its keys.

INDEXES

continued from page 24

the record. If the user changes a key field, **DeleteKey** is called with the original value of the key in order to delete the old key. Then **AddKey** is called with the new key value, in order to add the new key.

The value of **OK** is not checked after these operations for two reasons: First, **PutRec** does not affect **OK**; and second, although **DeleteKey** and **AddKey** update **OK**, **OK** will be **True** after the operations. **DeleteKey** returns **False** in **OK** if either the requested key is not found, or (if duplicates are allowed) the requested key is found, but the requested record number is not. Since the record is already successfully retrieved, **DeleteKey** finds the key and record number passed to it. **AddKey** returns **False** in **OK** only if you try to add a duplicate key when duplicates are not allowed. Duplicates are allowed in this application, so **AddKey** always returns **True** in **OK**.

DELETING RECORDS

Deleting a record is much like changing a record. Assuming that the user has entered the name to be deleted, take these steps:

1. Build the key from the last name;
2. Search the index for the key;
3. If successful, retrieve the record pointed to by the index;
4. Show the user the record and ask for verification;
5. Delete the key or keys; and
6. Delete the record.

Since we've already been through steps 1 through 4, Figure 12 shows steps 5 and 6. **DeleteRec** does just what its name implies—when passed the filename and a record number, it deletes the record. **DeleteRec** does not affect the global Boolean flag, **OK**. **DeleteKey** affects **OK**, but there is virtually no chance that it would return **OK = False** in this situation because the key value was used to retrieve the record in the first place.

This completes the discussion of the seven basic database functions. Now, let's investigate some advanced topics.

continued on page 26

continued from page 25

UNORDERED SEQUENTIAL SEARCH

The fact that the data file is separate from the index file (or files) allows some flexibility in dealing with the data file. It is quite possible to access the data file independently of the indexes. This can be useful if you don't need to access the data records in order. For instance, the process of counting the number of records that meet a certain criterion—such as zip codes within a certain range—is made faster by going through the file sequentially, rather than alphabetically. Figure 13 shows such an operation.

The new procedure shown in Figure 13 is **FileLen**, which returns the number of records in the data file as a long integer. Remember that the records are numbered from 0, not 1, so a file of 100 records contains records numbered 0–99. The **WHILE..DO** condition tests the number of records; it succeeds for record 99 and fails for record 100. Notice the use of the **Status** field in the master record—this field is a long integer whose value is 0 if the record has not been deleted. If the record has been deleted, the value of **Status** reflects the record's position in the free list of deleted records.

REBUILDING AN INDEX

Eventually, you will need to rebuild damaged or incomplete index files. Index files can be damaged in a variety of ways, ranging from bad disk media to user error (such as turning off or rebooting the computer before properly exiting the program). If a machine failure occurs after one index is updated but before another one is updated, then the indexes are incomplete. You can rebuild an index by using this procedure to read the data file sequentially and write new index files:

1. Delete the old index files;
2. Create new index files;
3. Open the data file; and
4. Read the data files sequentially. For each data file whose **Status** field contains 0 (i.e., is not deleted), perform these steps:

```
{ Code fragment -- unordered sequential search counting zip codes }
```

```
var
tot_recs,                { total records in file }
num_found  : integer ;   { number found that match }
beg_zip,    { beginning zip code }
end_zip    : string[9] ; { ending zip code }

{ User enters beginning and ending zip codes in range to count }

num_found := 0 ;          { initialize counter }
tot_recs := filelen(mf_file); { get number of records in data file }
rec_num := 1 ;           { skip record 0; it contains no user data }
while rec_num < tot_recs do
begin
  getrec (mf_file,rec_num,master) ;
  if (master.status = 0)           { zero = undeleted record }
  and (copy(master.zip,1,5) >= beg_zip)
  and (copy(master.zip,1,5) <= end_zip) then
    num_found := succ(num_found) ;
  rec_num := succ(rec_num)
end ; { while }
writeln ('Number found = ',num_found) ;
```

Figure 13. Counting records with an unordered sequential search.

```
{ Code fragment -- unordered sequential search to rebuild index files }
```

```
var
tot_recs,                { total records in file }

{ open files, create new indexes, etc. }

tot_recs := filelen(mf_file); { get number of records in data file }
rec_num := 1 ;               { skip record 0; it contains no user data }
while rec_num < tot_recs do
begin
  getrec (mf_file,rec_num,master) ;
  if (master.status = 0) then   { zero = undeleted record }
  begin
    key1 := build_key1 (master.last_name) ;
    key2 := build_key2 (master.last_name,master.zip) ;
    addkey (ix1_file,rec_num,key1) ;
    addkey (ix2_file,rec_num,key2)
  end ;
  rec_num := succ(rec_num)
end ; { while }
```

Figure 14. Rebuilding index files with an unordered sequential traversal of the database.

- a. Construct the key for the first index, call **AddKey** to add it to the index file, and
 - b. Repeat step a for each index.
5. When you are done, close all files.

The code in Figure 14 demonstrates this process.

A BASE FOR YOUR DATABASE

This article has described the basic database functions and their implementation using Turbo Access, which is part of the Turbo Pascal Database Toolbox. I've focused on the use of multiple indexes and low-level Turbo Access procedure calls. Turbo Access pro-

vides a wide range of procedures and functions for manipulating data files and their associated indexes. Use of the Toolbox can relieve you of considerable tedium and intellectual effort when building these procedures and functions yourself. The examples here should be enough to get you started on your own application. ■

William Meacham is a systems analyst in Austin, Texas and a part-time freelance author and programmer. He is the author of the Reliance Mailing List program, a shareware application that uses Turbo Access. Contact him at 1004 Elm Street, Austin, TX, 78703.

CATCH AND THROW IN TURBO PASCAL

Mark a point in your program and return to it from anywhere—instantly!

Jon Shemitz



WIZARD

Procedures and functions are two of the most powerful constructs Pascal offers. These two constructs let us break down complex actions into successive layers of ever-simpler code in order to write clear, maintainable programs. By reusing the code in several places, our programs act consistently.

Sometimes, however, a neatly procedural structure fails us. Consider these scenarios:

- A complex file transaction, involving perhaps several reads and writes, fails at any point across many levels of procedure nesting;
- A spreadsheet program encounters a divide by zero during recalculation of a chain of dependent cells;
- A confused user presses a “go home” key to escape to the main menu from a deeply nested tangle of options and suboptions.

These scenarios have one element in common: An exception that occurs deep inside a chain of procedure and function calls, when it's unacceptable to simply halt the program and return to DOS. We want the opportunity to do some cleaning up, or perhaps give the user a chance to take some corrective action; ultimately we want to return to the main input-action loop as though nothing out of the ordinary has happened. How do we return from all of those nested calls?

In many cases, the simplest, most compact solution is to write code as though all the subprograms always work, and then to “magically” undo all the calls and invoke an exception handler when they don't.

Clearly, this “magic” solution needs two routines: a setup routine that establishes a target or “home base” to return to, and a second routine that returns control to the target. A return involves resetting the stack to a previous state; therefore, the most sensible place to return to is the place where we set the target in the first place—the setup routine.

This method makes the setup routine a thoroughly strange creature: we only enter it once, but we can “return” from it any number of times. The first return is normal—we've just set a return point. Any subsequent “returns” are magical—we've encountered an exception and called the second routine to pass control

back to the target point. Obviously, on return from the setup routine we need to know whether we just did a setup or whether we have an exception to handle. The setup routine is a function that specifies which type of “return” it is. The call to the setup routine looks like this:

```
IF {setup} THEN {proceed normally}
ELSE {handle exception}
```

In C, these two routines are usually called **SetJump** and **LongJump**; in LISP these routines are usually called **Catch** and **Throw**. I have implemented **Catch** and **Throw** for Turbo Pascal 4.0 as a unit called **Xception**, (Listing 1); one part of **Xception** is an assembly language external, **XCEPTION.ASM** (Listing 2).

Turbo Pascal 4.0 never changes the data or stack segments, nor does it make any assumptions from statement to statement about the contents of the extra segment or the general purpose and index registers. Consequently, **Catch** and **Throw** can totally ignore SS, DS, ES, SI, DI, and the general purpose registers, and still do an effective job of saving and restoring the “system state.” This means that **Catch** need only store its return address and the stack and base pointers (SP and BP), while **Throw** need only restore SP and BP and do a long jump to **Catch**'s return address (see Listing 2). However, since an **interrupt** procedure may well use a different stack, **Catch** and **Throw** do, in fact, save and restore the stack segment as well.

The skeletal program in Figure 1 illustrates the fundamental use of **Catch** and **Throw**. You must:

1. Include **Xception** in your **USES** statement;
2. Declare a variable of type **Target**; and
3. Initialize the variable **Target** by calling **Catch**. (Note: throwing control to a target that hasn't been set will almost certainly cause a spectacular system crash.)

Once you have initialized the **Target** variable **Exception**, calling **Throw(Exception)** at any point in your program causes control to return through **Catch**, except that **Catch** now returns the value **ExceptionUsed**, not **ExceptionSet**.

Like any other variable, the target variable can be used *only* within its scope; therefore, it's normally global

continued on page 28

LISTING 1: XCEPTION.PAS

```

unit Xception; ( Exception handling via CATCH and THROW )
{$D+}

interface

type
  Target = record
    Private: array[1..10] of byte; ( "Abstract data type" )
    Point: pointer; ( The THROWing point )
  end;
  ExceptionMode =
    (ExceptionSet, ExceptionUsed);

function Catch(var Exception: Target): ExceptionMode;
procedure Throw(var Exception: Target);
function CanonicThrowingPoint(var Exception: Target): pointer;

implementation

{$L Xception.obj }

function Catch(var Exception: Target): ExceptionMode; external;
procedure Throw(var Exception: Target); external;

function CanonicThrowingPoint(var Exception: Target): pointer;
type
  DWord = record
    Lo, Hi: word;
  end;
begin
  Dec(DWord(Exception.Point).Hi, PrefixSeg + $10);
  CanonicThrowingPoint := Exception.Point;
end;

end.

```

LISTING 2: XCEPTION.ASM

```

;XCEPTION.ASM -- Jon Shemitz
; Assemble by: MASM XCEPTION;
;
; public Catch
; public Throw

BreakPnt struc
cSS dw ?
cSP dw ?
cBP dw ?
cIP dw ? ; offset is low word
cCS dw ?
tIP dw ?
tCS dw ?
BreakPnt ends

code segment word public
Catch proc far
  pop dx ; Ofs(return address)
  pop bx ; Seg(return address)
  pop di ; Ofs(Break)
  pop es ; Seg(Break)
  cld ; we use STOSW to save
  ; a few bytes
  mov ax,ss ; get the stack segment
  stosw ; save the stack ptr
  mov ax,sp ; get the stack ptr
  stosw ; save the stack ptr
  mov ax,bp ; get the base ptr
  stosw ; save the base ptr
  mov ax,dx ; get Ofs(Return address)
  stosw ; save Ofs(Return address)
  mov ax,bx ; get Seg(Return address)
  stosw ; save Seg(Return address)
  xor ax,ax ; Return BreakPointSet
  jmp dword ptr es:[di-4] ; eIP
Catch endp

Throw proc far
  pop ax ; Ofs(return address)
  pop dx ; Seg(return address)
  pop di ; get Ofs(BreakPnt)
  pop es ; get Seg(BreakPnt)
  mov es:[di].tCS,dx ; save the Throw-ing point
  mov es:[di].tIP,ax
  mov ss,es:[di].cSS ; restore Catch's SS
  mov sp,es:[di].cSP ; restore Catch's SP
  mov bp,es:[di].cBP ; restore Catch's BP
  mov ax,1 ; Return BreakPointUsed
  jmp dword ptr es:[di].cIP
Throw endp

code ends
end

```

CATCH AND THROW

continued from page 27

```

PROGRAM Fragment;

USES Xception;

VAR
  Exception: Target;

BEGIN
  IF Catch(Exception) = ExceptionUsed THEN
    BEGIN
      . . .
      {handle exceptions};
      . . .
    END;

    { Main input/action loop: }
    REPEAT
      . . .

      Throw(Exception);

    UNTIL Quit;
  END.

```

Figure 1. A simple use of *Catch* and *Throw*. The first time that *Catch* is called, it returns a value of *ExceptionSet* and control drops through to the rest of the program. Later, if the *Catch* function returns the constant *ExceptionUsed*, it means that *Catch* caught a "throw" from somewhere else in the program, and must handle the exception before continuing.

to the entire program. (Obviously, if the target is local to a subprogram, it can only be used within that subprogram or within nested subprograms.) Of course, you can reset a target so that the same exception is treated differently at different points in the program's execution. For example, if you call *Catch(Exception)* more than once, *Throw(Exception)* passes control to the point where you last called *Catch(Exception)*.

You can also have more than one target set at any one time, such as *FileException* and *UserException*. *Throw* passes control to the point where its parameter (which may be any initialized target) was set.

Finally, be aware that *Throw's* ability to restore the system's state is *absolute*. If *Throw* is executed to a target within a unit's initialization section during the main program block's execution, your program will effectively restart partway through the chain of unit initialization sections, rerunning the initialization sections of any "downstream" units, and finally restarting execution of the main program block. However, remember that *TURBO.TPL* executes prior to the earliest point at which you can set a target, and thus will not be reinitialized.

All of this is quite unstructured, of course, and I would be the last to recommend heavy use of *Catch* and *Throw*. Used with discretion, however, *Catch* and *Throw* can make your code faster, simpler, and easier to read. ■

Jon Shemitz is a consultant in Santa Cruz, California. He can be reached at (408) 479-9916 (voice) or (408) 476-4945 (BBS).

Listings may be downloaded from CompuServe as CATCH.ARC.

Now Supports Quick BASIC 4.0, Turbo Pascal 4.0

Programmers: Go home early!

C, BASIC, Pascal, dBASE®, FORTRAN and Modula-2 programmers: be more productive by clarifying and documenting your source code.

“Occasionally, a utility comes along that makes a programmer’s life much easier. SOURCE PRINT is such a program. It contributes to the programmer’s job by organizing code into a legible format and by helping to organize the documentation and debugging process.”

— PC Magazine

Source Print and Tree Diagrammer both have easy-to-use menus with point-and-shoot selection, and let you search for files containing a given string. For IBM PC and compatibles with 256K.

Join thousands of programmers who are working more efficiently using Source Print and Tree Diagrammer. Order these indispensable tools today. We ship immediately, and there’s no risk with our 60-day money-back guarantee. **Order both and save. Only \$155.00.**

800-257-5773 Dept. T6

MasterCard, VISA, American Express, COD. Add \$5 for shipping/handling.

or see your local dealer!

Source Print and Tree Diagrammer are trademarks of Powerline, Inc. dBASE is a trademark of Ashton Tate. Prices subject to change without notice.

Source Print™

organizes your source code, simplifies debugging, and makes documentation a snap! It lists one or more source files with informative page headings and optional line numbers, while offering invaluable features: **The Index** (Cross-Reference list) saves you time by showing exactly where variables are used and where functions, procedures, and routines are called.

```

1 source ()
2 while (lar < nres && ares[lar][0] == 0)
3   if ((d = ares[lar][1]) == 0)
4     p = A[ares[lar][1]];
5     while (d = sp)
6       loop++;
7     lar++;
8   }
9 }
10
11
12
13
14
15
16

```

C

Before

```

150 FOR INDX = 1 TO 100
160 IF TB(INDX) = 0 THEN X = 5
170 C = 50: WHILE K <= 1000: TB(K) = 0: K = K + X: WEND
180 GOSUB 2000
190 XT(C) = K: T2(C) = K: C = C + 1
200 NEXT INDX

```

BASIC

After

Wed 12-31-86 07:22:03 INDEX (Cross Ref)
all identifiers

inrecord	4,191	9=396	19,825	19=826
	21,889	22,922	22,953	23=978
	23,990			
ins	53,2293	53=2309	53=2319	53,2325
	54,2331	54,2332	54,2336	54=2346
	54,2354	54,2364	54,2365	54,2366
intext	4,193	9=395	43,1796	43,1815
	43=1820	45=1902		

Index

\$97⁰⁰

Locations where new values may be assigned to variables are shown, making it easy to track down that mysterious value change.

Structure Outlining solves the problem of hard-to-see nested control structures by automatically drawing lines around them.

Automatic Indentation of source code and listings reduces your editing time and ensures indentation accuracy.

Plus . . . Source Print generates a table of contents listing functions and procedures. Keywords can be printed in boldface on most printers. Multi-statement BASIC lines can be split for readability. Functions and procedures can be drawn by name from one or more source files to form a new file.

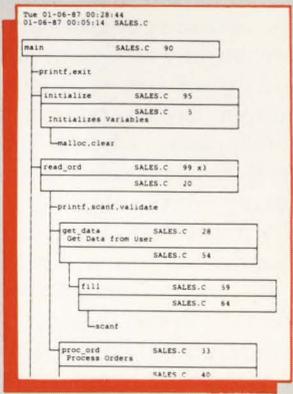


Tree Diagrammer™

shows your program’s overall organization at a glance. Ordinary program listings merely display functions, procedures, and subroutines sequentially, but do not display the relationships between these routines. Our revolutionary new Tree Diagrammer automatically creates an “organization chart” of your program showing the hierarchy of calls to functions, procedures, and subroutines. Recursive calls are indicated and designated comments in the source code will appear on the chart.

Tree Diagrammer helps you organize your program more logically. And you’ll be amazed at how easy it is to debug when you see how your routines interact.

\$77⁰⁰



Powerline, Inc. 2531 Baker Street, San Francisco, CA 94123 415-346-8325

YES! Rush me Source Print (at \$97. _____) Tree Diagrammer (at \$77. _____)

Both \$155. Ship/Handling \$5. For CA add 6% tax _____ Total _____

Name _____

Company _____

Address _____

City _____ State _____ Zip _____

Check enclosed VISA MasterCard American Express

Card # _____ Exp. Date _____

Signature _____ Phone # _____

RECURSING WITHOUT CURSING

Call yourself anytime—just know when to stop.

Jeff Duntemann

On an overwarm September day in 1980, I was sweating into my spiral notebook while Amtrak's Lake Shore Limited wobbled its way across the Ohio hinterlands toward Chicago. Having covered several pages with a tangle of boxes and

SQUARE ONE

arrows, I suddenly felt Carol's hand on my arm.

"You're turning green. Are you carsick?"

"No," I grumbled. "I'm trying to learn recursion."

An old hacker's chestnut kept running through my head on that train: *To iterate is human; to recurse, divine.* Iteration I understood: to repeat a process some number of times, as in a **FOR** loop, a **REPEAT..UNTIL** loop, or a **WHILE..DO** loop. Recursion, on the other hand, is one of those peculiar concepts that just refuses to come clear in the mind until eventually some small spark of understanding happens, and then, *wham!*, it becomes simple or even obvious. A great many people have trouble understanding recursion at first glance, so if you do too, don't think less of yourself for it. We all start out human. Divinity takes a little work.

Recursion is when a function or procedure invokes itself. It seems somehow intuitive to beginners that having a procedure call itself is either impossible or else an invitation to disaster. These fears are unfounded, of course. Let's look at them both.

Recursion is indeed possible. From a coding perspective, in fact, having a procedure call itself is no different than having a procedure call any other procedure. What happens when a procedure calls another procedure? Only this: First, the called procedure is *instantiated*; that is, its formal parameters and local variables are allocated on the system stack. Next, the return address (the location in the code from which the procedure was called and to which it must return control) is "pushed" onto the system stack. Finally, control is passed to the called procedure's code.

When the called procedure is finished executing, it retrieves the return address from the system stack and then clears its variables and formal parameters off of the stack by a process called "popping." Next, the procedure returns control to the code that called it by branching to the return address.

None of this changes when a procedure calls itself. Upon a recursive call to itself, new copies of the procedure's formal parameters and local variables are instantiated on the stack. Then control is passed to the start of the procedure again.

The potential problem shows up when execution reaches the point in the procedure where it calls itself. A third instance of the procedure is allocated on the stack, and the procedure begins running again. This is followed by a fourth instance, and a fifth...and after a few hundred recursive calls the stack has grown so large that it collides with something important in memory, and the system crashes. If you run the following kind of procedure, such a thing would happen very quickly:

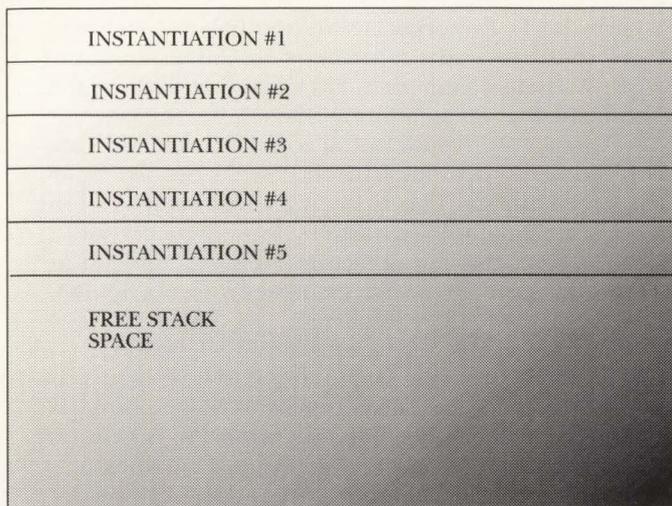
```
PROCEDURE Fatal;
```

```
BEGIN
  Fatal
END;
```

This situation is an unlimited feedback loop. It is this possibility that makes newcomers feel uneasy about recursion.

Obviously, the important part of recursion is knowing when to stop.

A recursive procedure must test some condition before it calls itself, to see if it still needs to call itself to complete its work. This condition could be a comparison of a counter against a predetermined num-



◀ SSEG + \$M STACK ALLOCATION VALUE; DEFAULTS TO 16,384

Figure 1. Each instantiation of a recursive routine reduces the amount of available stack space.

◀ STACK POINTER

◀ SSEG

ber of recursive calls, or some Boolean condition that becomes **True** (or **False**) when the time is right to stop recursing and go home.

When controlled in this way, recursion becomes a very powerful and elegant way to solve certain programming problems.

Let's go through a simpleminded example of a controlled recursive procedure. Read through the code in Listing 1 *very* carefully.

The program itself is nothing more than setting a counter to 1 and calling the recursive procedure **Dive**. **Dive** prints the word "Push!" when it begins executing, and the word "Pop!" when it ceases executing. In between, it prints the value of the variable **Depth** and then increments it.

Note the integer constant, **Levels**. If the incremented value of **Depth** is less than **Levels**, **Dive** calls itself. Each call to **Dive** increments **Depth** by 1, until at last **Depth** is greater than **Levels**. Then recursion stops.

Running program **PushPop** produces the following output. Can you tell yourself *exactly* why?

```

Push!
Our depth is now 1
Push!
Our depth is now 2
Push!
Our depth is now 3
Push!
Our depth is now 4
Push!
Our depth is now 5
Pop!
Pop!
Pop!
Pop!

```

Follow the execution of **PushPop** through all of its steps, until the output makes sense to you.

NUMBERS? NUMBERS!

Certain programming problems simply cry out for recursive solutions. Perhaps the simplest and best-known is the matter of calculating factorials. (The ! operator indicates the factorial operation, rather than any sort of numeric enthusiasm.) A factorial is the product of a digit multiplied by all of the digits less than itself, down to one:

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

A little scrutiny here will show that 5! is the same as $5 \times 4!$, and that 4! is the same as $4 \times 3!$, and so on. In the general case, $N! = N \times (N-1)!$ Whether you see it immediately or not, we have already expressed the factorial algorithm recursively by defining it in terms of a factorial. This will become a little clearer when we express it in Pascal, as is done in Listing 2.

There isn't a great deal to function **Factorial**. The function body is a single statement, and we express it as a conditional statement because there must always be something to tell the code when to stop recursing. Without the $N > 1$ test, the function merrily decrements **N** down past zero and recurses away until the system crashes.

The way to understand this function is to work it out for $N = 1$, then $N = 2$, $N = 3$, and so on. For $N = 1$, the $N > 1$ test returns **False**, so **Factorial** is assigned the value 1. No recursion is involved: $1! = 1$. For $N = 2$, a recursive call to **Factorial** is made: **Factorial** is assigned the value $2 * \text{Factorial}(1)$. As we saw above, **Factorial(1) = 1**. So $2! = 2 \times 1$, or 2. For $N = 3$, two recursive calls are made: **Factorial** is assigned the value $3 * \text{Factorial}(2)$. **Factorial(2)** is computed (as we just saw) by evaluating (recursively)

continued on page 32

LISTING 1: PUSHPOP.PAS

```

PROGRAM PushPop;

CONST
  Levels = 5;

VAR
  Depth : Integer;

PROCEDURE Dive(VAR Depth : Integer);

BEGIN
  Writeln('Push!');
  Writeln('Our depth is now: ',Depth);
  Depth := Depth + 1;
  IF Depth <= Levels THEN Dive(Depth);
  Writeln('Pop!')
END;

BEGIN
  Depth := 1;
  Dive(Depth);
END.

```

LISTING 2: FACTORL.SRC

```

FUNCTION Factorial(N : LongInt) : LongInt;

BEGIN
  IF N > 1 THEN Factorial := N * Factorial(N-1)
  ELSE Factorial := 1
END;

```

RECURSIVE

continued from page 31

Factorial(1). Factorial(1) simply equals 1. Catching on? One interesting thing to do is to add (temporarily) a **Writeln** statement to **Factorial** that displays the value of **N** at the beginning of each invocation.

A note on the power of factorials: Calculating anything over 7! overflows Turbo Pascal's two-byte integer type **Integer**. This is why **Factorial** returns a long integer. Actually, long integers don't help all that much, since the largest factorial representable in a long integer is 16!, which evaluates to 2,004,189,184.

THE HAZARDS OF CALLING YOURSELF

Even when you build machinery into a recursive routine that terminates the recursion at some point, recursion carries with it a certain hazard to the unwary. Knowing when to stop is the key; and the obvious answer is to stop when the work is done. However, there is the danger that you may run out of space on the stack for a new instantiation of the recursive routine *before* the work is done. So the problem devolves to this: How do you know when your stack is running low?

The good news is that Turbo Pascal can tell you how much stack space you have left.

The bad news is that that may not help you very much.

The good news comes in the form of a predefined function named **Sptr**, which quite simply returns the current value of the stack pointer register. Without getting into too many gritty details, the stack in your PC looks like Figure 1. The stack begins at a location in memory called **SSeg**, and continues upward in memory, usually to a length of 16,384 bytes. This stack size value may be increased to 65,520 bytes with the **\$M** compiler directive (see Appendix C in the *Turbo Pascal Owner's Handbook*). The stack pointer is the 8088's thumb in the stack; it indicates where the next available byte of stack space falls. Turbo Pascal sets up its stack so that the stack pointer starts off at the high end of the stack. As stack space is used up, the stack pointer is moved closer and closer to the bottom of the stack. When the value of the stack pointer is 0, you're out of stack and out of luck.

Now, by using **Sptr** to check the value of the stack pointer before each recursive call, you can theoretically see 0 coming and stop recursing before it's too late. *However*, you have no good way to know how much stack space each instantiation of your recursive routine will demand. Thus, while you can test whether the stack pointer is greater than zero, you

don't really know how close you can cut it before runtime error #202 (Stack Overflow) puts your program out of its misery.

Now wait, it gets worse. Suppose you're in the middle of some recursive task, and you notice that your stack is about gone. It's time to stop recursing and pop your way back up to reality—except that you will be partway through some job that now will not be completed. You may have traversed a binary tree partway, or partially filled a graphics screen, or done something else partway, but you may not know how far you've gotten, and you may have changed things that can't easily be undone.

An excellent example of this was provided by Fred Robinson in "Filling Regions with the Turbo Pascal Graphix Toolbox," (*TURBO TECHNIQ*, March/April, 1988). Fred showed us **Flood_Fill**, a very small, very fast routine for filling irregular areas on a graphics screen. This routine uses recursion, but the number of recursive calls required to fill a given area cannot be predicted ahead of the fact. A small area might be filled successfully, while a slightly larger area could exhaust the stack, leaving the program crashed and the area only partly filled.

Fred decided that recursion was not an appropriate way to fill regions of an arbitrary size, and designed a different, nonrecursive routine to do the job. You may also need to make that decision at some point. There are only a few useful guidelines that I can provide on creating a successful recursive procedure or function:

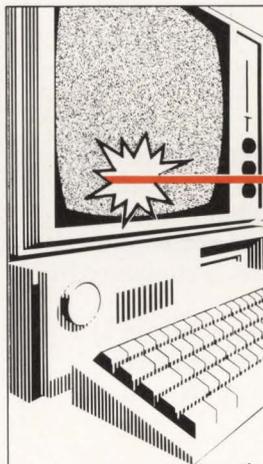
- Use as few procedure parameters and local variables as you can. The idea is to minimize the use of stack space, and every parameter and local variable must be allocated on the stack *each* time the routine calls itself.
- Use recursion *only* in situations where the number of recursive calls needed to get the job done is relatively low. The best situation is where the application limits the number of recursive calls, as in the **Factorial** function discussed earlier (limited to 16 levels), or in a routine that traverses the DOS directory structure with one recursive call per nested subdirectory. Subdirectories are rarely nested more than four or five levels deep.
- As a corollary to the above, applications where the number of recursive calls is measured in the hundreds—rather than the dozens—are always bad medicine. Use a nonrecursive algorithm.

NEITHER HUMAN NOR DIVINE

Some people believe that recursion is inherently slow, or else inherently fast. In fact, recursion is neither—it imposes no more of a performance burden on your programs than does a procedure call, which is all that recursion is. On the other hand, a procedure call takes some grumbling by the CPU—getting things onto the stack and off again—that **WHILE..DO** or **FOR** loops do not require. Using recursion is slower than using a **FOR** loop, so if a **FOR** loop is called for, use it.

On the other hand, there is a species of problem that simply falls out in recursive terms, as Douglas Hofstadter has argued in his deep but fascinating book *Goedel, Escher, Bach: An Eternal Golden Braid* (New York: Basic Books, Inc., 1979), which, I must warn, is *not* Square One material! The trick in using recursion lies in recognizing those problems, and not recasting iterative problems in recursive terms merely for the self-referential strangeness of it all. ■

Listings may be downloaded from CompuServe as RECURS.ARC.



Get To Know Your Programs Inside! and Out!

Now you can analyze your programs with unprecedented detail with Inside!, a new software package from Paradigm Systems.

Inside! allows you to examine the route your programs take through execution counts, minimum, maximum and total elapsed times and a count of how many times each source line executes—function by function—for Turbo Pascal and Turbo C!

New Product Offer:

Call Paradigm Systems before Sept. 1 and get your easy-to-use Inside! software for only \$65.00. Inside!, which is also available in other languages, will sell for \$75.00 after this special introductory offer.



Paradigm Systems Incorporated
P.O. Box 152 Milford, MA 01757
(800)537-5043 (617)478-0499

Turbo C and Turbo Pascal are registered trademarks of Borland International Inc. Inside! is a trademark of Paradigm Systems Incorporated.

CUSTOM TEXT FILE DEVICE DRIVERS

Create pseudo-files with special properties using this new Turbo Pascal 4.0 feature.

Neil Rubenking



WIZARD

Turbo Pascal 4.0's new text file device driver (TFDD) feature gives you full control of the routines that open, close, read from, and write to a text file. TFDDs are well suited to a range of applications, including console drivers for specialized monitors, serial port I/O, or any kind of text device I/O that isn't handled by the ordinary Turbo Pascal text file functions.

By contrast, the input and output of logical devices in Turbo Pascal 3.0 was controlled via user-created I/O drivers that could only be altered for a text device, not for a text file variable. To change all I/O, you had to modify the CON device; to change only certain I/O operations, you modified the USR device.

Turbo Pascal 4.0's TFDD provides access to the file handle and other internals of a **Text** variable that were accessed in 3.0 via the file interface block. Listing 1 shows the standard type definition for a **TextRec**, which is the record that corresponds to a text file variable. This type definition is contained in the **DOS** unit.

Notice the 16-byte array, called **UserData**, in Listing 1. With Turbo Pascal 4.0, you can create your own **TextRec** type, and replace **UserData** with any other fields that total 16 bytes.

A TFDD FOR STRING CONVERSION

All variables that are written to the screen or to a text file become character strings. When you want to convert variables to strings yourself (perhaps as input to a string-manipulation routine), you can select from numerous string conversion methods. One method, which handles components in a piecemeal fashion, uses the **Str** procedure to convert numeric values into strings and then concatenates the various substrings into a single string. This method is not a general solution, because it must be hand-coded to fit each individual situation.

A better, and far more general, method involves writing a TFDD to do string conversion. **UsrFile** (Listing 2) demonstrates this technique.

The UsrFile TFDD. **UsrFile** is a write-only TFDD—you can't **Reset** it or **Read** from it. Instead, you create a pseudo text file using the special **AssignUsr** procedure, **Rewrite** that file, and write anything you wish converted into a string to that file. Any data type that can be written either to the screen or to an ordinary text file can be written to **UsrFile** (including numeric values and Boolean values, but not enumerated types, arrays, records, or sets). Data that has been written to **UsrFile** is converted to string data and concatenated into a single string value that (as with all Turbo Pascal strings) may be up to 255 characters in length. A single call to the function **ReadUsr** returns the string value that has accumulated in the file and clears the file. The genuinely clever thing about **UsrFile** is that the actual process of conversion to string data is handled by Turbo Pascal's **Write** statement—**UsrFile** intercepts characters that are already converted by **Write**, before those characters reach their typical destinations on the screen or in an ordinary text file.

Customizing TextRec. A TFDD requires a customized **TextRec**. Examine **UsrFile**'s redefined **TextRec** type in Listing 2—rather than containing 16 bytes of **UserData**, **TextRec** has three new fields. **UFilePos** and **UFileSize** are **Word** fields that contain the current file position and size. The **Data** field is a pointer to the device's string data area. **TextRec** contains two 2-byte words, one 4-byte pointer, and eight bytes of unused space. This unused space in the original **UserData** is declared to be an 8-byte array called **UserData** in order to hold its space. Keep in mind that if you accidentally create a **TextRec** type comprising the wrong number of bytes, you'll get an "Invalid type cast" error when you try to compile your I/O routines.

Customizing I/O routines. In addition to the customized **TextRec**, a TFDD requires several customized I/O routines, which must follow a very specific format. First, these routines must all use the *far* call model (so we enable *far* calls throughout the definitions of the I/O routines by bracketing their definitions with the **\$F+** and **\$F-** compiler directives). Second, these I/O routines must all take a single **VAR** parameter of type **TextRec**. Third, they must return an integer result. A result that is anything other than zero is reported at runtime as an I/O error.

UsrFile requires three custom I/O routines. One routine opens the pseudo-file, the second routine closes the pseudo-file, and the third routine writes to the pseudo-file. The **UsrOpen** routine sets the file size and file position to zero. In a full read/write TFDD, we would have to deal with three different kinds of file opening routines: **Reset**, **Rewrite**, and **Append**. However, since **UsrFile** is a write-only version, it only has to handle **Rewrite**. If you try to **Reset** or **Append** the file, the **Open** routine triggers an I/O error.

You could omit a file-close routine in the case of this particular TFDD, because there's little point in closing a pseudo-file such as the one used in **UsrFile**. However, if you *did* attempt to close the pseudo-file without initializing the value of the **CloseFunc** field of the file's **TextRec** record, your program would crash. The **UsrClose** routine exists to prevent such crashes. **UsrClose** is the smallest size that it could possibly be, and it simply returns the integer zero value to indicate success.

UsrOutput. Procedure **UsrOutput** is called whenever you write to the pseudo-file. During such a write operation, Turbo Pascal's Runtime code takes the data passed to the **Write** statement, converts that data into a string of characters, and puts the characters into the file variable's buffer. (This buffer is the referent of the **BufPtr** field in the file variable's **TextRec**.)

The **BufPos** field of the pseudo-file's **TextRec** reports how many characters were placed in the buffer. The purpose of **UsrOutput** is to do *something* with those characters and then set **BufPos** back to zero, so that the next write operation will not overwrite data written to the buffer during the previous write operation.

UsrOutput first checks that adding more characters to its internal string buffer from the pseudo-file's buffer won't overrun the buffer's 255-byte limit. If the current file position, plus the number of new characters in the buffer (**BufPos**), is greater than the buffer's size, **UsrOutput** returns a "File Full" error. Also, if the file mode is anything other than **fmOutput**, **UsrOutput** returns an appropriate error code. If there is no error, **UsrOutput** moves the characters from the file's internal buffer to its own internal string buffer and updates the pseudo-file's position and size fields in the **TextRec**.

A special Assign procedure. **UsrOpen**, **UsrClose**, and **UsrOutput** are the only custom routines needed to create the TFDD. How do these routines get attached to a text variable? You have to write a special version of Turbo Pascal's familiar **Assign** procedure. The key to the process lies in four special pointer fields present in all **TextRec** variables that belong to all files of type **Text**. These pointer fields—**OpenFunc**, **InOutFunc**, **FlushFunc**, and **CloseFunc**—are normally initialized to the standard text file I/O routines that reside in the Turbo Pascal Runtime Library. To associate a text file with a suite of custom TFDD I/O routines, reassign these pointer values to point to your custom I/O routines. Certain other fields in the **TextRec** must also be initialized, just as they would be if the file variable was passed to the standard **Assign** procedure.

In the **UsrFile** program, procedure **AssignUsr** sets up all the necessary fields in **TextRec**. The initial mode is **fmClosed**; since the filename is irrelevant, we make it a null string. File position and size are initialized to zero. The real ac-

tion of **AssignUsr** lies in its modification of the **TextRec**'s four pointers to I/O routines. Note that both **InOutFunc** and **FlushFunc** point to the custom **UsrOutput** routine. This ensures that text is sent to the internal string buffer **Data** at the end of every **Write** statement. Without a **FlushFunc** routine, the converted text would not necessarily be moved out to **Data** until your program either flushed or closed the file.

Notice that procedure **ReadUsr** is *not* a TFDD routine. In one single operation, **ReadUsr** returns a string containing everything that you've written to the pseudo-file and clears the file. This application is simple enough that it doesn't require a true **UsrInput** routine.

The main body of **UsrFile** demonstrates how to write a TFDD to do string conversion. Any combination of variables, constants, or function results (or *anything* that can be written to the screen) that doesn't exceed 255 characters in length can be written to **UsrFile**. **UsrFile** also demonstrates an I/O error from our special I/O routines—proving that this error is just like an I/O error generated during I/O to a normal text file.

Next, let's examine a more complex TFDD that handles a variety of activities.

A RAM FILE WITHOUT A RAM DISK

MEMFILE.PAS (Listing 3) demonstrates a complete TFDD with all functions. You can **Rewrite**, **Reset**, **Append**, **Read**, or **Write** to **MEMFILE**, which creates a "file" on the heap of a size that you specify. In the example, the maximum size of the file is 4096 bytes; you can change that value to any amount of available heap memory by changing the constant **UsrSize**.

The special **AssignUsr** procedure is much like the one belonging to the **UsrFile** program described previously. However, fields **InOutFunc** and **FlushFunc** aren't actually reinitialized to point to

continued on page 36

custom routines until you open the file, because the file may be opened with **Reset**, **Rewrite**, or **Append**. The routines whose addresses are assigned to **InOutFunc** and **FlushFunc** vary, depending on the file mode in force when the file is opened by a call to **UsrOpen**.

If **UsrOpen** is called by **Reset**, the file mode is **fmInput**, which means that the file is to be opened for *reading*. The **Erased** flag in the modified **TextRec** type lets you know if the file in fact exists—if it doesn't exist, an error value is returned. If the file exists, we point the **InOutFunc** pointer to the **UsrInput** routine, point **FlushFunc** to a dummy routine called **UsrIgnore** that simply returns a zero value, and put the file position at zero.

On a **Rewrite** call, the file size and position are both set to zero. If **Erased** is equal to **True**, then RAM hasn't been allocated for it yet, so that step is also performed now. Both the **InOutFunc** and **FlushFunc** pointers are set to point to the output routine, **UsrOutput**.

Append is a curious combination of both **Reset** and **Rewrite**. **Append** first checks to be sure that the file exists—if it doesn't exist, an error code is returned. If the file *does* exist, the file pointer is pointed to the very end of the file by setting **FilePos** to **FileSize**. The mode is changed to **fmOutput**, and again both **InOutFunc** and **FlushFunc** are set to point to **UsrOutput**.

The output routine, **UsrOutput**, is identical to the one by that name in the **UsrFile** program described earlier. In this case, however, the internal buffer is an entire RAM file existing on the heap. The process of transferring data to the internal buffer from the **TextRec**'s temporary buffer after

each write operation is the same in both cases. Only the pointer referents differ.

The **UsrInput** routine is a bit more complicated. It returns characters if any more characters exist in the file, and signals End of File (**EoF**) if no characters exist after the file pointer. If the current file position is either at or past the file size, then we're at **EoF**. This is indicated by setting both **BufPos** and **BufEnd** to zero. Otherwise, the process is almost exactly the reverse of sending output to the RAM file, except that the possibility of overrunning the file variable's relatively small internal buffer is avoided. If more characters are available than can fit in the buffer, we process one buffer-full at a time.

Once you no longer need a memory file, procedure **EraseUsr** removes it by deallocating the heap memory that held the file's buffer, and setting the **Erased** flag to **True**. Without this procedure, a new memory file's heap allocation would be lost for the duration of the program each time that the new memory file was opened.

You can use the memory file concept in any program (such as a text sorter or a translation program) that requires *temporary* text files during its operation. If you're currently running your program from a RAM disk for better performance, consider the possibility of using a RAM file instead. Without any DOS overhead, it's likely to be significantly faster.

ALTERING AN EXISTING TEXT DEVICE

In Turbo Pascal 3.0, the internal variable **ConOutPtr** contained the offset of the internal routine that output a single character to the console. Console output was redirected to a custom driver by setting the **ConOutPtr** to the address of your custom console output routine. A custom **ConOut** procedure simply accepted a single character and did something with it. With Turbo Pascal 4.0, however, the console output driver must be modified according to TFDD

standards. To illustrate this process, we'll create **BackUnit** (Listing 4).

BackUnit causes output from all **Write** and **WriteLn** statements to appear backward on the screen. This slightly frivolous example doesn't require any special hardware—try it out on any system with a CRT that responds to standard PC BIOS calls for video output. As the program demonstrates, the **TextColor** and **TextBackground** statements work fine on backward text, and you'll find that **GotoXY**, **WhereX**, and **WhereY** perform correctly as well.

The initialization section. A unit can have an initialization section containing code that executes automatically at the start of any program that uses the unit. This code lies between the **BEGIN..END** pair at the very end of the unit. In **BackUnit**, the initialization section saves the current addresses of the I/O functions that are to be changed, and sets up an exit procedure that gets control at the end of any program that uses the unit. **BackUnit** has a minimal exit procedure—it turns off backward writing by restoring the saved I/O routine addresses, and then chains to the previously active **ExitProc**. (For more on unit initialization sections and exit procedures, see "Custom Exit Procedures," *TURBO TECHNIX*, March/April, 1988.)

Changing pointers in TextRec. In order to change the output action of a text file, we have to change the **InOutFunc** and **FlushFunc** pointers in the file's **TextRec**. Turbo Pascal 4.0's strong type casting ability makes it easy for us to access the fields of the **TextRec** record that corresponds to **Output**. We refer to the record as **TextRec(Output)**, which recasts **Output** to a **TextRec** type. (Under 3.0, we had to declare a **TextRec** variable as **ABSOLUTE** at the same address as **Output**.)

Since the existing addresses for **InOutProc** and **FlushProc** were saved, it's easy to turn the new ver-

continued on page 38

PolyAWK™ – The Toolbox Language™

For C, Pascal, Assembly & BASIC Programmers.

We call PolyAWK our "toolbox" language because it is a general-purpose language that can replace a host of specialized tools or programs. You will still use your standard language (C, Pascal, Assembler or other modular language) to develop applications, but you will write your own specialized development tools and programs with this versatile, simple and powerful language. Like thousands of others, you will soon find PolyAWK to be an indispensable part of your toolbox.

A True Implementation Under MS-DOS

Bell Labs brought the world UNIX and C, and now professional programmers are discovering AWK. AWK was originally developed for UNIX by Alfred Aho, Richard Weinberger & Brian Kernighan of Bell Labs. Now PolyAWK gives MS-DOS programmers a true implementation of this valuable "new" programming tool. PolyAWK fully conforms to the AWK standard as defined by the original authors in their book, *The AWK Programming Language*.

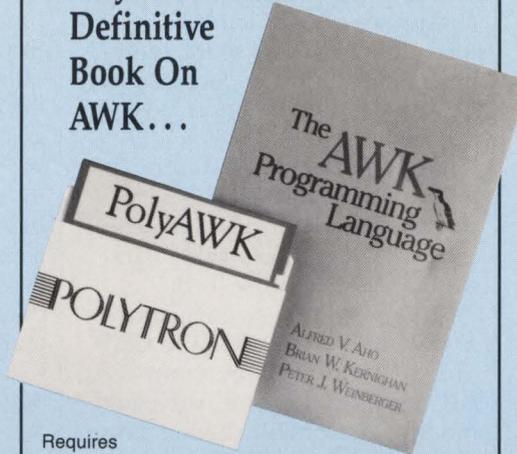
A Pattern Matching Language

PolyAWK is a powerful pattern matching language for writing short programs to handle common text manipulation and data conversion tasks, multiple input files, dynamic regular expressions, and user-defined functions. A PolyAWK program consists of a sequence of patterns and actions that tell what to look for in the input data and what to do when it's found. PolyAWK searches a set of files for lines matched by any of the patterns. When a matching line is found, the corresponding action is performed. A pattern can select lines by combinations of regular expressions and comparison operations on strings, numbers, fields, variables, and array elements. Actions may perform arbitrary processing on selected lines. The action language looks like C, but there are no declarations, and strings and numbers are built-in data types.

Saves You Time & Effort

The most compelling reason to use PolyAWK is that you can literally accomplish in a few lines of code what may take pages in C, Pascal or Assembler. Programmers spend a lot of time writing code to perform simple, mechanical data manipulation — changing the format of data, checking its validity, finding items with some property, adding up numbers and printing reports. It is time consuming to have to write a special-purpose program in a standard

PolyAWK Comes With The Definitive Book On AWK...



Requires MS-DOS 2.0 or above & 256K RAM.

\$99

When you order PolyAWK you receive a copy of *The AWK Programming Language* written by the authors of the original UNIX-based AWK. The book begins with a tutorial that shows how easy AWK is to use, followed by a comprehensive manual. Because PolyAWK is a complete implementation of AWK as defined by the book's authors, you will use this book as the manual for PolyAWK.

You can purchase PolyAWK and the book, *The AWK Programming Language*, for \$99. If you already have the book, you can order PolyAWK software only for \$85, which is \$14 off the regular \$99 purchase price. (The book serves as the User's Manual, so you should already have a copy of the book if you are ordering the software only.)

PolyShell Bonus!

PolyShell gives you 57 of the most useful UNIX commands and utilities under MS-DOS in less than 20K. You can still use MS-DOS commands at any time and exit or restart PolyShell without rebooting. MS-DOS programmers — discover what you have been missing! UNIX programmers — switch to MS-DOS painlessly! PolyShell and PolyAWK are each \$99 when ordered separately. Save \$50 by ordering the PolyShell + PolyAWK combination package for \$149. *Not copy-protected.*

30-Day

Money Back Guarantee

Credit Card Orders:

1-800-547-4000

Ask for Dept. TTX

Send Checks and P.O.s To:

POLYTRON Corporation

1700 NW 167th Place, Beaverton, OR 97006

(503) 645-1150 — FAX: (503) 645-4576

language like C or Pascal each time such a task comes up. With PolyAWK, you can handle such tasks with very short programs, often only one or two lines long.

Prototype With PolyAWK, Translate To Another Language

The brevity of expression and convenience of operations make PolyAWK valuable for prototyping even large-sized programs. You start with a few lines, then refine the program, experimenting with designs by trying alternatives until you get the desired result. Since programs are short, it's easy to get started and easy to start over when experience suggests a different direction. PolyAWK has even been used for software engineering courses because it's possible to experiment with designs much more readily than with larger languages. It's straightforward to translate a PolyAWK program into another language once the design is right.

Very Concise Code

Where program development time is more important than run time, AWK is hard to beat. These AWK characteristics let you write short and concise programs:

- The implicit input loop and the pattern-action paradigm simplify and often entirely eliminate control flow.
- Field splitting parses the most common forms of input, while numbers and strings and the coercions between them handle the most common data types.
- Associate arrays use ordinary strings as the index in the array and offer an easy way to implement a single-key database.
- Regular expressions are a uniform notation for describing patterns of text.
- Default initialization and the absence of declarations shorten programs.

Large Model Implementation

PolyAWK is a large model implementation and can use all of available memory to run big programs or read files greater than 64K.

Math Support

PolyAWK also includes extensive support for math functions such as strings, integers, floating point numbers and transcendental functions (sin, log, etc.) for scientific applications. Conversion between these types is automatic and always optimized for speed without compromising accuracy.

POLYTRON

High Quality Software Since 1982

®

DEVICE

continued from page 36

sion of the predefined file **OutPut** ON and OFF. To turn it ON, set both pointers to the address of the new **UsrOutput** procedure. To turn it OFF, restore the original values.

Converting with a ConOut replacement. The **UsrOutput** procedure is a model for any program that is converted from a 3.0 program by using a **ConOut** replacement. **UsrOutput** sends each character in the file's temporary buffer to the **ConOut** procedure. **UsrOutput** always returns the 0 value of success. The output portion of the serial driver in Chapter 26 of the *Turbo Pascal Owner's Handbook* works in the same way.

The **ConOut** procedure performs the real work of "backward writing." **ConOut** writes a character to the screen through a call to BIOS interrupt 10H, and then moves the cursor to the left of the character that was just written. **ConOut** can't simply use a **Write** statement to write the character,

since each **Write** calls the **UsrOutput** routine, which in turn calls **ConOut**. Since this type of endless loop is the stuff of which system crashes are made, **ConOut** makes a BIOS call to interrupt 10H instead. In addition to writing a character and moving the cursor, **ConOut** also handles a few special characters—ASCII carriage return, line feed, backspace, and bell. To scroll the screen, **ConOut** deletes the top line and repositions the cursor.

Program **Backward** (Listing 5) demonstrates **BackUnit**. Notice that writing in a forward direction is much faster than writing backward. This is because Turbo Pascal uses direct video memory I/O whenever you **USE** the **CRT** unit, whereas **ConOut** always uses BIOS calls. (Of course, you could modify the **ConOut** routine to use direct video I/O. BIOS calls are used here because they do most of the work for you.) Again, **BackUnit** is merely an example of how to modify a standard file, optimiz-

ing a TFDD before it proves itself truly useful to you would be premature.

IN THE DRIVER'S SEAT

Turbo Pascal 4.0's text file device drivers offer many opportunities for specialized input and output. These drivers replace Turbo Pascal 3.0's user-written I/O drivers, but give you considerably more control. TFDDs can be entirely new files or devices, or they can replace standard files or devices with new ones that are customized to fit your needs. With a little imagination, you can divert a text stream to any reasonable destination—and make it more than just a series of characters marching from here to there. ■

Neil Rubenking is a professional Pascal programmer and writer. He is a Contributing Editor for PC Magazine, and can be found daily on Borland's CompuServe Forum answering Turbo Pascal questions.

Listings may be downloaded from CompuServe as TFDD.ARC.



Discover Paradise

Programmer's Paradise Gives You Superb Selection, Personal Service and Unbeatable Prices!

Welcome to Paradise. The microcomputer software source that caters to your programming needs. Discover the Many Advantages of Paradise...

- Lowest price guaranteed
- Huge inventory, latest versions
- Technical support
- Immediate shipment
- 30-day money-back guarantee*
- Knowledgeable sales staff

Over 500 brand-name products in stock—if you don't see it, call!

We'll Match Any Nationally Advertised Price.

LIST OURS	LIST OURS	LIST OURS	LIST OURS
ARTIFICIAL INTELLIGENCE	300 199	FINALLY!	99 90
MULISP-87 INTERPRETER	95 86	FLASH-UP	89 80
PC SCHEME	100 85	FLASH-UP TOOLBOX	49 46
SMALLTALK/V	50 45	GRAPHPAK	69 60
EGA/VGA COLOR OPTION	50 45	MICROHELP UTILITIES	59 49
GOODIES DISKETTE	50 45	PEKS & POKES	45 39
SMALLTALK/COMM	495 435	QBASE	99 90
TI PROCEDURE CONSULTANT	150 109	QBASE REPORT	69 59
TURBO PROLOG V.2.0	100 69	QUICKBASIC	99 69
TURBO PROLOG TOOLBOX	100 90	QUICK-TOOLS	130 111
VP-EXPERT		QUICKPAK	69 60
		QUICKPAK II	49 45
ASSEMBLY LANGUAGE		QUICKWINDOWS	99 90
EZ_ASM	70 66	TRUE BASIC	100 90
MS MACRO ASSEMBLER	150 99	W/RUNTIME	150 135
OPTASM	195 172	TURBO BASIC	100 69
THE VISIBLE COMPUTER:8088	80 66	DATABASE TOOLBOX	100 69
THE VISIBLE COMPUTER:80286	100 90	EDITOR TOOLBOX	100 69
TURBO EDITASM	99 86	TELECOM TOOLBOX	100 69
BASIC		C LANGUAGE	175 137
DB/LIB	139 121	C ASYNCH MANAGER	139 121
EXIM SERVICES TOOLKIT	100 90	C-TERP FOR TURBO C	

TURBO POWER SCREEN
NEW powerful screen management for Turbo Pascal 4.0. Reliable, lightning fast data entry screens and menus to create your own sophisticated window oriented applications. Design and maintain screens and menus exactly as you want them to appear in your final application.

LIST: \$129 OURS: \$101

TURBO HALO	100 86	TURBO PASCAL	100 69
TURBO WINDOW/C	95 80	TURBO PLUS	100 89
PASCAL LANGUAGE		TURBO POWER SCREEN	129 101
ASCII TURBO PROGRAMMER	289 259	TURBO POWER UTILITIES	95 79
AZATAR DOS TOOLKIT	75 70	TURBO PROFESSIONAL 4.0	99 80
DOS/BIOS & MOUSE TOOLS	75 70	TURBO WINDOW PASCAL	95 80
MACH 2	100 90	UNIVERSAL GRAPHICS LIBRARY	150 121
METRABYTE D/A TOOLS	100 90		
OVERLAY MANAGER	45 40	OTHER LANGUAGES	
W/SOURCE CODE	90 80	LAHEY PERSONAL FORTRAN 77	95 86
SCIENCE & ENGINEERING TOOLS	75 69	LOGITECH MODULA-II COMP PACK	99 81
SCREEN SCULPTOR	125 95	MICROFOCUS PERSONAL COBOL	149 121
SYSTEM BUILDER	150 131	PC/FORTH	150 109
IMPEX	100 90		
REPORT BUILDER	130 116	UTILITIES	
T-DEBUG PLUS	45 39	DAN BRICKLIN'S DEMO PROGRAM	75 59
W/SOURCE CODE	90 80	DAN BRICKLIN'S DEMO PROG. II	195 179
TURBO ADVANTAGE	50 45	FANSI CONSOLE	75 66
TURBO ADVANTAGE COMPLEX	90 80	FETCH	55 49
TURBO ADVANTAGE DISPLAY	70 66	MACE UTILITIES	99 90
TURBO ANALYST	75 59	NORTON COMMANDER	75 56
TURBO.ASM	99 70	NORTON EDITOR	75 70
TURBO ASYNCH PLUS	129 101	NORTON UTILITIES	100 61
TURBO GEOMETRY LIBRARY	150 140	NORTON ADVANCED UTILITIES	150 101
TURBO HALO	95 80	NORTON GUIDES	100 65
TURBO MAGIC	99 90		

BORLAND PRODUCTS

EUREKA	167 119
REFLEX: THE ANALYST	150 109
SIDEKICK	85 59
SIDEKICK+	200 139
SUPERKEY	100 69
TURBO BASIC COMPILER	100 69
TURBO BASIC DATABASE	100 69
TURBO BASIC EDITOR TB	100 69
TURBO BASIC TELECOM TB	100 69
TURBO C	100 69
TURBO LIGHTNING AND	
LIGHTNING WORD WIZARD	150 109
TURBO PASCAL	100 69
TURBO PASCAL DBASE TOOLBOX	100 69
TURBO PASCAL DEV. TOOLKIT	395 289
TURBO PASCAL EDITOR TOOLBOX	100 69
TURBO PASCAL GAMEWORKS TB	100 69
TURBO PASCAL GRAPHIX TB	100 69
TURBO PASCAL NUM. METHODS	100 69
TURBO PASCAL TUTOR	70 45
TURBO PROLOG COMPILER	150 109
TURBO PROLOG TOOLBOX	100 69

Call or Write for Latest Free Catalog!

1-800-445-7899

In NY: 914-332-4548

Customer Service:
914-332-0869

International Orders:
914-332-4548

Telex: 510-601-7602

Programmer's Paradise™

A Division of Hudson Technologies, Inc.
42 River Street, Tarrytown, NY 10591



Terms and Policies

*We honor MC, VISA, AMERICAN EXPRESS

No surcharge on credit card or C.O.D. Payment by check New York State residents add applicable sales tax. Shipping and handling \$3.50 per item within the U.S. sent UPS ground. Rush and international service available. Call for prevailing rates.

*Programmer's Paradise will match any current nationally advertised price with exceptions terms for the products listed in this ad.

*Prices and Policies subject to change without notice.

*Hours 9AM EST - 7PM EST

*Mail Orders include your phone number

*Ask for details. Some manufacturers will not allow returns once disk seals are broken.

Dealers and Corporate Buyers—Call for special discounts and benefits!

LISTING 1: TEXTREC.DEF

```

TYPE
CharBuf = array[0..127] of char;
TextRec = RECORD
  Handle      : Word;
  Mode       : Word;
  BufSize    : Word;
  Private    : Word;
  BufPos     : Word;
  BufEnd     : Word;
  BufPtr     : ^CharBuf;
  OpenFunc   : pointer;
  InOutFunc  : pointer;
  FlushFunc  : pointer;
  CloseFunc  : pointer;
  UserData   : Array[1..16] of byte;
  Name       : Array[0..79] of char;
  Buffer      : CharBuf;
END;

( File mode magic numbers )
CONST
fmClosed = $D7B0;
fmInput  = $D7B1;
fmOutput = $D7B2;
fmInOut  = $D7B3;

```

LISTING 2: USRFILE.PAS

```

($R-,I+,N-)
PROGRAM UsrFile;

USES Crt, DOS;

CONST
  UsrSiz      = 255;
  IO_NotOutput = 105;
  IO_FileFull = 101;
  IO_Invalid  = 6;

TYPE
String255 = STRING[255];
CharBuf = ARRAY[0..127] OF Char;
FakeFile = ARRAY[0..UsrSiz] OF Char;
TextRec = RECORD
  Handle      : Word;
  Mode       : Word;
  BufSize    : Word;
  Private    : Word;
  BufPos     : Word;
  BufEnd     : Word;
  BufPtr     : ^CharBuf;
  OpenFunc   : pointer;
  InOutFunc  : pointer;
  FlushFunc  : pointer;
  CloseFunc  : pointer;
  UFilePos   : Word;
  UFileSiz   : Word;
  Data       : ^FakeFile;
  UserData   : ARRAY[1..8] OF Byte;
  Name       : ARRAY[0..79] OF Char;
  Buffer      : CharBuf;
END;

VAR
  UFile : Text;
  CH    : Char;
  N, D  : Integer;

($F+)(Start making all routines FAR)

FUNCTION UsrOpen(VAR F : TextRec) : Integer;

BEGIN
  UsrOpen := 0;
  WITH F DO
    IF Mode = fmOutput THEN
      BEGIN
        UFileSiz := 0;
        UFilePos := 0;
      END
    ELSE UsrOpen := IO_Invalid;
  END;
END;

FUNCTION UsrClose(VAR F : TextRec) : Integer;

BEGIN
  UsrClose := 0;
END;

FUNCTION UsrOutput(VAR F : TextRec) : Integer;

BEGIN
  UsrOutput := 0;
  WITH F DO
    IF Mode = fmOutput THEN

```

```

      BEGIN
        IF UFilePos+BufPos >= UsrSiz THEN UsrOutput := IO_FileFull
        ELSE
          BEGIN
            Move(BufPtr^, Data[UFilePos], BufPos);
            UFilePos := UFilePos+BufPos;
            IF UFilePos > UFileSiz THEN UFileSiz := UFilePos;
            BufPos := 0;
          END;
        END;
      ELSE
        IF Mode = fmClosed THEN UsrOutput := IO_NotOutput
        ELSE UsrOutput := IO_Invalid;
      END;
END;

($F-)(Stop making all routines FAR)

FUNCTION ReadUsr(VAR F : Text) : String255;

VAR Temp : String255;

BEGIN
  WITH TextRec(F) DO
    BEGIN
      Move(Data^, Temp[1], UFileSiz);
      Temp[0] := Chr(UFileSiz);
      UFileSiz := 0;
      UFilePos := 0;
    END;
    ReadUsr := Temp;
  END;

PROCEDURE AssignUsr(VAR F : Text);

BEGIN
  WITH TextRec(F) DO
    BEGIN
      Mode      := fmClosed;
      BufSize   := 127;
      BufPtr    := @Buffer;
      OpenFunc  := @UsrOpen;
      CloseFunc := @UsrClose;
      InOutFunc := @UsrOutput;
      FlushFunc := @UsrOutput;
      Name[0]   := #0;
      UFileSiz  := 0;
      UFilePos  := 0;
      New(Data);
    END;
  END;

BEGIN
  ClrScr;
  Write('Now writing several variables to "UFile" -- ');
  WriteLn('they will become a single STRING.');
```

LISTING 3: MEMFILE.PAS

```

($R-,I+,N-)
PROGRAM Memory_File;

USES Crt, DOS;

CONST
  BufSiz      = 127;
  UsrSiz      = 4095;
  IO_Invalid  = 6;
  IO_FileFull = 101;
  IO_NotOpen  = 103;
  IO_NotInput = 104;
  IO_NotOutput = 105;

```

```

TYPE
  FFileBuffer = ARRAY[0..UsrSiz] OF Char;
  FFilePointer = ^FFileBuffer;
  CharBuf = ARRAY[0..BufSiz] OF Char;
  TextRec = RECORD
    Handle      : Word;
    Mode       : Word;
    BufSize    : Word;
    BufPos     : Word;
    Private    : Word;
    BufEnd     : Word;
    BufPtr     : ^CharBuf;
    OpenFunc   : pointer;
    InOutFunc  : pointer;
    FlushFunc  : pointer;
    CloseFunc  : pointer;
    UFilePos   : Word;
    UFileSiz  : Word;
    FileData  : FFilePointer;
    Erased    : boolean;
    UserData  : ARRAY[1..7] OF Byte;
    Name      : ARRAY[0..79] OF Char;
    Buffer     : CharBuf;
  END;

VAR
  UsrFile : Text;
  line    : STRING[255];

($F+) (Make routines with FAR calls from here on)
FUNCTION UsrIgnore(VAR F : TextRec) : Integer;

BEGIN
  UsrIgnore := 0;
END;

FUNCTION UsrInput(VAR F : TextRec) : Integer;

  FUNCTION Min(A, B : Word) : Word;
  BEGIN
    IF A < B THEN Min := A ELSE Min := B;
  END;

  BEGIN
    UsrInput := 0;
    WITH F DO
      IF Mode = fmClosed THEN UsrInput := IO_NotOpen
      ELSE IF Mode = fmInput THEN
        BEGIN
          IF UFilePos >= UFileSiz THEN
            BEGIN
              BufEnd := 0;
              BufPos := 0;
            END
          ELSE
            BEGIN
              BufEnd := Min(UFileSiz-UFilePos, BufSiz);
              Move(FileData^ [UFilePos], BufPtr^, BufEnd);
              UFilePos := UFilePos+BufEnd;
              BufPos := 0;
            END;
          END
        ELSE IF Mode = fmOutput THEN UsrInput := IO_NotOutput
        ELSE UsrInput := IO_Invalid;
      END;

FUNCTION UsrOutput(VAR F : TextRec) : Integer;

BEGIN
  UsrOutput := 0;
  WITH F DO
    IF Mode = fmClosed THEN UsrOutput := IO_NotOpen
    ELSE IF Mode = fmOutput THEN
      BEGIN
        IF UFilePos+BufPos >= UsrSiz THEN UsrOutput := IO_FileFull
        ELSE
          BEGIN
            Move(BufPtr^, FileData^ [UFilePos], BufPos);
            UFilePos := UFilePos+BufPos;
            IF UFilePos > UFileSiz THEN UFileSiz := UFilePos;
            BufPos := 0;
          END;
        END
      ELSE IF Mode = fmInput THEN UsrOutput := IO_NotInput
      ELSE UsrOutput := IO_Invalid;
    END;

FUNCTION UsrOpen(VAR F : TextRec) : Integer;

BEGIN
  UsrOpen := 0;
  WITH F DO
    IF Mode = fmInput THEN
      (* ===== *)
      (* RESET : open for input from the *)
      (* "file". If size is 0, say the file *)
      (* doesn't exist. Otherwise, set InOut *)
      (* for INPUT and put the FilePos at 0. *)
      (* ===== *)

```

```

      BEGIN
        IF erased THEN UsrOpen := IO_NotInput
        ELSE
          BEGIN
            FlushFunc := @UsrIgnore;
            InOutFunc := @UsrInput;
            UFilePos := 0;
          END;
        END
      ELSE IF Mode = fmOutput THEN
        (* ===== *)
        (* REWRITE -- open for output TO the *)
        (* "file". Set FileSize and FilePos to *)
        (* 0 and allocate space for the file's *)
        (* data to reside in *)
        (* ===== *)
        BEGIN
          UFileSiz := 0;
          UFilePos := 0;
          IF erased THEN
            BEGIN
              erased := false;
              New(FileData);
            END;
            InOutFunc := @UsrOutput;
            FlushFunc := @UsrOutput;
          END
        ELSE IF Mode = fmInOut THEN
          (* ===== *)
          (* APPEND -- if the file doesn't exist *)
          (* yet, say so. Otherwise, point the *)
          (* FilePos at the FileSize, so new *)
          (* WRITE statements will append to the *)
          (* "file" *)
          (* ===== *)
          BEGIN
            IF erased THEN UsrOpen := IO_NotOutput
            ELSE
              BEGIN
                UFilePos := UFileSiz;
                InOutFunc := @UsrOutput;
                FlushFunc := @UsrOutput;
                Mode := fmOutput;
              END;
            END
          ELSE UsrOpen := IO_Invalid;
        END;

($F-) (Stop making routines with FAR calls)

PROCEDURE EraseUsr(VAR F : Text);

BEGIN
  WITH TextRec(F) DO
    BEGIN
      erased := true;
      dispose(FileData);
      mode := fmClosed;
    END;
  END;

PROCEDURE AssignUsr(VAR F : Text);

BEGIN
  WITH TextRec(F) DO
    BEGIN
      Mode      := fmClosed;
      BufSize   := BufSiz;
      BufPtr    := @buffer;
      OpenFunc  := @UsrOpen;
      CloseFunc := @UsrIgnore;
      Name[0]   := #0;
      UFileSiz := 0;
      UFilePos := 0;
      Erased    := true;
    END;
  END;

BEGIN
  ClrScr;
  AssignUsr(UsrFile);
  Rewrite(UsrFile);
  Write(UsrFile, 'I ');
  Write(UsrFile, 'am ', 1.234, ' feet high. ');
  WriteLn(UsrFile);
  WriteLn(UsrFile, 'The value of pi is ', Pi:1:11);
  Close(UsrFile);
  Reset(UsrFile);
  WriteLn
    ('I have written some lines to the "fake file". I can get them!');
  WriteLn
    ('back by READING the "fake file" and writing to the screen. ');
  WriteLn('HERE they come:');
  WHILE NOT(Eof(UsrFile) OR KeyPressed) DO
    BEGIN
      ReadLn(UsrFile, line);
      WriteLn(line);
    END;
  END;

```

```

WriteLn;
WriteLn('Now going to APPEND -- ...');
Append(UsrFile);
WriteLn(UsrFile, 'What is 1/4 of pi? Is it ', Pi/4:1:11, '?');
Close(UsrFile);
Reset(UsrFile);
WHILE NOT(Eof(UsrFile) OR KeyPressed) DO
  BEGIN
    ReadLn(UsrFile, line);
    WriteLn(line);
  END;
END.

```

LISTING 4: BACKUNIT.PAS

```

UNIT BackUnit;

Interface

Uses Crt, Dos;

PROCEDURE Backward_On;
PROCEDURE Backward_Off;

Implementation

VAR
  SaveExit : Pointer;
  HoldOutput, HoldFlush : pointer;

PROCEDURE ConOut(C : Char);

CONST
  CR = #0D; {carriage return}
  LF = #0A; {line feed}
  BEL = #7; {bell character}
  BKS = #8; {backspace}

VAR
  regs : Registers;
  X, Y : Byte;

BEGIN
  WITH regs DO
    BEGIN
      CASE C OF
        CR : GotoXY(80, WhereY);
        LF : BEGIN
              Y := succ(WhereY);
              IF Y > 25 THEN
                BEGIN
                  GotoXY(1,1);
                  DelLine;
                  Y := 25;
                END;
              GotoXY(WhereX, Y);
            END;
        BEL : BEGIN Sound(750); Delay(300); NoSound; END;
        BKS : BEGIN
              X := WhereX;
              IF X < 80 THEN
                GotoXY(succ(X), WhereY);
              END;
            ELSE
              (just write the character )
              AH := 9;
              AL := Ord(C);
              BH := 0;
              BL := TextAttr;
              CX := 1;
              Intr($10, regs);

              {now reposition the cursor}
              X := WhereX; Y := WhereY;
              IF X > 1 THEN Dec(X)
            ELSE
              BEGIN
                X := 80;
                Inc(Y);
              END;
              IF Y > 25 THEN
                BEGIN
                  GotoXY(1,1);
                  DelLine;
                  Y := 25;
                  X := 80;
                END;
              GotoXY(X, Y);
            END; {CASE}
          END;
        (PROCEDURE ConOut(C : Char);)
      END;

  ($F+) FUNCTION UsrOutput(VAR F : TextRec) : Integer; ($F-)

VAR N : Byte;

BEGIN
  WITH F DO
    BEGIN

```

```

FOR N := 0 TO Pred(BufPos) DO ConOut(BufPtr[N]);
BufPos := 0;
END;
UsrOutput := 0;
END;

```

```
PROCEDURE Backward_On;
```

```

BEGIN
  TextRec(Output).InOutFunc := @UsrOutput;
  TextRec(Output).FlushFunc := @UsrOutput;
END;

```

```
PROCEDURE Backward_Off;
```

```

BEGIN
  TextRec(Output).InOutFunc := HoldOutput;
  TextRec(Output).FlushFunc := HoldFlush;
END;

```

```
($F+)PROCEDURE MyExitProc;($F-)
```

```

BEGIN
  Backward_Off;
  ExitProc := SaveExit;
END;

```

```
{Initialization section}
```

```

BEGIN
  SaveExit := ExitProc;
  ExitProc := @MyExitProc;
  HoldOutput := TextRec(Output).InOutFunc;
  HoldFlush := TextRec(Output).FlushFunc;
END.

```

LISTING 5: BACKWARD.PAS

```

($R-, I+, N-)

PROGRAM Backward;

Uses Crt, Dos, BackUnit;

VAR
  dummy : char;

PROCEDURE Write_Explanation;

BEGIN
  WriteLn(' BACKWARD WRITING DEMO');
  WriteLn(' =====');
  WriteLn(' This program demonstrates a Text');
  WriteLn(' File Device Driver that replaces the');
  WriteLn(' standard driver for the OUTPUT ');
  WriteLn(' device in Turbo Pascal 4.0. When the');
  WriteLn(' replacement TFDD is activated, ');
  WriteLn(' all "Write" and "WriteLn" statements ');
  WriteLn(' will appear on the screen backward. ');
  WriteLn;
END;

BEGIN
  TextColor(LightGray); TextBackground(black);
  ClrScr;
  Write_Explanation;
  WriteLn('--PRESS a key to activate BACKWARD');
  dummy := ReadKey;

  Backward_On;
  TextColor(White);
  GotoXY(80,1);
  Write_Explanation;
  WriteLn('--PRESS a key to DEactivate BACKWARD');
  dummy := ReadKey;
  Backward_Off;
  GotoXY(1,14);
  Write(' The TextColor and TextBackground ');
  WriteLn('procedures work just as they normally');
  Write(' would, and text scrolls when it ');
  WriteLn('reaches the bottom of the screen. Press');
  Write(' a key to demonstrate this, and ');
  WriteLn('press a key again to stop. ');
  Dummy := ReadKey;
  Backward_On;
  WriteLn;

  REPEAT
    TextColor(random(16));
    TextBackground(random(8));
    Write('Press any key to STOP....');
  UNTIL KeyPressed;
  dummy := ReadKey;
  Backward_Off;
END.

```

MOUSE MYSTERIES, PART II: GRAPHICS

Add the graphics touch to your Turbo C and Turbo Pascal applications by combining the BGI with your mouse.

Kent Porter



PROGRAMMER

In the first article of this two-part series on mouse programming (see "Mouse Mysteries, Part I: Text," *TURBO TECHNIX*, May/June, 1988), I used a Turbo Pascal unit and a Turbo C library to explore how a Microsoft/Logitech-compatible mouse can be incorporated into the user interface of text-oriented programs. In this final article of the series, I'll discuss the techniques of mouse programming in graphics mode. While the calls and many of the programming techniques are the same for both the text and the graphics mode, you'll discover that there are some dramatic differences in the capabilities of the mouse, especially with respect to cursor management.

In Part I, I discussed the two kinds of mouse text cursors. Type 0 (in the call to `mTextCursor`) creates a software cursor whose appearance can be customized; this cursor operates independently of the display adapter's normal text cursor. Mouse cursor type 1, on the other hand, places the hardware cursor under the control of the mouse, although the call to `mTextCursor` doesn't actually alter the location of the next text input/output operation.

With mouse programming in graphics mode, we meet a third type of mouse cursor. This cursor is similar in some respects to the software text cursor (Type 0), but more flexible. The graphics cursor also has a new characteristic, known as the *hot spot*, which is the exact location of the cursor's point of registration.

The graphics cursor is a 16×16 -bit object that tracks in response to the mouse. Unlike the cursor in text mode, the graphics cursor doesn't jump a character cell at a time; instead, it moves smoothly, pixel-by-pixel. As you move the mouse, the cursor is repeatedly erased at its present position and redrawn at the next location. The redraw operation occurs so rapidly that the eye doesn't detect it; the cursor appears to move continuously.

Different graphics modes interpret combinations of bits differently. For example, in CGA high-resolution (640×200 monochrome) graphics, each bit is mapped to a single pixel. Therefore, a 16×16 -bit object is 16 pixels wide by 16 pixels high. On the other hand, CGA four-color mode (320×200) uses two adjacent bits to select a color from the active palette, so that a 16×16 -bit data object becomes a visual object that is 8 pixels wide by 16 pixels high, with each pixel's color dependent upon the two-bit pattern. These different interpretations of bit combinations have implications that I'll discuss later in this article. The important thing to remember right now is that a mouse cursor's workspace is 16×16 bits, which (depending on the mode) might be 8 or 16 pixels wide. Incidentally, the physical proportions don't change between these modes, since one pixel in four-color mode is the same physical width as two pixels in high-resolution mode.

DEFINING THE CURSOR MASKS

A graphics mouse cursor consists of two related bit maps (one superimposed on the other), called the cursor mask and the screen mask. The *cursor mask* defines the appearance of the cursor itself, while the *screen mask* specifies how the underlying pixels are treated. The mouse cursor is always *in front* of whatever is on the display, so that the screen mask selectively passes through or blocks background information as the cursor moves across fixed objects. Most often, the screen mask blocks the background pixels all around the cursor shape to create a border, thus making the cursor visible even when it's in front of an object of the same color.

Specifically, the mouse device driver ANDs the screen mask with the 16×16 -bit display area where the cursor is being placed. Then the mouse device driver XORs the cursor mask with the result of the AND. This process produces the screen display that is summarized in Table 1.

SCREEN MASK BIT	CURSOR MASK BIT	DISPLAY BIT
0	0	0
0	1	1
1	0	Unchanged
1	1	Inverted

Table 1. Effects of the screen mask and the cursor mask on the display area.

If a screen bit is 1 and the corresponding bit in the cursor mask is 0, the background bit *shines through*. A 0/0 combination in corresponding positions blocks the background bit, while a 0/1 forces the display bit ON. Although the 1/1 combination is seldom used, we'll see an application for it later in this article in connection with the I-beam shape.

Given these rules, let's see what a pair of masks actually looks like in binary. The mask set shown in Figure 1 defines an arrow pointing northwest, as the eye can discern from the patterns of 1s and 0s.

Hex	Screen mask	Cursor mask	Hex
3FFF	0011111111111111	0000000000000000	0000
1FFF	0001111111111111	0100000000000000	4000
0FFF	0000111111111111	0110000000000000	6000
07FF	0000011111111111	0111000000000000	7000
03FF	0000001111111111	0111100000000000	7800
01FF	0000000111111111	0111110000000000	7C00
00FF	0000000011111111	0111111000000000	7E00
007F	0000000001111111	0111111100000000	7F00
003F	0000000000111111	0111111110000000	7F80
001F	0000000000011111	0111111111000000	7FC0
01FF	0000000111111111	0111110000000000	7C00
10FF	0001000011111111	0100011000000000	4600
30FF	0011000011111111	0000011000000000	0600
F87F	1111100001111111	0000001100000000	0300
F87F	1111100001111111	0000001100000000	0300
FC3F	1111110000111111	0000000110000000	0180

Figure 1. Values for the screen mask and the cursor mask.

Note that the screen mask describes an outline for the cursor mask; the region of 0s is one bit greater in all directions than is the shape defined by 1s in the cursor. For example, the first three screen mask rows are as follows:

```
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1
```

In contrast, the first three rows for the cursor mask are shown below:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Thus, the tip of the arrow cursor is located in the second column of the second row. The screen mask blocks the background pixels above and to both sides of the cursor with a 0/0 combination.

This northwest-pointing arrow is the *default graphics cursor*. If you don't specify otherwise, this is the cursor that is automatically shown in a graphics mode. A custom cursor can always be replaced by the

PINPOINTING THE CURSOR'S LOCATION

In text mode, the mouse cursor always occupies one character cell. Since the text mouse moves one full cell at a time, the cursor is either in or out of a given cell. In Part I of this series, I showed that position inquiry calls to the device driver return coordinates that correspond to the CGA high-resolution graphics screen, which measures 640 × 200. The cursor is mapped to a character position by using the algorithm

```
mRowRange (0, GetMaxY);
```

along both axes to determine the text row and column values.

The granularity of a graphics screen is much finer than that of a screen in text mode. However, since the cursor spans an area of either 8 × 16 or 16 × 16 pixels, a problem arises: How can we identify the precise location of the cursor? The solution is the cursor's hot spot.

The *hot spot* is a pixel position within the mask that maps to a single pixel on the display. The device driver ascertains the cursor position by locating and reporting the coordinates of the display pixel that is currently overlaid by the hot spot. In the case of the default (northwest-pointing) graphics cursor, this display pixel is located at the position represented by 1,1 (where 0,0 represents the upper left corner of the mask). In other words, this position is the very tip of the arrow, which is located in the second row and the second column of the mask. If the cursor was represented by a cross-hair, the hot spot would logically occur where the hairs intersect. If the cursor was represented by a pointing hand, the hot spot would be at the tip of the index finger.

Thus, the hot spot is a relative location defined with reference to the upper left corner of the cursor

continued on page 46

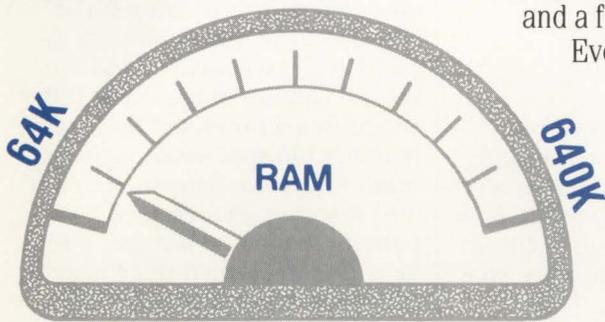
default graphics cursor by resetting the mouse and then showing the cursor again. It should never be necessary to specify this default mask set in your programs. (It's included here only to illustrate what a mask set looks like.)

I'll present several predefined cursor shapes shortly, and show you how to activate them. First, let's explore the graphics cursor's hot spot.

Lots of software packages help you work,

Moving ahead takes more than hard work, it takes smart work. There are stacks of productivity software you can buy for your PC. But to work smart, you only need one: SideKick® Plus. It's the newest member of Borland's professional series, from the same people who brought you the original SideKick: the program that introduced more than a million PC users to the convenience of using their computer as an organizing tool.

To buy productivity applications like those in SideKick Plus *separately*, you'd spend almost a thousand dollars and drain your computer's memory dry. SideKick Plus takes as little as



SideKick Plus puts you in control . . .
for as little as 64K!

64K of your computer's RAM; you decide exactly how much.

You can select just the productivity applications you need. Like a sophisticated telecommunications package, a powerful DOS manager, nine

notepads, a versatile outliner, four different calculators, support for both EMS and extended memory. And lots more.

You decide how to use SideKick Plus, too. Put your applications on your hard disk to call up when you need them, or leave them in RAM for instant availability. Either way, they're always at your fingertips. Accessible over any *other* application you're working in. Amazingly affordable. And very, very smart.

Here's What You Get!

- **The PhoneBook:** complete data and voice communications that you can set to take place in the background, with auto-dialing, an encrypted glossary, and a full Script language.

Even if you don't have a modem, it keeps your names, addresses, and phone numbers at your fingertips

- **Outlook:**
The Outline Processor: nine Outliners with automatic numbering, tree charts, and table of contents
- **The File Manager:** extended DOS file and directory management

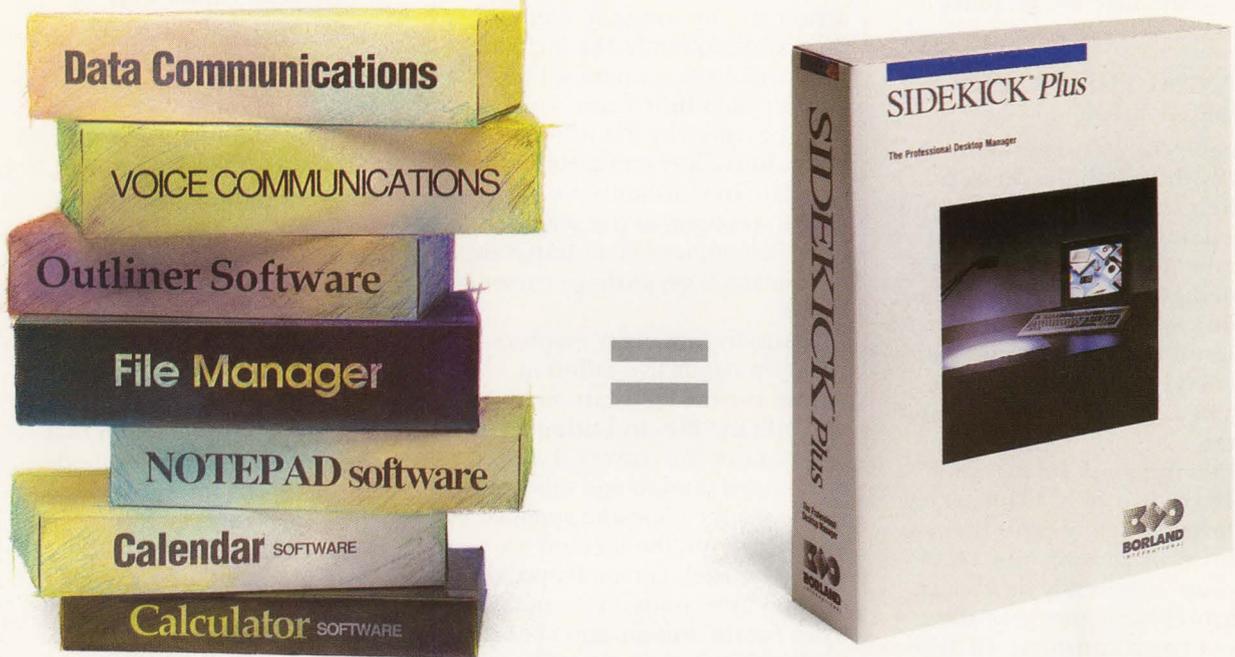
- **The Calculator:** four types: business, scientific, programmer and formula
- **The Clipboard:** for copy-and-paste integration between files and with other applications
- **The Time Planner:** includes a Calendar, Appointment Book, and Schedule window, plus alarms, repeating appointments, and attached agenda. Supports networks via a common calendar
- **The Notepad:** nine file-editor Notepads, up to 11,000 words each
- **The ASCII Table:** to find and paste characters quickly and easily
- **Supports both EMS and extended memory:** if you have expanded memory or the Intel Above™Board, you can load the SideKick Plus desk accessories into expanded memory and leave even more of your conventional memory for your other applications.

Only \$199.95
(not copy protected)

60-day money-back guarantee*

For a brochure, the dealer nearest you, or to order
Call (800) 543-7543

Only one helps you work smarter...



SideKick Plus!

“ What I like most is new ideas in software. Historically, one of the best new ideas was Borland’s SideKick. SideKick Plus is a much more powerful program than the original. It adds a lot of new features and gives you a lot more flexibility over how you use its many features.

Michael J. Miller, InfoWorld ”



continued from page 43

mask. The hot spot's range along the horizontal axis is 0.7 for CGA four-color mode, and 0.15 for all others (CGA high-resolution, Hercules, EGA, VGA, etc.). The hot spot's range along the vertical axis is always 0.15.

The mouse device driver needs to know the relative coordinates of the masks and the hot spot in order to manage the graphics cursor.

DEFINING THE GRAPHICS MOUSE CURSOR

Since the cursor's description is actually the combination of several items, it's convenient to group these items into a Pascal record or a C structure. This object (which is called **gCursRec** in the accompanying listings) consists of a pointer to the mask set and two unsigned integers that give the hot spot's relative X and Y coordinates.

While the mask set logically consists of two 16×16 -bit arrays, the device driver regards them as one array of 16×32 bits. These arrays are inherently fixed data. In Turbo Pascal, they take the form of typed constants. In Turbo C, they take the form of the static unsigned type. The arrays can be initialized with hex numbers that represent the bit patterns at definition.

Listings 1 and 2 include files for Turbo Pascal and Turbo C, respectively, that furnish five common shapes for mouse cursors: a check mark, a left-pointing arrow, a cross, a pointing hand, and an I-beam. The latter shape is usually associated with text I/O on the Macintosh and in Mac-like PC-based programs such as Xerox Ventura Publisher. The other shapes are used for pointing, drawing, selecting, and so forth.

Customarily, mouse-driven programs use different cursor shapes to identify the current mode. Such actions tend to be highly application-dependent.

The headings of these include files also initialize the **gCursRec** structures for each cursor type.

Note that the structure declarations cannot initialize the pointers during compilation; the pointers can only be set to **nil** (the **NULL** constant in Turbo C). This is because the addresses of the static arrays are not assigned until link time. Consequently, the include files contain equivalent subprograms called **InitGCurs**, which must be called by the using program. **InitGCurs** completes the structure initializations by plugging in pointers to the static arrays. Remember to call **InitGCurs**, or else you'll get garbage cursor forms.

Naturally, not every graphics program needs five different mouse cursors. You can rename and edit the files in Listing 1 and 2 to remove the cursors that you don't need (and to add others if you wish) for a specific application. However, the memory expense for each cursor shape is only 38 bytes (plus a few more bytes for the initialization code), so there's not a great penalty for including unused shapes.

The **GMOUSCUR** include files also furnish a mouse event handler. This handler is identical to those used for text mode in Part I of this article series. To install the handler, use the **mInstTask** call.

GETTING THE GRAPHICS CURSOR UP

The process of moving from a program's initial text mode to a mouse-driven graphics environment involves several steps. The following is a step-by-step guide.

1. Initialize the cursor descriptor structure(s) with a call to **InitGCurs**;
2. Reset the mouse with **mReset**; and

3. If the mouse exists, then:
 - a. Install the event handler with **mInstTask**;
 - b. Enter the desired graphics mode;
 - c. Build the environment display;
 - d. Tell the device driver which cursor shape you want via a call to **mGraphCursor** (skip this step if you're using the default arrow cursor);
 - e. Show the cursor with **mShow**;
 - f. Reset the mouse event flag to 0; and
 - g. Enter a loop to process mouse events.

The work of the program is done in substep g. This activity might involve drawing, text processing, file I/O, dialog boxes, and other operations. (As examples, consider PC Paintbrush and Microsoft Windows applications.) The loop must have an exit such as a menu or icon selection that can be clicked to graphics mode. The process of withdrawing is much simpler than that of setting up, and requires these steps:

- h. Reset the mouse with **mReset**; and
- i. Revert to text mode.

If a mouse isn't present in the system, none of the substeps a-i can occur (unless you use the cursor keys as an alternative to the mouse).

The **mGraphCursor** routine is furnished by the Pascal **MOUSE** unit and the C file **MOUSE.I**. This call takes four parameters:

- The X and Y relative coordinates of the hot spot; and
- The segment and offset of the mask set.

As mentioned earlier, the mask set is a 16×32 -bit array (i.e., a 32-word list). The first half of the array holds the screen mask, and

continued on page 48



A Deal You Can't Refuse ...700 Functions, 20 Disks, Free Software

Entelekon's C Business Library or C STARTER PACKAGE

FREE*
TURBO C^(R)
Borland

or

FREE*
QUICKCTM
Microsoft

or

FREE*
C MATH TOOL BOX
89 advanced
math/stat functions

*OR FREE REFUND if you already own one, see special offer (limited time)

What You Get With Entelekon Libraries



A C COMPILER without a good add-on library is like a PC without a keyboard...
it won't do what you want it to do!



GAIN C POWER Add capabilities your compiler library does NOT have. e.g.:

- ☛ New! Qwick Menuing—full 1-2-3 like menus & more
- ☛ Flexible powerful windowing + new Qwick windows
- ☛ Powerful cursor, video and attribute control
- ☛ Time and date arithmetic
- ☛ Sample code and working examples
- ☛ New! Qwick Data Entry with dialog boxes
- ☛ Formatted, fully validated data entry
- ☛ Display default field values
- ☛ Calculator style entry option
- ☛ 700 functions you need



SAVE TIME, TIME, TIME: man-years on development, calendar months on schedule!



SAVE MONEY: Lowest Cost, Highest Quality Library/Windows Available!



SMALLER PROGRAM SIZE: your application program can be up to 50% smaller!



EASY for beginners! **POWERFUL** for professionals!



INSTANT INSTALLATION UTILITY included!



SUPERB DOCUMENTATION: time saving, helpful, clear, complete, instructive.



BUSINESS USERS: FREE 3 machine site license (C Library & Power Windows).



FULL SOURCE CODE included! **NO ROYALTIES** on products you develop.



FREE UTILITY: To convert Turbo Pascal code to C code.

20 DISKS FULL
Source code, documentation
utilities, sample programs
FREE DEMO DISK AVAILABLE

SAVE MONEY!

SAVE TIME!

DON'T WAIT!

ORDER NOW!

SATISFACTION GUARANTEED

(Direct from Entelekon only)

CALL (713) 468-4412

POWER WINDOWSTM
MOST POWERFUL YET
POP-UP/PULL DOWN/OVERLAP
Menus/Overlays
Messages/Alarms
ZAP ON/OFF SCREEN
FILE-WINDOW MANAGEMENT
Horizontal & Vertical Scrolling
Word Wrap & Line Insertion
Cursor/Attributes/Borders

Full source code \$159.95

*** SPECIAL OFFER**

Free Turbo C or QuickC or C Math Tool Box with purchase of C Starter Package or C Business Library. Even if you already own Turbo C or QuickC or C Math Tool Box, we will refund up to the full purchase price of one of these packages with the purchase of C Starter Package or C Business Library.

C FUNCTION LIBRARY
BEST YOU CAN GET
OVER 500 FUNCTIONS
FULLY TESTED
BETTER FUNCTIONS

Full source code \$159.95

C BUSINESS LIBRARY
INCLUDES C FUNCTION LIBRARY, POWER WINDOWS, SUPERFONTS FOR C, B-TREE LIBRARY, ISAM

ALL for \$299.95
(A \$500.00 VALUE)

B-TREE LIBRARY & ISAM DRIVER

POWERFUL DATA MANAGER
FAST! EASY TO USE!
16.7 MILLION RECORDS/FILE
16.7 MILLION KEYS/FILE
Fixed/Variable length records.

Full source. No royalties \$129.95
Multi-User option available.

C STARTER PACKAGE

INCLUDES C FUNCTION LIBRARY, POWER WINDOWS, SUPERFONTS FOR C
ALL for \$199.95
(A \$370.00 VALUE)

EntelekonTM

SINCE 1982

12118 Kimberley, Houston, TX 77024

713-468-4412

VISA-MASTERCARD-CHECK-COD

the second half contains the cursor mask.

When the **mGraphCursor** routine is called, the mouse device driver immediately displays the cursor image and tracks the cursor's position using the hot spot. By issuing a single call when the program enters a new mode, cursor shapes can be instantly changed. This process is shown by the **demo** routine in Listings 3 and 4, which displays the name of the new mouse cursor and then calls **mGraphCursor** to change the cursor image (which is already visible by the time **demo** is called).

The GMICE programs are written in Turbo Pascal and Turbo C, using the MOUSE units presented in Part I and the include files given earlier in this article. Since the two GMICE programs are functionally identical (except for details of handling the mouse event), we can speak of them as one program.

GMICE uses CGA high-resolution mode, which is common to all IBM graphics display adapters (but not to the Hercules). The program displays a white block in the center of the screen. You can use GMICE to view the default cursor and the five cursors that are defined in **GMOUSCUR**, by moving the mouse to see the cursor against both light and dark backgrounds. Each time you click a mouse button, the cursor and its identifier change. The program ends and reverts to text mode when you click while the I-beam cursor is displayed.

MORE ABOUT MASKS

The I-beam cursor is unique among the shapes given here in that its screen mask consists entirely of one-bits. By combining the one-bits and the cursor mask, the background shows through when the cursor shape is the in-

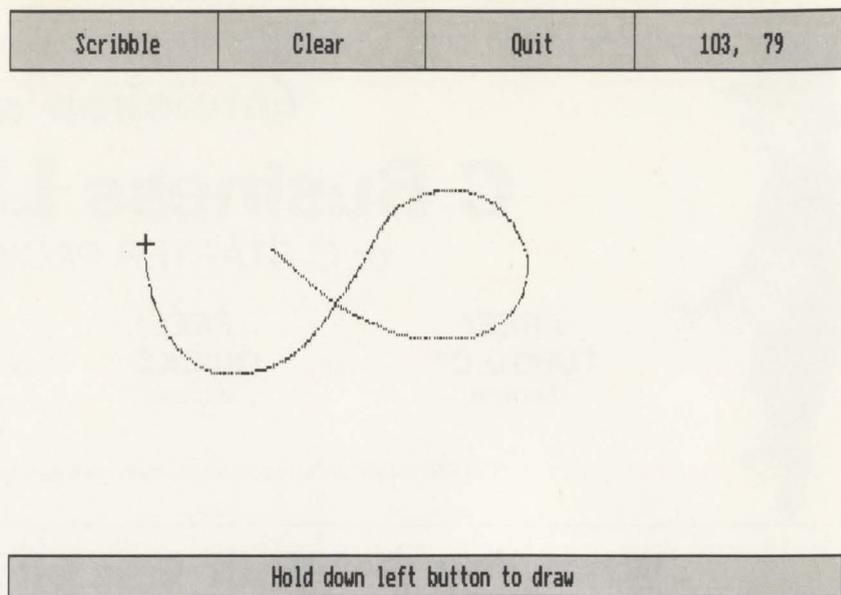


Figure 2. Screen display of the SCRIBBLE program.

verse of the background. No outlining occurs when the 0/0 combination is used because the I-beam is a spindly object that lacks any solid mass. An outline fattens and distorts the shape against a white background, so it works best to use the 1/1 inversion combination for this particular image.

As an experiment, select some of the other cursors with a cursor mask that consists entirely of one-bits. The results are interesting, but not as satisfactory. The use of the mouse in CGA four-color graphics (mode **CGAC1** or **CGAC2** with the BGI) is also unsatisfactory. Change the assignment of the mode variable to CGA four-color mode and try it; you'll see how the cursor picks up undesirable colors as side effects due to the two-bit pixels in this mode.

DRAWING WITH THE MOUSE

SCRIBBLE is a program that incorporates the principles I've discussed here. This simple drawing program (of the sort that forms the foundation for software packages such as PC Paintbrush) lets you do exactly what its name implies—scribble.

Figure 2 shows the display created by SCRIBBLE, which is a menu with the selections Scribble, Clear, and Quit. A meter in the upper right corner of the screen continually tracks the display position of the cursor's hot spot. The help box at the bottom of the screen contains instructions. When the program first displays on the screen, the cursor has the form of the pointing hand. To enter drawing mode, you move to the Scribble menu choice and click any button.

Once you've entered drawing mode, the cursor switches to the cross form. It can be moved around the large work area. To draw, hold down the left button and move the mouse. (This step is shown within the help box in Figure 2.)

When the Clear selection is clicked, the program clears the work area, restores the hand cursor, and redisplay the initial help message. This step resets the program so that you can start a new doodle. To end the program, click Quit.

Listings 5 and 6 show SCRIBBLE in Pascal and C. These are equivalent programs that include the appropriate version of GMOUSCUR and call on the MOUSE library routines presented in Part I of this article series.

The subprograms **MenuBox**, **Help**, and **SetUpScreen**, which draw upon the Turbo Pascal/C graphics library, handle the respective activities that their names suggest. **UpdateMeter** converts the numeric coordinates of the cursor's hot spot into text and then displays them in the meter box. The main body of SCRIBBLE initializes the environment and dispatches the **Work** subprogram, which controls the main operation of the program.

In Listing 5, **Work** consists chiefly of a large **CASE** statement (in Listing 6, **switch** is the equivalent C construct). This is enclosed inside a loop that repeats until the user clicks the **Quit** selection. After resetting the event flag, the loop waits for a mouse event to occur. When that event happens, **CASE** (in Turbo Pascal) or **switch** (in Turbo C) evaluates the event flag and takes action as appropriate. The event mask that is passed to the mouse driver is **\$55** in Pascal, or **0x55** in C. This mask triggers an event any time the mouse moves, or when any button is released. Since the sum of the case selectors is hex 55, they selectively act on any possible event.

The first event is **0x01**, which signifies a movement of the mouse. This case updates the meter to indicate the new cursor position. Before updating the meter, however, the case checks the status of the buttons. If the left button is down and the cursor is inside the work area (i.e., not in the menu or help areas), the routine draws a dot at the point that is covered by the hot spot. While

the pixel is being drawn, the cursor must be hidden in order for the display adapter to control the area under the cursor.

The second case is entered when any button release occurs. Since the release of a button may indicate a menu selection, this case checks if the cursor is inside the menu area. If so, the case examines the cursor's horizontal position to determine which selection the user has made. If the user chooses Scribble, the cursor switches to the cross image and displays the second help message; if the user selects Clear, the program resets the work area and restores the hand cursor and initial help message. If Quit is chosen, the program toggles the value of the **thru** variable to **TRUE**. The end of the loop checks this Boolean and repeats the process if the Boolean is still **FALSE**; otherwise, the program quits.

THE GRAPHICS MOUSE WITH THE EGA AND VGA

The mouse cursor's appearance is governed by the current graphics mode (as you already noticed if you switched to CGA four-color mode). In EGA and VGA modes, mouse cursors shrink along the horizontal axis because these modes place more Y units into the same physical screen area (350 or 480 units, versus 200 units in CGA). Thus, a cursor image that is tall and skinny in CGA high-resolution mode becomes relatively shorter and fatter as the number of Y coordinates increases.

Mouse vendors strive to keep up with evolving display technology by releasing new versions of software device drivers as new adapters come onto the market. Microsoft and Logitech currently ship device drivers that handle the common adapters up through VGA 640 × 480 multicolor graphics.

The latest released version of the Logitech driver at the time of

this writing is 3.4. For this article series, I first used Logitech version 3.2 on a VGA; however, I found the driver to be slightly unreliable in EGA modes, and useless for true VGA. The problem in EGA mode is that the cursor sometimes refuses to drop below row 199. The solution is simple: after entering graphics mode, issue the statement:

```
mRowRange (0, GetMaxY);
```

This apparently reassures the mouse driver that it's okay to use the area beyond that which CGA graphics provides. In VGA graphics, the mouse cursor fails to appear at all. Both problems disappear with the updated device driver.

Unlike DOS, the de facto standard Microsoft Mouse device driver has no function for ascertaining its own version level. (That's a shame, but that's how it is.) If you develop commercial software and you use a higher-resolution adapter, advise your customers to use a device driver that's up to the proper level.

As I said at the beginning of this series on mouse programming, it's not difficult to incorporate a mouse into your user interface. However, it is different because you have to think in terms of events and status checks. The small extra effort is very worthwhile, because a mouse can add tremendous power and ease of use to your software. The tools and techniques in this series should make it easier for you to develop superlative mouse-based user interfaces in Turbo Pascal and Turbo C. ■

Kent Porter is the author of Stretching Turbo Pascal and numerous other programming books. He is a frequent contributor to TURBO TECHNIQ.

Listings may be downloaded from CompuServe as CMOUS2.ARC.

LISTING 1: GMOUSCUR.INC

```

( GMOUSCUR.INC is an include file that defines several graphics )
( mouse cursor patterns. All are typed constants. )
( This file also contains an event handler called by the mouse )
( device driver, and a global event record variable 'theEvents' )
( ----- )

($F+) ( Force the including program to use far calls )

TYPE eventRec = record ( mouse event record )
    flag, button, col, row : WORD;
END;
gCursPtr = ^gCurs; ( pointer to cursor image )
gCurs = ARRAY [1..32] OF WORD; ( cursor image array )
gCursRec = record ( graphics cursor descriptor )
    image : ^gCursPtr;
    hotX, hotY : WORD;
END;

( Check mark image )
CONST checkIm : gCurs = ($FFF0, $FFE0, $FFC0, $FF81, ( screen mask )
    $FF03, $0607, $000F, $001F,
    $C03F, $F07F, $FFFF, $FFFF,
    $FFF, $FFF, $FFF, $FFF,
    $0000, $0006, $000C, $0018, ( cursor mask )
    $0030, $0060, $70C0, $1D80,
    $0700, $0000, $0000, $0000,
    $0000, $0000, $0000, $0000);

( Left arrow image )
LArrIm : gCurs = ($FE1F, $F01F, $0000, $0000, ( screen mask )
    $0000, $F01F, $FE1F, $FFFF,
    $FFF, $FFF, $FFF, $FFF,
    $FFF, $FFF, $FFF, $FFF,
    $0000, $00C0, $07C0, $7FFE, ( cursor mask )
    $07C0, $00C0, $0000, $0000,
    $0000, $0000, $0000, $0000,
    $0000, $0000, $0000, $0000);

( Cross image )
crossIm : gCurs = ($FC3F, $FC3F, $FC3F, $0000, ( screen mask )
    $0000, $0000, $FC3F, $FC3F,
    $FC3F, $FFF, $FFF, $FFF,
    $FFF, $FFF, $FFF, $FFF,
    $0000, $0180, $0180, $0180, ( cursor mask )
    $7FFE, $0180, $0180, $0180,
    $0000, $0000, $0000, $0000,
    $0000, $0000, $0000, $0000);

( Pointing hand image )
handIm : gCurs = ($E1FF, $E1FF, $E1FF, $E1FF, ( screen mask )
    $E1FF, $E000, $E000, $E000,
    $0000, $0000, $0000, $0000,
    $0000, $0000, $0000, $0000,
    $1E00, $1200, $1200, $1200, ( cursor mask )
    $1200, $13FF, $1249, $1249,
    $1249, $9001, $9001, $9001,
    $8001, $8001, $8001, $FFFF);

( I-beam image )
iBeamIm : gCurs = ($FFF, $FFF, $FFF, $FFF, ( screen mask )
    $FFF, $FFF, $FFF, $FFF,
    $FFF, $FFF, $FFF, $FFF,
    $FFF, $FFF, $FFF, $FFF,
    $F00F, $0C30, $0240, $0240, ( cursor mask )
    $0180, $0180, $0180, $0180,
    $0180, $0180, $0180, $0180,
    $0240, $0240, $0C30, $F00F);

( Graphics cursors )
check : gCursRec = (image : nil; hotX : 6; hotY : 7);
arrow : gCursRec = (image : nil; hotX : 0; hotY : 3);
cross : gCursRec = (image : nil; hotX : 7; hotY : 4);
hand : gCursRec = (image : nil; hotX : 5; hotY : 0);
iBeam : gCursRec = (image : nil; hotX : 7; hotY : 7);
( ----- )

VAR theEvents : eventRec; ( global variable )

PROCEDURE EventHandler
(Flags, CS, AX, BX, CX, DX, SI, DI, DS, ES, BP : WORD);

( Mouse event handler called by device driver )

INTERRUPT;

Begin
theEvents.flag := AX;
theEvents.button := BX;
theEvents.col := CX;
theEvents.row := DX;

inline ( ( Exit processing for far return to device driver )
    $8B/$E5/ ( MOV SP, BP )
    $5D/ ( POP BP )
    $07/ ( POP ES )
    $1F/ ( POP DS )
    $5F/ ( POP DI )
    $5E/ ( POP SI )
    $5A/ ( POP DX )
    $59/ ( POP CX )

```

```

    $5B/ ( POP BX )
    $5B/ ( POP AX )
    $CB); ( RETF )
END;
( ----- )

```

```
PROCEDURE InitGCurs;
```

```

( Initialize pointers in graphics cursor descriptors )
( Pointers can only be initialized at run time )

```

```

BEGIN
check.image := @checkIm;
arrow.image := @LArrIm;
cross.image := @crossIm;
hand.image := @handIm;
iBeam.image := @iBeamIm;
END;

```

```
( End of gmouscur.inc )
```

LISTING 2: GMOUSCUR.I

```

/* GMOUSCUR.I is an #include file defining several graphics mouse */
/* cursor patterns. All are statics. */
/* This file also contains an event handler called by the mouse */
/* device driver, and a global event record variable 'theEvents' */
/* Must #include MOUSE.I above this #include file. */
/* ----- */

typedef struct ( /* mouse event record */
    unsigned flag, button, col, row;
) EVENTREC;

typedef struct ( /* graphics cursor descriptor */
    unsigned *image;
    unsigned hotX, hotY;
) GCURSREC;

/* check mark image */
static unsigned checkIm [32] = (
    0xFFFF, 0xFFE0, 0xFFC0, 0xFF81, /* screen mask */
    0xFF03, 0x0607, 0x000F, 0x001F,
    0xC03F, 0xF07F, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0x0000, 0x0006, 0x000C, 0x0018, /* cursor mask */
    0x0030, 0x0060, 0x07C0, 0x1D80,
    0x0700, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000);

/* Left arrow image */
static unsigned LArrIm [32] = (
    0xFE1F, 0xF01F, 0x0000, 0x0000, /* screen mask */
    0x0000, 0xF01F, 0xFE1F, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
    0x0000, 0x00C0, 0x07C0, 0x7FFE, /* cursor mask */
    0x07C0, 0x00C0, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000);

/* Cross image */
static unsigned crossIm [32] = (
    0xFC3F, 0xFC3F, 0xFC3F, 0x0000, /* screen mask */
    0x0000, 0x0000, 0xFC3F, 0xFC3F,
    0xFC3F, 0xFFF, 0xFFF, 0xFFF,
    0xFFF, 0xFFF, 0xFFF, 0xFFF,
    0x0000, 0x0180, 0x0180, 0x0180, /* cursor mask */
    0x7FFE, 0x0180, 0x0180, 0x0180,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000);

/* Pointing hand image */
static unsigned handIm [32] = (
    0xE1FF, 0xE1FF, 0xE1FF, 0xE1FF, /* screen mask */
    0xE1FF, 0xE000, 0xE000, 0xE000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,
    0x1E00, 0x1200, 0x1200, 0x1200, /* cursor mask */
    0x1200, 0x13FF, 0x1249, 0x1249,
    0x1249, 0x9001, 0x9001, 0x9001,
    0x8001, 0x8001, 0x8001, 0xFFFF);

/* I-beam image */
static unsigned iBeamIm [32] = (
    0xFFF, 0xFFF, 0xFFF, 0xFFF, ( screen mask )
    0xFFF, 0xFFF, 0xFFF, 0xFFF,
    0xFFF, 0xFFF, 0xFFF, 0xFFF,
    0xFFF, 0xFFF, 0xFFF, 0xFFF,
    0xF00F, 0x0C30, 0x0240, 0x0240, ( cursor mask )
    0x0180, 0x0180, 0x0180, 0x0180,
    0x0180, 0x0180, 0x0180, 0x0180,
    0x0240, 0x0240, 0x0C30, 0xF00F);

```

```

/* Graphics cursor descriptors */
static GCURSREC check = (NULL, 6, 7);
static GCURSREC arrow = (NULL, 0, 3);
static GCURSREC cross = (NULL, 7, 4);
static GCURSREC hand = (NULL, 5, 0);
static GCURSREC iBeam = (NULL, 7, 7);

/* Global far pointer to mouse event record */
static EVENTREC far *theEvents;
/* ----- */

void far handler (void) /* event handler called by device driver */
{
EVENTREC far *save; /* pointer to save area in diff segment */
unsigned a, b, c, d; /* temp storage of registers */

a = AX, b = BX, c = CX, d = DX; /* save registers */
save = MK_FP (_CS-0x10, 0x00C0); /* point to PSP user area */
save->flag = a; /* stuff registers into it */
save->button = b;
save->col = c;
save->row = d;
} /* ----- */

void initGCurs (void) /* initialize ptrs in cursor descriptors */
{
check.image = checkIm;
arrow.image = LArrIm;
cross.image = crossIm;
hand.image = handIm;
iBeam.image = iBeamIm;
} /* ----- */
/* End of GMOUSCUR.I */

```

LISTING 3: GMICE.PAS

```

Program gmice;
( Illustrates the four graphics mouse cursors from gmouscur.inc, )
( plus the default graphics cursor. )

Uses mouse, graph;

($i gmouscur.inc)

CONST eventMask = $54; ( mask to trip event handler when any
                        mouse button is released )

VAR theMouse : resetRec;
    driver, mode : INTEGER;
/* ----- */

PROCEDURE Identify (title : string);
( Write name of cursor near top of screen )

VAR x : INTEGER;

BEGIN
SetViewPort (0, 0, GetMaxX, 30, TRUE);
ClearViewPort;
SetTextStyle (DefaultFont, HorizDir, 1);
x := (GetMaxX - TextWidth (title)) DIV 2;
OutTextXY (x, 20, title);
SetViewPort (0, 0, GetMaxX, GetMaxY, TRUE);
END;
/* ----- */

PROCEDURE GraphicScreen (title : string);
( Creates a graphics screen and shows the title )

VAR x, y : INTEGER;
    prompt : string [30];

BEGIN
InitGraph (driver, mode, '\DRIVERS'); ( set graphics mode )
IF GraphResult = gOK THEN BEGIN
Identify (title);
Prompt := 'Click any button to continue!';
x := (GetMaxX - TextWidth (prompt)) DIV 2; ( start of prompt )
OutTextXY (x, GetMaxY - 20, prompt);

( Prepare to draw a rectangle as a lighted backdrop for cursor )
SetFillStyle (SolidFill, 1);
SetColor (1);
x := (GetMaxX DIV 2) - 50;
y := (GetMaxY DIV 2) - 50; ( set corners )
Rectangle (x, y, x+100, y+100); ( draw )
FloodFill (GetMaxX DIV 2, GetMaxY DIV 2, 1); ( fill )
END;
/* ----- */

PROCEDURE demo (cursor : gCursRec; title : STRING);
( Show the indicated graphics cursor )

```

```

BEGIN
Identify (title);
mGraphCursor (cursor.hotX, cursor.hotY,
              seg (cursor.image^), ofs (cursor.image^));
theEvents.flag := 0;
REPEAT UNTIL theEvents.flag <> 0;
END;
/* ----- */

BEGIN
( Set up for run )
Driver := CGA;
Mode := CGAHi;
InitGCurs; ( initialize the cursor images )
mReset (theMouse);
IF theMouse.exists THEN BEGIN
mInstTask (eventMask, seg (EventHandler), ofs (EventHandler));
END;

( Show default cursor )
GraphicScreen ('Default cursor!');
mShow;
theEvents.flag := 0;
REPEAT UNTIL theEvents.flag <> 0;
END;

( Show the custom cursors )
Demo (check, 'Check cursor');
Demo (arrow, 'Left arrow cursor');
Demo (cross, 'Cross cursor');
Demo (hand, 'Pointing hand');
Demo (iBeam, 'I-Beam cursor');
mReset (theMouse);
CloseGraph;
END.

```

LISTING 4: GMICE.C

```

/* GMICE.C: Illustrates the default graphics mouse cursor, plus
/* the four from GMOUSCUR.I */

/* INCLUDES AND DEFINES */
#include <stdio.h>
#include <dos.h>
#include <graphics.h>
#include <mouse.i>
#include <gmouscur.i>
#ifdef TRUE
#define TRUE -1
#define FALSE 0
#endif
#define EVENTMASK 0x54 /* event when any mouse button released */

/* LOCAL PROTOTYPES */
void demo (GCURSREC, char*);
void graphicScreen (char*);
void identify (char*);

/* GLOBALS */
union REGS reg;
int driver = CGA, mode = CGAHI;
char path [] = "C:\TC";

void main ()
{
resetRec *theMouse;

/* Set up for run */
initGCurs(); /* initialize the cursor images */
theMouse = mReset(); /* reset the mouse */
if (theMouse->exists) {
mInstTask (EVENTMASK, FP_SEG (handler), /* install handler*/
          FP_OFF (handler));
theEvents = MK_FP (_psp, 0x00C0); /* point to event record */
}

/* Show default cursor */
graphicScreen ("Default cursor");
mShow(); /* turn on cursor */
while (theEvents->flag == 0) /* wait for click */
;

/* Show the custom cursors */
demo (check, "Check");
demo (arrow, "Left arrow");
demo (cross, "Cross");
demo (hand, "Pointing hand");
demo (iBeam, "I-Beam");
theMouse = mReset(); /* reset mouse */
closegraph();
} /* ----- */

```

```

void demo (GCURSOR cursor, char title[]) /* show graphics cursor */
(
  identify (title);
  mGraphCursor (cursor.hotX, cursor.hotY, (unsigned) (_DS),
    (unsigned) (cursor.image));
  theEvents->flag = 0;
  while (theEvents->flag == 0) /* wait for click */
  ;
) /* ----- */

void graphicScreen (char title[]) /* set up graphics screen */
(
  int x, y;
  char prompt [30];
  initgraph (&driver, &mode, path);
  if (graphresult() == grOk) (
    identify (title);
    strcpy (prompt, "Click any button to continue");
    x = (getmaxx() - textwidth (prompt)) / 2;
    outtextxy (x, getmaxx() - 20, prompt);

    /* draw rectangle as backdrop for cursor */
    setfillstyle (SOLID_FILL, 1);
    setcolor (1);
    x = (getmaxx()/2) - 50;
    y = (getmaxy()/2) - 50;
    rectangle (x, y, x+100, y+100);
    floodfill (getmaxx()/2, getmaxy()/2, 1);
  )
) /* ----- */

void identify (char *title)
(
  int x;

  setviewport (0, 0, getmaxx(), 30, TRUE);
  clearviewport();
  settxtstyle (DEFAULT_FONT, HORIZ_DIR, 1);
  x = (getmaxx() - textwidth (title)) / 2;
  outtextxy (x, 20, title);
  setviewport (0, 0, getmaxx(), getmaxy(), TRUE);
) /* ----- */

```

LISTING 5: SCRIBBLE.PAS

```

Program Scribble;

( A simple utility for drawing with the mouse in graphics mode )

USES mouse, graph;

($I gmousecur.inc)

CONST EventMask = $55; ( Any button released, or mouse moved )
Menu = 20; ( bottom of menu area )
HelpTop = 185; ( top of help area )
Box1 = 160; ( right end of each menu box )
Box2 = 320;
Box3 = 480;
HelpMsg1 = 'Click any button on Scribble to begin drawing';
HelpMsg2 = 'Hold down left button to draw';

VAR theMouse : resetRec;
mouses : locRec;

( ----- )

PROCEDURE MenuBox (x1, x2 : INTEGER; Item : STRING);

( Create a menu box between indicated x's at top of screen )

BEGIN
  SetViewPort (x1+1, 1, x2-1, menu-1, FALSE); ( local to help area )
  ClearViewPort;
  OutTextXY (50, 7, item); ( display text )
  SetViewPort (0, 0, GetMaxX, HelpTop, FALSE); ( drawing work area )
END;
( ----- )

PROCEDURE Help (Message : STRING);

( Display help message at bottom of screen )

VAR x : INTEGER;

BEGIN
  x := (GetMaxX - TextWidth (message)) DIV 2; ( For centering )
  SetViewPort (1, helpTop+1, 638, GetMaxY - 1, FALSE);
  ClearViewPort;
  OutTextXY (x, 3, message); ( write help message )
  SetViewPort (0, menu, 639, HelpTop, FALSE); ( drawing work area )
END;
( ----- )

FUNCTION SetUpScreen : BOOLEAN;

( Prepare screen, return TRUE if done, FALSE if can't )

```

```

VAR driver, mode : INTEGER;

BEGIN
  Driver := CGA; ( use CGA hi-res mode )
  Mode := CGAhi;
  InitGraph (driver, mode, '\TP'); ( set graphics mode )
  IF GraphResult = grOk THEN ( if successful... )
  BEGIN
    SetColor (1); ( initialize )
    SetTextStyle (DefaultFont, HorizDir, 1);
    MenuBox (0, box1, 'Scribble'); ( make menu boxes )
    Rectangle (0, 0, box1, menu);
    MenuBox (box1, box2, 'Clear');
    Rectangle (box1, 0, box2, menu);
    MenuBox (box2, box3, 'Quit');
    Rectangle (box2, 0, box3, menu);
    Rectangle (box3, 0, 639, menu); ( box for meter )
    Rectangle (0, HelpTop, 639, GetMaxY); ( box for help )
    Help (HelpMsg1); ( initial help message )
    SetUpScreen := TRUE; ( successful )
  END
  ELSE
    SetUpScreen := FALSE; ( unsuccessful )
  END;
( ----- )

PROCEDURE UpdateMeter (x, y : INTEGER);

( Update mouse position meter in upper right corner of display )

VAR Position : STRING [8];
Number : STRING [3];

BEGIN
  Str (x : 3, number); ( convert position to string )
  Position := number;
  Str (y : 3, number);
  Position := position + ' ' + number;
  MenuBox (box3, 639, position); ( display it )
END;
( ----- )

PROCEDURE Work;

( Draw with mouse until user clicks on Quit selection )

VAR thru : BOOLEAN;

BEGIN
  Thru := FALSE;
  REPEAT
    TheEvents.flag := 0; ( clear event flag )
    REPEAT UNTIL theEvents.flag <> 0; ( wait for mouse event )
    CASE theEvents.flag OF
      $0001: BEGIN ( mouse has moved )
        IF ((theEvents.row > menu) AND
          (theEvents.row < HelpTop)) THEN ( in work area )
          IF theEvents.button = 1 THEN BEGIN ( and left down )
            mHide;
            PutPixel (theEvents.col, theEvents.row, 1); {draw}
            mShow;
          END;
          UpdateMeter (theEvents.col, theEvents.row); { update }
        END;
      $0004,
      $0010,
      $0040: BEGIN ( any button released )
        IF theEvents.row < menu THEN ( if in menu area )
          IF theEvents.col < box1 THEN ( Scribble? )
          BEGIN
            WITH cross DO
              mGraphCursor (hotX, hotY, seg (image`),
                ofs (image`));
            Help (HelpMsg2);
          END
          ELSE
            IF theEvents.col < box2 THEN ( Clear? )
            BEGIN
              mHide;
              SetViewPort (0, menu+1, GetMaxX,
                helpTop-1, TRUE);
              ClearViewPort;
              mShow;
              WITH hand DO
                mGraphCursor (hotX, hotY, seg (image`),
                  ofs (image`));
              Help (HelpMsg1);
            END
            ELSE
              IF theEvents.col < box3 THEN ( Quit? )
                thru := true;
            END; ( of outer IF )
          END; ( of CASE )
        UNTIL thru;
      END;
    ( ----- )

  BEGIN
    InitGCurs; ( Initialize cursor images )
    mReset (theMouse); ( Initialize mouse )
    IF theMouse.exists THEN

```

```

BEGIN
theEvents.flag := 0;
IF SetUpScreen THEN          ( if in graphics mode... )
  BEGIN
  minstTask (EventMask, seg (EventHandler),
             ofs (EventHandler)); ( install handler )
  WITH hand DO              ( show pointing hand )
  mGraphCursor (hotX, hotY, seg (image`), ofs (image`));
  mShow;
  mPos (mouses);           ( Get mouse position )
  UpdateMeter (mouses.column, mouses.row);
  Work;                    ( do what the program does )
  mReset (theMouse);       ( shut down the mouse )
  CloseGraph;              ( back to text mode )
  END
ELSE
  WRITELN ('Graphics mode not available. Program ended.');
```

LISTING 6: SCRIBBLE.C

```

/* SCRIBBLE.C: Simple utility for drawing with mouse in graphics */

/* INCLUDES */
#include <stdio.h>
#include <dos.h>
#include <graphics.h>
#include <mouse.h>
#include <mousecur.h>

/* DEFINE TRUE/FALSE */
#ifndef TRUE
#define TRUE -1
#define FALSE 0
#endif

/* DEFINE CONSTANTS */
#define eventMask 0x55 /* any button released, or mouse moved */
#define menu 20 /* bottom of menu area */
#define helpTop 185 /* top of help area */
#define box1 160 /* right end of each menu box */
#define box2 320
#define box3 480
#define helpMsg1 "Click any button on Scribble to begin drawing"
#define helpMsg2 "Hold down left button to draw"

/* GLOBALS */
resetRec *theMouse;
locRec *mouses;

/* LOCAL FUNCTIONS */
void menuBox (int, int, char*);
void help (char*);
int setUpScreen (void);
void updateMeter (int, int);
void work (void);

/* ----- */
void main ()
{
theEvents = MK_FP (_psp, 0x00C0); /* point to event buffer */
initGCurs (); /* initialize cursor images */
theMouse = mReset (); /* initialize mouse */
if (theMouse->exists) {
theEvents->flag = 0;
if (setUpScreen()) /* if in graphics mode... */
  minstTask (eventMask, FP_SEG(handler), /* install handler */
             FP_OFF(handler));
  mGraphCursor (hand.hotX, hand.hotY, _DS, /* graphics curs */
               (unsigned) hand.image);
  mShow (); /* cursor on */
  mouses = mPos (); /* get position */
  updateMeter (mouses->column, mouses->row);
  work (); /* do mouse stuff until thru */
  theMouse = mReset (); /* reset mouse */
  closeGraph (); /* back to text mode */
} else
  puts ("\nGraphics mode not available. Program ended.");
} else
  puts ("Mouse not active. Program ended.");
} /* ----- */

void menuBox (int x1, int x2, char *item)
/* create a menu box between x1 and x2 at top of screen */
{
setviewport (x1+1, 1, x2-1, menu-1, FALSE);
clearviewport ();
outtextxy (50, 7, item); /* display text */
setviewport (0, 0, getmaxx(), helpTop, FALSE); /* drawing area */
} /* ----- */
```

```

void help (char *message)
/* Display help message at bottom of screen */
{
int x;
x = (getmaxx() - textwidth(message)) / 2; /* For centering */
setviewport (1, helpTop+1, 638, getmaxy() - 1, FALSE);
clearviewport ();
outtextxy (x, 3, message); /* write help message */
setviewport (0, menu, 639, helpTop, FALSE); /* drawing area */
} /* ----- */

int setUpScreen (void)
/* Prepare screen, return TRUE if done, FALSE if can't */
{
int driver = CGA, mode = CGAHI; /* use CGA hi-res mode */
char path [6] = "C:\TC"; /* path to drivers */

initgraph (&driver, &mode, path); /* set graph mode */
if (graphresult () == grOk) { /* if successful... */
setcolor (1); /* initialize */
settextstyle (DEFAULT_FONT, HORIZ_DIR, 1);
menuBox (0, box1, "Scribble"); /* make menu boxes */
rectangle (0, 0, box1, menu);
menuBox (box1, box2, "Clear");
rectangle (box1, 0, box2, menu);
menuBox (box2, box3, "Quit");
rectangle (box2, 0, box3, menu);
rectangle (box3, 0, 639, menu); /* box for meter */
rectangle (0, helpTop, 639, getmaxy()); /* box for help */
help (helpMsg1); /* initial help message */
return (TRUE); /* successful exit */
} else
return (FALSE); /* unsuccessful */
} /* ----- */

void updateMeter (int x, int y)
/* Update mouse position meter, upper right corner */
{
char position [9];
sprintf (position, "%3d, %3d", x, y); /* convert pos to string */
menuBox (box3, 639, position); /* display it */
} /* ----- */

void work (void)
/* Draw with mouse until user clicks on Quit selection */
{
int thru = FALSE;
do {
theEvents->flag = 0; /* clear mouse event flag */
while (theEvents->flag == 0); /* wait for mouse event */
switch (theEvents->flag) {
case 0x0001: /* mouse has moved */
if ((theEvents->row > menu) &&
    (theEvents->row < helpTop)) /* in work area */
if (theEvents->button == 1) { /* and left down */
mHide ();
putpixel (theEvents->col, theEvents->row, 1); /* draw */
mShow ();
}
updateMeter (theEvents->col, theEvents->row); /* update */
break;
case 0x0004:
case 0x0010:
case 0x0040: /* any button released */
if (theEvents->row < menu) /* if in menu area */
if (theEvents->col < box1) { /* Scribble? */
mGraphCursor (cross.hotX, cross.hotY, _DS,
              (unsigned) cross.image);
help (helpMsg2);
}
else
if (theEvents->col < box2) { /* Clear? */
mHide ();
setviewport (0, menu+1, getmaxx(), helpTop-1, TRUE);
clearviewport ();
mShow ();
mGraphCursor (hand.hotX, hand.hotY, _DS,
              (unsigned) hand.image);
help (helpMsg1);
} else
if (theEvents->col < box3) /* Quit? */
thru = TRUE;
break;
}
} while (!thru);
} /* ----- */
```

FORMATTING OUTPUT IN TURBO C

Turbo C's output formatting capabilities may surprise you.

Peter Aitken

All computer programs have one thing in common—output. A program must send information somewhere, be it to a video display, a printer, or a modem. In many cases, the ease with which we can use that information depends upon its arrangement and appearance—in other words, upon its format. No matter how skilled you are at writing tight code and efficient algorithms, your programs cannot reach their full potential until you're able to format output to its best advantage.

Turbo C has an entire family of functions that produce formatted output. These functions are known collectively as the **..printf** functions, and differ with respect to where they send output. The functions **printf()** and **vprintf()** send output to **stdout**, and **cprintf()** sends output directly to the console. (**stdout** normally is the console, but output to **stdout** can be redirected at the DOS level, whereas output direct to the console cannot be redirected.) The functions **fprintf()** and **vfprintf()** place output in a named stream, while **sprintf()** and **vsprintf()** place output in memory. Rather than describing how these functions differ, this article focuses on one thing that they have in common—how the format of their output is controlled. The **printf()** function is used in examples throughout this article, but remember that the discussion applies to the other functions in the **..printf** family as well.

THE FORMAT STRING

Information is passed to a C function by means of one or more arguments, which are enclosed in the parentheses following the function name. One of the arguments to the **printf()** function is the format string. This series of instructions, which is enclosed in double quotes, tells the **printf()** function about the size and appearance of the data that is being output. If you understand the various components of format strings, you'll have a great deal of control over the appearance of the output produced by the **..printf** functions.

One component of the format string can be literal text, which is text that you want output exactly as shown. In fact, a format string can consist of literal text alone, as in the following example:

Statement:
`printf("Hello there!");`

Output:
Hello there!

But what about outputting data that is contained in a variable? To do this, you must add two components to the **printf()** arguments. The first component, of course, is the name of the variable (C is a powerful language, but it can't read your mind!). The second component is a format specifier, which tells **printf()** the type of data being output and how to format that data. In the example below, assume that **number** is an integer variable with a value of 8:

Statement:
`printf("The value is %d",number);`

Output:
The value is 8

The **%d** in the format string is the *format specifier*. Format specifiers begin with a percent sign and end with a letter. The letter following the percent sign is the character code for the type of data that is being output. In this case, we use **d**, which stands for a decimal (i.e., base 10) integer. C provides character codes for all of the types of numeric data that you could output; these codes are listed in Table 1.

Note that some of the character codes in Table 1 can be upper- or lowercase. This allows you to specify the case of any output alphabetic characters. Specifically, this applies to formats where the output contains letters, such as hexadecimal notation. Any letters in the output will be in the same case as the character code in the associated format specifier.

continued on page 56

The Official Endorsed Books On Quattro®

Over 175,000 Copies In Print



"Borland-Osborne/McGraw-Hill offers you the only full line of endorsed books on Quattro. These titles combine Borland's own technical expertise with Osborne/McGraw-Hill's publishing savvy. With these official Quattro titles, you'll have a comprehensive library that keeps pace with you as you develop greater skills with Quattro."

Philippe Kahn, President & CEO, Borland International, Inc.

Quattro® Made Easy
by Lisa Biow

Guides you through a step-by-step introduction
\$19.95 600 pp.
ISBN: 0-07-881347-6

**Using Quattro®
The Professional Spreadsheet**
by Stephen Cobb

Gets you up and running fast with basic to more advanced techniques.
\$21.95 584pp.
ISBN: 0-07-881330-1

Quattro®: Secrets, Solutions, Shortcuts
by Craig Stinson

Unveils a clever selection of Quattro tricks.
\$21.95 650pp.
ISBN: 0-07-881400-6
Available: 8/88

Quattro®: Power User's Guide
by Stephen Cobb

Unlocks Quattro's full power for serious business.
\$22.95 600pp.
ISBN: 0-07-881367-0

Quattro®: The Complete Reference
by Yvonne McCoy

Details every Quattro feature, command, and function.
\$24.95 666pp.
ISBN: 0-07-881337-9

Quattro®: The Pocket Reference
by Stephen Cobb

Puts essential commands and features at your fingertips.
\$5.95 128pp.
ISBN: 0-07-881378-6

ORDER TODAY!

Available at Fine Book Stores and Computer Stores Everywhere or

**CALL TOLL FREE
800-227-0900**

Visa, MasterCard, & American Express Accepted

BORLAND-OSBORNE/McGRAW-HILL

BUSINESS SERIES

Quattro is a registered trademark of Borland International, Inc. Copyright©1988, McGraw-Hill, Inc.



Osborne McGraw-Hill
2600 Tenth Street
Berkeley, CA 94710

FORMATTING

continued from page 54

TYPE CHARACTER	INPUT DATA TYPE	OUTPUT FORMAT
d or i	integer	signed decimal (base 10) integer
u	integer	unsigned decimal (base 10) integer
o	integer	unsigned octal (base 8) integer
x or X	integer	unsigned hexadecimal (base 16) integer
f	floating pt.	signed value [-]dddd.dddd
e or E	floating pt.	scientific notation [-]d.dddd e [+/-]ddd
g or G	floating pt.	same as f or e, depending on value and precision

Table 1. Input types and output formats.

Look at the examples in Table 2—you should now be able to understand the connection between the format specifier and the resulting output. But what about those two strange results, where the computer doesn't seem to know the difference between -1 and 65535? If you're familiar with the difference between signed and unsigned variables in C, you'll understand the cause of the unexpected output (otherwise, refer to the accompanying sidebar for more information).

DATA	FORMAT SPECIFIER	OUTPUT
188	%d or %i	188
65535	%d	-1 (unexpected output)
1	%u	1
-1	%u	65535 (unexpected output)
99	%o	143 (143 is 99 decimal in octal)
56789	%x	ddd5
56789	%X	DDD5
18.405	%f	18.405000
18.405	%e	1.840500e+001
18.405	%E	1.840500E+001

Table 2. Examples of Turbo C's output formatting.

A format string must contain one format specifier for each variable in the variable list. The format specifiers are applied to the variables in order. For example:

```
int num = 2;
float root;
root = sqrt(num);
printf("%d = %f", num, root);
```

When compiled and executed, this code displays the output:

```
/2 = 1.414214
```

FORMATTING CHARACTER OUTPUT

Formatting character output is much simpler than formatting numeric output because fewer choices can be made. Characters come singly or in strings, and C offers a format specifier for each. Use %c to output a single character, and use %s to output a string. Some examples are shown in Table 3.

DATA	FORMAT SPECIFIER	OUTPUT
"X"	%c	X
"The cat\0"	%s	The cat

Table 3. Output formatting for character and string data.

The "0" at the end of the string is a reminder that C doesn't have a special variable class for strings, and stores them instead as arrays of characters. The 0 represents the null character and is the terminating character that marks the end of a string.

OUTPUTTING POINTERS

Turbo C has a special formatting character for outputting pointers. Although pointer values are rarely, if ever, a part of a finished program's output, it's often necessary to list them as part of the debugging process. The format specifier %p outputs a pointer (in hexadecimal) as YYYY (offset only) for near pointers, and as XXXX:YYYY (segment:offset) for far pointers.

OPTIONAL FORMAT CONTROLS

So far, we've discussed how to output literal text, and have covered the minimum format specifiers needed to output numeric and character variables. C offers a variety of optional format controls that let you specify the appearance of your program's output in more detail. One or more of these optional format controls can be placed in a format specifier, between the leading % and the type character. The components of a format specifier, including the optional format controls, are:

```
% [flags] [width] [.precision]
[size modifier] type
```

Flags. The first optional format control is the flags component. *Flags* control justification, numeric signs, decimal points, octal and hexadecimal prefixes, and trailing zeros.

The minus (-) flag causes the output to be left-justified in its field, and padded on the right with blanks if needed. If no flag is given, the default is right justification. Justification applies only if the output is narrower than the specified field width.

The plus (+) flag causes numeric output to be preceded by the appropriate sign (+ or -). The default is that only negative numbers are preceded by a sign.

With the blank () flag, positive numbers are preceded by a blank (space); negative numbers are not affected.

The pound sign (#) specifies that the argument is to be formatted using a so-called "alternate form." Alternate forms exist for certain type characters, as shown in Table 4.

Width. The width specifier determines the minimum width of the field in which the output is placed. The word "minimum" is important here—if the output is wider than the specified field width, the field expands as necessary in order to contain the output. The output is *never* truncated to fit a too-small field.

TYPE CHARACTER	ALTERNATE FORM
c, d, i, s, u o	none (flag has no effect) O will appear before nonzero argument
x or X	Ox or OX will appear before argument
e, E, or f	decimal point included even if no digits follow it
g or G	same as e and E, but trailing zeros are not removed

Table 4. Alternate forms.

The width specifier is simply a number that indicates the width (in spaces) of the output field. If the output value is narrower than the field, the value is padded with enough spaces to fill the field. As the default for this specifier, the value is right-justified within the field. If the "-" flag is included, the value is left-justified. If the width specifier is preceded with a 0, the value is right-justified and padded on the left with zeros. Several examples are shown below:

format specifier	output
-----	-----
%12d	←width 12→ 123456
%-12d	123456
%012d	000000123456

Precision. The precision specifier sets the number of digits that are printed to the right of the decimal point. When used with string variables, precision determines the maximum number of output characters. The precision specifier always begins with a period (.) to separate it from the preceding width specifier (if any). If no precision specifier is given, the default value is used. The default precision is 1 for the **d**, **i**, **o**, **u**, **x**, and **X** types; 6 for **e**, **E**, and **f** types; and all significant digits for **g** and **G** types. For strings, the default precision is the full length of the string (i.e., all characters up to the terminating null character).

When precision is set to 0, no decimal point is printed for **e**, **E**, and **f** types; other types are not affected, and precision remains at the default value. Setting precision to a number *n* causes *n* digits to the right of the decimal point, or *n* characters, to be output. If the output value is wider than *n* characters, strings are truncated and numbers are rounded in order to meet the specified precision.

Both width and precision can be specified with a variable argument to the **printf()** function, rather than as part of the format string. To use a variable argument, place an asterisk in the format string at the position of the width or precision specifier, then place an integer variable in the argument list just before the variable that is being formatted. Both width and precision can be controlled at the same time with variable arguments. In the following code fragment, both **printf()** statements produce the same output:

```
int width = 12;
int prec = 4;
printf("%12.4f",value);
printf("%*.*f",width,prec,value); );
```

Size. The final optional control involves input size modifiers, which apply only to numeric and pointer variables. Size modifiers tell the **printf()** function to interpret the argument's size as other than the default size. For pointers, the default size (near or far) is determined by the memory model in use. The input size modifiers **F** and **N** cause a pointer argument to be interpreted as a far or near pointer, respectively.

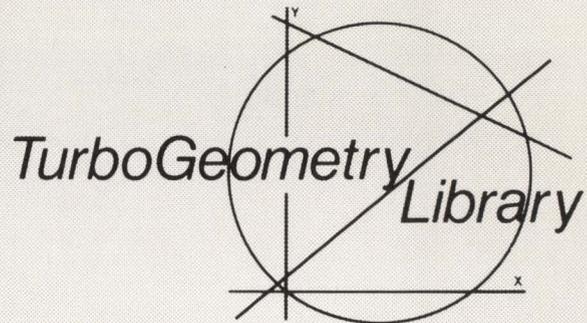
For numeric variables, the modifier **h** applies only to integer arguments (types **d**, **i**, **o**, **u**, **x**, and **X**), and causes the argument to be interpreted as a short **int**. The modifier **l** applies to either integer (types **d**, **i**, **o**, **u**, **x**, and **X**) or floating point (types **e**, **E**, **f**, **g**, and **G**), and causes an integer argument to be interpreted as a **long int** (long integer) and a floating point argument to be interpreted as a **double**. The input size modifiers have no effect on character (**c**, **s**) types.

ESCAPE SEQUENCES

An escape sequence changes the meaning of certain characters in a string. For example, you might want a double quotation mark to be output as part of a string rather than to be interpreted as the end of the string. Use an escape sequence of a double

continued on page 58

"The cost involved, in writing one of these geometric routines, is more than the price of the TurboGeometry Library."



Are you programming or planning to program CAD/CAM or graphics applications? Many hours, even days, can be spent in writing and debugging geometric routines. TurboGeometry Library can relieve you of those time consuming tasks that are part and parcel of every CAD/CAM or graphics program. There are over 150 routines in the library, supported by example programs and a 400 page manual. The source code is included. 30 day guarantee. Need IBMPC or Compatible, Turbo Pascal 4.0, Turbo C, or MS C. \$149.95 plus \$5.00 S&H in US. VISA, MasterCard, Check, PO, MO. No COD's. Send for additional information or call 214-423-7288.

**Disk Software, Inc., 2116 E. Arapaho #487
Richardson, Texas USA 75081**

"In CAD/CAM or graphics, it all comes down to using geometry"

NUMBERS AND THEIR NOTATION

Hexadecimal is a number system that uses base 16, rather than the base 10 used by the everyday decimal notation with which we're all familiar. In decimal notation, each position moving from right to left indicates a successive power of ten:

456 (decimal)

6	$\times 10^0 = 6 \times 1$	=	6
5	$\times 10^1 = 5 \times 10$	=	50
4	$\times 10^2 = 4 \times 100$	=	400
		sum =	456

Hexadecimal works the same way, except that each position in the number represents a power of 16:

456 (hexadecimal)

6	$\times 16^0 = 6 \times 1$	=	6
5	$\times 16^1 = 5 \times 16$	=	80
4	$\times 16^2 = 4 \times 256$	=	1004
		sum in decimal =	1090

Because hexadecimal uses powers of 16, it needs single digits to represent numbers up to 15 (decimal). The regular digits 0-9 represent themselves, and the (decimal) numbers 10-15 are represented by the letters A-F. Thus, counting up from 0 in hexadecimal (with decimal equivalents), we have:

Hex: 1 ... 9 A B C D E F 10 11 12 ... 1E
1F 20 ... FF 100

Dec: 1 ... 9 10 11 12 13 14 15 16 17 18 ...
30 31 32 ... 255 256

And, as used in larger numbers:

BCF (hexadecimal)

11	$\times 16^0 = 11 \times 1$	=	11
12	$\times 16^1 = 12 \times 16$	=	192
15	$\times 16^2 = 15 \times 256$	=	3840
		sum in decimal =	4043

Hexadecimal notation is a favorite among programmers because the binary bit patterns that are used internally by computers translate directly into hex digits. A single hex digit represents 1 nybble (4 bits), a pair of hex digits represents 1 byte, and 4 hex digits are perfect for representing 16-bit memory addresses:

Dec.	Hex	Binary
0	00	0000 0000
1	01	0000 0001
.	.	.
.	.	.

254	FE	1111 1110
255	FF	1111 1111
.	.	.
65534	FFFE	1111 1111 1111 1110
65535	FFFF	1111 1111 1111 1111

SCIENTIFIC NOTATION

Scientific notation was developed to represent the very large and very small numbers that are often used by scientists and engineers. Scientific notation expresses any value as a number between 1 and 10 that is multiplied by a power of 10, as shown in these examples:

Value	As power of 10	In scientific notation
5,450,000,000	5.45×10^9	5.45E+009
0.0000000789	7.89×10^{-8}	7.89E-008

SIGNED AND UNSIGNED VARIABLES

To avoid possible errors and confusion resulting from mixing signed and unsigned variables, you need to understand the differences between these variables and the way that your computer represents them internally. In Turbo C, the two data types **int** (integer) and **short** are each stored internally as two bytes. The 16 bits in 2 bytes can represent a total of 2^{16} (or 65536) different values. If an **int** variable has been declared as type **unsigned**, it can be assigned values between 0 and 65535. With unsigned variables, the relationship between bit pattern and value is straightforward binary:

0000 0000 0000 0001	1
0000 0000 0000 0010	2
.	.
.	.
0111 1111 1111 1110	32766
0111 1111 1111 1111	32767
.	.
.	.
1111 1111 1111 1110	65534
1111 1111 1111 1111	65535

If the variable is of the default type **signed**, however, things are different. The 16 bits in the computer still represent 65536 different values, but the permissible range is -32768 to 32767. The positive values between 0 and 32767 are represented by the same bit patterns that represent

an unsigned integer, but the negative values are represented as *two's-complements* of the corresponding positive value. A two's-complement is formed by first reversing all of the bits (all 0s become 1s and vice versa) and then adding 1, as shown below:

0011 0010 1001 1111	+12959
1100 1101 0111 0000	reverse all bits
	+1 add 1
1100 1101 0111 0001	-12959

Here's where the possible confusion arises: some bit patterns can represent two different values, depending upon whether they are interpreted as a signed or unsigned quantity. The following comparison shows how Turbo C interprets a single binary bit pattern as either a signed or an unsigned quantity:

Binary	Signed	Unsigned
1000 0000 0000 0000	-32768	32768
1000 0000 0000 0001	-32767	32769
1000 0000 0000 0010	-32766	32770
.	.	.
.	.	.
1111 1111 1111 1110	-2	65534
1111 1111 1111 1111	-1	65535
0000 0000 0000 0000	0	0
0000 0000 0000 0001	1	1
0000 0000 0000 0010	2	2
.	.	.
.	.	.
0111 1111 1111 1110	32766	32766
0111 1111 1111 1111	32767	32767

Note that for signed variables, the "high-order" bit (i.e., the bit farthest to the left) is 1 for all negative values and 0 for all positive values. Appropriately enough, this bit is called the *sign bit*.

For values between 0 and 32767, bit patterns are interpreted as having the same value for both signed and unsigned variables. Outside that range—for any bit pattern in which the high-order bit is 1—bit patterns are interpreted differently. We must be careful, therefore, to use the appropriate format specifier when outputting integer variables. Use **%u** only for variables that have been declared as **unsigned**, and use **%d** (or **%i**) only for signed variables. ■

—Peter Aitken

FORMATTING

continued from page 57

quotation mark preceded by a backslash (\") to tell the compiler to interpret the quotation mark as a literal character rather than as the string delimiter. This is shown in the following example:

```
Statement:
printf("She said \"hello!\"");
```

```
Output:
(Will not compile)
```

```
Statement:
printf("She said \\\"hello!\\\"");
```

```
Output:
She said "hello!"
```

Escape sequences are also used for the so-called nonprinting characters, such as tabs and line feeds, that are used to control printers and other output devices. For example, \n is the escape sequence for a newline. All of Turbo C's escape sequences are listed in Table 5. The \n sequence can be used to insert a newline between the several variables passed as arguments to printf, as shown in the following example:

```
Statement:
printf
("%d %d %d", val1, val2, val3);
```

```
Output:
1 2 3
```

```
Statement:
printf
("%d\n%d\n%d", val1, val2, val3);
```

```
Output:
1
2
3
```

The last two escape sequences shown in Table 5 permit you to include any character in a string by specifying its octal or hexadecimal value.

You may think that something is missing from the list of escape sequences—the percent sign. Since % marks the beginning of a format specifier, don't we

SEQUENCE	INTERPRETATION
\a	bell (beeps speaker)
\b	backspace
\f	formfeed
\n	linefeed
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\'	single quote
\"	double quote
\?	question mark
\DDD	DDD = octal value
\xDDD	DDD = hexadecimal value

Table 5. Escape sequences.

need to use the escape sequence \% to put a literal % in the output? There's no escape sequence for %, however, because the percent sign has a special meaning only in C format strings. Escape sequences are used only for characters, such as quotation marks, that have special meanings in all C strings. To output a literal % with one of the ..printf functions, simply use two consecutive percent signs: %%.

USE OF VARIABLES AS FORMAT STRINGS

Up to this point, we've specified format strings as text enclosed in quotation marks. This method is perfectly adequate in many situations, but it limits flexibility because the format string is hard-coded into the program and cannot be modified at runtime. To write programs that modify their own output format, use a string variable for a format string.

As an example, let's say that we need to output a numeric variable whose value can span a wide range. If the variable's value is less than 1 million, we want it to be output in decimal integer format; if the value is greater than 1 million, we want the output to be in scientific notation. This is accomplished by the code in Figure 1.

```
char fmt_string[20];

if (value > 1000000)
    strcpy(fmt_string, "The value is %E");
else
    strcpy(fmt_string, "The value is %d");
printf(fmt_string, value);
```

Figure 1. Using variables as format strings.

READY TO OUTPUT

You should know enough about format strings now to successfully tailor the output of your Turbo C programs to suit your needs. Recall that a format string contains one or more of these components:

1. Literal text that is output exactly as shown;
2. Escape sequences for special characters and control codes; and
3. One format specifier for each argument in the variable list.

The *Turbo C Reference Guide's* section on printf() contains a short program listing that generates a variety of output formats. By experimenting with this program, or with your own variant of it, you can quickly translate the information presented in this article into a working knowledge of format strings. ■

Peter Aitken is an assistant professor at Duke University Medical Center, and is the author of DigScope, a scientific software package. He writes and consults in the microcomputer field.

ALLOCATING FULL 64K BLOCKS IN TURBO C

Benefit from every byte of 64K memory allocation—without using huge pointers.

Michael Abrash



WIZARD

Although Turbo C's familiar **malloc** function can only allocate memory blocks smaller than 64K (even in the Huge model), the **farmalloc** function can allocate blocks of memory that are 64K bytes (and larger) in every model other than the Tiny model.

Memory blocks that are 64K or smaller in size can be accessed with far pointers; blocks that are larger than 64K must be accessed with huge pointers. Since code that uses huge pointers is considerably less efficient than code that uses far pointers, far pointers should be employed whenever possible. As a result, the optimum size for large memory blocks is exactly 64K.

The use of **farmalloc** to allocate memory blocks that are exactly 64K in size, however, presents a small problem. The far pointers to the memory blocks that **farmalloc** returns don't necessarily point to the start of a segment. In other words, allocated far pointers don't necessarily have a zero-offset portion; this is beneficial since the use of a nonzero-offset portion saves up to 15 bytes per memory block. Also, only the offset portion of a far pointer can change. Thus, if a far pointer that is allocated with **farmalloc** is used to access the high end of a 64K block, the offset portion of that pointer might accidentally wrap around to zero and let the program overwrite unrelated data in an adjacent segment.

The solution is simple: Adjust the far pointer that is returned by **farmalloc** so that the pointer has a zero-offset portion. This step advances the address to which the far pointer points by up to 15 bytes; thus we have to ask **farmalloc** for an *additional* 15 bytes. While this process wastes 15 bytes, it's nothing compared with the code space saved by using far rather than huge pointers.

Listing 1, **GetFarBlock**, returns a zero-offset far pointer to any block that is up to 64K bytes in size, thereby solving all of the problems that are associated with accessing 64K blocks by way of far pointers. Call **GetFarBlock** whenever you want to allocate any far blocks between 65,518 and 65,536 (inclusive) bytes in size. ■

Michael Abrash is a senior software engineer at Orion Instruments in Redwood City, California.

Listing can be downloaded from CompuServe as FULL.ARC.

LISTING 1: FULL64K.C

```
#include <alloc.h> /* required for farmalloc */
#include <dos.h> /* required for MK_FP, FP_SEG, FP_OFF */

/* Returns a far pointer to a block allocated on the far heap.
The offset portion of the pointer is guaranteed to be zero.
Don't forget that BlockSize must be a long, not an int! */
char far * GetFarBlock(unsigned long BlockSize)
{
    char far *temp;

    /* Get a block 15 bytes larger than needed */
    temp = farmalloc(BlockSize + 15);
    /* Adjust the pointer up to the start of the next segment
and force the offset portion to zero */
    return(MK_FP(FP_SEG(temp)+(FP_OFF(temp)+15)/16, 0));
}
```

WORTH THE WAIT

Waiting for one event or another—when neither may actually occur—takes some Turbo C finesse.

Jonathan Sachs



WIZARD

I once had to write a specialized telecommunication program to call another computer, log on, transfer some files, and log off again—all automatically. The process was straightforward, but it depended upon a lengthy sequence of question-and-answer exchanges between my program and the other computer's program. At any point, a malfunction could make the other system just "go away"—forever fail to respond to my program's last message. My program had to be able to deal with that problem gracefully. Above all, it could never be allowed to lock up my machine, keeping it permanently waiting for a response that would never come.

In solving this problem, I developed a set of Turbo C routines that accomplish the following tasks:

- Wait for the passing of a specified time interval, or the occurrence of a specified event (such as the arrival of an incoming character on a serial port), whichever comes first;
- Read and write characters to a serial port; and
- Wait for the passing of a specified time interval alone.

These functions provide a generalized way to wait for an event that should occur within a certain length of time, but due to some error condition, might never occur at all. This is a common requirement in programs that communicate with a modem or other serial device. The methods used in these routines demonstrate how function pointers can be used to simplify and generalize Turbo C code.

TO WAIT OR NOT TO WAIT

When I began to design the program, I immediately saw that it had to use code that follows the logic shown in Figure 1 in order to read incoming data from the modem.

As the design progressed, this apparently simple logic became marvelously complex. It popped up all over the program, always in a slightly different form. For example, waiting "longer than we should have

```
while (there's nothing to read yet)
{
    if (we've waited longer than we should have to)
    {
        indicate a "time-out";
        break;
    }
}
if (no time-out)
    read the next character;
```

Figure 1. The timeout algorithm.

to" meant different things in different cases. The time limit might be 10 seconds for a response to a log-on request, 2 seconds for the first character of an incoming message, 0.5 seconds for subsequent characters of a message, and so on. In some cases, "longer than we should have to" meant that the operator had gotten impatient and pressed a key on the keyboard!

I needed to encapsulate the logic shown in Figure 1 into a set of C functions flexible enough to meet all of the program's various requirements. The resulting functions are described in this article.

A MINI-TOOLBOX

The following functions comprise a mini-toolbox for dealing with time delays in serial communications situations:

- **wait** waits for an event to occur, or for a specified time interval to elapse (whichever comes first).
- **delay** waits for a specified time interval to elapse without reference to an event.
- **ticsm** returns the number of centiseconds (hundredths of a second) since midnight. Both **wait** and **delay** call **ticsm**.
- **sgetc** reads a character from a serial port. If no character is ready, it returns -1, which is the customary value of the symbol **EOF** (End Of File). My application uses this function to report the event (the arrival of an incoming character) to **wait**.

continued on page 62

continued from page 61

- **sputc** writes a character to a serial port. This function is not directly related to **wait**, but is presented here for completeness. Programs that perform serial input usually perform serial output as well.

HOW THEY WORK

Since **wait** is the most significant function of the group, we'll look at it first. Listing 1 provides the source code for **wait**, which expects two parameters:

1. **n** specifies a timeout interval, measured in centiseconds (a centisecond is the smallest interval measured by DOS's "get time" function). **n** is small enough to give reasonably precise control, but large enough to let an **unsigned short** value describe reasonable intervals (up to 655.35 seconds, or about 11 minutes).
2. **event** is a pointer to a function that tests whether the event of interest has occurred. If the event has occurred, the function returns a non-negative **short** value. If not, the function returns **EOF** (-1).

wait simply calls **event** over and over until **event** returns a non-negative value, or until **n** centiseconds have elapsed. **wait** reads the time of day before the first call to the event function and again after each call, computing elapsed time as the difference between the times.

wait deals with a time interval that spans midnight by adding the number of centiseconds in a day to the time of day after an event function call occurs. Thus, the "before" and "after" times are expressed relative to zero at midnight on the same day.

When **wait** is done, it returns the value last returned by the **event** function. This is **EOF** if the timeout occurred; otherwise, it's the value that represents the event (for example, the character read from the serial port).

Notice how the **event** function is identified: a pointer to **event** is **wait**'s second parameter. In C, a pointer to a function is represented by the function's name, with

no parentheses after it. Thus, a call to **wait** that specifies an event function named **com_in** might look like this:

```
result = wait(500,com_in);
```

For comparison, a direct call to **com_in** looks like this:

```
result = com_in();
```

Inside **wait**, the **event** parameter is declared as a pointer to a function that returns an **int**. Near the end of the **for** loop, the function pointed to by **event** is called with no parameters.

wait would be easier to understand if we made it call **sgetc** directly by name, dispensing with the function pointer altogether. But **wait** would then be limited to testing for a single specific named event. To test for another event, we would have to write another version of **wait** to call a corresponding **event** function. A function pointer enables **wait** to test for an unlimited variety of events, which are selectable at runtime rather than at compile time. The result is a much more useful function that is only fractionally more complex and less efficient.

TICSM.C (Listing 2) contains the source code for **ticsm**. This function calls Turbo C's **gettime** function, which returns the time of day in a **time** structure. **ticsm** then converts the time contained in the **time** structure to hundredths of a second.

DELAY.C (Listing 3) contains the source code for **delay**. It follows logic similar to that of **wait**, except that **delay** doesn't call an event function. Instead, **delay** simply checks the time over and over again, until the timeout interval has elapsed.

If you understand the value of reusable code, you're probably wondering why **delay** doesn't simply call **wait** with an event that never happens, as shown in the following code:

```
int no_event()
{ return(EOF); }
```

```
void delay(n)
  unsigned n;
{ return( wait(n,no_event) ); }
```

... or even with a macro, like this:

```
#define delay(n) wait(n,no_event)
```

In fact, I initially made **delay** call **wait**. That approach caused debugging problems, so I later

backed off and made **delay** a separate function. I'll discuss this problem in a moment.

SGETC.ASM (Listing 4) and **SPUTC.ASM** (Listing 5) contain the source code for **sgetc** and **sputc**, respectively. These listings are written in assembler because they manipulate hardware and need to be as time-efficient as possible. An important note: Assembler **SGETC.ASM** and **SPUTC.ASM** with the **/MX** assembler command in force, so that the Turbo C linker recognizes the symbols during the link pass. For those who wish to experiment with these routines but do not have an assembler, the **.OBJ** files are provided in the listings archive on CompuServe.

A complete explanation of these functions is outside the scope of this article, since both functions are intimately tied to the PC's serial communications hardware. In essence, **sgetc** checks the status of a COM port. If the receive buffer contains a character, **sgetc** returns the character; if not, **sgetc** returns -1. **sputc** simply places a character in the transmit buffer; if the buffer is already full, the PC's hardware forces the function to wait.

THE ROUTINES IN USE

COMTEST.C (Listing 6) illustrates the use of the functions described in this article. When **COMTEST** receives a character from the serial port, it displays the character's hexadecimal value and "echoes" a character whose value is one greater than the received character. If the timeout interval expires without a character coming in, the program sends an impatient message.

COMTEST uses a "help function" named **comin** to call **sgetc**. This is necessary because **sgetc** expects a parameter that gives a communication port number, and the parameter list of a call to **wait** has no place for one.

To read a character, the program calls **wait** with two parameters: the timeout interval and a pointer to **comin**.

COMKEY2.C (Listing 7) contains a function (**com_key**) that is similar to **comin** but tests for two

continued on page 65

TURN UP THE POWER WITH TURBO TOOLBOXES!

Add power to your Turbo language programs with the Borland Turbo Toolboxes.* They provide you with source code and routines to be added into your programs so you don't have to reinvent the wheel. And you don't pay royalties on your own compiled programs that include the Toolboxes' routines.

TURBO C®

TURBO C RUNTIME LIBRARY SOURCE CODE

An indispensable tool for serious Turbo C programmers! The Runtime Library Source Code lets you get even more out of Turbo C's flexibility and control, with a library of more than 350 functions you can customize or use as is in your Turbo C programs. You get the source for the standard C library, math library and batch files to help with recompiling and rebuilding the libraries.*

TURBO PASCAL®

Turbo Pascal Runtime Library Source Code coming soon!

TURBO PASCAL DATABASE TOOLBOX

With the Turbo Pascal Database Toolbox you can build your own powerful, professional-quality database programs. Included is a free sample database with source code and two powerful problem-solving modules.

Turbo Access™ quickly locates, inserts, or deletes records in a database using B+ trees—the fastest method for finding and retrieving database information.

Turbo Sort™ uses the Quicksort method to sort data on single items or on multiple keys. Features virtual memory management for sorting large data files.

TURBO PASCAL NUMERICAL METHODS TOOLBOX

Turbo Pascal Numerical Methods Toolbox implements the latest high-level mathematical methods to solve common scientific and engineering problems. Fast. Every time you need to calculate an integral, work with Fourier Transforms, or incorporate any of the classical numerical analysis tools into your programs, you don't have to reinvent the wheel. It's a complete collection of Turbo Pascal routines and programs that gives you applied state-of-the-art math tools. Includes two graphics demo programs to give you the picture along with the numbers. Comes with complete source code.

TURBO PASCAL TUTOR

Turbo Pascal Tutor is everything you need to start programming in Turbo Pascal. It consists of a manual that takes you from the basics up to the most advanced tricks, and a disk containing sample programs as well as learning exercises.

It comes with thousands of lines of commented source code on disk, ready for you to compile and run. Files include all the sample programs from the manual as well as several advanced examples dealing with window management, binary trees, and real-time animation.

TURBO PASCAL EDITOR TOOLBOX

Turbo Pascal Editor Toolbox gives you three different text editors. You get the code, the manual, and the know-how. We provide all the editing routines. You plug in the features you want.

MicroStar™: A full-blown text editor with a complete pull-down menu user interface.

FirstEd™: A complete editor equipped with block commands, windows, and memory-mapped screen routines.

Binary Editor: Written in assembly language, a 13K "black box" that you can easily incorporate into your programs.

TURBO PASCAL GRAPHIX TOOLBOX

Turbo Pascal Graphix Toolbox is a collection of tools that will get you right into the fascinating world of high-resolution monochrome business graphics, including graphics window management. Draw both simple and complex graphics. Store and restore graphic images to and from disk.

TURBO PASCAL GAMEWORKS

Explore the world of state-of-the-art computer games with Turbo Pascal GameWorks. Using easy-to-understand example games, it teaches you theory and techniques to quickly create your own computer games. Comes with three ready-to-play games: Turbo Chess,™ Turbo Bridge,™ Turbo Go-Moku.™

TURBO PROLOG®

TURBO PROLOG TOOLBOX IS SIX TOOLBOXES IN ONE

More than 80 tools and 8,000 lines of source code help you build your own Turbo Prolog applications. Includes toolboxes for menus, screen and report layouts, business graphics, communications, file-transfer capabilities, parser generators, and more!

TURBO BASIC®

TURBO BASIC DATABASE TOOLBOX

With the Turbo Basic Database Toolbox you can build your own powerful, professional-quality database programs. Includes *Trainer*, a demonstration program that graphically displays how B+ trees work and a free sample database with source code. The Toolbox enhances your programming with 2 problem-solving modules:

Turbo Access quickly locates, inserts, or deletes records in a database using B+ trees—the fastest method for finding and retrieving database information.

Turbo Sort uses the Quicksort method to sort data on single items or on multiple keys.

TURBO BASIC® EDITOR TOOLBOX

Turbo Basic Editor Toolbox will help you build your own superfast editor to incorporate into your Turbo Basic programs. We provide all the editing routines. You plug in the features you want! We've included two sample editors with complete source code.

MicroStar: A full-blown text editor with a pull-down menu user interface and all the standard features you'd expect in any word processor.

FirstEd. A complete editor with windows, block commands, and memory-mapped screen routines, all ready to include in your programs.

System requirements: All Turbo Toolboxes for the IBM PS/2™ and the IBM® family of personal computers and all 100% compatibles. PC-DOS (MS-DOS®) 2.0 or later. Turbo C Runtime Library Source Code requires Turbo C 1.5 or later. Turbo Pascal Toolboxes require Turbo Pascal 4.0 or later and 256K RAM. Turbo Prolog Toolbox requires Turbo Prolog 1.1 or later and 384K RAM. Turbo Basic Toolboxes require Turbo Basic 1.0 or later and 640K RAM.

*Does not include source for graphics or floating point emulator.



For the dealer nearest
you or to order, call
(800) 543-7543

LISTING 1: WAIT.C

```

/* WAIT.C: wait() */
#include <stdio.h>

extern unsigned long ticsm();

/*****
 * Wait a length of time or until an event occurs,
 * whichever comes first.
 * IN:  "n" = maximum time to wait, in centiseconds.
 *      "event" points to a function that returns -1 if the timeout
 *      occurs, or a non-negative integer if the event occurs.
 *      This is designed primarily for use with "cgetc", which
 *      returns -1 for "no character ready."
 * RETURNS: -1 if the time-out occurred.
 *      The value returned by (*event()) if the event occurred.
 *****/
int wait(n,event)
    unsigned n;
    int (*event)();
{
    unsigned long start_time, timeout_time, current_time;
    int i;

    /* Compute start time & timeout time. If the timeout time overflows
    midnight, that's OK; a test inside the loop takes care of it. */
    start_time = ticsm();
    timeout_time = start_time + n;

    /* Loop until the timeout happens or the event occurs. */
    for ( ; ; )
        {

        /* Compute current time. If we've wrapped past midnight, add a day's
        worth of centiseconds. */
        current_time = ticsm();
        if ( current_time < start_time )
            current_time += 8640000L;

        /* Check for timeout. */
        if ( current_time >= timeout_time )
            return(-1);

        /* Do the function. */
        i = (*event)();
        if ( i!=EOF )
            return(i);
        }
}

```

LISTING 2: TICSM.C

```

/* TICSM.C: ticsm() */
#include <dos.h>

/*****
 * Compute the current time of day in centiseconds since midnight.
 * RETURNS: centiseconds since midnight.
 *****/
long ticsm()
{
    struct time tod;

    gettime(&tod);
    return( 360000L*tod.ti_hour + 6000L*tod.ti_min +
           100*tod.ti_sec + tod.ti_hund );
}

```

LISTING 3: DELAY.C

```

/* DELAY.C -- unconditional wait. */
#include <stdio.h>

extern unsigned long ticsm();

/*****
 * Unconditional wait for "u" centiseconds. This is the same as wait()
 * except that it doesn't look for an event. Calling wait() would be
 * easier, but would not permit us to replace wait() with a routine
 * that doesn't look at time when debugging.
 *****/
void delay(n)
    unsigned n;
{
    long start_time, timeout_time, current_time;

    /* Compute start time & timeout time. */
    start_time = ticsm();
    timeout_time = start_time + n;

    /* Loop until the timeout happens. */
    for ( ; ; )
        {

        /* Compute current time. If we've wrapped past midnight, add a day's
        worth of centiseconds. */
        current_time = ticsm();
        if ( current_time < start_time )
            current_time += 8640000L;

        /* Check for timeout. */
        if ( current_time >= timeout_time )
            return;
        }
}

```

LISTING 4: SGETC.ASM

```

page 62,120
; SGETC.ASM -- read a character from a serial port.
; BE SURE TO ASSEMBLE WITH THE /MX COMMAND!!
;
;int sgetc(port)
; unsigned port;

;IN:  port = serial port number. Any even number is interpreted
;      as 0, and any odd number is interpreted as 1.
;RETURN:EOF if no character is ready; else the character (0-0xFF).

bios equ 21H
EOF equ -1

PUBLIC _sgetc
_TEXT SEGMENT BYTE PUBLIC 'CODE'
ASSUME CS:_TEXT
_sgetc PROC NEAR
    push bp
    mov bp,sp
    push di
    push si

    mov ax,[bp+4] ; DX = port.
    dec ax
    and ax,1 ; AX = port-1, reduced to range [0-1]
    push ds ; Note we save DS after usual stuff!

    mov si,ax
    add si,si ; SI = 2*(port-1).
    mov dx,40H
    mov ds,dx ; *** DS points to device table. ***
    mov dx,[si]
    add dx,5 ; DX = status port address.
    in al,dx ; Read line status.
    test al,1 ; Receive buffer full?
    jz sgetcEOF ; No.
    xor ax,ax
    mov dx,[si] ; DX = COMn data port address.
    in al,dx ; read byte.

    jmp SHORT sgetc2

sgetcEOF: ; No character ready;
    mov ax,EOF ; return AX = EOF.

sgetc2: ; **** DS restored. ****
    pop ds
    pop si
    pop di
    mov sp,bp
    pop bp
    ret

_sgetc endp
_TEXT ends

end

```

listing continued on page 66

WAIT

continued from page 62

alternative events: either input on a serial port, or input on the keyboard. **com_key** uses the standard library function **kbhit** to test for the presence of keyboard input. Since **com_key** returns two pieces of data and cannot have parameters (remember, it's called through a function pointer), it deposits the data in global variables. This example gives you some idea of how readily **wait** can be adapted to complex situations.

WAITDEMO.C (Listing 8) is a simplified version of the program in Listing 6. WAITDEMO tests for input from the keyboard instead of input from the serial port. It's useful for trying out **wait** without attaching a communications device to a serial port.

WHY **delay** DOESN'T CALL **wait**

I mentioned earlier that I had written an original version of **delay** that called **wait** with a pointer to an event function that always returns **EOF**. This removed any reference to external events, and caused **wait** to act like a simple time-delay function. In true economical programming style, I had built upon the services provided by **wait**, instead of duplicating them.

While debugging my specialized communications program, however, I found it useful to call another computer that was running an ordinary "plain vanilla" modem program, rather than a second copy of my communications program. I watched the other computer receive my program's messages, and entered appropriate responses through the keyboard. In this testing situation, a normally functioning **wait** would have made debugging impossible, since I couldn't possibly type responses fast enough to prevent my program from timing out when I didn't want it to.

I solved this problem by writing a special version of **wait** that waits for an event forever if the timeout interval is specified as an even number. The original timeout test is shown below:

```
if ((current_time >= timeout_time))
    return(-1);
```

The modified timeout test looked like this (remember, **n** is the timeout interval):

```
if ((n%2) &&
    (current_time >= timeout_time))
    return(-1);
```

Since timeout intervals ordinarily are whole numbers, my program ignores all of the even-valued timeouts without making a single change in its source code. I tested the program's timeout processing by temporarily changing the timeout intervals to odd values, in a few selected cases at a time.

This trick had a side effect, though: Because the original **delay** called **wait**, **delay** ignored even-valued timeout intervals, too. Since **delay** had no event to wait for, this made even-valued **delay** calls wait forever!

I could have given all my **delay** calls odd-valued timeout intervals, but only by making numerous, scattered changes in source code. This was an unattractive prospect. Instead, I wrote a separate version of **delay** that doesn't call **wait**, and thus is not affected by the change in the debugging version of **wait**.

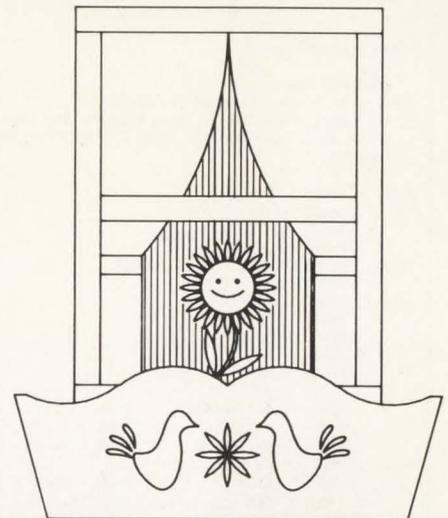
I could have put the new version of **delay** in a separate library along with the older, debugging-oriented version of **delay**, but I foresaw that the problem I had just solved would recur whenever I had to debug a program that uses **wait** and **delay**. Therefore, it seemed wise to leave the new, independent version of **delay** in my working library, rather than to have to remember the "gotcha" caused by the original version each time I debugged a program.

The moral: Principles like "design your code for reusability" are excellent guides to good software design, but few of them are infallible, especially in sophisticated applications. There's no substitute for using your own hard-won experience as a guide. When a rule doesn't work—break it! ■

Jonathan Sachs has worked as a software developer and technical writer since 1971. He operates a consulting company near San Francisco.

Listings may be downloaded from CompuServe as WAIT.ARC.

THE WINDOW BOX



WINDOW BOX (n):

1. A flower box that enhances the beauty of a window.
2. A windowing toolbox for C programmers.

Enhance the beauty of your C applications with **THE WINDOW BOX**.

ADD SOME PIZAZZ!

THE WINDOW BOX lets you **ELECTRIFY** your programs with pop-up windows, pull-down menus with highlight bar selection, and context sensitive help. Watch your screen go blank when your program is idle. Assign functions to the function keys. Much more!

ADD SOME POWER!

Read many fields with one operation. Data entry windows offer many formats, complete cursor navigation, and let you tie verification functions to any field. Use scrolling and text-editing windows, too. Print a window, not necessarily the whole screen. (Super for mailing labels!) Much more!

SOURCE CODE PROVIDED.

Contains no assembler code! Only standard C code. See how things work. Understand how things work. Change how things work. Compatible with all major C compilers. Requires MS-DOS/PC-DOS.

REASONABLE PRICE.

And no royalties. Only \$49.50 including shipping and tax. Or, try the demo disk and inspect the manual for only \$10. Like what you see, and apply this \$10 to the purchase price. Overseas add \$5.00 per order and we will Air Mail.

SATISFACTION GUARANTEED, or return in 30 days for a full refund.

Mastercard/Visa: Call **412-487-4282**.

Or, send checks (U.S. funds) to:

Vertical Horizons Software
113 Lingay Drive
Glemshaw, PA 15116

```

        page      62,120
; SPUTC.ASM -- write a character to a serial port.
;
;          BE SURE TO ASSEMBLE WITH THE /MX COMMAND!
;
;char sputc(ch,port)
; char ch;
; unsigned port;
;IN:      ch = the character to put.
;         port = the serial port number. Any even number is interpreted
;         as 0, and any odd number is interpreted as 1.
;RETURN:ch.

bios     equ      21H
EOF      equ      -1

        PUBLIC   _sputc
_TEXT    SEGMENT BYTE PUBLIC 'CODE'
        ASSUME  CS:_TEXT
_sputc   PROC     NEAR
        push    bp
        mov     bp,sp
        push    di
        push    si

        mov     al,[bp+4]      ; AL = "ch".
        mov     ah,1          ; AH = 1 (for "output").
        mov     dx,[bp+6]      ; DX = port
        dec     dx
        and     dx,1          ; DX = port-1, reduced to range [0-1]
        int     14H           ; Do it.
        mov     al,[bp+4]      ; AL = "ch"...
        xor     ah,ah         ; AX = "ch".

        pop     si
        pop     di
        mov     sp,bp
        pop     bp
        ret

_sputc   endp
_TEXT    ends

        end

```

LISTING 6: COMTEST.C

```

/* COMTEST.C -- a test of the SGETC and SPUTC functions. */
#include <stdio.h>

extern int wait();
extern void delay();
extern int sgetc(), sputc();

/*****
*This is the event function. It returns the next COM1 character,
* or EOF if none is waiting.
*****/
int comin()
{
return( sgetc(1) );
}

main()
{
int i;

printf( "Press a key on a serial device connected to COM1;" );
printf( "\nPress ENTER on the serial device to quit." );

while ( ( i = wait(500,comin) ) != 0x0D )
{
if ( i == EOF )
printf( "\nCome on, press a key! " );
else
{
printf( "\nThe hex value of that key is 0x%2X.", i );
sputc(i+1,1);
}
}

delay( 50 );
printf( "\nGoodbye, world!\n" );
}

```

LISTING 7: COMKEY2.C

```

#include <stdio.h>

/*****
*This is the event function. If a keyboard character is ready it
* returns the character in "keyready"; else it returns EOF.
* Similarly, if a character is waiting on the serial port it returns
* character in "comready"; else it returns EOF. The function waits
* for one of the above to be ready, giving preference to the com
* port, or for a timeout. It handles function keys, etc., in the
* customary way: by returning 0 in the low byte and the scan code
* in the high byte. The function returns 0 if EITHER a keyboard
* character or a serial character is read; EOF if NEITHER is ready.
*****/
int com_key()
{
/* Initialize keyready. */
keyready = EOF;

/* Set comready to serial data or EOF. If EOF, return. */
comready = sgetc(1);
if ( comready != EOF )
return( 0 );

/* No serial data. If no keyboard data, return. */
if ( !kbhit() )
return( EOF );

/* Keyboard data is available. If it's ASCII, getche() returns its
value. If it's an extended key, getche() returns 0 & a subsequent
getch() returns its scan code. */
keyready = getche();
if ( !keyready )
keyready = getch() << 8;
return( 0 );
}

```

LISTING 8: WAITDEMO.C

```

/* WAITDEMO.C -- a demonstration of the WAIT and DELAY functions. */
#include <stdio.h>

extern int wait();
extern void delay();

/*****
*This is the event function. It returns the next keyboard character,
* or EOF if none is waiting. It handles function keys, etc., in the
* customary way: by returning 0 in the low byte and the scan code in
* the high byte.
*****/
int keyin()
{
int i;

if ( kbhit() )
{
/* A key was pressed. If it's an ASCII key, getche() returns
its value. If it's an extended key, getche() returns 0 & a
second getche() returns its scan code. */
i = getche();
if ( i )
return(i);
else
return( getch() << 8 );
}
else
return( EOF );
}

main()
{
int i;

printf( "Press a key and I'll tell you its hex value, or ENTER to quit. " );

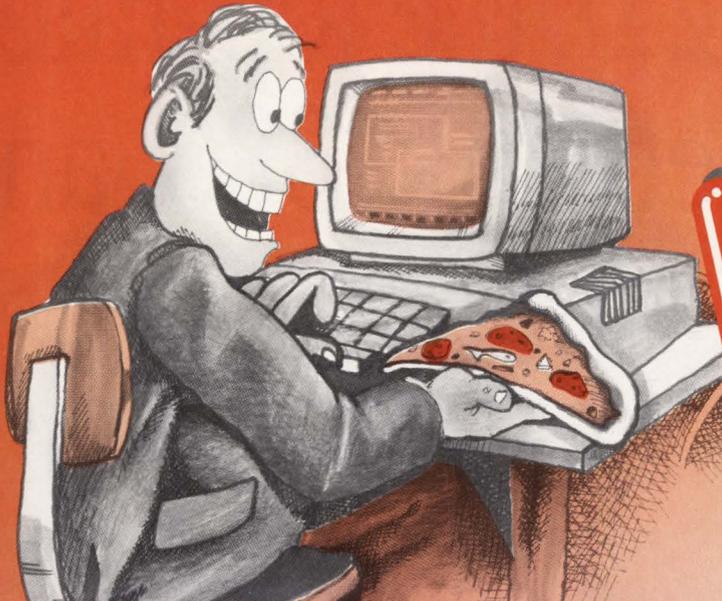
while ( ( i = wait(500,keyin) ) != 0x0D )
{
if ( i == EOF )
printf( "\nCome on, press a key! " );
else
{
printf( "\nThe hex value of that key is 0x%2X.", i );
printf( "\nPress another key. Press ENTER to quit. " );
}
}

delay( 50 );
printf( "\nGoodbye, world!\n" );
}

```

Multi-Edit vs. **PIZZA**

With EVERYTHING!



- Is your editor **OUT TO LUNCH?**
 - Does it handle **ALL OF YOUR NEEDS?**
 - Is it flexible, programmable and reconfigurable?
 - **MOST IMPORTANTLY**, is it **EASY TO USE?**
- OR WOULD YOU RATHER BE EATING PIZZA?**

Only MULTI-EDIT tastes this good!

Full automatic Windowing and Virtual Memory
 Edit multiple files regardless of physical memory size
 Easy cut-and-past between files
 View different parts of the same file

Powerful, EASY-TO-READ high-level macro language
 Standard language syntax
 Full access to ALL Editor functions

Language-specific macros for C, PASCAL, BASIC and MODULA-2
 Smart Indenting
 Smart brace/parenthesis/block checking
 Template editing
 More languages on the way

Terrific word-processing features for all your documentation needs
 Intelligent word-wrap
 Automatic pagination
 Full print formatting with justification, bold type, underlining and centering
 Flexible line drawing
 Even a table of contents generator

Compile within the editor
 Automatically positions cursor at errors
 Allocates all available memory to compiler

Complete DOS Shell.
 Scrollable directory listing
 Copy, Delete and Load multiple files with one command
 Background file printing

Regular expression search and translate
Condensed Mode display, for easy viewing of your program structure

Pop-up FULL-FUNCTION Programmer's Calculator and ASCII chart

and MOST IMPORTANT, the BEST USER-INTERFACE ON THE MARKET!

Extensive context-sensitive help
 Choice of full menu system or logical function key layout
 Function keys are always labeled on screen (no guessing required!)
 Keyboard may be easily reconfigured and re-labeled
 Extensive mouse support
 Easy, automatic recording and playback of keystrokes
 Anchovies easily removed

MULTI-EDIT COMBINES POWER WITH EASE OF USE LIKE NO OTHER EDITOR ON THE MARKET TODAY.

MULTI-EDIT \$99 COMPLETE
 VERSION 2

	Multi-Edit	BRIEF 2.0	Norton Editor	Vedit Plus	PIZZA WITH EVERYTHING
Edit 20+ Files larger than memory	Yes	Yes	No	Yes	12 slices
Powerful high level macro language	Yes	Yes	No	Yes	Italian
Full UNDO	Yes	Yes	No	No	No
Visual marking of blocks	Yes	Yes	Yes	No	Looks Good
Line, stream and column blocks	Yes	Yes	No	No	Use Knife
Automatic file save	Yes	Yes	No	No	No
Online help	Extensive	Limited	Limited	Limited	Extensive
Choice of keystroke commands or menu system	Yes	No	No	Yes	Menu Available
Function Key assignments labeled on screen (may be disabled)	Yes	No	No	No	No
Word processing functions	Extensive	Limited	Limited	Extra Cost	Difficult
Complete DOS shell	Yes	No	No	No	Deep Dish
Pop-up Programmer's Calculator and ASCII Table	Yes	No	No	No ASCII	No
Unlimited 'Off the Cuff' keystroke macros	Yes	No	No	Yes	Sauce on Cuff often
Allocates all available memory to compiler when run from within editor	Yes	No	No	No	Lots of bytes
Intelligent indenting, template editing and brace/parenthesis/block matching and checking for C, PASCAL, BASIC and MODULA-2	Yes	C Only	No	Limited	Limited Intelligence
Flexible condensed mode display	Yes	No	Yes	No	Definitely
PRICE	\$99	\$195	\$50	\$185	About \$12

Get Our FULLY FUNCTIONAL DEMO Copy for only \$4

To Order, Call 24 hours a day:
 1-800-221-9280 Ext. 951
 In Arizona: 1-602-968-1945
 Credit Card and COD orders accepted.

American Cybernetics
 1228 N. Stadem Dr.
 Tempe, AZ 85281

Requires IBM/PC/XT/AT/PS2 or full compatible, 256K RAM, PC/MS-DOS 2.0 or later. Multi-Edit and American Cybernetics are trademarks of American Cybernetics. BRIEF is a trademark of Underware, Inc. Norton Editor is a trademark of Peter Norton Computing, Inc. Vedit is a registered trademark of CompuView Products Inc. Copyright 1987 by American Cybernetics.

Paradox 2.0, the top-rated Network, 386, and



Paradox® is both the first family in DBMS and the top-rated relational database. Software Digest has ranked Paradox #1 for the past 2 years; PC Magazine gave Paradox its "Editor's Choice" award and InfoWorld named it 1987 "Product of the Year" for Database Systems.

Now there's OS/2

Paradox OS/2 is the newest member of the Paradox family—more are on the way and they're all 100% compatible with each other.

Paradox OS/2 allows you to take advantage of powerful OS/2 features such as addressing up to 16 megabytes of memory and running concurrent sessions. And Paradox OS/2 even lets you start new OS/2 sessions from within Paradox.

*Customer satisfaction is our main concern; if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks of their respective holders. Copyright ©1988 Borland International, Inc. BU 1228A

Harness the power of 386

Paradox 386 is powerful new DOS software for your powerful new hardware and it's designed exclusively for 80386-based systems. It also lets you ignore the old 640K limits and races through your data 32 bits at a time instead of just 16. It's a perfect solution for anyone faced with very large tables (tens of thousands of records or more) and/or large applications.

“As proof of Borland's commitment to delivering compatibility across diverse hardware and software environments, Paradox 386 and Paradox 2.0 can share the same databases and applications on a network.”

Giovanni Perrone, PC Week

Paradox . . . it's the PC database-management system equivalent to turbo-charging an M-series BMW.

Giovanni Perrone, PC WEEK ”

The Paradox Network really works

Network users, you need Paradox's multiuser capabilities. The network runs smoothly, intelligently and so transparently that multiusers can access the same data at the same time—without getting in each other's way. (But safeguards prevent multiple users from altering the same data at the same time.) And with screen refresh you get real-time data updates on your screen.

“[Paradox is] a true network application, a program that can actually take advantage of a network to provide more features and functions, things that can't be done with a standalone PC.”

Aaron Brenner, LAN Magazine

[Paradox] elegantly handles all the chores of a multiuser database system with little or no effort by network users.

*Mark Cook and Steve King
Data Based Advisor* ”

relational database, has now OS/2 versions!



“Query-by-Example” gives you the right answer, right now

Our “Query-by-Example” (QBE) technique is just one illustration of the technological leadership offered by Paradox for the past 2 years.

QBE is fast and simple to use. Simply call up a form and check off the information you want.

/(F6) to include a field in the ANSWER; (F5) to give an Example

SALES	Stock	Customer	Init	Street
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				

PRODUCTS	Stock	Description	Price	Quant
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				

ANSWER	Customer	Init	Description	Price
1	Chevalier		Hink handkerchiefs (13)	12,995.00
2	Ejseph, III		Robot-valet	149,995.00
3	Fahd		Twitching panthers	576,000.00
4	Hanover		Digital grandfather clock	4,995.00
5	Hanover		Robot-valet	149,995.00
6	Hathems		Robot-valet	149,995.00
7	Hathems		Stretch III Beetle	58,450.00
8	Rogor		Robot-valet	149,995.00
9	Ratier		Hink handkerchiefs (13)	12,995.00
10	Ratier		Robot-valet	149,995.00

Without having to write a line of code, you can, for example, get answers to queries like: *Find all the items we sold for more than \$1000 and tell me who ordered them.*

An artificial intelligence technique called “heuristic query

optimization” gives Paradox’s QBE the ability to figure out not just the right answer, but also the fastest way to get the right answer.

QBE makes high-speed links between one piece of data and another and quickly sees the relationships your question calls for.

PAL:™ A powerful programming language

PAL, the Paradox Application Language, is a full-featured, high-level, structured database programming language that lets you write sophisticated Paradox programs (scripts) and applications. It includes such powerful features as looping constructs, arrays, branching, procedures, and a full set of functions.

“Most people we meet who give Paradox a try, end up switching to it . . .

Mark Cook and Steve King
Data Based Advisor”

There’s a Paradox 2.0 version for you

Whether you’re a DOS or OS/2 user, there’s a Paradox version for you.



60-Day Money-back Guarantee*

For a brochure or the dealer nearest you, call (800) 543-7543



CERTAINTY FACTORS IN TURBO PROLOG

Applications in chemistry: An infrared spectroscopy peak-matching facility for the identification of small organic compounds.

Tom Castle

A major aspect of analytical chemistry deals with obtaining experimental data in order to identify or quantify a chemical compound. One of the main tools used by a chemist to gather this data is spectroscopy. For those who slept through

Chemistry 101, *spectroscopy* is an instrumental method that measures the interaction between a material and electromagnetic radiation (most commonly, light). Some spectroscopic methods rely upon the absorption of light by a material; other methods depend on either scattering or fluorescence.

Infrared (IR) spectroscopy is a standard tool of the analytical chemist. The spectrum of infrared light is of sufficient energy to cause various vibrational excitations of organic molecules when the light is absorbed. Depending on the chemical nature of the bonds within the molecule, different frequencies are absorbed. In fact, one type of bond may absorb several different frequencies of infrared light, depending on the vibrational mode of excitation. Thus, even a simple molecule possesses a complex and unique infrared absorption spectrum. Because of this uniqueness, the technique for identifying small, organic molecules is extremely valuable. A typical IR spectrum is shown in Figure 1.

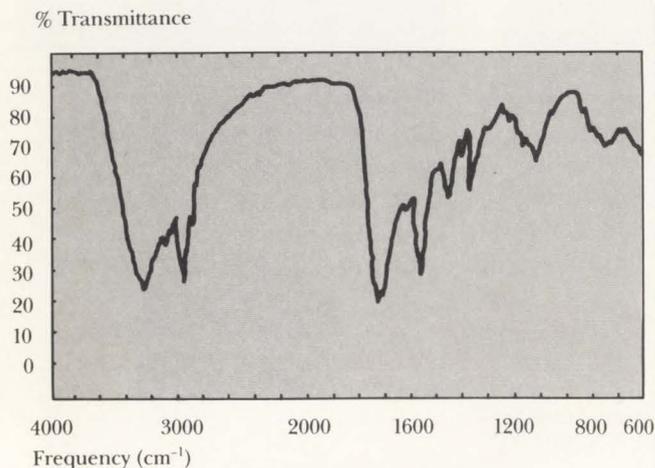


Figure 1. A typical infrared absorption spectrum.

The complexity of an IR spectrum, and the unique relationship between a compound and its spectrum, are a blessing and a curse. On the positive side, matching an experimental spectrum to the spectrum of a known compound is unequivocal evidence for an identification. On the negative side, the sheer complexity of a spectrum only allows estimates of substructure classes based upon visual inspection of the spectrum. Luckily, computers appeared on the scene.

Since that time, chemical structure elucidation has become heavily dependent on computers. The chemist not only uses the computer to perform literature searches of various compounds, families, and substructures, but also accesses huge databases that contain virtually any physical or chemical property of most compounds.

An expert system can help maximize the benefits of computers and computer-based information. An expert system is simply a machine equivalent of a person who is knowledgeable about a specific range of subject matter. The system must be able to make decisions based upon the facts in its database, to answer queries, and to explain its answers to the user. (For a detailed look at expert systems, see *TURBO TECHNIX*, March/April, 1988.)

IR PEAK MATCHING PROGRAM

Listing 1 shows a bare-bones expert system for the identification of organic compounds using the process of peak matching the infrared spectra of experimental and known compounds. For the sake of simplicity, I've kept the database to a minimum. Each record contains a compound name and a list of the integer values of the absorption peaks for that compound. Intensity values for the various peaks are not considered. The permanent database (IRPEAKS.DAT) is shown in Listing 2.

Run the program in Listing 1 and enter experimental values. The program compares those values to the lists of peak frequencies in the database. The **intersect** rule (adapted from the **difference** rule found in Ivan Bratko's book, *Prolog Programming for Artificial Intelligence*) is used to find the common elements in the two lists. Once the number of common elements is determined, a score is assigned based on the degree of matching. The compounds with scores greater than a threshold value (arbitrarily set at 50 percent) are asserted into a temporary database.

Data reliability hinges on three separate conditions: accuracy, precision, and reproducibility.

Once all of the "hits" have been asserted, they are converted into a list (using **db_list**) and sorted. The sorted list is then displayed by the **write_list** predicate. The user also has the opportunity to examine the actual database information for any record that came up in the "hit" list. Although the program is simple, it exemplifies an important consideration in data analysis—uncertainty.

DATA RELIABILITY

Data reliability hinges on three separate conditions: accuracy, precision, and reproducibility. For numerical data, these are commonly recognized terms. *Accuracy* is the conformity of a measure to a true or standard value. *Precision*

continued on page 72

LISTING 1: IRANAL.PRO

```

/*****
/*
/*                               IRANAL.PRO                               */
/*                               Using Turbo Prolog for the                */
/*                               identification of small organic chemicals  */
/*                               by Infrared Spectroscopy                  */
/*                               copyright (c) by Tom Castle, 1988        */
/*
/*****

/*****
/* The following program was designed to provide assistance in the */
/* structure determination of small, organic molecules from simple */
/* data sets obtained from infrared spectroscopy. The program is a */
/* simple example of a typical EXPERT SYSTEM. Our input data is */
/* compared against facts stored in a database by use of the */
/* designed INFERENCE ENGINE. Two important ideas are approached */
/* by this program. The first is CERTAINTY. The CERTAINTY SCORE */
/* is an approximation of how confident we feel about the data */
/* presented or the answers obtained from the INFERENCE ENGINE. */
/* The peak matching facility generates a CERTAINTY SCORE which */
/* is calculated from the total number of matches and mismatches */
/* encountered between experimental data and database information. */
/* The second idea probed is GATING for searching unreliable */
/* numerical values in a database. Much of what science tells us */
/* is wrong. Wrong in the sense that it is generally an */
/* approximation although sometimes a very precise approximation. */
/* Wrong also in the sense that facts are generally a statistically */
/* derived value of observations continually being refined to more */
/* closely reflect the truth. We must use GATING to offer */
/* ourselves a window of variable width with which to search for */
/* facts in the database since databases only recognize exact */
/* instances of unification. Life generally isn't that black and */
/* white.
/*****

DOMAINS
    sampling = real
    certainty, data = integer
    name = string
    target = t(name, certainty, sampling)
    targetlist = target*
    datalist = data*
INCLUDE "tdoms.pro"

DATABASE
/* This is the permanent database predicate kept in the */
/* file "irpeaks.dat" : compound name and ir peak list */

    compound(name, datalist)

/* The next few database predicates are temporary items */
/* to hold the analysis and search results. */

    certain_thresh(integer)
    exp_data(datalist)
    gate_width(integer)
    hits(target)

PREDICATES
    adjust_cursor(ROW, COL, ROW, COL)
    bubblesort(targetlist, targetlist)
    db_list(targetlist)
    examine_answers
    gated_member(data, datalist)
    get_data
    intersect(datalist, datalist, datalist)

```

```

list_len(datalist, integer)
makelist(datalist, ROW, COL)
match
process(integer)
process1(integer)
retract_temps
swap(targetlist, targetlist)
write_answers(targetlist)
write_hits(targetlist, integer)
write_list(datalist, ROW, COL)
INCLUDE "tpreds.pro"
INCLUDE "menu.pro"

GOAL
consult("irpeaks.dat"),
assert(gate_width(10)),
assert(certain_thresh(50)),
makewindow(1,32,7," IR Peak Matching Facility ",0,0,25,80),
repeat,
clearwindow,
menu(10,30,23,7,
["Enter IR Data",
"Search",
"Change Gate Width",
"Exit"],
"",1, Choice),
process(Choice).

CLAUSES
/*****
/*
/*          USER INTERFACE          */
/* The PROCESS clauses are the responses to the menu items in the */
/* GOAL. The PROCESS1 clauses respond to the second menu for    */
/* examining database records.                                     */
*****/

process(0):- fail.
process(1):- get_data, !, fail.
process(2):- match,
             db_list(List),
             bubblesort(List, SortedList),
             write_answers(SortedList),
             examine_answers,
             retract_temps, !, fail.

process(3):-
    gate_width(GateWidth),
    write("The current variability gate for the "),
    write("search is set at ", GateWidth, "."), nl,
    write("Enter a new gate size or hit <Return> : "),
    readint(New),
    retract(gate_width(_)),
    asserta(gate_width(New)), !, fail.
process(4):- exit.

get_data:-
    retract_temps,
    clearwindow,
    cursor(1,10),
    write("Enter data obtained from IR analysis"),
    cursor(2,10),
    write("      Hit <Return> to quit."),
    makelist(Datalist,4,3), /* a writelist with cursor */
    asserta(exp_data(Datalist)).

write_answers(List):-
    clearwindow,
    field_str(0,2,20,"Compound"),
    field_str(0,25,20,"Certainty Score"),
    field_str(0,45,20,"Sampling Score"),
    write_hits(List,2).

```

CERTAINTY FACTORS

continued from page 71

is the degree of tolerance or refinement of a measurement. *Reproducibility* is generally a statistically derived value that represents the variation of a parameter between different measurement events.

The collection and processing of IR peak data illustrate how experimental data can become unreliable. The peak frequencies can be obtained by having the spectrophotometer generate peak tables from the spectrum. Alternatively, peak lists can be generated by either visual inspection or mechanical digitization of a published spectrum. The method used to obtain peak lists dictates the degrees of accuracy, precision, and reproducibility.

Non-numerical data can also be subject to these conditions of data reliability. If questioned about a given quality, the answer could be wrong (accuracy), vague (precision), or change from one query to another query (reproducibility). It's also difficult to establish a hierarchy for descriptive terms such as "good," "ok," "satisfactory," "partly cloudy," and so forth. To increase data reliability, this type of subjective data is often converted into numerical terms.

How do we take data reliability into account? A quick-and-dirty method to compensate for unreliability is *gating*, where all experimental values are given a plus or minus tolerance of a specified amount. Experience dictates how wide the gate should be. Some situations are amenable to statistical analysis for determining the appropriate gate width. I've provided a way to adjust the gate width in Listing 1.

To incorporate gating into the program, I changed the common **member** predicate to the **gated_member** predicate. The new rule succeeds if a value is within a range that is specified by the gate width on either side of any member in a list.

DATA SAMPLING

One area of data reliability, data sampling, doesn't involve the numerical values of the data themselves. Using the IR spectrum again as an example, there are many small intensity and partially overlapping peaks in any IR spectrum. These data may or may not

The program must assign a score that reflects the amount of matching from each database record. This score is often called a certainty or confidence factor.

be entered into peak lists. For instance, one chemist might count a different number of peaks than another chemist, or obtain the data from a slightly different spectrum of the same compound, or use a different method of generating peak lists for different compounds.

The user should have some indication if there are differences in sample size between the experimental data set and the database records. Although there are rigorous statistical tools for evaluating sampling, the sampling score has been kept simple. The equation is:

$$\text{Sampling_Score} = 1 / (\text{abs}(\text{Db_len} - \text{Exp_len}) + 1)$$

Db_len and **Exp_len** represent the number of members in the database and experimental lists, respectively. The **abs** operator obtains the absolute value of the

continued on page 74

```
write_hits([],_).
write_hits([t(N,C,S)|T],Row):-
    field_str(Row,2,20,N),
    Row1 = Row + 1,
    cursor(Row,30),
    write(C),
    cursor(Row,50),
    writef("%3.2",S),
    write_hits(T,Row1).

examine_answers:-
    repeat,
    menu(15,50,7,7,
        ["Examine Database Entry",
         "Return"], "",1,Choice),
    process1(Choice).

process1(0):- fail.
process1(1):-
    nl,nl,
    write("Enter the compound name you want to examine : "),
    nl, readln(Name),nl,
    cursor(Row,_),
    compound(Name,Exp_list),
    write_list(Exp_list,Row,3), !,
    fail.
process1(2).

retract_temps:-
    retract(exp_data(_)), fail.
retract_temps.

/*****
/*          LIST MANIPULATIONS          */
/* Most of these are derivatives of the common */
/* predicates: readlist, writelist, listlen */
/* found in the TProlog Reference Manual or the*/
/* TProlog Toolbox TPRED.S.PRO file.      */
*****/

makelist([H|T],Row,Col):-
    cursor(Row,Col),
    readint(H), !,
    adjust_cursor(Row,Col,Row1,Col1),
    makelist(T,Row1,Col1).
makelist([],_,_).

write_list([],_,_):- !.
write_list([H|T],Row,Col):-
    cursor(Row,Col),
    write(H),
    adjust_cursor(Row,Col,Row1,Col1),
    write_list(T,Row1,Col1).

adjust_cursor(R,C1,R,C2):-
    C1 < 70, !,
    C2 = C1 + 6.
adjust_cursor(R,_,R1,C):-
    C = 6,
    R1 = R + 1.

list_len([],0).
list_len([_|T],N):-
    list_len(T,X),
    N = X + 1.
/* the listlen predicate in the */
/* Toolbox TPRED.S.PRO file has a */
/* stringlist domain for its arg.*/
/* This has an integerlist.      */
```

continued from page 73

```

/*****
/*
/* INFERENCE ENGINE */
/* These are the rules which govern the search capabilities of the */
/* program along with the processing and decision-making functions.*/
/* The main rule is MATCH which counts the number of matches and */
/* mismatches between the ir experimental data set and the */
/* database data sets. A CERTAINTY or CONFIDENCE value is */
/* calculated along with a SAMPLING DIFFERENCE value. The database*/
/* entries with a CERTAINTY SCORE over a threshold value will be */
/* stored in a list. That list is then presented to the user. An */
/* opportunity to inspect a database record is also available. */
/*****

```

```

match:-          /* this is what the psychologists*/
  exp_data(Exp_list), /* call free association.*/
  list_len(Exp_list,Exp_len), /* It is unrestrained */
  cursor(1,30), /* backtracking. */
  write("Searching Database"),
  compound(Name,Db_list),
  intersect(Exp_list,Db_list,Common_list),
  list_len(Db_list,Db_len),
  list_len(Common_list,Com_len),
  min(Db_len,Exp_len,Min_len),
  Certainty = (Com_len * 100) / Min_len,
  certain_thresh(T),
  Certainty > T,
  Sampling = 1 / (abs(Db_len - Exp_len) + 1),
  asserta(hits(t(Name,Certainty,Sampling))),
  fail.

```

```

match.

```

```

/*****
/*
/* MORE LIST STUFF */
/* The GATED_MEMBER is a take-off of the MEMBER */
/* predicate found in the TProlog Reference Manual. */
/* The INTERSECT rule finds items from the first */
/* list that are common to the second list and */
/* places them in a third "common" list. It is */
/* derived from the DIFFERENCE rule found in Ivan */
/* Bratko's Prolog Programming for Artificial Intel.*/
/* The DB_LIST predicate converts the HITS database */
/* records into a list. That seemed easier than */
/* creating a list in the first place. */
/* The BUBBLESORT and SWAP rules are also from */
/* Bratko. (Steal from the best.) */
/*****

```

```

gated_member(X, [Y|_]):-
  gate_width(GateWidth), /* retrieve the gate width */
  X < Y + GateWidth, /* is X within the range of*/
  X > Y - GateWidth, !. /* the dbase value +/- the */
gated_member(X, [_|Tail]):- /* gate? */
  gated_member(X,Tail).

```

```

intersect([],_,[]).
intersect([X|L1],L2,[X|L]):- /* if X is within the range */
  gated_member(X,L2), !, /* of any of the items in the*/
  intersect(L1,L2,L). /* second list, add it to the*/
intersect([_|L1],L2,L):- /* third. If not, go on to */
  intersect(L1,L2,L). /* the next member of the 1st*/

```

difference. This guarantees that the denominator is always positive. The value of 1 is added to the absolute difference of the list sizes to avoid division by zero errors, and to give a value of 1 for lists of equal size.

SEARCH CERTAINTY

The experimental data will seldom, if ever, match a database record exactly. This will be true even when gating is applied. Therefore, the program must assign a score that reflects the amount of matching from each database record. This score is often called a *certainty* or *confidence factor*. The certainty factor, like the sampling score, has nothing to do with probability statistics; rather, it's a general indication of the certainty of a given answer. To calculate the certainty score, take the number of elements in the experimental list that are common to a database record. Then divide by the smaller number of elements of the two lists, and multiply by 100 to get a percent value. This score determines whether a database record is a "hit."

IMPROVEMENTS

Dealing with uncertainty is a common feature of most expert systems. A true expert system, however, should be able to tell the user if additional information is required to make a better identification. A more sophisticated system might keep database records of a companion list of peaks in database records that just barely missed selection with the defined gate. The program could then inform the user of better matches with slightly wider gates. Commercial IR peak-matching programs always tell the user to compare the actual hardcopy spectra before making an identification. This is a sound approach.

The program could also be improved to make better use of

the IR spectra information. Sub-structure information for each compound could be incorporated into the database, since each absorption peak in a spectrum indicates a specific chemical bond

Other analytical techniques could be used to enhance the identification methods, such as ultraviolet-visible spectroscopy, which aids in structure identification.

type and environment. The program could then interpret the peaks and report on the substructures, and could also report the interpretation of the experimental spectrum even if good matches were not found in the database.

Other analytical techniques could be used to enhance the identification methods, such as ultraviolet-visible spectroscopy, which aids in structure identification. Additional techniques that provide very specific information about compounds include mass spectrometry and nuclear magnetic resonance spectroscopy.

Keep in mind, however, that more information necessitates more sophisticated programming. Several new problems will emerge if you try to make a single identification from several analytical methods. These are the dragons of incomplete data and conflicting data. ■

Tom Castle is a chemist in Kalamazoo, Michigan. He writes software reviews and C programming articles for Atari ST magazines.

Listings may be downloaded from CompuServe as IRANAL.ARC.

```
db_list([t(Name,Conf,Samp)|T]):-
  hits(t(Name,Conf,Samp)), /* retrieve the db*/
  retract(hits(t(Name,Conf,Samp))),!, /* info, add to */
  db_list(T). /* list & discard.*/
db_list([]).

bubblesort(Unsort,Sort):-
  swap(Unsort,L),!,
  bubblesort(L,Sort).
bubblesort(Sort,Sort).

swap([t(N,C,S),t(N2,C2,S2)|T],[t(N2,C2,S2),t(N,C,S)|T]):-
  C < C2.
swap([t(N,C,S)|T1],[t(N,C,S)|T2]):-
  swap(T1,T2).
```

LISTING 2: IRPEAKS.DAT

```
compound("acetophenone",[3400,3350,3070,3000,1683,1595,1580,1450,
1356,1265,1175,1075,1020,953,755,687])
compound("ammonium benzoate",[3000,1710,1600,1550,1385,1068,1022,
835,720,708,685,680])
compound("anisole",[3060,3030,3000,2950,2835,1920,1840,1770,1690,
1590,1480,1460,1450,1330,1300,1245,1075,1030,875,775,750,
680])
compound("benzyl alcohol",[3300,2985,2857,1960,1875,1825,1497,1471,
1453,1208,1017,735,697])
compound("cyclohexane",[2910,2840,2550,1443,1251,900,855])
compound("cyclohexanone",[2930,2840,1710,1450,1425,1340,1310,1222,
1120,1053,1018,908,853,748])
compound("1-decene",[3049,1645,1470,1390,986,907,720])
compound("heptanoic acid",[3000,2950,2920,2850,1715,1455,1408,1280,
1230,1200,1100,930])
compound("hexane",[2980,2920,2860,1468,1378,725])
compound("1-hexyne",[3268,2941,2857,2110,1470,1430,1247,1105,667])
compound("isobutyramide",[3350,3170,2960,1640,1468,1425,1290,1140,
650])
compound("leucine-(D,L)",[2965,2910,2840,2500,2140,1610,1580,1505,
1455,1405,1350,1300,1285,1225,1130,848,765,675])
compound("mesitylene",[3003,2940,2874,1760,1720,1610,1475,1390,1038,
837,687])
compound("nitrobenzene",[3100,3080,2860,1610,1605,1520,1478,1345,
1315,1108,1070,1022,935,852,793,702,680])
compound("octylamine",[3365,3290,3200,2910,2850,2817,1620,1458,1370,
1063,790])
compound("2-pentanol",[3333,2907,1460,1361,1142,1101,1058,1030,1000,
948,905,890,828,757])
compound("2-pentanone",[2955,2930,2866,1725,1465,1430,1370,1295,
1273,1240,1172,965,900,727])
compound("phenol",[3333,3045,1925,1840,1770,1700,1580,1495,1468,
1359,1223,1067,1020,998,805,745,685])
compound("phenyl acetate",[3070,3040,1770,1593,1493,1360,1205,1183,
1068,1025,1010,923,892,813,748,695])
compound("2-phenyl propionaldehyde",[3077,3040,2985,2941,2874,2825,
2717,1730,1600,1497,1453,1389,1070,1020,8980,860,749,699])
compound("propionic anhydride",[2990,2950,2880,1825,1758,1465,1420,
1348,1265,1090,1040])
```

FAILING WITH GRACE

Replace recursion with iteration—and save memory.

Edward B. Flowers

Prolog's descriptive nature—and its ability to perform recursion—make it an ideal language for modeling ideas. However, since recursion tends to use a great deal of memory, the programmer must keep memory usage in mind when developing large programs.

Programs that employ a menu-driven front end for obtaining and directing user input are particularly interesting challenges in Prolog. Such front ends use a loop to handle the process of displaying a menu, allowing the user to select an item from the menu, carrying out the action required by the menu choice, and returning the program to the menu so that the user can make other selections. In this article, I'll show how to economize the use of memory by embedding menus within **repeat..fail** and **repeat..condition** loops. I'll also briefly discuss the role of recursion in memory usage. Finally, I'll present iteration as an alternative to recursion, and will cover methods for controlling the side effects caused by backtracking in a **repeat..fail** loop.

RECURSION VERSUS ITERATION

The beauty of recursion is that it allows us to describe an iterative process in just a few statements. Recursion is logically simpler than iteration, because recursion more closely models the way we think. Recursion is used to solve problems that contain another problem of the same kind within the larger problem. For example, an algorithm for taking the average of the numbers one through ten can be stated in the following five steps.

1. Check if ten values have been summed together.
2. If so, divide the sum by ten in order to take the average of those values, and then quit.
3. Otherwise, add the value one to the current number that is being added to the sum.
4. Add that value to the current sum.
5. Start over with step 1.

In Turbo Prolog, this process translates into:

```
average(10,S):-
  Average = S/10,
  write("The average is ",Average),nl.
average(X,Y):-
  N      = X + 1,
  NewSum = Y + N,
  average(N,NewSum).
```

Notice how closely these clauses match the algorithm.

Recursion, then, provides a natural and logical way to describe a problem. Unfortunately, recursion can use considerable amounts of memory because Turbo Prolog creates a record in memory, called a "stack frame." The *stack frame* maintains the value of variables at each recursion, as well as the pointers to which the program returns after the particular recursion. As a recursive call proceeds, the program uses up more and more memory as additional frames are created. If enough calls are made to exhaust available memory, then the program fails.

Turbo Prolog recognizes a special case of recursion, called *tail recursion*, which occurs when the recursive call is the last call in the clause (as shown in the previous code sample). In such cases, Turbo Prolog uses optimization techniques to minimize memory demands. (For more on this topic, refer to "The Tail Recursion Tiger," *TURBO TECHNIX*, January/February, 1988.) Recursive clauses, however, cannot always be made tail recursive.

continued on page 78

Reflex: the database that maximizes your decision power

To get ahead in business, you must make decisions. And you must make them right... the first time. That's why you need the power of Reflex:® The Analyst. It's a no-nonsense flat-file database that stores and organizes your information. Then works like a spreadsheet for sophisticated

what-if analyses. Then shows you your information from every angle and perspective, uncovering all its hidden meanings. Then lets you move full speed ahead!

The view is up to you

Let Reflex break your data down. Turn it around. Show it off. And add it up. All with simple menus and commands that don't turn you inside out.

View your data one record at a time. Or all together in columns and rows. Create five different kinds of graphs. A numeric summary. Or a sophisticated report that makes everything fall into place. Even get a split screen that shows a form and graph at the same time. As you edit the form, your changes are instantly reflected in the graph!

Crosstabs show you the big picture

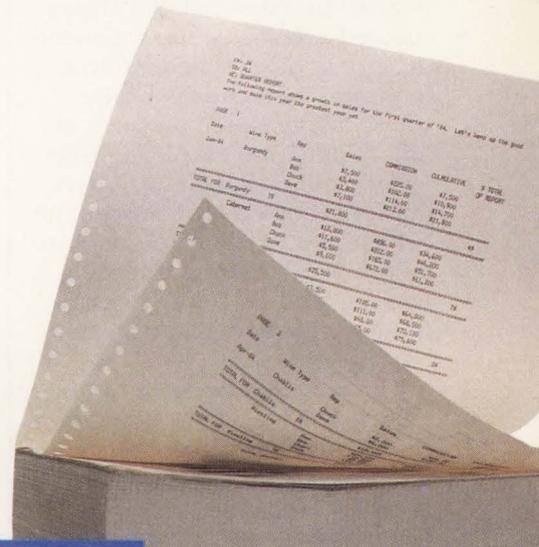
Reflex can give you a unique Crosstab view of your information—a powerful numeric summary divided into categories and displayed in a numeric summary table. Use it to pinpoint trends and relationships among the data and ask what-if questions. Your Crosstab shows Bob's average percent margin is higher than the other sales reps. Is he pushing too hard on price at the expense of volume? Quickly change the Crosstab for a second analysis, and Reflex will give you a whole new perspective.

Even generate reports for 1-2-3 and dBASE files

The Reflex Report view is a powerful report generator that can also accept files from popular applications like Lotus 1-2-3®, dBASE®, and PFS: File. Use Reflex to generate everything from mailing labels to sophisticated custom reports—it's the only report generator you need!

The Workshop's templates make it easier

The Reflex Workshop, available separately, gives you 22 master templates for running almost any kind of business. The formats are all there; you just plug in the numbers.



Views Edit Print/File Records Search Crosstab

CROSSTAB

Summary: @AUG Field: % Margin

Rep

	"Alan"	"Bob"	"Cathy"	"Dave"	ALL
"Paddles"	(12.8)	27.8	9.8	15.7	9.7
"Silent"	47.4	47.3	43.7	36.8	43.6
"Sport"	28.6	31.9	38.9	25.8	29.1
"Swiftwater"	43.8	48.5	43.7	36.7	43.8
ALL	26.6	38.7	31.8	28.4	31.4

Use the Reflex Crosstab view to get the whole picture...

Views Edit Print/File Records Search Graph Type

LIST

Date	Rep	Product	Quantity	Sales \$	Avg Price	Unit Cos
Jan-85	Alan	Paddles	81	\$6,558	\$81	\$77
Jan-85	Alan	Silent	16	\$16,835	\$1,052	\$578
Jan-85	Alan	Sport	18	\$4,976	\$498	\$398
Jan-85	Alan	Swiftwater	9	\$6,672	\$741	\$437
Jan-85	Bob	Paddles	51	\$5,235	\$183	\$77

FORM

Quantity: 81 \$ Margin

Sales \$: \$6,558 Avg \$ Ma

Avg Price: \$81 % Margin

Unit Cost: \$77 Commis

GRAPH

\$ Margin

... or split your Reflex screen to show several views at once—and watch your data change as you edit!

Maximize your decision power!

Give your decisions the power of Reflex. See your Borland dealer today.

“If you need an analytical tool that's powerful, versatile, easy to use, and with the right price, Reflex is for you.

Bob Weeks, Chicago Computer Guide”

60-Day Money-back Guarantee*

For the dealer nearest you
Call (800) 543-7543



REPEAT LOOPS

The alternative to recursion is the use of backtracking in repeat loops to perform the iterative process. To create a repeat loop, set a backtrack point by creating a nondeterministic call that always succeeds. Next, add the statements that are to be executed within the loop, and then end the clause with a failing condition (such as **fail**). When Turbo Prolog encounters the failing condition, it backtracks to the nondeterministic call (which says, "go look for more solutions") and executes the loop once again. In this way, no stack frames are required and memory remains intact.

The **menu** predicate in Listing 1 provides an example of an unconditional repeat loop. These loops are commonly called **repeat..fail** loops, because they begin with the programmer-defined **repeat** predicate and end with the standard predicate **fail**. The following **menu** clause demonstrates the overall structure of a **repeat..fail** loop.

```
menu if
  repeat,
  ...
  fail.
```

repeat is a nondeterministic clause whose sole purpose is to set a backtrack point. (Remember, a backtrack point is established whenever more than one possibility for a solution exists, and Turbo Prolog must decide on one of two search paths.) **repeat** is defined in the context of the program as:

```
repeat.
repeat:- repeat.
```

The first clause establishes the backtrack point. The second **repeat** clause (which is tail recursive) calls the first clause to set another backtrack point whenever a "fail" occurs and the program backtracks across the original call to **repeat**. This process generates an infinite number of possible solutions.

In the **menu** example above, **repeat** is followed by a number of program statements and finally a call to **fail**. **fail** forces Turbo Prolog to backtrack to **repeat**.

Since **repeat** generates an infinite number of solutions, the search continues back down the clause to the **fail** again. This process continues ad infinitum until the <q>uit option is selected.

Figure 1 diagrams how the program in Listing 1 uses the main menu to route processing to groups of related predicates through the **choice** clauses. Memory is restored as control first re-

continued on page 80

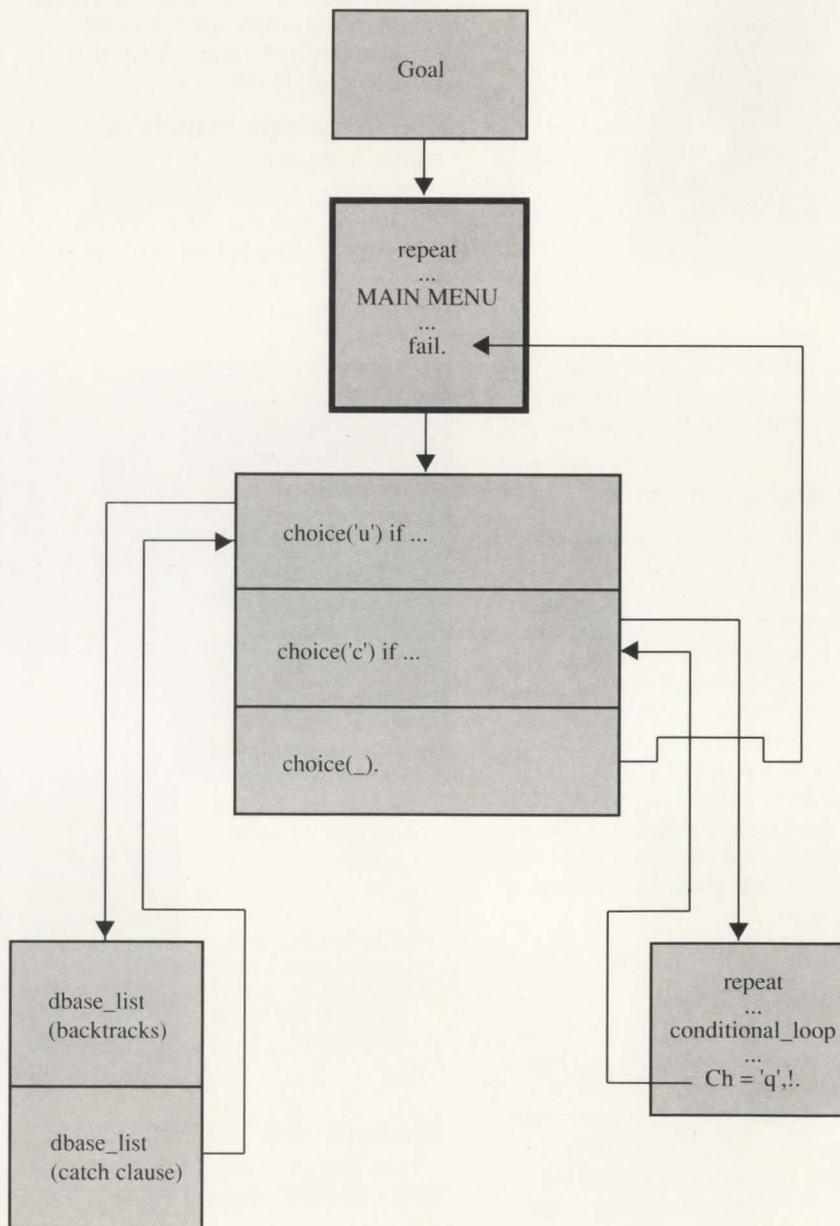
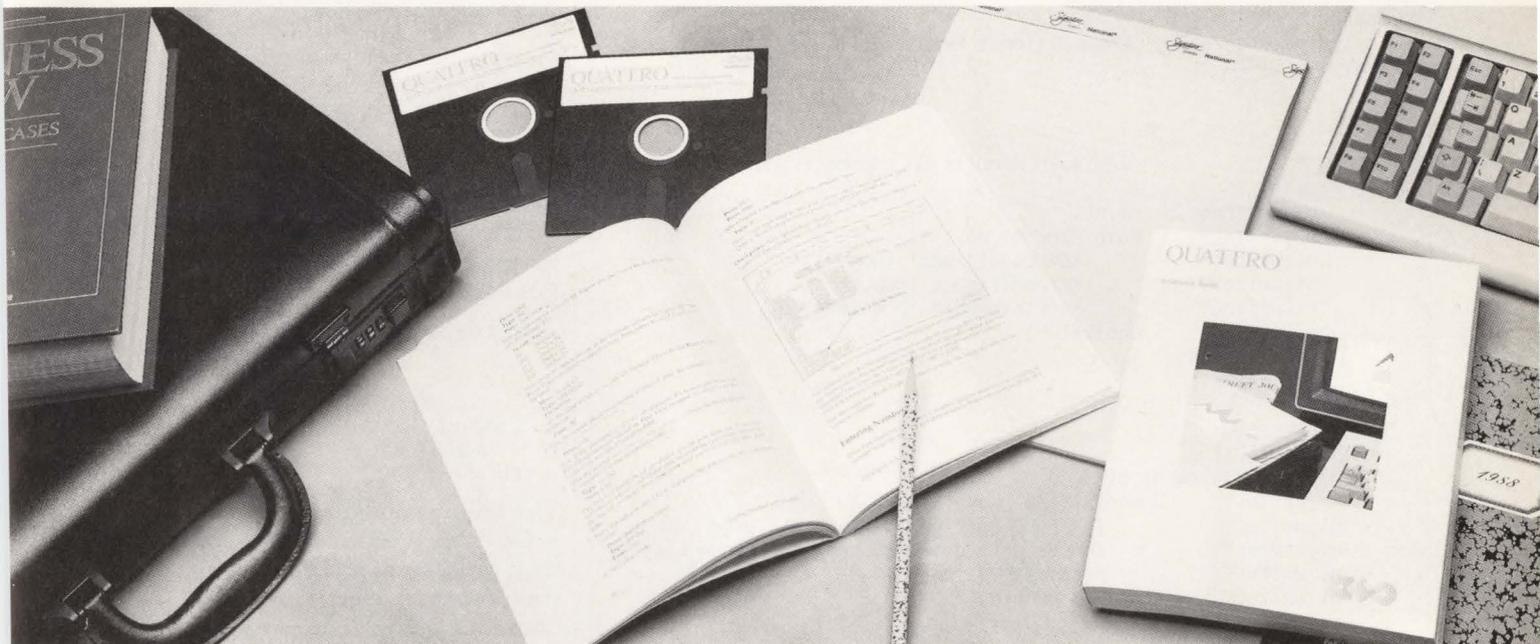


Figure 1. The flow of program control in Listing 1.

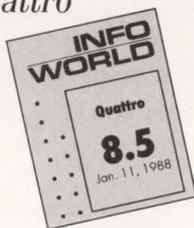
“Those who are considering purchasing 1-2-3 will be better off with Quattro”

—John Walkenbach, InfoWorld



Here's what InfoWorld had to say about Quattro

“There are some clear advantages in choosing Quattro over the 1-2-3 of today: easier installation, no copy protection, improved speed, much better macros, excellent graphics, a customizable command interface, and direct compatibility with industry standard file formats. If cost is a factor, you can get five copies of Quattro for the same money that would buy two 1-2-3 packages.”



Quattro includes SQZ!® Plus data compression

A special implementation of SQZ! Plus, the spreadsheet file compression utility, is built into Quattro and comes to you absolutely free. SQZ! Plus for Quattro automatically compacts and expands Quattro spreadsheets by up to 95% during file saving and retrieving.

Features: Improving the industry standard

“Quattro takes the industry standard and improves upon it in the areas that count most. It addresses many of the weaknesses of 1-2-3 and adds quite a few of its own unique touches.”

“Perhaps Quattro's main advantage over most other spreadsheets is its minimal recalculation capability. When you make a change in your spreadsheet, only affected cells are recalculated, greatly speeding things up in most cases.”

“Other Quattro features that improve upon the 1-2-3 standard include auto-record macros, vastly superior graphics, and easy installation.”

Performance: Markedly superior to 1-2-3

“Our benchmark tests show Quattro markedly superior to 1-2-3 in file saves and retrieves.”

“Quattro's graphics are a sight to behold.”

“Quattro makes working with macros practically painless. If you're into complex 1-2-3 macros, the debugging feature alone is good reason to make the switch to Quattro.”

“No one can argue that Quattro is anything less than an excellent spreadsheet value.”

Excerpts from John Walkenbach's review of Quattro in InfoWorld, January 11, 1988.

60-Day Money-back Guarantee*
Includes 3½" and 5¼" disks.

For the dealer nearest you or a brochure call (800) 543-7543



Reprinted with permission from InfoWorld

*Customer satisfaction is our main concern. If within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. SQZ! is a registered trademark of Symantec Corp./Turner Hall Publishing division. Other brand and product names are trademarks of their respective holders. Copyright © 1988 Borland International, Inc. 81 121248W

```

/* Conditional and unconditional repeat..fail loops */

database
    sum(integer)

predicates
    menu
    repeat
    choice(char)
    dbase_list
    fact(string)
    conditional_loop
    adder(char)

goal
    asserta(sum(0)),
    makewindow(1,26,30," Unconditional Repeat-Fail Loop ",0,0,24,80),
    menu.

clauses

    menu if          /* the main menu is an unconditional loop */
        repeat,
        clearwindow,nl,nl,nl,
        nl,write("    Main Menu:"),nl,
        nl,write("        <u>nconditional loop."),
        nl,write("        <c>onditional loop."),
        nl,write("        <ctrl><break> to quit."),nl,
        nl,write("    Choice: "),
        readchar(Ch),
        choice(Ch),
        fail.

    choice('u') if
        clearwindow,nl,nl,nl,
        nl,write("    Unconditional Loop:"),nl,
        nl,write("        This unconditional \"choice\" predicate"),
        nl,write("        routes data processing to the dbase_list"),
        nl,write("        predicate."),nl,
        nl,write("    <any> to continue: "),
        readchar(_),nl,
        dbase_list.
    choice('c') if
        clearwindow,
        retract(sum(_)),
        asserta(sum(0)),!,
        conditional_loop.
    choice(_).

    dbase_list if          /* no fail is necessary after */
        fact(String),      /*the write statement. The main */
        nl,write("    ",String). /* menu repeat-fail loop causes */
    dbase_list if          /* dbase_list to backtrack */
        nl,                /* through the fact() database */
        nl,write("    <any> to continue: "),
        readchar(_).

    conditional_loop if
        clearwindow,nl,nl,
        nl,write("    Conditional Loop:"),nl,
        nl,write("        This conditional repeat-fail"),
        nl,write("        loop adds one to the Sum on"),
        nl,write("        each increment."),nl,
        nl,write("    Choices:"),nl,
        nl,write("        <any> to continue adding."),
        nl,write("        <q>uit."),nl,
        repeat,

```

continued from page 78

turns to the **choice** predicate and then returns to **fail** in the main **menu**.

CONDITIONAL REPEAT LOOPS

A *conditional repeat loop* might be called a "**repeat..condition loop**" because the conditions of the loop begin with **repeat** and end with a condition that can either succeed or fail. If the condition fails, the loop repeats. On the other hand, meeting the requirements of the condition causes the loop to terminate and processing continues. Therefore, a conditional repeat loop is similar to a **DO..WHILE** loop.

As an example, consider the following **conditional_loop** predicate (also shown in Listing 1):

```
conditional_loop if
```

```
...
```

```
repeat,
    adder(Ch),
    Ch = 'q',!.
```

The failing condition in this case is **Ch = 'q'**. If this condition fails, then processing proceeds again from **repeat** (which is located above the **adder** command). Therefore, **Ch = 'q'**, takes the place of **fail**.

Another characteristic of **repeat** loops is illustrated below by the loop structure in the **economies** predicate of Listing 3:

```
economies if
```

```
...
```

```
readchar(Graphit), clearwindow,

repeat,
    both(StopCont),
    graphit(StopCont,Graphit),
    StopCont = "stop", !.
```

If, for instance, the "<g>raph the data" option is chosen, the variable **Graphit** is instantiated to 'g' and retains that value while the loop continues its search for **StopCont = "stop"**. The reason that **Graphit** is not freed during the backtracking process is that **Graphit** is instantiated outside of the loop. On the other hand, **StopCont** is instantiated within the loop, so this variable is freed each time that Turbo Prolog backtracks across it.

SIDE EFFECTS OF BACKTRACKING

When a clause is executed from within a **repeat** loop, backtracking takes place in all of the downstream predicate links. The problem then becomes how to isolate and control downstream backtracking alternatives. The cut operator, **!**, is used to prevent backtracking into useless alternatives.

For example, the **dbase_list** predicate in Listing 1 does not contain a **fail**. Nonetheless, **dbase_list** calls the **fact** database predicate until all alternatives are exhausted. Turbo Prolog does not stop after the first solution is found because **fail** (in the main menu's repeat loop) "reaches down" to **dbase_list** through **choice('u')** to exhaust all of the **fact** alternatives. This is true even

continued on page 82

```

    adder(Ch),
Ch = 'q',!. /* adding continues until Ch = 'q' is true */
    adder(Ch) if !,
        retract(sum(Sum)),
        nl,write("    Sum = ",Sum),
        Sum2 = Sum + 1,
        asserta(sum(Sum2)),
        readchar(Ch).

repeat.
repeat if repeat.

fact("The Main Menu repeat-fail loop uses").
fact("the dbase_list predicate to retrieve").
fact("all of the fact predicates that contain").
fact("these statements.").
fact("").
fact("Conditional and unconditional").
fact("repeat-fail loops have very").
fact("low memory overhead.").

```

LISTING 2: RMENU.PRO

```

/* Combining repeat loops */

database
    key_number(integer)

predicates
    menu
    repeat
    menu2(char)
    choice(char)
    combinations(integer)
    data_number_one(integer)
    data_number_two(integer)

goal
    asserta(key_number(0)),
    makewindow(1,26,30," Menus and Combinations ", 0,0,24,80),
    menu.

clauses

menu if
    repeat,
        clearwindow,nl,nl,nl,
        nl,write("    Main Menu:"),nl,
        nl,write("        repeat-<f>fail menu."),
        nl,write("        repeat-<c>ondition menu (with !)."),
        nl,write("        simple <p>redicate ! menu."),
        nl,write("        simple p<r>edicate menu (without !)."),
        nl,write("        <ctrl><break> to quit."),nl,
        nl,write("    Choice: "),
        readchar(Ch),
        menu2(Ch),
    fail.

```

```

menu2('f') if
  repeat,
    clearwindow,nl,nl,nl,
    nl,write(" Repeat-Fail Menu:"),nl,
    nl,write(" <c>ombinations."),
    nl,write(" <ctrl><break> to quit."),nl,
    nl,write(" Choice: "),
    readchar(Ch),
    choice(Ch),
  fail. /* the fail in repeat */
menu2('c') if
  repeat,
    clearwindow,nl,nl,nl,
    nl,write(" Repeat-Condition Menu:"),nl,
    nl,write(" <c>ombinations."),
    nl,write(" <q>uit."),nl,
    nl,write(" Choice: "),
    readchar(Ch),
    choice(Ch),
  Ch = 'q', !. /* the condition */
menu2('p') if
  repeat, /* has no effect */
    clearwindow,nl,nl,nl,
    nl,write(" Simple Predicate Menu (with terminating !):"),
    nl,
    nl,write(" <c>ombinations."),
    nl,write(" <program terminates after first success>."),nl,
    nl,write(" Choice: "),
    readchar(Ch),
    choice(Ch),nl,
    nl,write(" <any> to continue: "),
    readchar(_), !.
menu2('r') if
  repeat, /* has no effect */
    clearwindow,nl,nl,nl,
    nl,write(" Simple Predicate Menu (without terminating !):"),
    nl,
    nl,write(" <c>ombinations."),
    nl,write(" <ctrl><break> to quit."),nl,
    nl,write(" Choice: "),
    readchar(Ch),
    choice(Ch).

choice('c') if
  clearwindow,nl,nl,
  nl,write(" Find database combinations with key number: "),
  readint(KeyNumber),nl,
  combinations(KeyNumber).
choice(_) if
  nl,nl,write(" <any> to continue: "),
  readchar(_).

combinations(KeyNumber) if
  data_number_one(One),
  data_number_two(Two),
  nl,write(" ",Keynumber," ",One," ",Two).

data_number_one(1).
data_number_one(2).
data_number_one(3).

data_number_two(1).
data_number_two(2).
data_number_two(3).

```

though neither **choice('u')** nor **dbase_list** contains a **fail**. Backtracking in **dbase_list** is caused by the **repeat..fail** loop in the main menu, and ends only when all of the alternatives in the chain of predicates leading to **dbase_list** have been exhausted. This "remote" cause of backtracking can be confusing in a large program.

This same effect is also evidenced in the **unconditional_loop** clause in the earlier example. The use of the cut (!) after the **Ch = 'q'** condition is essential. Otherwise, the loop continues indefinitely, searching for 'q'. To see the effect of the cut operator on the loop, try removing the cut and running the program.

COMBINATIONS OF REPEAT LOOPS

Listing 2 provides a menu of secondary menus that illustrate some of the quirks of conditional and unconditional repeat loops. Each of the subsidiary menus routes processing to the following **combinations** clause:

```

combinations(KeyNumber) if
  data_number_one(One),
  data_number_two(Two),nl,
  write(" ",KeyNumber," ",One," ",Two).

```

Find database combinations with key number: 1

```

1 1 1
1 1 2
1 1 3
1 2 1
1 2 2
1 2 3
1 3 1
1 3 2
1 3 3

```

<any> to continue:

Figure 2. The output generated by the combinations predicate.

If backtracking occurs, **combinations** writes out the nine combinations of the numbers one through three, since **data_number_one** and **data_number_two** are non-deterministic. In this case, **combinations** generates the output shown in Figure 2. All but one of the example menu combinations cause backtracking in the **combinations** predicate.

The "repeat-<f>ail" option leads into a menu that is similar to the main menu and contains a **repeat..fail** loop. This secondary menu repeats endlessly; pressing the Ctrl-Break sequence breaks the **repeat..fail** loop. This menu's **repeat..fail** loop causes backtracking, and prints the number combinations shown in Figure 2.

The "repeat-<c>ondition" option routes processing to a menu that has a conditional repeat loop. When the condition **Ch = 'q'** is satisfied (by selecting <q>uit), the cut operator tells the loop to exit back to the main menu. This secondary menu's **repeat..condition** loop also causes backtracking, and forces **combinations** to print out the nine number combinations.

The main menu choice "simple <p>redicate ! menu" leads into a menu where **repeat** is followed by neither a terminating **fail** nor a failing condition. This menu operates as a simple predicate, and the cut stops this loop after the loop's first success. The cut prevents backtracking from occurring, so that only the first combination is written.

Choosing "simple <p>redicate menu (without !)" sends processing to a menu where **repeat** is followed by neither a condition nor

continued on page 84

```
repeat.
repeat if repeat.
```

LISTING 3: TMODEL.PRO

```
/* Trade Model */

database
    economy(integer,real,real,real,real,real,real,real)

predicates
    menu
    repeat
    choice(char)
    clearbase
    s_string(integer,integer,string,integer)
    graphit(string,char)
    economies
    us_economy
    japan_economy(string)
    both(string)
    terminate(string,string)

goal
    makewindow(1,26,30," Trade Model ",0,0,24,80),
    menu.

clauses

    menu if
        repeat,
        clearwindow,
        s_string(3,5,"Main Menu:",29),
        nl,write("          <r>un an econometric simulation."),
        nl,write("          <ctrl><break> to quit."),
        nl,nl,write("          Choice: "),
        readchar(Ch), clearwindow,
        choice(Ch),
        fail.

    choice('r') if
        clearbase,
        assertz(economy(0,1,4,3,2,3,2,1)),
        clearwindow,
        economies, !.

    economies if
        s_string(3,5,"New Simulation -----",29),
        nl,write("          <g>raph the data."),
        nl,write("          <ctrl><brk> to quit."),
        nl,write("          <any> to continue: "),
        nl,nl,write("          Choice: "),
        readchar(Graphit), clearwindow,
        repeat,
        both(StopCont),
        graphit(StopCont,Graphit),
        StopCont = "stop", !.

    both(StopCont) if
        us_economy,
        japan_economy(StopCont).
```

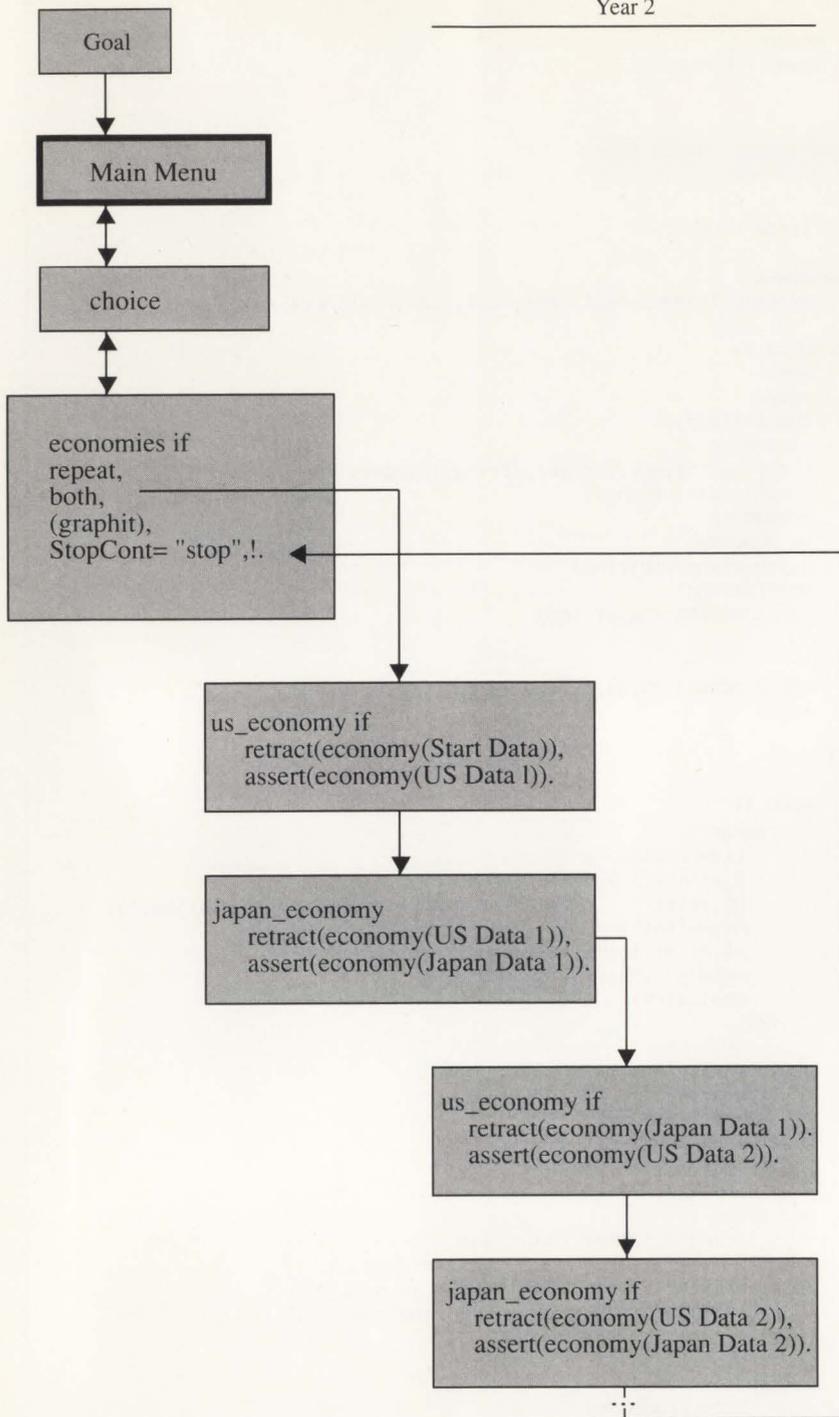


Figure 3. The flow of data in the *Trade Model* program of Listing 3.

FAILING WITH GRACE

continued from page 83

by a terminating cut. This menu operates under the direction of the main menu **repeat..fail** loop, which runs the "simple p<r>edicate" loop repeatedly. In this case, the main menu's **repeat..fail** loop causes backtracking.

PASSING VARIABLES

There are times when processed data must be passed from one iteration of the **repeat..fail** loop to another. In such instances, the **repeat..fail** loop must use **assert** and **retract** to pass variable values from one iteration to the next. Listing 3 is an example in which a large number of econometric variables must be passed back and forth between the **us_economy** and **japan_economy** predicates. This process is necessary because the Gross National Product (GNP) of each economy is dependent upon the variables of the other economy. In the current year, a portion of the U.S. GNP is added into the Japanese GNP. In the next year, a portion of Japanese GNP, **JGnp2**, is added into that year's U.S. GNP through the **LastGnp** variable. Notice that only one **economy** database predicate is necessary to pass all of these variable values from one iteration to the next:

```

us_economy if
    retract(economy(Iter1,LastGnp,
                Uc1,Ui1,Ug1,
                Jc1, Ji1, Jg1)),
    ...
    assertz(economy(Iter2,Ugnp2,
                Uc2,Ui2,Ug2,
                Jc1, Ji1, Jg1)).

japan_economy(StopCont) if
    retract(economy(Iter1,Ugnp2,
                Uc2,Ui2,Ug2,
                Jc1, Ji1, Jg1)),
    ...
    assertz(economy(Iter2,Jgnp2,
                Uc2,Ui2,Ug2,
                Jc2, Ji2, Jg2)).
  
```

In the conditional loop of Listing 1, the internal database must be used to pass the value of the **Sum** variable from one iteration to another so that the **Sum** may accumulate the individual values. Notice that the **sum** database predicate, like **economy**, accumulates data in only *one* database variable. This is done in the database **sum(Sum)** predicate. These database predicates use very little memory, since old values are retracted before new values are asserted.

MEMORY SAVED, MEMORY EARNED

Although recursion has the virtue of stating algorithms elegantly, programs that rely heavily upon recursion often use a great deal of memory. In such applications, replacing recursive loops with **repeat** loops prevents the program from exhausting available memory. Remember to keep the possible side effects of backtracking in mind when converting recursive loops to **repeat** loops. In addition, you'll have to rewrite the loop to pass variables through the internal database, rather than through the parameters of a recursive call. All in all, however, you'll find that the extra effort is worthwhile. ■

REFERENCES

Shafer, Dan. *Turbo Prolog Primer (Revised edition)*, Indianapolis, IN: Howard W. Sams & Company, 1987.

Shafer, Dan. *Advanced Turbo Prolog Programming*, Indianapolis, IN: Howard W. Sams & Company, 1987.

Edward B. Flowers is an associate professor of economics and finance at St. John's University in New York City.

Listings may be downloaded from CompuServe as RPFAIL.ARC.

```

graphit("stop",_) if !.
graphit("cont",'g') if
  clearwindow,
  nl,nl,nl,write("    This routine graphs the data."),
  nl,nl,write("    <any> to continue:"),
  readchar(_).
graphit(_,_).

us_economy if
  retract(economy(Iter1,LastGnp,Uc1,Ui1,Ug1,Jc1, Ji1, Jg1)),
  Iter2 = Iter1 + 1,
  Uc2 = Uc1 + 1, Ui2 = Ui1 + 1, Ug2 = Ug1 + 1,
  Ugnp2 = Uc2 + Ui2 + Ug2 + (0.05*LastGnp),
  nl,writef(" Year: %2",Iter2),
  s_string(3,3,"United States -----",29),
  s_string(5,0,"          Gnp = C + I + G",21),
  writef(" %-6.2f = Ugnp2 = %-6.2f+%-6.2f+%-6.2f",
        Ugnp2,Uc2,Ui2,Ug2),
  assertz(economy(Iter2,Ugnp2,Uc2,Ui2,Ug2,Jc1, Ji1, Jg1)).

japan_economy(StopCont) if
  retract(economy(Iter1,Ugnp2,Uc2,Ui2,Ug2,Jc1, Ji1, Jg1)),
  Iter2 = Iter1,
  Jc2 = Jc1+(0.05*Uc2),
  Ji2 = Ji1+(0.05*Ui2),
  Jg2 = Jg1+(0.05*Ug2),
  Jgnp2 = Jc2 + Ji2 + Jg2 + (0.05*Ugnp2),
  assertz(economy(Iter2,Jgnp2,Uc2,Ui2,Ug2,Jc2, Ji2, Jg2)), !,
  s_string(11,3,"Japa -----",29),
  s_string(13,0,"          Gnp = C + I + G",21),
  writef(" %-6.2f = Jgnp2 = %-6.2f+%-6.2f+%-6.2f",
        Jgnp2,Jc2, Ji2, Jg2),
  nl,nl,write("    <m>ain menu, <any> to continue: "),
  terminate("text",StopCont), !.

terminate("text", "cont") if
  readchar(Ch),
  not(Ch = 'm'),
  cursor(0,0), !.
terminate("text", "stop") if clearwindow, !.

s_string(Row,Col,String,Attr) if
  cursor(Row,Col),
  str_len(String,Len),
  field_attr(Row,Col,Len,Attr),
  field_str(Row,Col,Len,String),
  Row2 = Row + 1,
  cursor(Row2,0).

clearbase if
  retract(economy(_,_,_,_,_,_,_)),
  fail.
clearbase.

repeat.
repeat if repeat.

```

IN GRAPHIC HARMONY

Using the BGI in your graphics routines means hardware compatibility for your program.

Alex Lane



PROGRAMMER

In last issue's Turbo Pascal section, Tom Swan discussed the new Borland Graphics Interface (see "Meet the BGI," *TURBO TECHNIX*, May/June, 1988). But Turbo Pascal users are not the only ones to reap the benefits of the BGI—the BGI is also accessible from Turbo C 1.5, and most recently, from Turbo Prolog 2.0. Although Turbo Prolog 2.0 still supports turtle graphics, the BGI offers a far more comprehensive library of graphics routines.

The BGI system furnishes the programmer with services ranging from high-level routines that create and manage viewports (virtual screens on the display); to routines that draw circles, ellipses, rectangles, and other shapes; to routines that let the programmer define patterns for filling shapes on the screen. Since the various features of the BGI are discussed at great length in "Meet the BGI," I won't repeat that information here. If you would like to know more about the BGI in general, I highly recommend Mr. Swan's article as a starting point.

In this article, we'll look at a number of graphics issues, such as the portability of graphics among computers equipped with different display hardware. We'll also explore programs for drawing and filling odd shapes, drawing lines, and labeling objects. The program in Listing 1 illustrates most of the points under discussion, and specifically addresses CGA, EGA, and VGA display adapters.

THE BGI AND GRAPHICS PORTABILITY

One of the thornier problems in programming is how to write graphics-based software that needs little or no modification in order to work with a variety of video adapters. This task is difficult because different adapters display graphics in several modes, all of which entail keeping tabs on the number of rows and columns on the screen, the number of colors

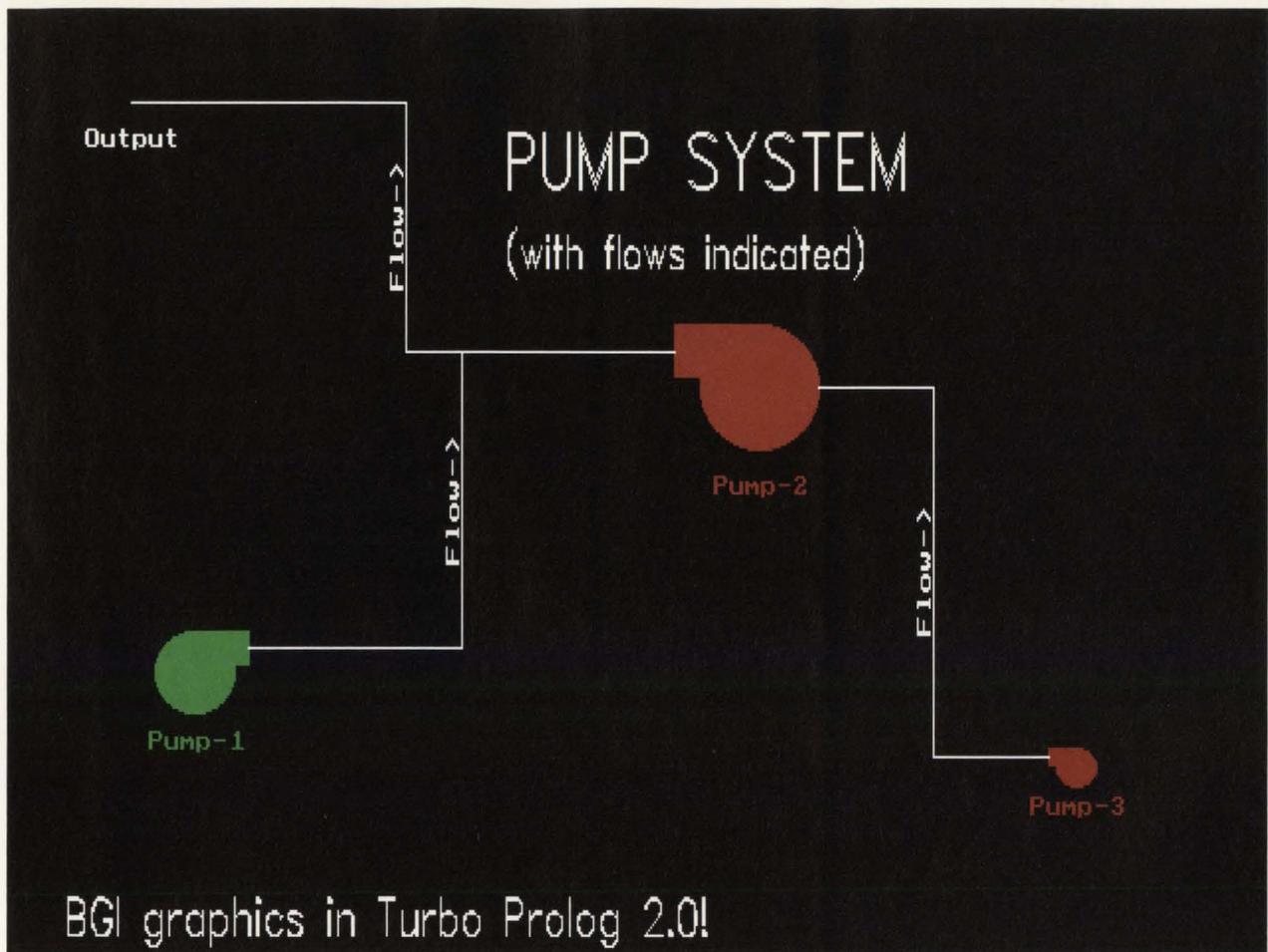
that can be displayed, and the screen's graphics resolution. For example, a low-resolution Color Graphics Adapter (CGA) can display four colors (including background) with a resolution of 320 pixels horizontally and 200 pixels vertically. When switched to a high-resolution mode, the same CGA display can then show 640 pixels horizontally, but can only support two colors (one of which is a background color). A computer with an Enhanced Graphics Adapter (EGA) card, on the other hand, can resolve 640 pixels across and 350 pixels down, while the resolution on Video Graphics Array (VGA) cards can go as high as 640 × 480.

To survive in this morass of adapters and modes, a program must do three things. Using a program that draws a circle in the exact center of the screen as an example, the program must first figure out what type of graphics adapter is installed in the computer. Second, it must initialize the appropriate graphics mode. Third, the program must locate the center of the screen and finally draw the circle. These tasks are relatively easy to perform with the Turbo Prolog BGI predicates.

INITIALIZING THE GRAPHICS MODE

The BGI predicate **detectgraph** checks the video hardware installed in the computer, determines which graphics driver and mode to use, and returns this information as two integer output parameters. To see which other modes are available, use the **getmoderange** predicate, which returns a low number and a high number that represent a range of modes that are available for a given driver. This is useful if you deliberately do not want to use the highest resolution screen mode for a particular adapter.

By the way, the use of integers to represent modes, sizes, fonts, fill patterns, and so forth is a common theme in the BGI system. While this makes execution easy for the computer, it makes things difficult for the programmer in terms of readability. Fortunately, a set of **constant** type declarations (a new feature of Turbo Prolog 2.0) in the file GRAPDECL.PRO



(which comes on the Turbo Prolog 2.0 distribution disk) lets you use symbolic names like **gothic_FONT** and **cga** in predicates, instead of using numbers like 4 and 0.

Once the graphics adapter type is determined, the program in Listing 1 selects a mode by calling the user-written predicate **set_mode**. The point of this predicate is to force a low-resolution mode with color palette 0 when a CGA card is found. In EGA and VGA modes, the highest resolution is set. Once the adapter type and mode are determined, they are asserted into the database via the user-written **graphic** database predicate, since they are accessed whenever a display-sensitive decision needs to be made.

Having established the adapter type and mode, the **initgraph** predicate initializes the graphics system. Initialization consists of loading the appropriate graphics driver from disk (or validating a driver that's already linked into the program), and actually switching the system into graphics mode. Although **initgraph** is normally supplied with a graphics driver name and mode, it can also be invoked to automatically detect the connected graphics hardware, and to select the highest possible resolution for that hardware.

DRAWING SHAPES

The BGI system offers a library of ready-made predicates that draw lines and basic shapes like circles, ellipses, and rectangles. Other predicates draw circular arc segments, pie slices, two- and three-dimensional

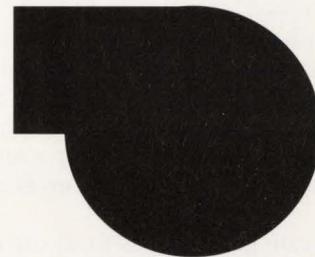


Figure 1. The symbol used to depict a centrifugal pump.

bars, and irregular polygons. With these tools, you can draw just about any shape.

An example from real-world engineering is a symbol that is used in a number of graphics-oriented control systems to depict a centrifugal pump (see Figure 1). The code for drawing this symbol is in the predicate **place_pump**. This predicate allows the programmer to specify the pump name (which is printed below the pump), the pump's position on the screen, the pump's size, the direction of the pump's outlet, and whether the pump is on or off. The position of the pump on the screen is expressed using an arbitrary 0 to 100 scale, both horizontally and vertically. The point (0,0) is in the upper left corner; (100,100) is located in the lower right corner. The point (50,50) is located exactly in the center of the display.

continued on page 88

continued from page 87

The idea behind using these coordinates is that a pump positioned at (50,50) appears in the center of the screen for CGA, EGA, and VGA displays without requiring the programmer to do any scaling. Scaling is performed with the user-written **scale** predicate, which uses the adapter and mode information to determine the resolution and, subsequently, the actual coordinates for positioning. The same fundamental technique is used by **adjust_size** (user-defined) to establish a radius for the arc and thus, in effect, to set the size of the pump.

The outline of the pump is basically a rectangle superimposed on a circle. However, if you draw it like this using the **circle** and **rectangle** predicates, you'll be obliged to fill the pump with the same color that you used to draw the circle and rectangle. Otherwise, the part of the rectangle that lies inside the circle, and the arc that lies inside the rectangle, will remain visible. An alternate way of drawing a pump is to separately draw an arc that represents the pump body, and then draw the three line segments that represent the outlet.

The BGI **arc** predicate draws arc segments in a counterclockwise direction and takes, among other parameters, angles that correspond to the start and end points of the arc. A point with an angle of 0 degrees corresponds to the 3 o'clock position on a clock face, with 90 degrees corresponding to 12 o'clock, 180 degrees to 9 o'clock, and 270 degrees to 6 o'clock.

I determined the angles for the pump symbol in the old-fashioned way, by first sketching the symbol on a piece of graph paper, and then using a protractor to find the angles for the endpoints of the arc. For a pump whose outlet faces to the right, the starting point is at 90 degrees and the arc continues around to about 12 degrees; the arc for a left-facing pump starts at about 168 degrees and ends at 90 degrees.

The BGI predicate **getarccoords** finds the coordinates of the start and end points. The predicate **getendcoords** rearranges these coordinates so that the variables **X0** and **Y0** in the **place_pump** predicate always denote the end under the pump outlet, and **X1** and **Y1** correspond to the top of the pump. Starting at (**X0,Y0**), complete the outlet by drawing a line segment away from the body, another line segment up from the body, and a third line segment in toward the body.

FILLING PATTERNS

The process of coloring the completed pump reveals some adapter-related problems, which are primarily due to the differences between the way that color is controlled in CGA and EGA/VGA hardware. For example, the CGA low-resolution modes allow you to choose from four predefined color palettes (see Table 1). In palette number 0, a pixel with a value of

PALETTE NUMBER	COLOR ASSIGNED TO PIXEL VALUE		
	1	2	3
0	light green	light red	yellow
1	light cyan	light magenta	white
2	green	red	brown
3	cyan	magenta	light gray

Table 1. A list of possible values for the Color Graphics Adapter.

1 appears light green on the display; a pixel value of 2 appears light red; and a pixel value of 3 appears yellow. The same pixel values in palette number 2 display light cyan, light magenta, and white, respectively. Clearly, if you want a symbol to display red when the symbol is on, green when it's off, and yellow when it's malfunctioning, palette 0 (which corresponds to mode 0) should be active.

In the CGA high-resolution mode, life becomes more difficult, since only two colors are available, and one must serve as a background color. One way to distinguish between pump states, in this case, is either to fill in the pump symbol or leave it as an outline figure.

The user-written predicate **adjust_color** uses information about the connected hardware and the current mode to set colors for on and off pump states. For low-resolution CGA, the **on** color is 2 (which corresponds to the light red, light magenta, red, and magenta of palettes 0 through 3, respectively), and the **off** color is 1 (light green, light cyan, green, and cyan, respectively).

With EGA and VGA hardware, 16 colors are available. For these adapters, the **on** color is **red**, and the **off** color is **green**.

Filling the pump symbols with color is the job of **fill_symbol**. This predicate also uses information about the graphics hardware to decide how the pump symbol should be filled. In the EGA, VGA, and low-resolution CGA modes, the BGI predicate **flood-fill** fills in the symbol solidly with the appropriate color. If you want to experiment with the BGI fill patterns, replace the **solid_FILL** argument in the **setfillstyle** call with, for example, **xhatch_FILL** to obtain a heavy cross-hatch fill pattern. The **fill_symbol** predicate also fills in the pump symbol in the high-resolution CGA mode whenever the pump is on, and leaves the pump symbol as an outline figure when the pump is off.

PUTTING WORDS ON THE SCREEN

The last visible step in the process of placing a pump symbol on the screen is to print the pump's name beneath the symbol. Once again, adjustments need to be made on a hardware-dependent basis. The technique for printing the name is simple: Add a number to the pump's **Y** coordinate that is equal to the radius of the pump body multiplied by some factor. In CGA and VGA modes, this factor is 1.5; for

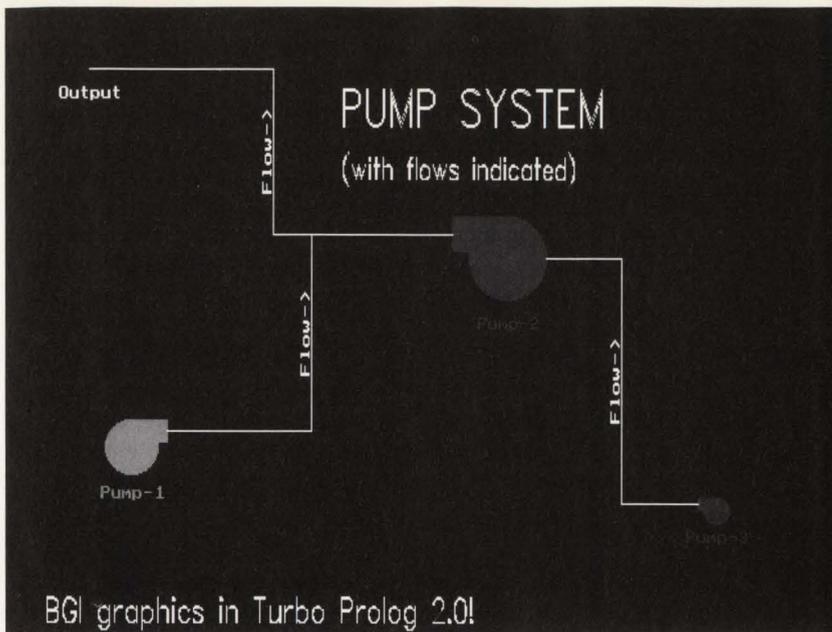


Figure 2. Diagram output by the program in Listing 1.

EGA displays, the factor needs to be smaller (I selected 1.1). The user-written predicate `adjust_label_distance` selects the appropriate factor based on the hardware.

The justification features of the BGI system let you center text with a simple call:

```
settextjustify(center_TEXT, top_TEXT)
```

Here, the first argument tells the BGI, "I want the text centered on the **X** coordinate given in `outtextxy`, and below the **Y** coordinate."

The same basic steps are taken to title the screen using `place_source_or_sink` and `place_title`. `place_source_or_sink` simply prints the name of a flow source or sink on the screen, to be used later when all of the components are connected with line segments to form a network. `place_title` prints a string at the specified coordinates at a specified size. The predicate `adjust_size_for_graphics` handles sizing and font selection for printing titles on the screen. (The sizing of titles is also hardware-dependent; larger sizes become unwieldy on CGA displays.) The auxiliary predicate `write_msg`, used by `place_title`, temporarily changes the title's font, the direction of the title's text display, its horizontal and vertical justification, and its font size. `place_title` restores the original settings for those parameters. Figure 2 shows the completed diagram generated by the program in Listing 1.

CONNECTING THE PARTS

In taking a closer look at Listing 1, notice that I assert instances of the `component_point` predicate in both `place_pump` and `place_source_or_sink`. In the case of `place_pump`, I determine the points on the symbol that represent the inlet (at the 9 o'clock position for a right-facing pump; at 3 o'clock for a left-facing pump) and outlet points. The coordinates of these points are asserted, along with their description and the pump's name. In the case of `place_source_or_sink`, the coordinates, their description, and the name of the source or sink point are asserted in similar fashion.

All of this leads to the `connect` predicate, which is called as:

```
connect("Pump-3", outlet, "Pump-2", inlet)
```

This call can be interpreted as "connect the outlet of device **Pump-3** to the inlet of device **Pump-2**," and results in a line (or collection of line segments) being drawn appropriately. The predicate `draw_lines`, which takes two pairs of coordinates and draws lines between them, uses `write_msg` to place the string "Flow->" vertically along any vertical line segment. With some modification to figure out the appropriate message (such as replacing "Flow->" with "←Flow") and to determine whether or not the message will fit, this labeling of line segments can be made *smarter*. In fact, with some added sophistication, the `connect` predicate in general can be made to act more intelligently and to require less care in the selection of coordinates for displayed symbols.

SUPPORTING THE HARDWARE

The acid test for this program is getting it to perform successfully on a variety of machines. I tested the code on an AT-compatible machine with an installed Video Seven Vega VGA card, and on an IBM XT with a CGA card. To force the card to emulate the CGA and EGA modes on the AT, I replaced `detect_graph(G_driver)` with:

```
G_driver = cga
G_driver = ega
```

This replacement forced the hardware into the respective mode. (I could have also used `cga` and `ega` as input parameters to `initgraph` with the same result, but that step would have required the driver and mode numbers to be asserted *by hand*.) All of the predicates that begin with `adjust_` were spawned by watching how the displays came up under various modes.

The BGI gives the programmer two fundamental ways to package graphics-driven code. The basic way is to compile the Turbo Prolog source to an .EXE file

continued on page 90

```

project "process1.prj"

% bgidriver "_CGA_driver_far"
% bgidriver "_EGAVGA_driver_far"
bgifont "_gothic_font_far"
bgifont "_small_font_far"
bgifont "_sansserif_font_far"
bgifont "_triplefont_far"

code = 4000

include "GRAPDECL.PRO"
DATABASE
component_point(string,symbol, integer, integer)
graphic(integer, integer)

PREDICATES
adjust_color(symbol, integer)
adjust_label_distance(real)
adjust_size(symbol, integer)
adjust_size_for_graphics(integer, integer, integer)
connect(string,symbol, string, symbol)
direction_factor(symbol, integer, integer, integer)
draw_lines(integer, integer, integer, integer)
fill_symbol(integer, integer, symbol, integer)
getendcoords(symbol, integer, integer, integer, integer)
lookup(integer, integer, integer, integer)
max(integer, integer, integer)
place_pump(string, integer, integer, symbol, symbol, symbol)
place_source_or_sink(string, symbol, integer, integer)
place_title(string, integer, integer, integer)
resolution(integer, integer)

scale(integer, integer, integer, integer)
set_mode(integer, integer)
size_factor(symbol, integer)
vertical_msg(integer, integer)
write_msg(string, integer, integer, integer, integer, integer,
           integer, integer)

GOAL
break(on),
/*****
The following line detects the video card in use in your
machine. If you've got a multimode card, you can set the driver
"by hand" by replacing the following line with, for example:
G_driver = vga,
*****/
detectgraph(G_driver, _),

set_mode(G_driver, G_mode),

/*****
A number of decisions must be made based on the driver type,
so we want to post this information for general consumption.
*****/
asserta(graphic(G_driver, G_mode)),
InitGraph(G_Driver, G_Mode, _),
break(on),
settextjustify(center_TEXT, top_TEXT),

/*****
The initialization phase is over.

Now, we proceed to put together a fairly simple display
consisting of an "Output" point, and three pumps.
*****/
place_source_or_sink("Output", outlet, 10, 10),
place_pump("Pump-1", 15, 70, medium, right, off),
place_pump("Pump-2", 60, 40, large, left, on),
place_pump("Pump-3", 85, 80, small, left, on),

/*****
Now set the color to white and connect these components.
*****/

setcolor(white),
connect("Output", outlet, "Pump-2", outlet),
connect("Pump-1", outlet, "Pump-2", outlet),
connect("Pump-3", outlet, "Pump-2", inlet),

/**** As a final touch, title the display *****/

place_title("PUMP SYSTEM", 40, 15, 4),
place_title("(with flows indicated)", 40, 25, 1),
place_title("BGI graphics in Turbo Prolog 2.01", 5, 95, 2),

moverel(0, 100),
readchar(_).

```

continued from page 89

and then distribute that file on a disk with the .BGI driver and the .CHR font files. While this is the simplest way to ensure portability of code across a variety of graphics hardware, it also exposes the largest *surface area* of your application to prospective users who now have to keep track of several files. (You probably have programs that require you to care for a number of auxiliary driver and configuration files. If you're like me, you try to leave them alone and hope nothing happens to them.)

The alternative is to use the **bgidriver** and **bgifont** compiler directives to declare which drivers and fonts are to be part of your program, and to create an appropriate project file to link driver files to the rest of your program. The big plus in using this approach is that the end user only needs to keep track of one file—this minimizes the work he or she has to do in order to use your product. On the down side, incorporating every driver into your program can add almost 30K to the program's size, with the possibility that up to 24K is generated but not used. Personally, unless your application is already groaning in size, I'd link in at least the CGA, EGA, and VGA drivers to cover the majority of prospective users.

There are in-between alternatives, too. For example, if you plan to write an application both for your VGA machine at home and your CGA machine at the office, you may choose to link in only those two drivers, and you'll still have to deal with only one program file. All in all, the ability to incorporate drivers into finished programs is a big plus.

PARTING WORDS

Up until now, programmers have been pretty much at the mercy of language publishers with respect to graphics. Most languages made only half-hearted attempts to support graphics, often requiring some pretty fancy programming in order to draw even simple shapes. The BGI, on the other hand, represents an extensive graphics environment that offers a variety of shape, size, style, color, and positioning features that programmers previously had to write themselves.

On a more basic level, the question of hardware compatibility has also been a problem not easily solved by programming. In the past, if a particular language supported your graphics hardware, you were in luck; if not, well, you either rolled your own code (which required a lot of specialized knowledge) or you bought the required hardware. The BGI furnishes the programmer with ready-written drivers that support a broad range of graphics hardware. Now the programmer is free to tackle the main task at hand—writing programs that get things done. ■

Alex Lane is a knowledge engineer living in Jacksonville, Florida. He is the moderator of the Prolog conference on the Byte Information Exchange (BIX).

Listings may be downloaded from CompuServe as PROBGI.ARC.

CLAUSES

```
/* place_title publishes a Title at a position X_percent over
from the left of the screen and Y_percent down from the top.
Magn denotes the size.
*/
```

```
place_title(Title,X_percent,Y_percent,Magn) :-
  scale(X_percent,Y_percent,X,Y),
  adjust_size_for_graphics(Magn,Size,Font),
  write_msg(Title,X,Y,Font,horiz_DIR,left_TEXT,center_TEXT,Size).
```

```
/* place_source_or_sink locates a point on the screen that serves
as a source or sink for the displayed system, labels it, and
keeps track of its location for future connection.
*/
```

```
place_source_or_sink(Name,Id,X_percent,Y_percent) :-
  scale(X_percent,Y_percent,X,Y),
  asserta(component_point(Name,Id,X,Y)),
  adjust_size(small,Delta),
  Yname = Y + Delta,
  outtextxy(X,Yname,Name),!
```

```
/* place_pump is designed to place a pump figure identified with
string Name at a position that is X_percent across and
Y_percent down the screen. The size is either small,medium or
large, the pump can face left or right, and the pump can be
either on or off. Once the pump has been displayed a global
database is updated reflecting the inlet and outlet points of
the pump for future reference.
*/
```

```
place_pump(Name,X_percent,Y_percent,Size,Direction,Status) :-
  scale(X_percent,Y_percent,X,Y),
  direction_factor(Direction,F,Start,Finish),
  adjust_color(Status,Color),
  adjust_size(Size,Radius),
  DX = -1.4 * Radius * F,
  HD = 0.42 * Radius * F,
  setcolor(Color),
  arc(X,Y,Start,Finish,Radius),
```

```
  getendcoords(Direction,X0,Y0,_,Y1),
  moveto(X0,Y0), /* inside corner */
```

```
  DY = Y1 - Y0,
  linerel(HD,0), /* out */
  linerel(0,DY), /* up */
  linerel(DX,0), /* in */
  fill_symbol(X,Y,Status,Color),
```

```
  Xout = X0 + HD, Yout = Y0 + DY/2,
  Xin = X - (F * Radius),
  asserta(component_point(Name,outlet,Xout,Yout)),
  asserta(component_point(Name,inlet,Xin,Y)),
  adjust_label_distance( LF ),
  Yname = Y + Radius * LF,
  outtextxy(X,Yname,Name).
```

```
adjust_color( Status,Color ) :-
  graphic(cga,cgaHI), Color = white;
  graphic(cga,_), Status = on, Color = 2;
  graphic(cga,_), Status = off, Color = 1;
  graphic(cga,cgac1), Status = on, Color = lightmagenta;
  graphic(cga,cgac2), Status = on, Color = red;
  graphic(cga,cgac3), Status = on, Color = magenta;
  graphic(vga,vgaHI), Status = on, Color = red;
  graphic(vga,vgaHI), Status = off, Color = green;
  graphic(ega,_), Status = on, Color = red;
  graphic(ega,_), Status = off, Color=green.
```

```
adjust_label_distance( 1.1 ) :- graphic( ega,_).
```

```
adjust_label_distance( 1.5 ).
```

```
adjust_size(Size,Radius) :-
  size_factor(Size,Factor),
  resolution(X,_),
  Radius=X/Factor,!
```

```
adjust_size_for_graphics(_,1,default_FONT) :- graphic(cga,_).
adjust_size_for_graphics(X,X,sans_serif_FONT).
```

```
connect(Component1,Point1,Component2,Point2) :-
  component_point(Component1,Point1,X1,Y1),
  component_point(Component2,Point2,X2,Y2),
  draw_lines(X1,Y1,X2,Y2).
```

```
direction_factor( left, -1, 168, 90 ).
direction_factor( right, 1, 90, 12 ).
```

```
draw_lines(X,Y1,X,Y2) :- line(X,Y1,X,Y2),!.
draw_lines(X1,Y,X2,Y) :- line(X1,Y,X2,Y),!.
draw_lines(X1,Y1,X2,Y2) :-
```

```
  MX = (X1 + X2) / 2, MY = (Y1 + Y2) / 2,
  line(X1,Y1,MX,Y1),
  line(MX,Y1,MX,Y2),
  vertical_msg(MX,MY),
  line(MX,Y2,X2,Y2),!.
```

```
fill_symbol(X,Y,Status,Color) :-
  graphic(_,cgaHI),
  Status = off, !;
  graphic(_,cgaHI),
  Status = on,
  setbkcolor(Color),
  floodfill(X,Y,Color),!;
  setfillstyle(solid_FILL,Color),
  floodfill(X,Y,Color),!.
```

```
getendcoords( left, X,Y,A,B) :-
  getarccords( _,_,X,Y,A,B).
getendcoords( right, X,Y,A,B ) :-
  getarccords( _,_,A,B,X,Y).
```

```
lookup( cga, cgaHI, 640, 200 ).
lookup( cga, _, 320, 200 ).
lookup( mcga, mcgaMED, 640, 200 ).
lookup( mcga, mcgaHI, 640, 480 ).
lookup( mcga, _, 320, 200 ).
lookup( ega, egaLO, 640, 200 ).
lookup( ega, egaHI, 640, 350 ).
lookup( ega64, ega64LO, 640, 200 ).
lookup( ega64, ega64HI, 640, 350 ).
lookup( egamono, _, 640, 350 ).
lookup( hercmono, _, 720, 348 ).
lookup( att400, att400HI, 640, 400 ).
lookup( att400, att400MED, 640, 200 ).
lookup( att400, _, 320, 200 ).
lookup( vga, vgaLO, 640, 200 ).
lookup( vga, vgaMED, 640, 350 ).
lookup( vga, vgaHI, 640, 480 ).
lookup( pc3270, _, 720, 350 ).
```

```
max(X,Y,X) :- X = Y, !.
max(X,Y,Y) :- X < Y, !.
```

```
resolution( X,Y ) :-
  graphic(Driver,Mode),
  lookup(Driver,Mode,X,Y),!.
```

```
scale(X_percent,Y_percent,X_absolute,Y_absolute) :-
  resolution(X,Y),
  X_absolute = X_percent/100 * X,
  Y_absolute = Y_percent/100 * Y,!
```

```
/* set_mode( cga, cgaHI) :- !. */ /* this mode isn't too useful */
set_mode( cga, cgaC0) :- !.
set_mode( ega, egaHI) :- !.
set_mode( vga, vgaHI) :- !.
```

```
size_factor(small,64).
size_factor(medium,32).
size_factor(large,21).
```

```
vertical_msg(X,Y) :-
  write_msg("Flow->",X,Y,
  default_FONT,vert_DIR,bottom_TEXT,center_TEXT,1).
```

```
write_msg(Text,X,Y,Dfont,Ddir,DHJ,DVJ,Dsize) :-
  gettextsettings(Font,Direction,Size,HJ,VJ),
  settxtstyle(Dfont,Ddir,Dsize),
  settxtjustify(DHJ,DVJ),
  outtextxy(X,Y,Text),
  settxtstyle(Font,Direction,Size),
  settxtjustify(HJ,VJ).
```

LOGIC AND TURBO PROLOG

Prolog's origins in logic are reflected in the mindset behind the language.

Alex Lane

"Contrariwise," continued Tweedledee, "if it were so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

—Lewis Carroll,

Through the Looking Glass

SQUARE ONE

Most introductory books on Prolog dutifully note that the name Prolog is derived from the phrase "programming in logic," and then they briskly move on to other subjects. Although most of us are familiar with the concept of logic, there are many interpretations.

To some, logic means disciplined, emotionless thinking, as exemplified by Mr. Spock from *Star Trek*. Others think of the electrical AND and OR circuits that are used to build computers. Still others view logic as a branch of philosophy, owing more to Aristotle than to Spock.

So, to clear the air: *Logic*, as discussed in this article, is the branch of mathematics that is concerned with the form of statements, and with the determination of truth via mechanical manipulation of formulas.

TRUTH

Truth is a fundamental idea in logic. Logical statements can have one of two truth values: true or false. One pitfall when dealing with truth as a logical concept is that it doesn't necessarily correspond to our everyday notion of truth as "conforming to fact," or "being forthright and sincere." Don't lose any sleep over this, but don't be surprised if mathematical logic isn't always intuitive, either.

PROPOSITIONS

Propositions are statements that can be evaluated as either true or false. They contain information about something. For example, the statement "Socrates" is not a proposition, because you cannot assign a truth value to it. Neither is the statement "is a man," for the same reason. On the other hand, the statement

"Socrates is a man" is a proposition that can be said to be true or false. The analysis of propositions using symbolic notation is called *propositional calculus*, and forms an important part of Prolog's logical foundation.

LOGICAL OPERATIONS

In arithmetic, individual numbers like 3 and 47 are useful, but you can only do so much with them. By analogy, you can go only so far with individual propositions in logic.

Again, in arithmetic, numbers are used in combination with the fundamental operations of addition, subtraction, multiplication, and division to express more than what the numbers alone can say. Similarly, fundamental logical operations can be used with propositions in order to express additional information.

There are four basic operations in logic: negation, conjunction, disjunction, and implication. (There is also a fifth connective—equivalence—which we will not consider here.) These operators are called "sentential" or "logical connectives." Their names and symbols (both in logic and in Turbo Prolog) are shown in Table 1.

CONNECTIVE	NAME	SYMBOL	TURBO PROLOG NOTATION
Negation	NOT	~	not(...)
Conjunction	AND	&	,
Disjunction	OR	+	;
Implication	IF-THEN	→	:-

Table 1. A list of logical connectives with their associated symbols and notation.

NOT	
A	$\sim A$
False	True
True	False

AND		
A	B	A & B
False	False	False
False	True	False
True	False	False
True	True	True

OR		
A	B	A + B
False	False	False
False	True	True
True	False	True
True	True	True

IF THEN		
A	B	A \rightarrow B
False	False	True
False	True	True
True	False	False
True	True	True

Table 2. Truth tables for the various logical operators.

By using these connectives with propositions, we express statements such as "Today is Monday AND the weather is sunny," and "IF Socrates is a man THEN Socrates is mortal."

If you look closely at the last part of the previous sentence, you'll notice how the "and" that connects the two propositions creates another, more complex proposition.

With the exception of NOT, all of these connectives are *binary*—they only work with two propositions. The NOT connective is *unary*, and is applied to one proposition.

Let's take a quick tour of these connectives. A set of truth tables that illustrate these operations appears in Table 2.

NOT. The effect of NOT is to invert the truth value of whatever it's applied to. Table 2 shows that if A is true, then $\sim A$ is false. Conversely, if A is false, then $\sim A$ is true. Applying the NOT connective twice to a proposition is the same as not applying it at all: $\sim\sim A$ is the same as A.

AND. The AND connective (represented as a comma "," in Turbo Prolog) operates on two propositions at a time, and is true only when both propositions are true (see Table 2). All other combinations of truth values result in a value of false. When evaluating a series of ANDed propositions,

One of the positive aspects of using logic for programming is the extent to which humans already incorporate these basic connectives into everyday thinking.

everything stops as soon as a false proposition is encountered. In the Prolog predicate shown below, **A** must prove true before Prolog will try to prove **B**:

```
Z :- A, B.
```

If **A** fails, no attempt will be made to prove **B**.

OR. The logical OR connective (represented by a semicolon ";" in Turbo Prolog) is false only when both of its associated propositions are also false (see Table 2). Otherwise, the OR of the two propositions is true. (This is different from the case where either one proposition or the other is true; this situation is called an *exclusive OR*, and is not addressed here.)

When ORed propositions are evaluated, the OR succeeds as soon as the first true proposition is found. Consider the following Turbo Prolog statement:

```
Z :- A; B.
```

If **A** is satisfied, no attempt at satisfying **B** is made unless Prolog backtracks to this predicate.

As just described, the AND and OR connectives are intuitive—they act the way that you would expect them to act. One of the positive aspects of using logic for programming is the extent to which humans already incorporate these basic connectives into everyday thinking. Yet a major source of confusion when discussing logic concepts is that some of the ideas are simply not intuitive.

IF-THEN

In logic, IF-THEN statements are expressed in the form $A \rightarrow B$, where the arrow means "implies." The concept of IF-THEN is central to our thinking; we are exposed to it daily. We use it in statements like: "If today is Wednesday, then you're reading this magazine."

Intuitively, the idea behind this statement is that IF the first part of the statement ("today is ...") is true, THEN the second part ("you're reading ...") is also true. But what if today isn't Wednesday? You may or may not be reading this magazine. Is our original IF-THEN statement true or false?

In logic, the statement $A \rightarrow B$ is defined (see Table 2) to be true when B is true (regardless of the value of A) or when A is false (regardless of B). Symbolically, $A \rightarrow B$ is equivalent to the expression $\sim A + B$. So, it turns out that by definition, the proposition is true if you're reading this magazine, no matter what day it is. Similarly, if today is not Wednesday, the statement is true regardless of whether or not you're reading this magazine.

If you're confused by this logic, remember that we're talking about the truth or falsity of the statement $A \rightarrow B$, not of the individual propositions A and B. In Prolog terms, given the statement $A \rightarrow B$ and the fact that A is false, we have:

```
B :- A.
```

continued on page 94

LOGIC

continued from page 93

Remember that in Prolog, the positions of the antecedent (the proposition at the arrow's tail) and the consequent (the proposition that the arrow points to) are reversed. Thus, in Prolog, $A \rightarrow B$ is expressed as:

```
B :- A.
```

If **A** is false, nothing can logically be said about **B** one way or another, and attempts to satisfy **B** in Prolog fail. Failure to satisfy a goal in Prolog is not the same as proving the goal to be false; it only demonstrates an inability to prove that the goal is true. This distinction may sound fine, but it is important. Failure to satisfy goal **B** using **A** does not prevent **B** from being satisfied in some other way. For example, if we know that **B** is true, we have:

```
B :- A.
B.
```

Now the goal **B** succeeds, regardless of the value of **A**.

THE RULE OF MODUS PONENS

The logical definition of IF-THEN leads to a very important result: Given $A \rightarrow B$ and the fact that **A** is true, then **B** is true. This method of mechanically obtaining a result (known as a *rule of inference*) is called *modus ponens* and is central to the way Prolog works. In Prolog, modus ponens is expressed as:

```
B :- A.
A.
```

Given this set of Prolog clauses, it is clear that the goal **B** can be satisfied.

PREDICATE CALCULUS

Ordinary propositional calculus is too limited for substantial use in logic programming. In Prolog, the statement "If Socrates is a man, then Socrates is mortal" is expressed in propositional form as:

```
socrates_is_mortal :-
  socrates_is_a_man.
```

The problem with using the propositional form for this statement is that a separate clause must then be generated for each individual that is considered; there is no way to generalize this relationship. You cannot express the idea "If <someone> is a man, then <someone> is mortal" in propositional calculus. In order to deal with objects in both an individual and a general way, we must use predicate calculus.

Remember that in Prolog, the position of the antecedent (the proposition at the arrow's tail) and the consequent (the proposition that the arrow points to) are reversed.

In both Prolog and logic, statements about objects (both by themselves and in relation to other objects) are called *predicates*. Predicates have a specified number (or arity) of arguments. Continuing with the Socrates example, let's define the predicates **is_mortal** and **is_a_man**. Each of these predicates takes one argument. Using the same notation as in propositional calculus, let's say:

```
is_mortal( socrates ) :-
  is_a_man( socrates ).
```

We can generalize this expression by not specifically identifying the predicate arguments, and by referencing unknowns (variables):

```
is_mortal( X ) :-
  is_a_man( X ).
```

This expression says: "If someone unknown (denoted by **X**) can be said to be **is_a_man**, then that same someone can be said to be **is_mortal**." The *instantiation* (or binding) of **X** to a specific value

is accomplished using a technique called "matching," or more formally, "unification."

Unification. For the purposes of our discussion, Prolog variables are *unifiable* if they can be matched together. Notice how variables are matched when trying to satisfy the **member** predicate:

```
member( X, [X, _] ). /* succeed if
                     X is the head
                     of the list */

/* otherwise... */
member( X, [_, T] ) :-
  member( X, T ). /* succeed if
                  X is a member
                  of the list
                  tail */
```

Given an object and a list of objects, **member** succeeds if the object is in the list; otherwise, **member** fails. Given the following goal, the first clause fails after the variable **X** is instantiated to 'a' and then cannot match the head of the list (the character 'x'):

```
member( 'a', [ 'x', 'a', 'z' ] ).
```

In the second clause, **X** matches 'a', and **T** matches ['a','z']. The following recursive call now succeeds, since the first argument and the head of the second argument match the specification in the clause:

```
member( 'a', [ 'a', 'z' ] ).
```

WRAP-UP

The mechanism used by Prolog is based on the idea of mechanical proof of logical statements. Given this mechanism, the challenge for the Prolog programmer is not to devise control strategies for program operation, but rather to simply formulate a collection of true relationships—and then to let the mechanical features of Prolog deliver an answer. ■

Alex Lane is a knowledge engineer living in Jacksonville, Florida. He is the moderator of the Prolog conference on the Byte Information Exchange (BIX).

THE ADVENTURES OF **TURBOMAN**

NEW!

Turbo Prolog 2.0 is
the Artificial Intelligence
breakthrough you've
been waiting for!



“ If I had to pick one
single recommendation for
people who want to try to
keep up with the computer
revolution, I'd say, 'Get and
learn Turbo Prolog.'

—Jerry Pournelle, *Byte* ”

NEW! Turbo Prolog 2.0

Artificial Intelligence to you

Turbo Prolog® puts the power of advanced Artificial Intelligence into your hands—whether you're a professional programmer or just getting started. You'll spin out high-level, real-world applications faster than you ever thought possible. Because Borland's advanced compiler technology drives you right to the cutting edge of Artificial Intelligence development!

Zoom into the future

With its natural, English-like syntax, state-of-the-art compiler, and integrated environment, Turbo Prolog puts the power of the future at your fingertips. Building advanced applications like expert systems, customized knowledge bases, natural language interfaces, and smart database management systems has never been this fast, or this easy.



With Turbo Prolog, you simply describe your problem, so you never get bogged down in procedural language. Slide through even the most complex applications using only about one tenth the code. Your finished programs are tight, readable, and easy to maintain!

New 2.0: the most powerful Prolog yet

New Turbo Prolog 2.0 takes programming to the limit. The new compiler is optimized to produce tighter and more efficient code than ever before.

The new two-volume documentation includes an in-depth tutorial rich with examples and instructions—to take you all the way from basic programming through advanced techniques.

And your fully-integrated environment is even more convenient—with a full-screen editor you can customize just the way you want it!

```

Files Edit Run Compile Options Setup
Editor Dialog
Trace Line 266 Col 14 C:\PROLOG\PROGRAMS\SHOT.
s_verbp([VERB:TOKL], TOKL1, [COL:COLL], COLL1,
verbp(COL, D_NOUNP)):-
  is_verb(VERB),
  s_nounp(TOKL, TOKL1, COLL, COLL1, NOUNP, D_N
s_verbp([VERB:TOKL], TOKL, [COL:COLL], COLL, ve
  is_verb(VERB).

check(WORD):-is_noun(WORD),?.
check(WORD):-is_det(WORD),?.
check(WORD):-is_rel(WORD),?.
check(WORD):-is_verb(WORD),?.
check(WORD):-write(">> Unknown word: ",WOR
nl, readchar(_).

Sentence Analyzer
Write a sentence: we love prolog

SENTENCE
├── NOUNP
│   ├── NOUN
│   │   └── we
│   └── VERBP
│       ├── VERB
│       │   └── love
│       └── NOUNP
│           └── NOUN
│               └── prolog

```

```

Message Trace
noun CALL: is_noun("we")
verb RETURN: is_noun("we")
rel CALL: is_noun("love")
det

```

F2-Save F3-Load F6-Switch F9-Compile Alt-X-Exit

Turbo Prolog's powerful development environment makes developing high-level applications quicker and easier than you've ever imagined. Consider this execution of a sentence analyzer. The results of the sentence analysis are shown in the output window while the source code in the edit window traces the execution. The trace window shows the predicates being traced.

delivers powerful or real-world applications!



Powerful new tools save time

Turbo Prolog 2.0 gives you new, powerful tools to take your programs anywhere your imagination wants to send them:

- *An external database system* for developing large databases. Supports B+ trees and EMS
- *Support for the Borland Graphics Interface*, the same professional-quality graphics in Turbo Pascal,* Turbo C* and Quattro.*
- *Source code to a fully-featured Prolog interpreter*. Plus step-by-step instructions to adapt it or include it as is in your own applications!
- *Full compatibility with Turbo C* so the two languages can call each other forward and backward freely—and you can program with two of the most powerful languages around!
- *Full window management system*
- *Powerful exception handling* and error trapping features
- *High-resolution video support*
- *And a lot more!*

Just \$149.95

Add the Turbo Prolog Toolbox: six powerful toolboxes in one make building applications even easier!

Here's another breakthrough: the Turbo Prolog Toolbox. You get more than 80 tools and 8,000 lines of source code to help build your own Turbo Prolog applications—including separate toolboxes for building menus, screen and report layouts, business graphics, communications, file transfer capabilities, parser generators, and more. Use the Toolbox code as is or modify it to suit your needs.

Toolbox requires Turbo Prolog 1.1 or later.

Just \$99.95

System Requirements: For the IBM PS/2™ and the IBM® family of personal computers and all 100% compatibles. PC-DOS (MS-DOS) 2.0 or later. 384K RAM.

*Customer satisfaction is our main concern; if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. A Borland Turbo Toolbox® product. Other brand and product names are trademarks or registered trademarks of their respective holders. Copyright ©1988 Borland International, Inc. BI 1263

Upgrade now for just \$64.95!

If you're a registered Turbo Prolog owner, you can upgrade to 2.0 now for the special introductory price of just \$64.95 plus \$5 for shipping and handling.

To charge your upgrade to your credit card, call us toll-free today at (800) 543-7543. Be sure to have your original Turbo Prolog serial number handy.

60-Day Money-back Guarantee*
For the dealer nearest you
Call (800) 543-7543



CAT AND MOUSE IN TURBO PROLOG: PART II

Adding a Mac-like user interface is easy with Turbo Prolog.

Safaa H. Hashim



WIZARD

The Turbo Prolog Toolbox contains a number of utilities for creating various types of menus such as pop-up menus, pulldown menus, scrolling menus, and more. With all of these menus, the user initiates an action by pressing a combination of arrow keys and the Enter key. For example, in a pulldown menu program, the user moves to a particular menu using the arrow keys, and then presses the Enter key to activate the menu. Next, the arrow keys are pressed to highlight a particular option. By pressing the Enter key once again, the option is selected.

While the menu programs greatly simplify the interaction between the program and the user, we can streamline the user's actions even further through the use of a mouse. With the mouse, the user points to the particular menu and selects an option in that menu (this process is known as "point and click").

In Part I of this article (see "Cat and Mouse in Turbo Prolog: Part I," May/June, 1988) we explored the basics of mouse programming with Turbo Prolog. Here in Part II, I'll discuss some of the menu programs in the Turbo Prolog Toolbox and will show you how to modify them to work with the Microsoft Mouse.

A SIMPLE MOUSE-BASED MENU

One of the programs in the Turbo Prolog Toolbox is MENU.PRO. This program implements two predicates, **menu** and **menu_leave**, which allow the programmer to create simple pop-up menus. In the following discussion, I'll show you how to use input from the mouse rather than from the keyboard. The program SMSMENU.PRO (Simple Mouse MENU) in Listing 1 is the modified version of MENU.PRO. Note that SMSMENU.PRO works only with a mouse—keyboard input has been disabled. In comparing SMSMENU.PRO with MENU.PRO, notice that all the predicates that recognize keyboard strokes have been replaced with others that interact with the mouse.

To see how the modified program behaves, run Listing 1. (To run the listings in this article, you must have MSM-DRV.PRO, which is presented in Part I.) Enter the goal:

```
mstest
```

The major predicates in Listing 1 are **smsmenu**, **menuinit**, and **smsmenu1**. Let's consider **smsmenu** first:

```
smsmenu(Button,ROW,COL,WATTR,
        FATTR,LIST,HEADER,CHOICE):-
    msm_init,
    msm_show,
    menuinit(ROW,COL,WATTR,FATTR,
            LIST,HEADER,NOOFROW,
            NOOFCOL),
    repeat,
        msm_stat(Button,R,C),
        MsR=R/8, MsC=C/8,
        smsmenu1(MsR,MsC,NOOFROW,
                NOOFCOL,CHOICE),!,
    removewindow,
    msm_hide.
```

In calling **smsmenu**, you can specify which mouse button will activate the pop-up menu (use **Button = 1** for the left button, and **Button = 2** for the right button).

The first two subgoals in **smsmenu** initialize the mouse and show its cursor. Then **menuinit** (which was taken directly from MENU.PRO) displays the window for the menu and a list of the menu's options. Each option is placed on a separate row.

The next subgoal, **repeat**, marks the beginning of the PSR (Press button, Select option, and Release button) loop. This loop provides the mechanism to pick an option by pressing the designated mouse button and holding it down, moving the cursor to the desired option, and selecting that option by releasing the mouse button.

In order for the program to know where the user stops in the list of menu options, **smsmenu1** is called within the **repeat** loop. **smsmenu1** calculates the cursor's row position within the menu window, as shown in the following clause:

```

smsmenu1(MsR,MsC,Nrows,
         Ncols,CHOICE):-
makewindow(X,_,_,Srow,Scol,_,_),
X=81,
SR1=Srow+1, SC1=Scol+1,
compare_pos(MsR,MsC,SR1,SC1,
            Nrows,Ncols,NewR,_,
            inside),
CHOICE = NewR + 1, !.

```

Much of the work here is handled by **compare_pos**. **compare_pos** takes the row and column of the cursor's current position, along with the position and dimensions of the active window, and checks to see if the cursor is within the active window. **compare_pos** also returns the row and column of the cursor's position within the active window.

DYNAMIC MOUSE MENU PROGRAM

One drawback of SMSMENU.PRO is its static nature. The pop-up menu position is fixed, and is independent of the cursor's position. It would be more useful to display the pop-up menu at the current cursor position, because you can then control where the menu appears.

To display a menu at the current cursor position, consider DMSMENU.PRO (Dynamic Mouse MENU) in Listing 2. The major difference between DMSMENU.PRO and SMSMENU.PRO lies in the main predicate **dmsmenu** and its subgoal **dmsmenu1**. A close look at **dmsmenu** reveals that it contains an additional **repeat** loop:

```

dmsmenu(Button,WATTR,FATTR,LIST,
        HEADER,STARTCHOICE,CHOICE):-
msm_init,
msm_show,
repeat,
msm_stat(Button,R,C),
ROW=R/8, COL=C/8,
menuinit(ROW,COL,WATTR,FATTR,
        LIST,HEADER,NOOFROW,
        NOFCOL),
ST1=STARTCHOICE-1,
max(0,ST1,ST2),
MAX=NOOFROW-1,
min(ST2,MAX,STARTROW),
assert(currentrow(STARTROW)),
reverseattr(WATTR,REV),
field_attr(STARTROW,0,
          NOFCOL,REV),
repeat,
msm_stat(B,R1,C1),
ROWa=R1/8, COLa=C1/8,
dmsmenu1(B,Button,ROWa,
        COLa,NOOFROW,
        NOFCOL,CHOICE),
!,removewindow,
msm_hide.

```

The first **repeat** loop waits for the user to position the cursor and press a button. The second **repeat** loop displays the menu at that position.

ADDING SCROLL BARS

Another useful feature in mouse applications is the scroll bar, which allows text to be scrolled up, down, left, or right. A scroll bar gives the visual effect of scrolling the text in relation to the size of the whole document. From the user's perspective, a scroll bar allows complete control of text movement in a window, without the need to memorize a set of commands.

In order to develop a scroll bar program, we need to build a tool that allows text to be scrolled through windows. This tool, which relies upon the built-in **scroll** predicate, is shown in Listing 3. **scroll** inserts a blank line at the top or bottom of the currently active window. This makes the text in the window appear to scroll down or up, respectively. In addition, **scroll** can be used to scroll text to the right and to the left of the currently active window. For instance, to scroll text up two rows, give the goal:

```
scroll(2,0).
```

To scroll text three rows down and five columns to the left, give the goal:

```
scroll(-3,5).
```

As you can see, **scroll** moves text both horizontally and vertically in the currently active window. Unfortunately, **scroll** doesn't allow the retrieval of text from a memory buffer. Therefore, once a portion of the text is scrolled outside of the boundary of the currently active window, that portion of text is lost.

To handle this problem, we can modify the **scroll** predicate to update the screen by adding the following **scr** clause:

```

scr(ROWS,COLS):-
file_text(STRlist),
makewindow(_____,RN,CN),
retract(pointer(Rpos,Cpos)),
refreshROWS(ROWS,Rpos,Cpos,
            STRlist,RN,CN),
NewRpos=Rpos+ROWS,
refreshCOLS(COLS,NewRpos,Cpos,
            STRlist,RN,CN),
NewCpos=Cpos+COLS,
assert(pointer(NewRpos,NewCpos)).

```

To see how **scr** works, let's test it with a predicate:

```

test_scr(ROW,COL):-
makewindow(1,7,0,"testing scr",
           0,0,10,40),
file_str("document.txt",STR),
window_str(STR),
assertFILEstr("document.txt"),
scr(ROW,COL),
readln(_).

```

Now, give the goal:

```
test_scr(5,10).
```

In the first three subgoals of **test_scr**, Turbo Prolog opens a window called "testing scr," reads the contents of the file DOCUMENT.TXT, and copies those contents to the string variable **STR**. The contents of **STR** then display in the window. The next subgoal, **assert-FILEstr**, converts the contents of DOCUMENT.TXT into a list of strings, with each string representing a line of text. Note that **assert-FILEstr** asserts the list of lines into the knowledge base, using the database fact **file_text**.

The next subgoal in **test_scr** is **scr**, where **file_text** retrieves **STRlist** from the knowledge base. **STRlist** represents a list of all lines in the file that are to be displayed in the currently active window. The scroll bar program (MSBAR.PRO) discussed below uses this list to track which parts of the file are visible or not visible in the current window. One way to do this is to use a database fact to register a pointer, **pointer(ROW,COLUMN)**, into the text file. This pointer relates the upper left corner of the active window to a row and column in the text file. The diagram in Figure 1 may help you to visualize this situation.

In order to scroll up five rows in the window, five rows need to be retrieved from the text file to fill the five rows of space at the bottom of the window. To scroll ten columns to the left of the window, ten columns must be retrieved from the text file to fill the ten columns of space to the right side of the scrolled text. Two predicates perform these "refresh" functions: **refreshROWS**, to refresh the rows, and **refreshCOLS**,

continued on page 100

UNLEASH YOUR 80386!

Your 80386-based PC should run two to three times as fast as your old AT. This speed-up is primarily due to the doubling of the clock speed from 8 to 16 MHz. The new MicroWay products discussed below take advantage of the real power of your 80386, which is actually 4 to 16 times that of the old AT! These new products take advantage of the 32 bit registers and data bus of the 80386 and the Weitek 1167 numeric coprocessor chip set. They include a family of MicroWay

80386 compilers that run in protected mode and numeric coprocessor cards that utilize the Weitek technology.

The benefits of our new technologies include:

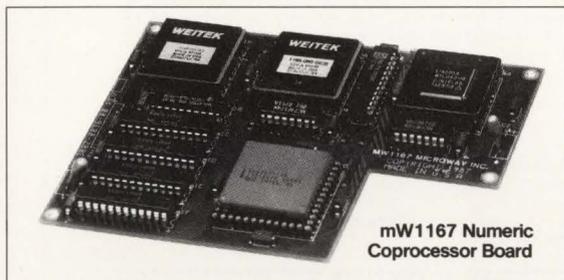
- An increase in addressable memory from 640K to 4 gigabytes using MS-DOS or Unix.
- A 12 fold increase in the speed of 32 bit integer arithmetic.
- A 4 to 16 fold increase in floating point

speed over the 80387/80287 numeric coprocessors.

Equally important, whichever MicroWay product you choose, you can be assured of the same excellent pre- and post-sales support that has made MicroWay the world leader in PC numerics and high performance PC upgrades. For more information, please call the Technical Support Department at

617-746-7341

After July 1988 call 508-746-7341



MicroWay® 80386 Support

MicroWay 80386 Compilers

NDP Fortran-386 and **NDP C-386** are globally optimizing 80386 native code compilers that support a number of Numeric Data Processors, including the 80287, 80387 and mW1167. They generate mainframe quality optimized code and are syntactically and operationally compatible to the Berkeley 4.2 Unix f77 and PCC compilers. MS-DOS specific extensions have been added where necessary to make it easy to port programs written with Microsoft C or Fortran and R/M Fortran.

The compilers are presently available in two formats: Microport Unix 5.3 or MS-DOS as extended by the Phar Lap Tools. MicroWay will port them to other 80386 operating systems such as OS/2 as the need arises and as 80386 versions become available.

The key to addressing more than 640 kbytes is the use of 32-bit integers to address arrays. NDP Fortran-386 generates 32-bit code which executes 3 to 8 times faster than the current generation of 16-bit compilers. There are three elements each of which contributes a factor of 2 to this speed increase: very efficient use of 80386 registers to store 32-bit entities, the use of inline 32-bit arithmetic instead of library calls, and a doubling in the effective utilization of the system data bus.

An example of the benefit of excellent code is a 32-bit matrix multiply. In this benchmark an NDP Fortran-386 program is run against the same program compiled with a 16-bit Fortran. Both programs were run on the same 80386 system. However, the 32-bit code ran 7.5 times faster than the 16-bit code, and 58.5 times faster than the 16-bit code executing on an IBM PC.

NDP FORTRAN-386™\$595
NDP C-386™\$595

MicroWay Numerics

The **mW1167™** is a MicroWay designed high speed numeric coprocessor that works with the 80386. It plugs into a 121 pin "Weitek" socket that is actually a super set of the 80387. This socket is available on a number of motherboards and accelerators including the AT&T 6386, Tandy 4000, Compaq 386/20, Hewlett Packard RS/20 and MicroWay Number Smasher 386. It combines the 64-bit Weitek 1163/64 floating point multiplier/adder with a Weitek/Intel designed "glue chip". The mW1167™ runs at 3.6 MegaWhetstones (compiled with NDP Fortran-386) which is a factor of 16 faster than an AT and 2 to 4 times faster than an 80387.

mW1167 16 MHz \$995
mW1167 20 MHz\$1595

Monoputer™ - The INMOS T800-20 Transputer is a 32-bit computer on a chip that features a built-in floating point coprocessor. The T800 can be used to build arbitrarily large parallel processing machines. The Monoputer comes with either the 20 MHz T800 or the T414 (a T800 without the NDP) and includes 2 megabytes of processor memory. Transputer language support from MicroWay includes Occam, C, Fortran, Pascal and Prolog.

Monoputer T414-20 with 2 meg¹ ... \$995
Monoputer T800-20 with 2 meg¹ ...\$1495

Quadputer™ can be purchased with 2, 3 or 4 transputers each of which has 1 or 4 megabytes of memory. Quadputers can be cabled together to build arbitrarily fast parallel processing systems that are as fast or faster than today's mainframes. A single T800 is as fast as an 80386/mW1167 combination!

Biputer™ T800/T414 with 2 meg¹ ... \$3495
Quadputer 4 T414-20 with 4 meg¹ ...\$6000

¹Includes Occam

80386 Multi-User Solutions

AT8™ - This intelligent serial controller series is designed to handle 4 to 16 users in a Xenix or Unix environment with as little as 3% degradation in speed. It has been tested and approved by Compaq, Intel, NCR, Zenith, and the Department of Defense for use in high performance 80286 and 80386 Xenix or Unix based multi-user systems.

AT4 - 4 users\$795
AT8 - 8 users\$995
AT16 - 16 users\$1295

Phar Lap™ created the first tools that make it possible to develop 80386 applications which run under MS-DOS yet take advantage of the full power of the 80386. These include an 80386 monitor/loader that runs the 80386 in protected linear address mode, an assembler, linker and debugger. These tools are required for the MS-DOS version of the MicroWay NDP Compilers. **Phar Lap Tools**\$495

PC/AT ACCELERATORS

287Turbo-10 10 MHz\$450
287Turbo-12 12 MHz\$550
287TurboPlus-12 12 MHz\$629
FASTCACHE-286 9 MHz\$299
FASTCACHE-286 12 MHz\$399
SUPERCACHE-286\$499

MATH COPROCESSORS

80387-20 20 MHz\$725
80387-16 16 MHz\$475
80287-10 10 MHz\$295
80287-8 8 MHz\$239
80287-6 6 MHz\$155
8087-2 8 MHz\$154
8087 5 MHz\$99

MicroWay

The World Leader in PC Numerics

P.O. Box 79, Kingston, Mass. 02364 USA (617) 746-7341
32 High St., Kingston-Upon-Thames, U.K., 01-541-5466
St. Leonards, NSW, Australia 02-439-8400

LISTING 1: SMSMENU.PRO

```

/* ***** */
/* smsmenu - simple mouse menu */
/* ***** */
smsmenu
Implements a pop-up menu with at most 23 possible choices.
For more than 23 possible choices use longmenu.

FLOW PATTERN: (i,i,i,i,i,i,o)

The arguments to menu are:

smsmenu(BUTTON,ROW,COL,WINDOWATTR,FRAMEATTR,STRINGLIST,
        HEADER,SELECTION)

        BUTTON is the mouse button used for making
        selections.
        (i.e. 1:left Button, 2:right, 3:middle)

        ROW and COL determine the position of the window

        WATTR and FATTR determine the attributes for the
        window and its frame - if FATTR is zero there will
        be no frame around the window.

        STRINGLIST is the list of menu items
        HEADER is the text to appear at the top of
        the menu window

Example: smsmenu(1,10,10,6,4,["Option A",
        "Option B",
        "Option C"],"test",X)

***** */
include "msdoms.pro" /* domains declarations */
include "msut.pro" /* utility predicates */
include "msm-drv.pro" /* the mouse-bios calls */

PREDICATES
compare_pos(INTEGER,INTEGER,INTEGER,INTEGER,INTEGER,
            INTEGER,INTEGER,INTEGER,SYMBOL)
within_boundary(INTEGER,INTEGER,INTEGER,INTEGER,
                INTEGER,INTEGER,INTEGER,SYMBOL)

CLAUSES
/* ***** compare_pos ***** */
/* "compare_pos" predicate takes the mouse position
(MsR,MsC), and the currently active window positions and
dimensions (SR,SC,NR,NC) and returns the position of the
mouse in the currently active window. */

compare_pos(MsR,MsC,SR,SC,NR,NC,NewMsR,NewMsC,POS):-
    MsR >= SR, MsC >= SC,
    within_boundary(MsR,MsC,SR,SC,NR,
                    NC,NewMsR,NewMsC,POS), !.
compare_pos(R,C,_,_,_,R,C,outside):-!.

within_boundary(MsR,MsC,SR,SC,NR,
                NC,NewMsR,NewMsC,inside):-
    NewMsR=MsR-SR, NewMsC=MsC-SC,
    NewMsR < NR, NewMsC < NC.
within_boundary(R,C,_,_,_,R,C,outside):-!.

/* ***** */
/* ***** smsmenu ***** */
/* ***** */
/* DATABASE
currentrow(INTEGER) */

PREDICATES
smsmenu(INTEGER,INTEGER,INTEGER,INTEGER,
        INTEGER,STRINGLIST,STRING,INTEGER)
menuinit(INTEGER,INTEGER,INTEGER,INTEGER,STRINGLIST,
         STRING,INTEGER,INTEGER)
smsmenu1(INTEGER,INTEGER,INTEGER,INTEGER,INTEGER)

CLAUSES
/* ***** smsmenu ***** */
smsmenu(Button,ROW,COL,WATTR,FATTR,LIST,HEADER,CHOICE):-
    msm_init,
    msm_show,
    menuinit(ROW,COL,WATTR,FATTR,LIST,
            HEADER,NOOFROW,NOOFCOL),
    repeat,
    msm_stat(Button,R,C),
    MsR=R/B, MsC=C/B,
    smsmenu1(MsR,MsC,NOOFROW,NOOFCOL,CHOICE), !,
    removewindow,
    msm_hide.

/* ***** ms_menu1 ***** */
smsmenu1(MsR,MsC,Nrows,Ncols,CHOICE):-
    makewindow(X,_,_,Srow,Scol,_,_),
    X=81,
    SR1=Srow+1, SC1=Scol+1,
    compare_pos(MsR,MsC,SR1,SC1,
                Nrows,Ncols,NewR,_,inside),
    CHOICE = NewR + 1, !.

```

```

/* Mouse button pressed outside menu window... quit and
do nothing */
smsmenu1(MsR,MsC,Nrows,Ncols,0):-
    makewindow(X,_,_,Srow,Scol,_,_),
    X=81,
    SR1=Srow+1, SC1=Scol+1,
    compare_pos(MsR,MsC,SR1,SC1,Nrows,Ncols,_,_,outside),
    !.

/* ***** menuinit ***** */
menuinit(ROW,COL,WATTR,FATTR,LIST,
        HEADER,NOOFROW,NOOFCOL):-
    maxlen(LIST,0,MAXNOOFCOL),
    str_len(HEADER,HEADLEN),
    HEADL1=HEADLEN+4,
    max(HEADL1,MAXNOOFCOL,NOOFCOL),
    listlen(LIST,LEN), LEN=0,
    NOOFROW=LEN,
    adjframe(FATTR,NOOFROW,NOOFCOL,HH1,HH2),
    adjustwindow(ROW,COL,HH1,HH2,AROW,ACOL),
    makewindow(81,WATTR,FATTR,HEADER,AROW,ACOL,HH1,HH2),
    writelist(0,NOOFCOL,LIST).

/* ***** */
/* ***** TESTING ***** */
/* ***** */

PREDICATES
mstest

CLAUSES
mstest:-
    /* Button=1, left button */
    smsmenu(1,10,10,6,4,["Option A",
        "Option B",
        "Option C"],"test",X),
    write(X).

/* ***** END OF SMSMENU.PRO ***** */

```

LISTING 2: DMSMENU.PRO

```

/* ***** */
/* ***** dmsmenu ***** */
/* ***** */
dmsmenu generates a disappearing menu at any position
on the screen that is pointed at by the mouse cursor.

dmsmenu(BUTTON,WATTR,FATTR,LIST,HEADER,STARTCHOICE,CHOICE)

BUTTON: This is the mouse button (1,2,3, ...,7) that will
activate the menu. If you press the particular
mouse button Turbo Prolog will display the menu at
the mouse-cursor position.

WATTR, FATTR: These are the "window" and "frame" attributes.

LIST: is the list of options in the menu, it is a list of
strings.

HEADER: Title of the menu.

STARTCHOICE: is the menu row to be first highlighted when
the menu is activated.

CHOICE: is an integer referring to the option we want to
select in the menu.

To run the program, give the goal: popup.

The mouse button is set to 2 which is the right button.
When you press the button you should get the test menu.
*/

include "msdoms.pro" /* domains declarations */
include "msut.pro" /* utility predicates */
include "msm-drv.pro" /* the mouse-bios calls */

PREDICATES
compare_pos(INTEGER,INTEGER,INTEGER,INTEGER,INTEGER,
            INTEGER,INTEGER,INTEGER,SYMBOL)
within_boundary(INTEGER,INTEGER,INTEGER,INTEGER,
                INTEGER,INTEGER,INTEGER,SYMBOL)

CLAUSES
/* ***** compare_pos ***** */
compare_pos takes the mouse position (MsR,MsC),
and the currently active window positions and dimensions
(SR,SC,NR,NC) and returns the position of the mouse in the
currently active window. */

```

listing continued on page 104

PROMOUSE

continued from page 100

cussed in Part I of this article, so I won't cover them in detail here.)

file_text then retrieves the list of lines (asserted by **assertFILE-str**), and **list_len** returns the number of lines (**L**) in the text file.

Another **makewindow** subgoal retrieves the number of rows in the actual scrolling window. After adding horizontal and vertical scroll bars, **addVbar** (in **test-BOTH**) creates a window for scrolling text; this window is located within the original window.

scrollUNIT returns a scroll unit **R**, which is the ratio between the total number of lines in the text file and the total number of rows in the currently active window. In other words, to scroll text vertically, every vertical-bar row corresponds to a value **R** in the text file. On the other hand, to scroll text horizontally, every horizontal-bar column corresponds to one column in the text file.

Finally, we come to the **repeat** loop. **msm_stat** gives the button that is currently being pressed, along with the row and column of the cursor. Then **ms_act** checks whether the pressed button is the button that activates the scroll bars program. Next, **ms_act** checks whether the cursor is in the vertical bar window or the horizontal bar window. Once **ms_act** knows which bar the user is using, it relates any movement of the bar indicator (mouse cursor movements) to the modified scroll predicate **scr**.

APPLICATIONS FOR THE MOUSE

With the basic mechanisms in place, there are a number of mouse features that you can add to your applications. For instance, the mouse can be used with Turbo Prolog windows to dynamically move windows on the screen. This allows the user to point to a window and use the mouse to drag it to a new position. The mouse could also be used to resize a window, or to shift between windows on the screen. Another useful feature you might add is the ability to use the mouse to move or copy strings of text.

In graphics mode, you can add the capabilities to draw points, lines, and polygons. In addition, the mouse can be used to indicate the direction for rotation, reflection, perspective, and other kinds of graphical transformations. Once you start programming the mouse with Turbo Prolog, I'm sure you'll think of many other creative ways to use the mouse. ■

Safaa H. Hashim is a graduate student in the computer science division at the University of California, Berkeley.

Files may be downloaded from CompuServe as MOUSE2.ARC.

REFERENCES

Carrol, John M. (ed.). *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, Boston, Massachusetts: MIT Press, 1987.

Solution Systems. "Experts' Views on the Human Interface Traits of Successful Commercial Software." The Developer's Publisher, 1987.

(For a copy of this report, write to Solution Systems, 541 Main Street, Suite 410, South Weymouth, MA 02198.)

Heckel, Paul. *The Elements of Friendly Software Design*, New York, New York: Warner Books, Inc., 1984.

King, Richard Allen. *The MS-DOS Handbook*, Berkeley, California: SYBEX Inc., 1986.

Nath, Sanjiva. *Turbo Prolog: Features For Programmers*, Portland, Oregon: MIS Press, Inc., 1986.

Nickerson, Raymond S. *Using Computers: Human Factors in Information Systems*, Boston, Massachusetts: MIT Press, 1987.

Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Reading, Massachusetts: Addison-Wesley Publishing Company, 1987.

4.0
TURBO

A Debugger and Overlays for Your Turbo Pascal 4.0

T-DebugPLUS 4.0 is the powerful symbolic run-time debugger that helps you find and fix bugs faster than ever before. You get ■ Access to global and local variables ■ Watch variables ■ Conditional breakpoints ■ Support for graphics modes and dual displays ■ Source and assembly language modes and much more. **T-DebugPLUS 4.0 is only \$45.**

Overlay Manager 4.0 brings overlays to Turbo Pascal 4.0. Develop programs larger than available memory, even larger than the 640K limit. Convert to the fantastic new Turbo 4.0 while retaining the overlays your program needs. Automatic EMS support. Source code included, no royalties. **Overlay Manager 4.0 is only \$45.**

Call toll-free for credit card orders:

1-800-538-8157 ext. 830 (1-800-672-3470 ext 830 in CA)

Satisfaction guaranteed or your money back within 30 days.

For upgrade or other information, call 408-438-8608.



Shipping and taxes prepaid in U.S. and Canada. Elsewhere add \$12 per item.

TurboPower Software P.O. Box 66747 Scotts Valley, CA 95066-0747

```

compare_pos(MsR,MsC,SR,SC,NR,NC,NewMsR,NewMsC,POS):-
  MsR >= SR, MsC >= SC,
  within_boundary(MsR,MsC,SR,SC,NR,NC,NewMsR,NewMsC,POS),
  !.
compare_pos(R,C,_,_,_,_R,C,outside):-!.

within_boundary(MsR,MsC,SR,SC,NR,NC,NewMsR,NewMsC,inside):-
  NewMsR=MsR-SR, NewMsC=MsC-SC,
  NewMsR < NR, NewMsC < NC.
within_boundary(R,C,_,_,_R,C,outside):-!.

/* ***** */
/* ***** dmsmenu ***** */
/* ***** */
/* DATABASE
currentrow(INTEGER) */

PREDICATES
dmsmenu(INTEGER,INTEGER,INTEGER,STRINGLIST,
  STRING,INTEGER,INTEGER)
menuinit(INTEGER,INTEGER,INTEGER,
  INTEGER,STRINGLIST,STRING,INTEGER,INTEGER)
dmsmenu1(INTEGER,INTEGER,INTEGER,INTEGER,
  INTEGER,INTEGER,INTEGER)

CLAUSES

/* ***** ms_menu ***** */
dmsmenu(Button,WATTR,FATTR,LIST,HEADER,STARTCHOICE,CHOICE):-
  msm_init,
  msm_show,
  repeat,
  msm_stat(Button,R,C),
  ROW=R/B, COL=C/B,
  menuinit(ROW,COL,WATTR,FATTR,LIST,
  HEADER,NOOFROW,NOOFCOL),
  ST1=STARTCHOICE-1,
  max(0,ST1,ST2),
  MAX=NOOFROW-1,
  min(ST2,MAX,STARTROW),
  assert(currentrow(STARTROW)),
  reverseattr(WATTR,REV),
  field_attr(STARTROW,0,NOOFCOL,REV),
  repeat,
  msm_stat(B,R1,C1),
  ROWa=R1/B, COLa=C1/B,
  dmsmenu1(B,Button,ROWa,COLa,
  NOOFROW,NOOFCOL,CHOICE), !,
  removewindow,
  msm_hide.

/* ***** ms_menu1 ***** */
dmsmenu1(B,B,MsR,MsC,Nrows,Ncols,_):-
  makewindow(X,WATTR,_,_Srow,Scol,_,_),
  X=81,
  SR1=Srow+1, SC1=Scol+1,
  compare_pos(MsR,MsC,SR1,SC1,Nrows,Ncols,NewRow,_ ,inside),
  currentrow(StartRow),
  StartRow <> NewRow,
  field_attr(StartRow,0,Ncols,WATTR),
  retract(currentrow(StartRow)),
  assert(currentrow(NewRow)),
  reverseattr(WATTR,REV),
  field_attr(NewRow,0,Ncols,REV), !,
  fail.
dmsmenu1(B,B1,MsR,MsC,Nrows,Ncols,CHOICE):-
  B <> B1,
  makewindow(X,_,_Srow,Scol,_,_),
  X=81,
  SR1=Srow+1, SC1=Scol+1,
  compare_pos(MsR,MsC,SR1,SC1,Nrows,Ncols,NewR,_ ,inside),
  CHOICE=NewR+1, !.
dmsmenu1(B,B1,MsR,MsC,Nrows,Ncols,0):-
  B <> B1,
  makewindow(X,_,_Srow,Scol,_,_),
  X=81,
  SR1=Srow+1, SC1=Scol+1,
  compare_pos(MsR,MsC,SR1,SC1,Nrows,Ncols,_ ,outside), !.

/* ***** menuinit ***** */
menuinit(ROW,COL,WATTR,FATTR,LIST,HEADER,NOOFROW,NOOFCOL):-
  maxlen(LIST,0,MAXNOOFCOL),
  str_len(HEADER,HEADLEN),
  HEADL1=HEADLEN+4,
  max(HEADL1,MAXNOOFCOL,NOOFCOL),
  listlen(LIST,LEN), LEN>0,
  NOOFROW=LEN,
  adjframe(FATTR,NOOFROW,NOOFCOL,HH1,HH2),
  adjustwindow(ROW,COL,HH1,HH2,AROW,ACOL),
  makewindow(81,WATTR,FATTR,HEADER,AROW,ACOL,HH1,HH2),
  writelist(0,NOOFCOL,LIST).

```

```

/* ***** */
/* ***** TESTING ***** */
/* ***** */

PREDICATES
popup

GOAL
  popup.

CLAUSES
  popup :-
    msm_init,
    msm_show,
    makewindow(1,7,0,0,0,25,80),
    /* BUTTON = 2, the right button is used */
    dmsmenu(1,71,124,["option 0 ",
    "option 1 ",
    "option 2 ",
    "option 3 ",
    "option 4 ",
    "option 5 ",
    "option 6 "]],
    "MOUSE POPUP",2,X),
    write(X).

/* ***** END OF DMSMENU.PRO ***** */

```

LISTING 3: SCROLL.PRO

```

/* ***** */
/* *** A scroll predicate with string saving buffer *** */
/* ***** */
NOTE: to test this program by itself, uncomment the
lines with the "@" symbol. */

/* @ include "msdoms.pro" */

PREDICATES
assertFILEstr(STRING) /* assert a text_file fact. */
getROWstr(STRINGLIST) /* get list of ROWS (strings) */

scr(INTEGER,INTEGER) /* this is my "scroll(i,i)" */
refreshRows(INTEGER,INTEGER,INTEGER,
  STRINGLIST,INTEGER,INTEGER)
refreshU(INTEGER,INTEGER,INTEGER,STRINGLIST,
  INTEGER,INTEGER,INTEGER)
refreshD(INTEGER,INTEGER,INTEGER,STRINGLIST,
  INTEGER,INTEGER,INTEGER)
refreshRpos(INTEGER,INTEGER,STRINGLIST,
  INTEGER,INTEGER,INTEGER,INTEGER)

refreshCOLS(INTEGER,INTEGER,INTEGER,
  STRINGLIST,INTEGER,INTEGER)
refreshL(INTEGER,INTEGER,INTEGER,
  STRINGLIST,INTEGER,INTEGER)
refreshR(INTEGER,INTEGER,INTEGER,STRINGLIST,INTEGER)
refreshCpos(INTEGER,INTEGER,INTEGER,
  INTEGER,STRINGLIST,INTEGER)

getROWS(INTEGER,INTEGER,
  STRINGLIST,STRINGLIST)
getROWS1(INTEGER,INTEGER,
  STRINGLIST,STRINGLIST)
getROWS2(INTEGER,STRINGLIST,STRINGLIST)
refreshCposA(INTEGER,INTEGER,INTEGER,INTEGER,
  STRINGLIST)

/* Utility Predicates */

/* @ repeat
erase */
analyseFILENAME(STRING,STRING,STRING)
extract_str(STRING,INTEGER,INTEGER,STRING)
extract_str1(STRING,INTEGER,INTEGER,INTEGER,STRING)
list_len(STRINGLIST,INTEGER,INTEGER)
pointerLIMIT(INTEGER,INTEGER,INTEGER,INTEGER)

/* @ testscr(INTEGER,INTEGER) */

CLAUSES

/* ***** assertFILEstr ***** */
assertFILEstr(InFILENAME) if
  openread(infile,InFILENAME),
  readdevice(infile),
  getROWstr(LIST),
  assert(file_text(LIST)),
  assert(pointer(0,0)),
  closefile(infile),
  readdevice(keyboard).

```

Another Lahey Computer Systems, Inc. Sets a New FORTRAN Standard!

Introducing the latest addition to our line of PC FORTRAN Language Systems—
Lahey Personal FORTRAN 77 Version 2.0

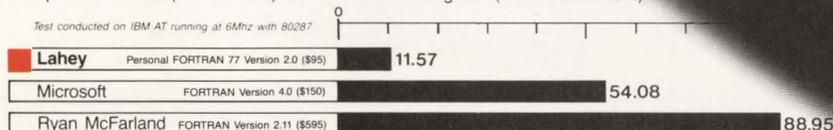
What You Get When You Purchase Lahey Personal FORTRAN:

Lahey Experience.

We are experts in designing and implementing FORTRAN Language Systems. Lahey has been producing mainframe implementations since 1967 and PC FORTRANs (F77L) since 1984. In fact, F77L was named the "EDITOR'S CHOICE" among PC FORTRANs by *PC Magazine*. This 20-year span of specialization has been incorporated into the design of our revolutionary Lahey Personal FORTRAN 77.

LAHEY SLASHES COMPILATION TIME.

Compilation times (in seconds) for Whetstone Program (WHETS3H.FOR)



Customer Support:

Our philosophy is that customer relationships begin, rather than end, at the point of sale. Services include free technical support, electronic bulletin board for fast service and information access, and newsletters to keep you up to date on our latest developments.

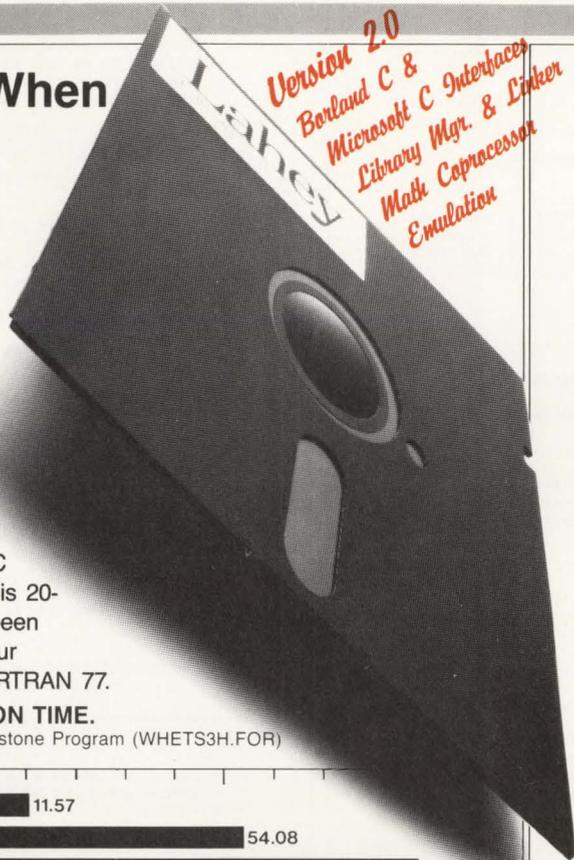
Purchasing the Lahey Personal FORTRAN 77 gives you software designed by FORTRAN experts, a feature-loaded product with industry-leading compilation speed, and quality technical support; all for **\$95**.

International Representatives: Australia: Comp. Transitions, Tel. (03)5372786 • Canada: Barry Mooney & Assoc., Tel. (902)6652941 • Denmark: Ravenholm Computing, Tel. (02)887249 • England: Grey Matter Ltd., Tel. (0364)53499 • Holland: Lemax Co. B.V. (02968)4210 • Japan: Microsoft Inc., Tel. (03)813822 • Norway: Polysoft A.S. (03)892240 • Switzerland: DST Comp. Services, Tel. (022)989188

MS-DOS & MS FORTRAN are trademarks of Microsoft Corporation.

We have a complete line of PC FORTRAN Language Systems. For developing or porting programs there is no substitute for a Lahey. If you would like information on any of these products, please call 1-800-548-4778.

Lahey Personal FORTRAN 77 — So much for so little	\$95
F77L — "Editor's Choice" PC Magazine	\$477
F77L-EM/16 — Ability to write programs as large as 15 MB on 80286	\$695
F77L-EM/32 — New 32-bit — Programs up to 4GB on 80386.	\$895



Feature Loaded:

- Full implementation of the ANSI X3.9-1978 FORTRAN Standard
- Fast Compilation (see chart)
- Popular Language Extensions highlighted in the manual
- Source On-Line Debugger
- English Diagnostics and Warning Messages
- LOGICAL*1, LOGICAL*4
- INTEGER*2, INTEGER*4
- REAL*4, REAL*8, and DOUBLE PRECISION
- COMPLEX*8, COMPLEX*16
- Recursion
- 31-Character Names
- Trailing Comments
- Cross Reference and Source Listings
- 64 KB Generated Code
- 64 KB Stack Storage
- 64 KB Commons, Constants and Saved Local Data
- Math coprocessor emulation runs with or without a math coprocessor chip
- 400-Page User Manual

SYSTEM REQUIREMENTS:
256K Ram MS-DOS (2.0 or later)

\$95

Lahey is setting the PC FORTRAN Standard.

TO ORDER

1-800-548-4778

(specify disk size)

Lahey Computer Systems, Inc.

P.O. Box 6091

Incline Village, NV 89450

Telephone: (702) 831-2500

TELEX: 9102401256

FAX: (702) 831-8123

Lahey
Computer Systems Inc.

```

/* ***** analyseFILENAME ***** */
analyseFILENAME(STRname,Name,Extension) if
fronttoken(STRname,Name,Rest),
fronttoken(Rest,".",Extension).

/* ***** getROWstr ***** */
getROWstr():- eof(infile), !.
getROWstr([Hrow|RESTRows]):-
    readln(Hrow),
    getROWstr(RESTRows), !.

/* ***** repeat ***** */
/* @ repeat.
repeat:- repeat.
*/
/* ***** SCROLL PREDICATE "scr" ***** */
/* ***** scr(ROWS, COLS) ***** */
scr(ROWS, COLS):-
    file_text(STRlist),
    makewindow(, , , , , RN, CN),

/* now start the operation of refreshing text */

/* pointer position (row and column),
the pointer position is the starting row and column
of part of the text file that will be displayed in
the currently active window. The pointer position
in text (say 3,0 or 4,5) always corresponds to
window position of 0,0. */

/* pointer position in the text */
retract(pointer(Rpos,Cpos)),

    refreshROWS(ROWS,Rpos,Cpos,STRlist,RN,CN),
    NewRpos=Rpos+ROWS,
    refreshCOLS(COLS,NewRpos,Cpos,STRlist,RN,CN),
    NewCpos=Cpos+COLS,
    assert(pointer(NewRpos,NewCpos)).

/* ***** refreshROWS ***** */
/* ***** refreshROWS ***** */
/* ***** refreshROWS ***** */

/* refresh the rows */
refreshROWS(0, , , , ,):- !.
refreshROWS(ROWS, Prow, Pcol, STRlist, NR, NC):-
    ROWS > 0,
    WR=NR-1, /* Assuming current window has no frame */
    refreshU(ROWS, Prow, Pcol, STRlist, WR, 0, NR, NC), !.
refreshROWS(ROWS, Prow, Pcol, STRlist, NR, NC):-
    ROWS < 0,
    refreshD(ROWS, Prow, Pcol, STRlist, 0, 0, NR, NC), !.

/* ***** refreshD ***** */
/* refresh Down, put text at bottom of window */
/* refresh from current position "Cpos" Nrows
going backward in the list of rows means
upward in the window. */

refreshD(0, , , , ,):- !.
refreshD(ROWNum, Prow, Pcol, STRlist, Wrow, Wcol, WNR, WNC):-
    TempPos=Prow-1,
    scroll(-1,0),
    refreshRpos(TempPos, Pcol, STRlist, Wrow, Wcol, WNR, WNC),
    NewRowNum=ROWNum+1,
    refreshD(NewRowNum, TempPos, Pcol, STRlist,
        Wrow, Wcol, WNR, WNC), !.

/* ***** refreshU ***** */
/* refresh Up, put text at top of window.
refresh from current position "Cpos" and Nrows going forward.
Forward in list of rows means downward in the window. */

refreshU(0, , , , ,):- !.
refreshU(ROWNum, Prow, Pcol, STRlist, WR, WC, NR, NC):-
    TempProw=Prow+1,
    scroll(1,0),

/* adjust the starting row to be pulled from
text file and placed in window. We do this by
adding the window depth, num of rows to the current
pointer into the text file. The skipped rows are
assumed to be reserved on the screen by the
built-in scroll predicate. */

AdjustedProw=Prow+NR,
refreshRpos(AdjustedProw, Pcol, STRlist, WR, WC, NR, NC),
NewRowNum=ROWNum-1,
refreshU(NewRowNum, TempProw, Pcol,
    STRlist, WR, WC, NR, NC), !.

/* ***** refreshRpos ***** */
/* write a certain row, the "Pos" numbered row in the
text file, at the field defined by the current window
row, Wrow, Wcol, WNC and of course the string of that text
file row, RowStr. */

```

```

refreshRpos( , , , , ,):- !.
refreshRpos(0, Pcol, [ROWstr|_], Wrow, Wcol, , WNC):-
    frontstr(Pcol, Rowstr, , STR),
    field_str(Wrow, Wcol, WNC, STR), !.
refreshRpos(Pos, Pcol, [_|RESTRows], WR, WC, RN, CN):-
    StepPos=Pos-1,
    refreshRpos(StepPos, Pcol, RestRows, WR, WC, RN, CN), !.

/* ***** refreshCOLS ***** */
/* ***** Refresh the columns ***** */

refreshCOLS(COLS, Prow, Pcol, STRlist, RowNum, ColNum)

COLS    number of columns to refresh
Prow    pointer row number
Pcol    pointer col number
STRlist list of rows in the text
RowNum  number of rows in screen window
ColNum  number of columns in screen window

/*
refreshCOLS(0, , , , ,):- !.
refreshCOLS(COLS, Prow, Pcol, STRlist, RowNum, ColNum):-
    COLS > 0, /* scrolling text to the left */
    scroll(0, COLS),
/* @ readln( , ) */
    refreshL(COLS, Prow, Pcol, STRlist, RowNum, ColNum), !.

refreshCOLS(COLS, Prow, Pcol, STRlist, RowNum, _):-
    COLS < 0, /* scrolling text to the right */
    scroll(0, COLS),
    refreshR(COLS, Prow, Pcol, STRlist, RowNum), !.

/* ***** refreshL ***** */
/* refresh to the left */
refreshL(COLS, Prow, Pcol, STRlist, RowNum, ColNum):-
/* pointer to start of refresh in Text */
    TStartCol=Pcol+ColNum,
/* pointer to start of refresh in Window */
    WStartCol=ColNum-COLS,
    refreshCpos(COLS, TStartCol, WStartCol,
        Prow, STRlist, RowNum).

/* ***** refreshR ***** */
/* refresh to the right */
refreshR(COLS, Prow, Pcol, STRlist, RowNum):-
    TStartCol=Pcol+COLS,
/* refreshing from left of screen starts at 0 */
    WStartCol=0,
    C = abs(COLS),
    refreshCpos(C, TStartCol, WStartCol,
        Prow, STRlist, RowNum).

/* ***** refreshCpos ***** */
/* refresh a part of text into certain parts of screen */

refreshCpos(COLS, TstartCOL, WstartCOL, Prow, InList, Rnum):-
    getROWS(Prow, Rnum, InList, OutList),
    WstartROW=0,
    refreshCposA(COLS, TstartCOL, WstartROW,
        WstartCOL, OutList).

/* ***** refreshCposA ***** */
refreshCposA( , , , , ,):- !.
refreshCposA(COLS, TstartC, WstartR,
    WstartC, [ROWstr|Rest]):-
    NextWstartR=WstartR+1,
    extract_str(ROWstr, TstartC, COLS, STR),
    str_len(STR, L),
    field_str(WstartR, WstartC, L, STR),
    refreshCposA(COLS, TstartC, NextWstartR, WstartC, Rest), !.

/* ***** getROWS ***** */
getROWS(0, NumofRows, InStrList, OutStrList):-
    list_len(InStrList, 0, Len),
    getROWS1(NumofRows, Len, InStrList, OutStrList), !.
getROWS(PointerROW, NumofRows, [_|Rest], OutList):-
    UpdatedPointerRow=PointerRow-1,
    getROWS(UpdatedPointerRow, NumofRows, Rest, OutList).

/* ***** getROWS1 ***** */
getROWS1(NumofRows, Len, InList, OutList):-
    NumofRows < Len,
    getROWS2(NumofRows, InList, OutList), !.
getROWS1( , , List, List):- !.

/* ***** getROWS2 ***** */
getROWS2(0, , ,):- !.
getROWS2(NumofRows, [H|T1], [H|T2]):-
    NewN=NumofRows-1,
    getROWS2(NewN, T1, T2), !.

```

```

/* ***** extract_str ***** */
/* Flow Patterns (i,i,i,0), (i,i,i,i)
extract the string of characters OutStr that is part
of the string InStr. OutStr starts at the column StartCol
in InStr. The length of OutStr is Length. */

extract_str(INSTR,SC,NC,OUTSTR):-
  str_len(INSTR,LENGTH),
  extract_str1(INSTR,LENGTH,SC,NC,OUTSTR).

extract_str1(LENGTH,SC,NC):- LENGTH < SC, !.
extract_str1(INSTR,LENGTH,SC,NC,OUTSTR):-
  LENGTH < SC+NC,
  frontstr(SC,INSTR,STR),
  OUTSTR=STR, !.
extract_str1(INSTR,SC,NC,OUTSTR):-
  frontstr(SC,INSTR,RESTSTR),
  frontstr(NC,RESTSTR,STR),
  OUTSTR=STR, !.

/* ***** list_len ***** */
list_len([],L):- !.
list_len([_|_],OldLength,CurrentLength):-
  AddLength=OldLength+1,
  list_len(AddLength,CurrentLength), !.

/* ***** pointerLIMIT ***** */
pointerLIMIT(RowNum,Length,PointerRow,NewRowNum):-
  RowNum > 0,
  Limit = Length - PointerRow - 1,
  RowNum >= Limit,
  NewRowNum = Limit, !.
pointerLIMIT(RowNum,_,PointerRow,NewRowNum):-
  RowNum < 0,
  Num = abs(RowNum),
  Num >= PointerRow,
  NewRowNum = - PointerRow, !.
pointerLIMIT(RowNum,_,_,RowNum):- !.

/* ***** test ***** */
/* @ testscr(X,Y):-
  makewindow(1,71,0,5,5,13),
  file_str("b:scr.tst",STR),
  window_str(STR),
  assertFILEstr("b:scr.tst"),
  scr(X,Y),
  readln(_).
*/
/* ***** END OF SCROLL.PRO ***** */

```

LISTING 4: MSBAR.PRO

```

/* ***** */
/* A scrolling-bar program */
/* ***** */
include "msdoms.pro" /* domains & database declarations */
include "msut.pro" /* utility predicates */
include "msm-drv.pro" /* the mouse-bios calls */
include "compare.pro" /* compare mouse & screen positions */
include "scroll.pro" /* the modified scroll, "scr" */

PREDICATES
  ms_act(INTEGER,INTEGER,INTEGER,INTEGER)
  msbarV(INTEGER,INTEGER,INTEGER,
          INTEGER,INTEGER,INTEGER)
  msbarH(INTEGER,INTEGER,INTEGER,INTEGER,INTEGER)

  addVbar(STRING)
  putVbar(STRING)
  addHbar(STRING)
  putHbar(STRING)

  exist_VBAR(INTEGER)
  exist_HBAR(INTEGER)
  INITbarV(INTEGER,INTEGER,INTEGER,
            INTEGER,INTEGER,INTEGER)
  INITbarH(INTEGER,INTEGER,INTEGER,
            INTEGER,INTEGER,INTEGER)

/* TESTING PREDICATES */
showBARS
testHORZ
testVERT
testBOTH

/* ADDITIONAL UTILITY PREDICATES */
erase
scrollUNIT(INTEGER,INTEGER,INTEGER)

CLAUSES

/* ***** */
/* "ms_act" Classifying and Executing mouse actions */
/* ***** */
If mouse button "Button" was pressed inside the vertical
bar window, then vertical scrolling is active, msbarV. */

```

```

ms_act(Button,MsR,MsC,RowUnit):-
  makewindow(WN,_,_,_),
  BN=WN+100,
  barV(BN,_,SR,SC,NR,NC),
  compare_pos(MsR,MsC,SR,SC,NR,NC,NewRow,_,inside),
  barROW(BR),
  BR = NewRow,
  repeat,
  msm_stat(B,R,C),
  Row=R/B, Col=C/B,
  msbarV(B,Button,Row,Col,BN,RowUnit), !.

```

```

/* If mouse button "Button" was pressed inside the horiz.
bar widow, then horizontal scrolling is active, msbarH */
ms_act(Button,MsR,MsC):-
  makewindow(WN,_,_,_),
  BN=WN+101,
  barH(BN,_,SR,SC,NR,NC),
  compare_pos(MsR,MsC,SR,SC,NR,NC,NewCOL,inside),
  barCOL(CO),
  CO = NewCOL,
  repeat,
  msm_stat(B,R,C),
  Row=R/B, Col=C/B,
  msbarH(B,Button,Row,Col,BN), !.

```

```

/* ***** */
/* ***** mouse controlled scrolling-bar program ***** */
/* ***** */

```

```

msbarV(B,B,Row,Col,BN,ScrollFactor):-
  barV(BN,BARattr,BOXattr,SR,SC,NR,NC),
  compare_pos(Row,Col,SR,SC,NR,NC,NewRow,_,inside),
  barROW(OldRow),
  NewRow <> OldRow,
  shiftwindow(BN),
  field_attr(OldRow,0,NC,BARattr),
  field_attr(NewRow,0,NC,BOXattr),
  WN = BN - 100,
  shiftwindow(WN),
  retract(barROW(OldRow)),
  assert(barROW(NewRow)),
  S = NewRow - OldRow,
  S1 = S * ScrollFactor,
  scr(S1,0), !,
  fail.

```

```

msbarH(B,B1,Row,Col,BN,ScrollFactor):-
  B <> B1,
  barV(BN,BARattr,BOXattr,SR,SC,NR,NC),
  compare_pos(Row,Col,SR,SC,NR,NC,NewRow,_,inside),
  barROW(OldRow),
  NewRow <> OldRow,
  shiftwindow(BN),
  field_attr(OldRow,0,NC,BARattr),
  field_attr(NewRow,0,NC,BOXattr),
  WN = BN - 100,
  shiftwindow(WN),
  retract(barROW(OldRow)),
  assert(barROW(NewRow)), !,
  S = NewRow - OldRow,
  S1 = S * ScrollFactor,
  scr(S1,0), !.
msbarV(B,B1,Row,Col,BN,_) :-
  B <> B1,
  barV(BN,_,SR,SC,NR,NC),
  compare_pos(Row,Col,SR,SC,NR,NC,_,_,outside), !.

```

```

/* ***** msbarH ***** */

```

```

msbarH(B,B,Row,Col,BN):-
  barH(BN,BARattr,BOXattr,SR,SC,NR,NC),
  compare_pos(Row,Col,SR,SC,NR,NC,NewCol,inside),
  barCOL(OldCol),
  NewCol <> OldCol, !,
  shiftwindow(BN),
  field_attr(0,OldCol,1,BARattr),
  field_attr(0,NewCol,1,BOXattr),
  WN = BN - 101,
  shiftwindow(WN),
  retract(barCOL(OldCol)),
  assert(barCOL(NewCol)),
  S = NewCol - OldCol,
  scr(0,S), !,
  fail.
msbarH(B,B1,Row,Col,BN):-
  B <> B1,
  barH(BN,BARattr,BOXattr,SR,SC,NR,NC),
  compare_pos(Row,Col,SR,SC,NR,NC,NewCol,inside),
  barCOL(OldCol),
  NewCol <> OldCol, !,
  shiftwindow(BN),
  field_attr(0,OldCol,1,BARattr),
  field_attr(0,NewCol,1,BOXattr),
  WN = BN - 101,
  shiftwindow(WN),
  retract(barCOL(OldCol)),
  assert(barCOL(NewCol)),

```

```

S = NewCol - OldCol,
scr(0,S), 1.
msbarH(B,B1,Row,Col,BN):-
B <> B1,
barH(BN,_,_,SR,SC,NR,NC),
compare_pos(Row,Col,SR,SC,NR,NC,_,_,outside), 1.
/* ***** addVbar & addHbar ***** */
addVbar(STR):-
makewindow(WN,Wattr,Fattr,_,R,C,RN,CN),
Fattr <> 0,
NewWN=WN+20,
NewRN=RN-2,NewCN=CN-2,NewR=R+1,NewC=C+1,
makewindow(NewWN,Wattr,0,_,NewR,NewC,NewRN,NewCN),
putVbar(STR), 1.
addVbar(STR):-
putVbar(STR), 1.

addHbar(STR):-
makewindow(WN,Wattr,Fattr,_,R,C,RN,CN),
Fattr <> 0,
NewWN=WN+20,
NewRN=RN-2,NewCN=CN-2,NewR=R+1,NewC=C+1,
makewindow(NewWN,Wattr,0,_,NewR,NewC,NewRN,NewCN),
putHbar(STR), 1.
addHbar(STR):-
putHbar(STR), 1.

/* ***** putVbar ***** */
/* check if vertical bar already exists for the current
window */
putVbar(_):-
makewindow(NW,_,_,_,_,_,_,_),
NW < 100, /* the window is not a bar window */
exist_VBAR(NW),
shiftwindow(NW),
write("\n vertical scrolling bar already exists"), 1.

/* If window has horizontal bar then top level window is
a horizontal bar window and needs be removed first then we
resize main window and recreate horiz. bar then add
vertical bar. */
putVbar(STR):-
makewindow(WN,Wattr,Fattr,Label,
Srow,Scol,Nrows,Ncols),
exist_HBAR(WN), /* horiz. bar. already exists */
VBWN=WN+100,
HBWN=WN+101,
NewNcols=Ncols-2,
Bcol=Scol+NewNcols,
Brow=Srow-1, /* save window status before removing window */
removewindow,
removewindow, /* remove old horiz. bar */
INITbarH(HBWN,120,16,Brow,Scol,NewNcols),
INITbarV(VBWN,120,16,Srow,Bcol,Nrows),
makewindow(WN,Wattr,Fattr,Label,
Srow,Scol,Nrows,NewNcols),
window_str(STR), 1.

/* If window has neither horizontal nor vertical bars,
then create vertical bar */
putVbar(STR):-
makewindow(NW,Wattr,Fattr,Label,
Srow,Scol,Nrows,Ncols),
VBWN=NBW+100,
NewNcols=Ncols-2,
Bcol=Scol+NewNcols,
/* removing old window */
removewindow,
/* create the vertical bar */
INITbarV(VBWN,120,16,Srow,Bcol,Nrows),
/* update main window */
makewindow(WN,Wattr,Fattr,Label,
Srow,Scol,Nrows,NewNcols),
window_str(STR), 1.

/* ***** putHbar ***** */
/* Check if horizontal bar already exist for the current window */
putHbar(_):-
makewindow(NW,_,_,_,_,_,_,_),
NW < 100, /* the window is not a bar window */
exist_HBAR(NW),
shiftwindow(NW),
write("\n vertical scrolling bar already exists"), 1.

```

```

/* If window has vertical bar then only resize window and recreate
vertical bar, then add horizontal bar. */
putHbar(STR):-
makewindow(WN,Wattr,Fattr,Label,
Srow,Scol,Nrows,Ncols),
exist_VBAR(WN), /* horiz. bar. already exists */
VBWN=WN+100,
HBWN=WN+101,
NewNrows=Nrows-1,
NewSrow=Srow+1,
VBcol=Scol+Ncols,
removewindow,
removewindow, /* remove old vert. bar */
INITbarV(VBWN,120,16,NewSrow,VBcol,NewNrows),
INITbarH(HBWN,120,16,Srow,Scol,Ncols),
makewindow(WN,Wattr,Fattr,Label,NewSrow,
Scol,NewNrows,Ncols),
window_str(STR), 1.

/* If window has neither horizontal nor vertical bars,
create a horizontal bar */
putHbar(STR):-
makewindow(WN,Wattr,Fattr,Label,
Srow,Scol,Nrows,Ncols),
HBWN=WN+101,
NewNrows=Nrows-1,
NewSrow=Srow+1,
removewindow,
INITbarH(HBWN,120,16,Srow,Scol,Ncols),
makewindow(WN,Wattr,Fattr,Label,
NewSrow,Scol,NewNrows,Ncols),
window_str(STR), 1.

/* ***** exist_VBAR & exist_HBAR ***** */
exist_VBAR(WN):-
BWN=WN+100,
existwindow(BWN).

exist_HBAR(WN):-
BWN=WN+101,
existwindow(BWN).

/* ***** INITbarV & INITbarH ***** */
/* Initialize the vertical scrolling bar */
INITbarV(WN,BARattr,BOXattr,Brow,Bcol,Blength) :-
makewindow(WN,BARattr,0,_,Brow,Bcol,Blength,2),
assert(barV(WN,BARattr,BOXattr,Brow,Bcol,Blength,2)),
field_attr(0,0,2,BOXattr),
assert(barROW(0)), 1.

/* Initialize the horizontal scrolling bar */
INITbarH(WN,BARattr,BOXattr,Brow,Bcol,Blength) :-
makewindow(WN,BARattr,0,_,Brow,Bcol,1,Blength),
assert(barH(WN,BARattr,BOXattr,Brow,Bcol,1,Blength)),
field_attr(0,0,1,BOXattr),
assert(barCOL(0)), 1.

/* ***** testing ***** */
/* ***** testing ***** */
showBARS:-
makewindow(1,70,2,"SHOWbars",0,0,25,80),
window_str(STR),
addVbar(STR),
readln(_),
addHbar(STR),
readln(_).

testHORZ:-
erase,
file_str("scr.tst",STR),
assert(FILEstr("scr.tst")),
makewindow(1,6,71,"testHORZ",10,5,10,65),
addHbar(STR),
msm_init,
msm_show,
file_text(ListOfRows),
list_len(ListOfRows,0,L),
makewindow(1,6,71,"testHORZ",10,5,10,65),
scrollUNIT(L,NR,RowUnit),
repeat,
msm_stat(B,R,C),
Row=R/8, Col=C/8,
B <> 0,
ms_act(B,Row,Col,RowUnit), fail.

testVERT:-
erase,
file_str("scr.tst",STR),
assert(FILEstr("scr.tst")),
makewindow(1,6,71,"testVERT",10,5,10,65),
addVbar(STR),
msm_init,
msm_show,

```

```

file_text(ListOfRows),
list_len(ListOfRows,0,L),
makewindow(_____,NR),
scrollUNIT(L,NR,RowUnit),
repeat,
  msm_stat(B,R,C),
  Row=R/8, Col=C/8,
  B <> 0,
  ms_act(B,Row,Col,RowUnit), fail.

testBOTH:-
  erase,
  file_str("scr.tst",STR),
  assertFILEstr("scr.tst"),
  makewindow(1,6,71,"testBOTH",10,5,10,65),
  addHbar(STR),
  addVbar(STR),
  msm_init,
  msm_show,
  file_text(ListOfRows),
  list_len(ListOfRows,0,L),
  makewindow(_____,NR),
  scrollUNIT(L,NR,RowUnit),
  repeat,
    msm_stat(B,R,C),
    Row=R/8, Col=C/8,
    B <> 0,
    ms_act(B,Row,Col,RowUnit), fail.

/* ***** */
/* ***** utility ***** */
/* ***** */
L= number of lines in text file.
N= number of rows in active window.
R= a unit used by the bar moving indicator, such that
for each window-row this indicators moves, the program
scrolls R lines in the text file. */

scrollUNIT(L,N,R):-
  0 = L mod N,
  R = L div N, 1.

scrollUNIT(L,N,R):-
  R = L div N + 1.

/* retract all knowledge base facts. */
erase:- retract(_), fail.
erase.

/* ***** END of msbar.pro ***** */

```

LISTING 5: MSDOMS.PRO

```

/* ***** */
/* ***** DOMAINS ***** */
/* ***** */

DOMAINS
STRINGLIST = STRING*
INTEGERLIST = INTEGER*
FILE = infile

DATABASE

file_text(StringLIST)
pointer(INTEGER,INTEGER)
barROW(INTEGER)
barCOL(INTEGER)

/* barV(ParentWindowNo,BarNo,Srow,Scol,Nrow) */
barV(INTEGER,INTEGER,INTEGER,INTEGER,
      INTEGER,INTEGER)
barH(INTEGER,INTEGER,INTEGER,INTEGER,
      INTEGER,INTEGER,INTEGER)

currentrow(INTEGER)

```

LISTING 6: MSUT.PRO

```

/* File name: "msut.pro,"

NOTE: This file is extracted from A TURBO PROLOG TOOLBOX
utility file. */

/* ***** */
/* ***** UTILITES ***** */
/* ***** */

/* ***** repeat ***** */

PREDICATES
nondeterm repeat

CLAUSES
repeat.
repeat:-repeat.

/* ***** */
/* ***** miscellaneous ***** */
/* ***** */

PREDICATES
/* The length of the longest string */
maxlen(StringLIST,INTEGER,INTEGER)
/* The length of a list */
listlen(StringLIST,INTEGER)
/* used in the menu predicates */
writelist(INTEGER,INTEGER,StringLIST)
/* Returns the reversed attribute */
reverseattr(INTEGER,INTEGER)
min(INTEGER,INTEGER,INTEGER)
max(INTEGER,INTEGER,INTEGER)

CLAUSES
maxlen([_],MAX,MAX1):-
  str_len(H,LENGTH),
  LENGTH>MAX,1,
  maxlen(T,LENGTH,MAX1).
maxlen([_],MAX,MAX1):- maxlen(T,MAX,MAX1).
maxlen([],LENGTH,LENGTH).

listlen([],0).
listlen([_],N):-
  listlen(T,X),
  N=X+1.

writelist(_____, []).
writelist(LI,ANTKOL,[H|T]):-
  field_str(LI,0,ANTKOL,H),
  LI1=LI+1,
  writelist(LI1,ANTKOL,T).

min(X,Y,X):-X<=Y,!.
min(_____,X,X).

max(X,Y,X):-X>=Y,!.
max(_____,X,X).

reverseattr(A1,A2):-
  bitand(A1,$07,H11),
  bitleft(H11,4,H12),
  bitand(A1,$70,H21),
  bitright(H21,4,H22),
  bitand(A1,$08,H31),
  A2=H12+H22+H31.

/* ***** */
/* adjustwindow takes a windowstart and a window size and
adjusts the windowstart so the window can be placed on
the screen. adjframe looks at the frameattribute: if it
is different from zero, two is added to the size of the
window */
/* ***** */

PREDICATES
adjustwindow(INTEGER,INTEGER,INTEGER,
             INTEGER,INTEGER,INTEGER)
adjframe(INTEGER,INTEGER,INTEGER,INTEGER,INTEGER)

CLAUSES
adjustwindow(LI,KOL,DLI,DKOL,ALI,AKOL):-
  LI<25-DLI,KOL<80-DKOL,!,ALI=LI,AKOL=KOL.
adjustwindow(LI,_____,DLI,DKOL,ALI,AKOL):-
  LI<25-DLI,!,ALI=LI,AKOL=80-DKOL.
adjustwindow(_____,KOL,DLI,DKOL,ALI,AKOL):-
  KOL<80-DKOL,!,ALI=25-DLI,AKOL=KOL.
adjustwindow(_____,_____,DLI,DKOL,ALI,AKOL):-
  ALI=25-DLI,AKOL=80-DKOL.

adjframe(0,R,C,R,C):-!.
adjframe(_____,R1,C1,R2,C2):-R2=R1+2,C2=C1+2.

/* ***** */

```

BINARY TO TEXT FOR COMMUNICATIONS

Transform binary files into transmittable Turbo Basic programs that can convert themselves back into their original form.

Robert E. Stearns, Jr.



PROGRAMMER

Many electronic mail services, including MCI Mail, handle only text files, sometimes with limited character sets. A *text file* as used here means a file that contains *only* the printable characters in the ASCII character set, plus a handful of control characters that are often called "whitespace" characters: carriage return, line feed, tab, and sometimes BEL (character 7). If you want to send a binary file (or a file collection, such as an .ARC archive file) via such an electronic mail service, you have to first convert the file to a text file. In addition, the recipient must have a corresponding program to re-create the text of your file in its original form.

TEXTIFY.BAS

TEXTIFY.BAS (Listing 1) streamlines both steps by converting any file to a Turbo Basic program that contains only the common displayable characters from the ASCII set, in lines less than 72 characters long. This new Turbo Basic program can then be re-constructed on the receiving end using a BASIC interpreter or compiler such as Turbo Basic, BASICA, or QuickBASIC (I've tested lengthy samples with all three). With these options, almost anyone with an IBM PC or compatible should be able to retrieve a file that has been encoded with TEXTIFY.

TEXTIFY, which is written in Turbo Basic, has three main parts: initialization, main file processing, and termination. In addition, a number of **DATA** statements contain the encoded data needed to re-create the file in its original form.

Initialization. Initialization involves several steps:

- Getting the source file name;
- Creating the destination filename;
- Opening both the source and the destination files;
- Initializing the translation table; and
- Writing the file re-creation program to the output file.

The translation table created by the initialization steps contains only upper- and lowercase alphabetic characters, the ten numeric digits, and the "@" and "\$" symbols. The main file processing converts 8-bit binary values into the 64 characters in the translation table.

Main file processing. Main file processing reads a section of the source file using **BINARY** mode file processing. If the character length of the last section of the source file is not equal to a multiple of three, that section is extended in order to equal that length. For every 3 bytes ($24 = 3 \times 8$ bits) of the input file, 4 integers in the range 0 to 63 ($24 = 4 \times 6$ bits) are created. Each of these integers is used as a subscript to access one of the characters from the translation table and then to write that character to disk.

The characters are written to disk as part of a series of **DATA** statements. As each section of the source file is converted, a checksum is calculated. After the last character for each section has been written to disk, this checksum is appended to the end of each **DATA** statement. After transmission, the file re-creation program also computes this checksum, and then compares its result to the checksum read from the **DATA** statement in order to determine if the file has been corrupted by the transmission process. This process continues until no more data is left in the input file. Approximately every 1000 bytes, the user is informed about the progress of the conversion.

Termination. Termination writes a final **DATA** statement to the file, closes both the input and output files, and then tells the user that the process is complete.

RE-CREATION

The file re-creation program, embedded in string form in the **DATA** statements of TEXTIFY, is the logical inverse of TEXTIFY. (It's written in a form that enhances its readability. If you often transfer small binary files, you may want to compress this file re-

creation program by eliminating spaces and using colons [“:”] to combine separate lines onto one line.)

Re-creating the transmitted file involves nothing more than loading the transmitted file into a compatible BASIC compiler or interpreter and then running the file. The re-creation program converts the data in its **DATA** statements into a duplicate of the original file.

The re-creation process works like this: Since every character in the translation table is unique, the table can be used by the file re-creation program to regenerate each corresponding integer value. The re-creation program takes 4 characters at a time from its **DATA** statements, and maps each character into an integer in the range 0 to 63 ($4 \times 6 = 24$ bits) using the **INSTR** function. It then converts these 24 bits to 3 bytes ($3 \times 8 = 24$) of data, and writes these bytes to the output file. A checksum is computed for each **DATA** statement's data. That checksum is compared to the checksum that has been embedded in the **DATA** statement by the conversion program. If the two checksums do not match, an error message is displayed, indicating that

the file took a “hit” during transmission through communication channels.

READY, SET, TEXTIFY

TEXTIFY may be used in two different ways. If you type its name as a command (with no operands), it prompts you for the name of the file to convert. Alternatively, you can type the following:

```
TEXTIFY <filename>
```

This command causes TEXTIFY to process the file named <filename>. This latter format is appropriate for use in either communications scripts or .BAT files.

The cost that TEXTIFY exacts for its service is an expansion in the size of the transmitted file,

since every three binary bytes are represented by four ASCII characters, plus the constant overhead of the embedded re-creation program. Still, it's a small price to pay for being able to finesse a long string of eight-bit bytes through a seven-bit communications channel. ■

Bob Stearns is employed by the University of Georgia's Advanced Computational Methods Center. He is also a consultant on the use of supercomputers of several different architectures, and acts as the public domain librarian for the ACMC and the local PC users group.

Listings can be downloaded from CompuServe as TEXTIFY.ARC.

continued on page 112



```

'
' This program will take any file and create a BASIC program
' which will recreate the file, but which contains no characters
' that could cause any communications link a problem. The only
' characters in the file are those from the 95 character graphic
' subset of the ASCII set, and many of the more obscure
' characters from that group have been eliminated as well. Any
' file created with program should pass through almost any
' communications link unscathed. I even made sure the maximum
' line length was less than 72.
'
defint a-z          ' all integers makes everything faster
dim table$(63)     ' the character conversion table
chunksize=36       ' handle the file in pieces this size
a$=command$        ' get the name of the file to convert
'
' if the file name is not present in the command line, get it
' from the user interactively.
'
if a$="" then
  input "Type the name of the file to process";a$
end if
open a$ for binary as 1
'
' the output file name will be the same as the input file name,
' including path, but with the extension BAS.
'
i=instr(a$,".")
if i=0 then b$=a$ else b$=left$(a$,i-1)
b$=b$+".BAS"
open b$ for output as 2
'
' move the selected characters to the array to simplify and
' even speed up their access.
'
read tb$,tbx$
tb$=tb$+tbx$
for i=0 to 63
  table$(i)=mid$(tb$,i+1,1)
next i
'
' read the conversion program from the data statements and
' write it as the prefix to the converted data.
'
do
  read bline$
  print #2,bline$
loop until bline$=" 9999 ""
'
' write the first data statement containing the file length and
' file name to the converted file.
'
filelen=lof(1)
print #2,using "##### DATA ";10000;
print #2,a$,".",filelen!
print filelen!;" bytes to do" ' Tell the user how much there is to do
lineno=10001                  ' The line number in the output program
filepos!=1                    ' Current position in the input file
lim!=1000                     ' When to tell how much we've done
'
' This is the main code of the program. It reads chunks of the
' input file, converts each group of three bytes to four
' characters in the output file, and writes the characters to
' the output file in the form of data statements. As the input
' is processed, a checksum is formed for each chunk and the
' checksum is written to the output file to be checked by the
' program which will reconstruct the file.
'
while(filepos!<=filelen!)
  seek #1,filepos-1
  if filepos!+chunksize-1<=filelen! then
    get$ #1,chunksize,t$
  else
    get$ #1,filelen-filepos!+1,t$
    i=len(t$) mod 3 ' for the rest of the code to
    if i=0 then i=3 ' work properly, there must be a
    t$=t$+left$(" ",3-i) ' multiple of 3 chars in t$
  end if
  print #2,using "##### DATA ";lineno;
  lineno=lineno+1
  checksum=0
  for i=1 to len(t$) step 3
    j1=asc(mid$(t$,i,1)) ' aaaaaaaaaabbbbbbbcccccccc
    j2=asc(mid$(t$,i+1,1)) ' 111111222222333333444444
    j3=asc(mid$(t$,i+2,1)) ' as the above bit map
    c1= j1 \ 4 ' shows, we will convert
    c2=((j1 and 3) * 16 )+j2 \ 16 ' 24 bits of three data
    c3=((j2 and 15) * 4 )+j3 \ 64 ' bytes to four numbers
    c4= (j3 and 63) ' between 0 and 63.
    print #2,table$(c1); ' Next, we print them as
    print #2,table$(c2); ' characters which can be
    print #2,table$(c3); ' converted back to the
    print #2,table$(c4); ' corresponding numbers.
    checksum = checksum+j1+j2+j3 ' always <= chunksize*255

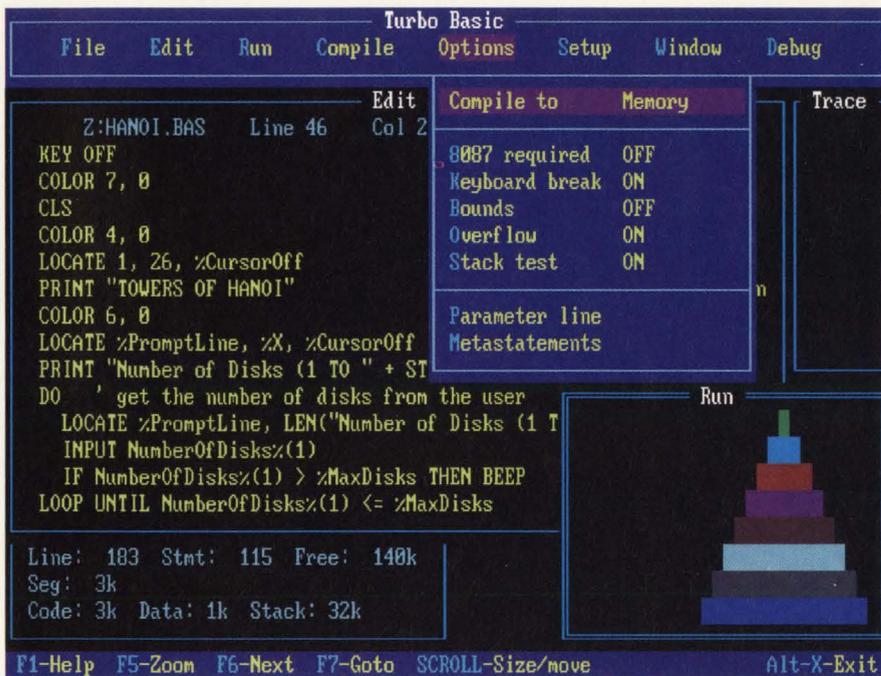
```

```

next i
print #2,"";checksum
filepos!=filepos!+chunksize
if filepos!>lim! then
  print filepos!;" bytes done"
  lim!=lim!+1000
end if
wend
'
' put on the final data statement indicating the end of the
' file, close the files and tell the user we are done
'
print #2,using "##### DATA ";lineno;
print #2,chr$(34);chr$(34);"0"
close #1
close #2
print "Files closed, job complete"
stop
'
' The characters to which the file is converted
'
data ABCDEFGHIJKLMNOPQRSTUVWXYZ
data abcdefghijklmnopqrstuvwxyz0123456789a$
'
' The file reconstruction program less the data statements
' which describe the file to be built. This program does the
' inverse transform of the program above. It processes each
' group of four characters into 6 bit integers, then concatenates
' consecutive groups of 8 bits into output characters. These
' output characters are then written to the output file until
' the original file size is reached.
'
data " 10 DEFINIT A-Z"
data " 15 READ TB$,TBX$,FC$,JS$,JES"
data " 16 TB$=TB$+TBX$"
data " 20 READ A$"
data " 30 OPEN A$ FOR OUTPUT AS #1"
data " 40 READ FS!"
data " 45 PRINT JS$;CHR$(32);A$"
data " 50 READ LNS,CS"
data " 55 L=10001"
data " 60 WHILE(LEN(LNS!)<>0)"
data " 65   CC=0"
data " 70   FOR I=1 TO LEN(LNS!) STEP 4"
data " 80     D1=INSTR(TB$,MID$(LNS,I,1))-1"
data " 90     D2=INSTR(TB$,MID$(LNS,I+1,1))-1"
data " 100    D3=INSTR(TB$,MID$(LNS,I+2,1))-1"
data " 110    D4=INSTR(TB$,MID$(LNS,I+3,1))-1"
data " 120    C1=((D1* 4) + (D2 \ 16)) AND 255"
data " 130    C2=((D2*16) + (D3 \ 4)) AND 255"
data " 140    C3=((D3*64) + D4 ) AND 255"
data " 145    CC=CC+C1+C2+C3"
data " 150    PRINT #1,CHR$(C1);"
data " 160    X1=X1+1"
data " 170    IF X1<FSI THEN PRINT #1,CHR$(C2); : X1=X1+1"
data " 180    IF X1<FSI THEN PRINT #1,CHR$(C3); : X1=X1+1"
data " 190    NEXT I"
data " 195    IF CC<>CS THEN PRINT FC$;L"
data " 200    READ LNS,CS"
data " 205    L=L+1"
data " 210 WEND"
data " 220 CLOSE #1"
data " 225 PRINT A$;CHR$(32);JES"
data " 230 STOP"
data " 1000 DATA ABCDEFGHIJKLMNOPQRSTUVWXYZ"
data " 1005 DATA abcdefghijklmnopqrstuvwxyz0123456789a$"
data " 1010 DATA FILE CORRUPTED AT"
data " 1020 DATA CREATING FILE"
data " 1030 DATA HAS BEEN CREATED"
data " 9999 ""

```

Basically speaking, there's one choice ... Turbo Basic!



Turbo Basic's development environment gives you overlapping windows, pull down menus, and the ability to run text-based applications in a window.

Turbo Basic® is the BASIC that lets even beginners write polished, professional programs almost as easily as they can write their names.

The others don't. When you really examine them, you'll find that even though they may be "quick," they make it hard to get where you're going. (Sort of like a car with an engine but no steering wheel.)

Turbo Basic takes you farther faster—in the comfort of a sleek development environment that gives you full control. Naturally it has a slick, fast compiler just like all Borland's technically superior Turbo languages. It also has a full-screen windowed editor, pull-down menus, and a trace debugging

system. And innovative Borland features like binary disk files, true recursion, and more control over your compiling. Plus the ability to create programs as large as your system's memory can hold.

The critics agree. The choice is basic. Turbo Basic from Borland.

“... What really makes Turbo Basic special is its blinding speed, small size, and many added commands. Programs compiled with Turbo Basic are often much faster and smaller than those produced by other compilers.

Ethan Winer, PC Magazine Best of 1987

Turbo Basic, simply put, is an incredibly good product.

William Zachman, Computerworld ”

Add another Basic advantage:
The Turbo Basic Toolboxes *New!*

- *The Database Toolbox* gives you code to incorporate into your own programs. You don't have to reinvent the wheel every time you write new Turbo Basic database programs. *New!*
- *The Editor Toolbox* is all you need to build your own text editor or word processor, including source code for two sample editors.

*60-Day Money-back Guarantee**

Compare the BASIC differences!

	<i>Turbo Basic 1.1</i>	QuickBASIC 4.0 Compiler	QuickBASIC 4.0 Interpreter
Compile & Link to stand-alone EXE	<i>3 sec.</i>	7 sec.	---
Size of .EXE	<i>28387</i>	25980	---
Execution time w/80287	<i>0.16 sec.</i>	16.5 sec.	21.5 sec.
Execution time w/o 80287	<i>0.16 sec.</i>	286.3 sec.	292.3 sec.

The Elkins Optimization Benchmark program from March 1988 issue of Computer Language was used. The Program was run on an IBM PS/2 Model 60 with 80287. The benchmark tests compiler's ability to optimize loop-invariant code, unused code, expression and conditional evaluation.

System Requirements: For the IBM PS/2™ and the IBM® family of personal computers and all 100% compatibles. Operating System: PC-DOS (MS-DOS) 2.0 or later. Toolboxes require Turbo Basic 1.1. Memory: 384K RAM for compiler, 640K RAM to compile Toolboxes.

*Customer satisfaction is our main concern: if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks of their respective holders. Copyright ©1988 Borland International, Inc. BI 1246



For the dealer nearest you
call (800) 543-7543

VIEWPORTS IN TURBO BASIC

Create a viewport—a screen within your screen—with the **VIEW** and **WINDOW** statements.

Peter Aitken

The screen display of graphics is an important part of the output of many computer programs. Whether it involves the simple shapes of a child's geometry tutoring program, or the complex engineering drawings of a CAD package, graphical display is a fundamental component of today's personal computers.

In Turbo Basic, the entire screen is normally used for graphics output. But what if we want to restrict the output to just a portion of the screen, so that the remainder of the screen can be used for something else? An example of how useful this can be is provided in Figure 1, which shows a screen from a simple program that I developed to quiz young children about basic geometrical shapes. Graphical output (the shapes) is restricted to the window, while the remainder of the screen is available for text output. This Turbo Basic program uses the **VIEW** statement to establish a rectangular region of the screen, called a *viewport*, to which all graphics output is sent.

In order to understand viewports and Turbo Basic's **VIEW** statement, you first need to understand the workings of the graphics screen and its coordinate system. If you already understand screen coordinates, feel free to skip ahead. If not, the following information will be useful.

THE GRAPHICS SCREEN

It's important to remember that viewports function only in graphics mode. In *graphics mode*, each of the individual dots, or *pixels*, on your screen can be controlled to create any pattern. In contrast, *text mode* displays predefined symbols, such as letters, numbers, and punctuation marks. While text can be displayed in graphics mode, individual pixels cannot be controlled in text mode.

All IBM and compatible computers support text mode; whether your computer also supports graphics mode depends upon its display adapter. The most common graphics display adapters are the Color

Graphics Adapter, or CGA, and the Enhanced Graphics Adapter, or EGA.

If we're going to control individual pixels, we need a way to specify their location. A pixel's location is specified by a pair of numbers, or *coordinates*, with the first, or X, coordinate giving the horizontal position, and the second, or Y, coordinate giving the vertical position. By convention, the pixel in the top left corner of the screen has coordinates (0,0). The X coordinate increases as you move to the right, and the Y coordinate increases as you move down, until you reach the pixel in the lower right corner of the screen. This pixel has the coordinates (XMAX-1, YMAX-1). The values XMAX and YMAX (which are used here simply for illustrative purposes, and are not predefined variables or constants) give the screen resolution. XMAX and YMAX, respectively, are the total number of pixels horizontally and vertically on the screen. The screen resolution varies depending upon the type of graphics adapter. With a CGA, XMAX and YMAX are 640 pixels and 200 pixels, respectively. With an EGA, they're 640 pixels and 350 pixels. For the remainder of this article, I'll assume CGA resolution; if you have an EGA, keep the difference in mind.

The coordinate system just described involves *physical coordinates*, which are understood directly by your computer's display hardware. You can also define a separate system of *logical coordinates* using Turbo Basic's **WINDOW** statement, as I'll describe a little later.

VIEWPORTS

Normally, graphics operations can utilize the entire screen. As I mentioned at the start of the article, Turbo Basic's **VIEW** statement lets you define a viewport, to which graphics drawing operations are limited. The syntax of the **VIEW** statement is:

```
VIEW [ [SCREEN] [(X1,Y1) - (X2,Y2)
      [, [color] [,boundary]]]
```

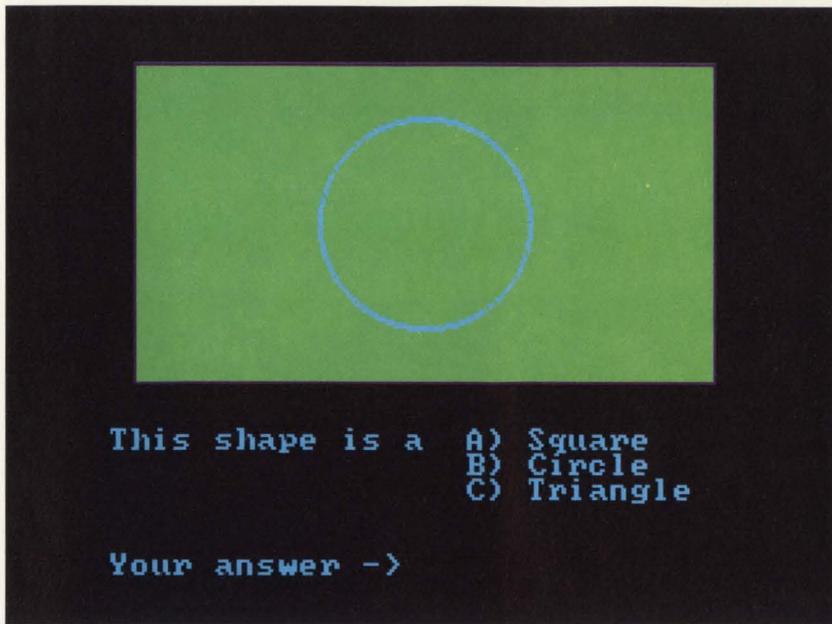


Figure 1. Screen display of a child's geometry tutor program written in Turbo Basic. The **VIEW** statement was used to define the rectangular viewpoint, where all graphics output appears. The remainder of the screen is used for text.

The coordinates **X1,Y1** and **X2,Y2** give the coordinates of the top left and bottom right corners of the viewport. These coordinates must be given in real screen coordinates, even if a **WINDOW** statement is in effect. I'll discuss screen coordinates in more depth shortly.

color and **boundary** are optional numerical arguments. If a **color** argument is included, the viewport is filled with that color and the previous contents of that screen region are erased. If no **color** argument is included, the previous contents of the viewport area are preserved. Including a **boundary** argument causes a border to be drawn around the viewport in the specified color.

The optional **SCREEN** keyword determines the reference point for coordinates within the viewport. If **SCREEN** is included, pixel 0,0 remains at the top left corner of the *screen*; if **SCREEN** is omitted, pixel 0,0 is at the top left corner of the *viewport*. Thus, omitting the **SCREEN** keyword has the effect of adding **X1,Y1** to any coordinates used in future graphics operations within the viewport.

Let's take a look at how this works. I wrote a simple program that defines a viewport with coordinates (160,60)-(600,180), and then draws a circle whose center is at (260,60). Figure 2 illustrates the different results obtained by including or omitting the **SCREEN** keyword in the **VIEW** statement.

The **VIEW** statement without arguments defines the entire screen as the viewport. This has the effect of returning things to the way they were before any viewports were defined. Using the **SCREEN** statement (*not* the **VIEW** statement's **SCREEN** keyword!) to change screen modes also cancels any **VIEW** setting.

CLIPPING

No, I haven't suddenly switched to talking about football penalties! Clipping is what happens to graphic output that falls outside the boundaries of

the active viewport: it is cut off, or *clipped*, and does not appear. Clipping applies to partial objects as well as to entire objects. For example, if you execute a **CIRCLE** statement, only that part of the circle that falls within the viewport appears. This could be the entire circle, part of the circle, or none of it at all.

TEXT AND VIEWPORTS

Often you'll want to put both text and graphics on the same screen. How does text output behave when a viewport is active? Just as it always does—as far as text is concerned, viewports don't exist. Text appears outside or inside a viewport, and can cross over the viewport boundary. Viewport clipping is not performed on text. If you want to display text on a graphics screen with an active viewport, it's up to you to insure that the text does not impinge on the viewport.

MULTIPLE VIEWPORTS

You can have as many viewports on the screen as you like, although only the most recently defined viewport receives graphic output. When you deactivate a viewport (by activating another viewport), the deactivated viewport's contents remain on the screen unless explicitly cleared.

CLEARING THE SCREEN

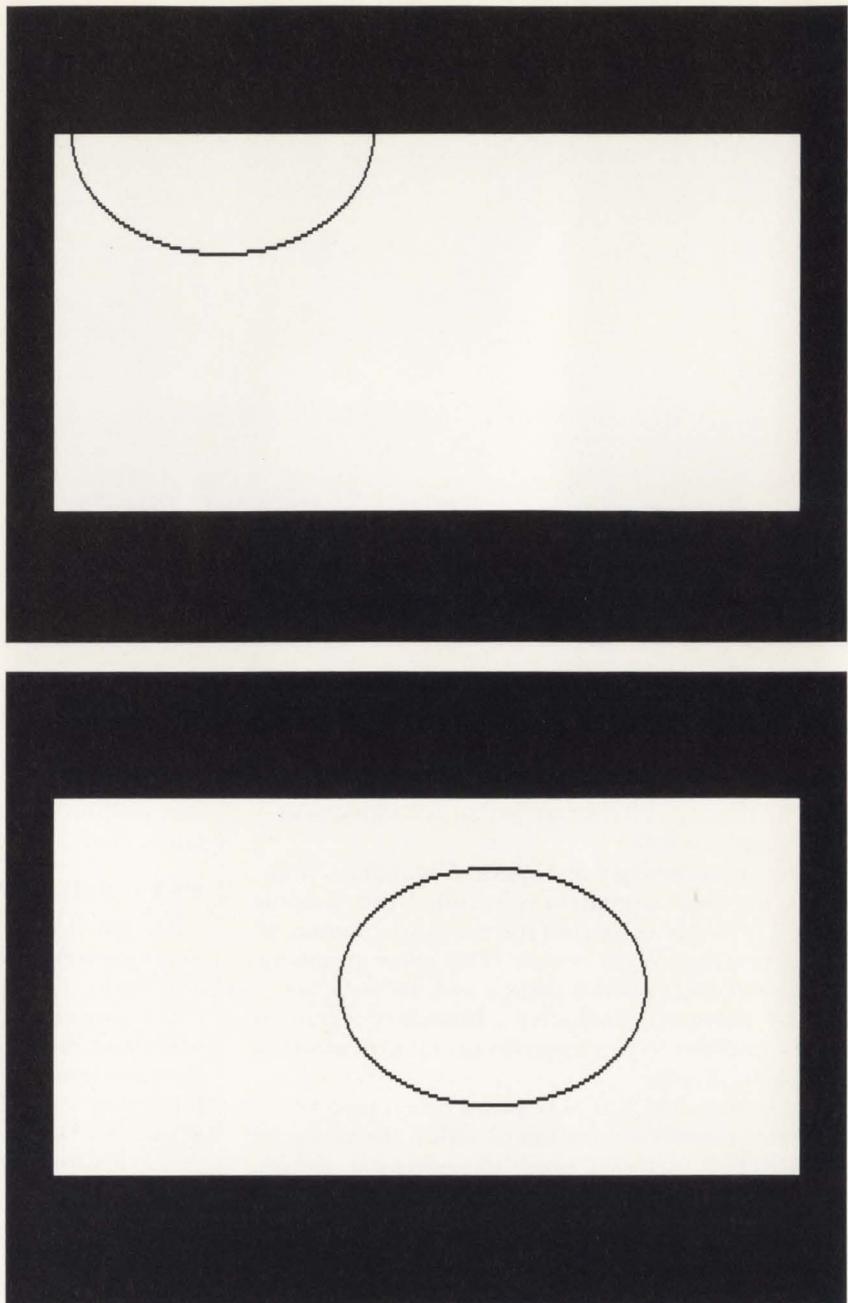
If a viewport is active, the **CLS** statement clears only the defined viewport, leaving the remainder of the screen unchanged. When it's first established, a viewport is cleared only if a background color was specified in the **VIEW** statement. To clear the entire screen, first use the **VIEW** statement without arguments to turn off the viewport.

GET AND PUT

What about using the **GET** and **PUT** statements with viewports? Both of these statements, which are used for screen animation and other interesting graphics

continued on page 116

Figure 2. The effects of including or omitting the **SCREEN** keyword in the **VIEW** statement. Both screens resulted from defining a viewport with corners at (160,60)-(600,180), and then drawing a circle whose center was at (260,60). In the first screen, the **SCREEN** keyword was included; in the second screen, it was not. In the second screen, note how the circle is shifted to the right and downward by the same amount of pixels that the viewport is shifted from the upper left corner of the screen.



VIEWPORTS

continued from page 115

effects, can be used with viewports—but with limitations. Briefly, **GET** copies a portion of the screen into an array in memory. **PUT** does the reverse; it copies graphics data from a memory array to the screen. However, when used with viewports, the screen region addressed by a **GET** or **PUT** statement must be entirely *within* the active viewport. If the region being copied to or from crosses a viewport boundary, an Illegal Function Call error results. (If you're not familiar with **GET** and **PUT**, refer to the *Turbo Basic Owner's Handbook*.)

REDEFINING SCREEN COORDINATES

What if you don't like the normal screen physical coordinate system of 640×200 (or 350 with the

EGA) pixels, with Y values increasing downward? The Turbo Basic **WINDOW** statement lets you redefine the coordinate system to anything you like. Such a custom coordinate system consists of *logical coordinates* that are remapped to your display's unchanging physical coordinates by Turbo Basic's runtime code. **WINDOW**'s syntax is:

```
WINDOW [[SCREEN] (X1,Y1)-(X2,Y2)]
```

X1,Y1 are the new coordinates of the lower left corner of the screen, and **X2,Y2** are the new coordinates of the upper right corner. If the **SCREEN** keyword is included, the Y coordinate system retains the default characteristic that larger Y values represent lower positions on the screen.

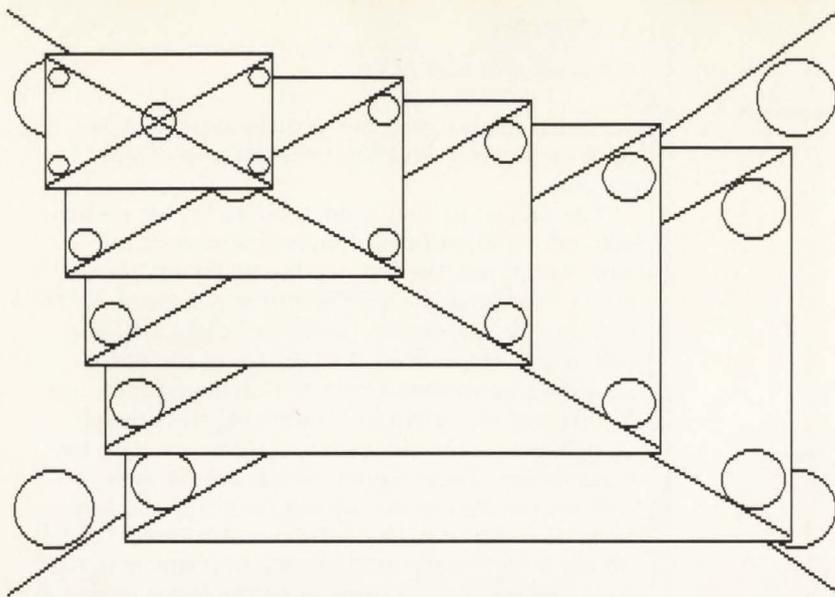


Figure 3. Output of *VIEW-DEMO.BAS*, which establishes a scaled viewport using the subroutine *ScaledViewport*. A graphics image is first drawn to the entire screen, then to a series of smaller viewports. Note how the full screen image is scaled to fit within each viewport.

Here's an example. After executing the statement **WINDOW (0,0)-(1000,1000)**, the statement **PSET (0,0)** illuminates the pixel in the lower left corner of the screen (at physical coordinates 0,199), **PSET (1000,1000)** illuminates the pixel in the upper right corner (at physical coordinates 639,0), and **PSET (500,500)** illuminates the pixel in the center of the screen (at physical coordinates 320,100). The **WINDOW** statement is extremely useful for customizing the screen coordinates to suit your needs.

As mentioned earlier, the coordinates of a viewport must be specified in physical screen coordinates even when a **WINDOW** statement is in effect. Plotting within a viewport, however, makes use of the coordinate system established by a **WINDOW** statement.

It's important to remember that the **WINDOW** statement does not change the physical resolution of the screen. After **WINDOW (0,0)-(1000,1000)**, the screen does not suddenly have 1000 pixels vertically and horizontally. When a window statement is active, Turbo Basic translates the window's logical coordinates into physical coordinates every time a drawing operation is performed.

SCALING VIEWPORTS

Why am I bringing up the **WINDOW** statement in an article on viewports? Although there is no direct connection between the two statements, proper use of the **WINDOW** statement can greatly enhance the use of viewports.

When used by itself, a **VIEW** statement sets up a viewport that shows only a portion of what would be drawn on the full screen. Anything that falls outside of the boundary of the viewport is clipped and therefore not seen. When the **WINDOW** statement is used appropriately, however, a viewport of any size or location can become a miniature screen. The entire screen graphics image (as it would have appeared *without* a **VIEW** statement) is shrunk and scaled to

fit entirely within the viewport. How do we do this?

Conceptually, it's quite simple. Normal full-screen graphics operations use coordinates in the range 0,0 through 639,199. All we need to do in order to "shrink" the full screen into a viewport is use the **WINDOW** statement to establish a coordinate system

continued on page 118

	EASY	FAST	PROFESSIONAL
DATA ENTRY	GW Basic	Basic 86	BASICA
WINDOWS		Quick Basic	Turbo C
MENUS			
HELP	MS Cobol	MS C	Turbo Basic
Lattice C	If you are serious about programming PLEASE try HI-SCREEN XL!		
Quick C			
Instant C			
	HI-SCREEN XL™		
\$149 only		C Basic	dBXL
Multilanguage support		RM-Cobol	Fox Base
No Royalties		Realia Cobol	Turbo Pascal
30-day risk free			
Mark Williams C	Call now for demo and information: 1-800-338-2852 in CA: (415) 397-4666		
RM-Fortran			
Microfocus Cobol			
MS MASM	dBase II, III, and III+	Turbo Prolog	
MS Pascal	MS Fortran	MS Basic	Quicksilver
"You may like other screen management tools, but you will love HI-SCREEN XL."			

Softway, Inc., 500 Sutter St., Suite 222, San Francisco, CA 94102

```

'Turbo BASIC program VIEWDEMO: demonstrates use of
'subroutine ScaledViewport to create scaled graphics viewports

screen 2

call DrawStuff
delay 1

for i=6 to 2 step -1
  call ScaledViewport(i*15,i*8,i*100,i*30)
  call DrawStuff
  delay 1
next i

while not instat : wend
end

SUB ScaledViewport(X1,Y1,X2,Y2)

'sets up a viewport with corners at X1,Y1 and X2,Y2. Full
'screen graphics output will be scaled to fit the viewport

  xscale = 640 / (X2-X1)
  yscale = 200 / (Y2-Y1)
  xoffset = -(X1 * xscale)
  yoffset = -(Y1 * yscale)
  xmax = (640 * xscale) + xoffset
  ymax = (200 * yscale) + yoffset
  WINDOW SCREEN (xoffset,yoffset) - (xmax,ymax)
  VIEW SCREEN (X1,Y1) - (X2,Y2),,1
  CLS

END SUB

SUB DrawStuff
  line (0,0)-(639,199)
  line (0,199)-(639,0)
  circle (320,100),50,1
  circle (35,30),30,1
  circle (605,170),30,1
  circle (605,30),30,1
  circle (35,170),30,1
END SUB

```

VIEWPORT

continued from page 117

such that the viewport has coordinates 0,0 at its upper left corner and 639,199 at its lower right corner.

The procedure may seem a bit tricky, but it's actually quite straightforward once you understand what's required. We first need scale factors for the X and Y coordinates. These factors are determined by dividing the screen size (width or height, 640 and 200 or 350, respectively) by the size of the corresponding viewport dimension. Calculate the X and Y offsets of the origin by multiplying the physical coordinates of the viewport's top left corner by the scale factors. These values, which are the new window coordinates of the top left corner of the screen, must be negative so that window coordinates 0,0 fall on the screen at the top left corner of the viewport. Next, the window coordinates of the lower right corner of the screen must be calculated. Finally, **WINDOW** and **VIEW** statements are executed to establish the coordinate system and the viewport.

Listing 1 provides a real example of how to create a viewport that is scaled to the full-screen coordinate system. The subroutine **ScaledViewport** does all of the interesting work. For demonstration purposes, **ScaledViewport** is embedded in a brief demonstration program that first draws a graphics image to the full screen, then draws the image to progressively smaller scaled viewports. The program's screen output is shown in Figure 3.

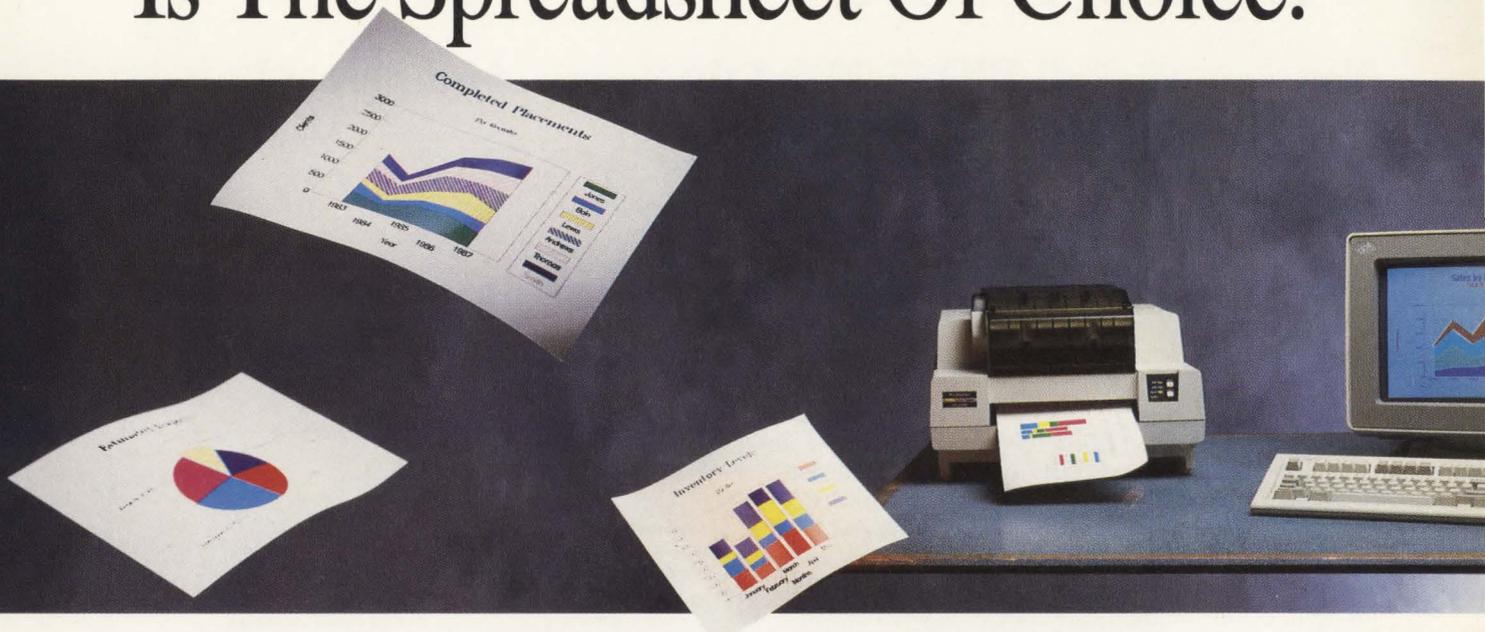
ScaledViewport accepts the coordinates of the desired viewport as arguments. It then performs the needed calculations, sets up the coordinate system, and establishes and clears the viewport. As written, the subroutine does *not* perform bounds checking on viewport dimensions, nor does it perform any other error checking.

To ensure that you can understand what **ScaledViewport** is doing, this subroutine does only what is necessary in order to create the scaled viewport. It could certainly be modified to be more general and more reliable by making certain additions. The vertical screen dimension is currently "hard-coded" at 200 pixels. A better method, however, would be to pass a parameter containing the vertical screen dimension, which may also be 350 pixels (on the EGA) or 480 pixels (with the VGA). It would also be a good idea to perform error checking on the coordinates that are passed as parameters to ensure that meaningful values are always passed. With a little additional work, **ScaledViewport** could become an important addition to your Turbo Basic graphics toolkit. ■

Peter Aitken is an assistant professor at Duke University Medical Center, and is the author of DigScope, a scientific software package. He writes and consults in the microcomputer field.

Listings may be downloaded from CompuServe as VIEWPT.ARC.

It's Easy To See Why Quattro Is The Spreadsheet Of Choice!



In fact, it's hard *not* to see. Because one look at Quattro® shows you a lot more for your money. More speed, more power, and the most spectacular presentation-quality graphics anywhere—built in.

Dazzling and diverse

If you went out looking, you'd be hard pressed to find spreadsheet graphics as dazzling and diverse as Quattro's. If you did, they'd be in a separate standalone package with a separate standalone price. And they still wouldn't be integrated with your spreadsheet's menu commands the way Quattro's are.

Brilliance built in

Quattro lets you choose from 10 different types of presentation-quality graphs and a huge selection of fonts, fill patterns and colors.

Quattro supports PostScript® too. So you can use today's most popular laser printers and typesetters to make your work—and yourself—look positively brilliant.

Hard copy made easy

Quattro makes it easy to get hard copies of your graphics—with a printer or plotter, directly from the spreadsheet. In fact, you don't even have to leave the spreadsheet.

Seeing is believing!

Dazzling graphics are just one of Quattro's eye-opening features; your dealer can show you the others. Quattro is easy to use and fully compatible; it even accepts familiar 1-2-3® compatible commands and uses data files created with other spreadsheets and databases. But Quattro gives you a lot more—in fact, twice the speed and power of the old standard. For only half the price.

60-Day Money-back Guarantee*

For the dealer nearest you call (800) 543-7543

“ Quattro contains the most comprehensive presentation graphics capability available in a spreadsheet . . . The graphs Quattro can produce surpass even those available through add-on products like Lotus Graphwriter or Freelance Plus. If Borland wanted to, it could certainly sell the graphics portion of the spreadsheet on its own merit as a standalone graphics application.

Robert Alonzo, Personal Computing

Quattro's presentation-quality graphics output capabilities rival those that 1-2-3 can obtain only in conjunction with separate presentation graphics software . . . For me, at least, Quattro has certainly become the character-oriented spreadsheet program of choice.

William Zachmann, Computerworld

In the few years since Lotus Development Corp. introduced 1-2-3, many companies have attempted to unseat the king of the spreadsheet hill. The latest contender, Borland International Inc.'s Quattro, succeeds where other spreadsheet packages have failed . . . Quattro is at least two steps ahead of 1-2-3.

Ricardo Birmele, PCResource ”

*Customer satisfaction is our main concern; if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Lotus and 1-2-3 are registered trademarks of Lotus Development Corp. Other brand and product names are trademarks of their respective holders. Copyright ©1988 Borland International, Inc. BI 1236A



CALLING BIOS SERVICES FROM TURBO BASIC

Now you can call BIOS without the agony of interpreted BASIC—just CALL INTERRUPT.

Ethan Winer



WIZARD

In the past, BIOS services could only be accessed from BASIC by using assembly language. This process required a knowledge of all the various BIOS functions and how to call them, plus a great deal of tedious coding. Worse, the only way that a BASIC program could communicate with an assembler routine was by passing variable addresses on the stack—so the programmer needed a solid understanding of both assembly language and the way that BASIC stores variables internally.

With the advent of Turbo Basic, the need to use assembler for BIOS calls is behind us—Turbo Basic calls interrupts directly. Even locating variables can now be performed entirely within Turbo Basic.

ENTER INTERRUPTS

How can a program call on BIOS routines if it doesn't know where to find them? The very first few bytes of the PC's memory map contain a table of addresses, called the *interrupt vector table*. These addresses point to interrupts. The first four bytes in the interrupt vector table hold the segment and address for interrupt 0, the next four bytes in the table point to interrupt 1, and so forth. Thus, a program can use the number of an interrupt to find that interrupt's service routine in memory.

The Intel 86 family of microprocessors has a software interrupt instruction (INT) for accessing routines through the interrupt vector table in low memory. Whenever the CPU encounters an instruction such as INT 10H, it goes to the interrupt table, obtains the appropriate address, and calls the routine automatically.

REGISTERS

BIOS services are grouped by function. For example, screen-oriented ("video") services fall under interrupt 10H. To identify the particular INT 10H service that is needed, a service number is first placed into the pro-

LISTING 1: SCROLLUP.BAS

```

***** ScrollUp.Bas
UL.Row = 5 : UL.Col = 5           'specify the corners
LR.Row = 18 : LR.Col = 74

%AX = 1 : %BX = 2                 'define the registers
%CX = 3 : %DX = 4                 ' as named constants

CLS
FOR X = 1 TO 24
  PRINT STRING$(80, X + 64);      'print a test pattern
NEXT

INPUT; "How many rows to scroll? (0 to clear) ", Rows

REG %AX, Rows + 256 * 6           'rows in AL, service 6 in AH
REG %BX, 7 * 256                  'color in BH
REG %CX, UL.Col + 256 * UL.Row    'UL.Col in CL, UL.Row in CH
REG %DX, LR.Col + 256 * LR.Row    'LR.Col in DL, LR.Row in DH

CALL INTERRUPT &H10              'call BIOS to do it

```

cessor's AH register. Other information required by the interrupt may be passed in additional registers.

A *register* is simply a memory location inside the microprocessor. You could think of registers as being the 8088's built-in "local variables." An understanding of the difference between the various registers is helpful when using Turbo Basic's **CALL INTERRUPT** instruction. The 8088 has 11 registers; we'll take a brief look at each of them.

General purpose registers. The four general purpose registers, called AX, BX, CX, and DX, are capable of holding operands for the simplest instructions (such as addition and subtraction). Each general purpose register also has its own specialty. For example, AX is the only general purpose register that can be multiplied.

Index registers. Two other registers, called SI (Source Index) and DI (Destination Index), can also be used for simple addition and subtraction. However, the real purpose of these two registers is to perform *indexing*—to point to an address that contains data.

Segment registers. Four segment registers, called CS, DS, ES, and SS, hold the current code, data, and stack segments, plus a spare or "extra" segment.

REGISTER ORGANIZATION

Before moving on to the actual interrupts, we must first discuss the way that some of these registers are organized. The four general purpose registers—AX, BX, CX, and DX—are all capable of holding a single 16-bit word. However, each of these registers can also be considered as two separate 8-bit registers.

When a general purpose register is treated as two separate 8-bit registers, each byte in the register can be accessed independently. For example, the high-byte portion of AX is referred to as AH, while the low-byte part of CX is called CL. This approach is important to understand, because Turbo Basic does not allow the individual 8-bit portions of each register to be set or read. Therefore, if a particular DOS service requires the AH register to be loaded first with a number, that number must be multiplied by 256 (which, in effect, shifts the 8-bit quantity "up" into AH), and then the number is loaded into AX. To load two separate quantities into AH and AL, the quantity that is intended for AH must first be multiplied by 256. Next, the resulting number must be *added* to the quantity that is intended for AL, and then the final sum is put into AX.

Registers are loaded and read with Turbo Basic's **REG** statement, which can be used either as a statement or as a function. Notice that **REG** doesn't really operate on the processor's registers; rather, **REG** reads or writes its parameters into a special area of memory. When **CALL INTERRUPT** is used, those values are transferred to or from each machine register.

TO CALL AN INTERRUPT

Interrupt 5, which performs a Print Screen, is the simplest PC interrupt. Unlike the other BIOS interrupts, interrupt 5 needs no additional setup or preloading of any registers. To access interrupt 5, issue the following call:

```
CALL INTERRUPT 5
```

That's all there is to it! This call simply accesses the code that is already built into ROM; the BIOS does the

real work. If the GRAPHICS.COM utility (which is shipped with DOS) is already loaded, then that utility is called instead of the ROM-based routine. Remember that whenever an interrupt is invoked, the 8088 looks at low memory for the address that contains the actual service routine. Since GRAPHICS.COM places its own address into RAM, interrupt 5 is routed automatically to the RAM-resident routine, rather than to the original code that is stored in the BIOS ROM.

SCROLL 'EM

Although BASIC has generally provided more features than any other compiled language, there are still several BIOS services that even Turbo Basic doesn't perform well or at all. An example of such a service is the process of quickly clearing or scrolling rectangular subsets of the text display screen.

BIOS handles the process of clearing and scrolling screen regions with two separate routines: Service 6 scrolls a region up, and service 7 scrolls a region down. When these routines are called, a number of parameters must be specified that indicate the upper left and lower right corners of the screen region, the number of lines to be scrolled, and an attribute to which the blanked lines, or lines within the region, will be set. Note that a screen region can be cleared by setting the number of scrolled lines in service 6 to zero.

SCROLLUP.BAS (Listing 1) provides a small demo program that scrolls up or clears a specified number of rows. Notice that all of the BIOS video routines assume that rows are numbered from 0 to 24, and that columns are numbered from 0 to 79. Also notice the use of Turbo Basic's named constants (which are preceded by %) to provide more meaningful names and to make register identification easier.

When the upper and lower halves of each register are loaded, an extra step is necessary. Since Turbo Basic doesn't provide direct access to each register half, the register halves must be loaded by the multiplication and addition steps shown in SCROLLUP. Note that a color or attribute parameter can be placed in the BH register to select the background. Service 7 of interrupt 10H operates similarly to service 6; with both services, all parameters are passed in the same registers and carry the same meaning. The only difference between service 6 and service 7 is that the latter scrolls downward, rather than upward.

WORK SMART

There are many other useful BIOS services, all of which may be accessed through **CALL INTERRUPT**. Keep in mind, however, that most BIOS functions (such as reading the cursor position, writing an individual pixel dot, and so forth) can be accessed more easily with Turbo Basic's built-in routines. Once you understand the power of BIOS and Turbo Basic, **CALL INTERRUPT** will let you work as hard as you need to—without working harder than you have to. ■

Ethan Winer owns Crescent Software, and is the author of the QuickPak utilities for Turbo Basic and Microsoft QuickBASIC.

Listings may be downloaded from CompuServe as SCROLL.ARC.

DATE FORMATTING WITH SPRINT

Make a date with Sprint to explore the basics of building a "soft" user interface.

Neil Rubenking

 Behind Sprint's "soft interface" lies a powerful programming language. Sprint programs (called *macros*) can build a complete custom user interface (UI) that makes Sprint look and act just like another word processing program. That's no mean feat, considering the wide variation in features among current word processors. Here's how Sprint handles one particular feature of its WordPerfect UI.

PROGRAMMER

HOW ABOUT A DATE?

In WordPerfect, pressing Shift-F5 and "1" inserts the current date into the text. Pressing Shift-F5 and "2" allows you to edit the date format. Codes that control the format are contained in a special format string; each numeric character from "0" to "9" selects a different form of the time and date. For example, the default format string "3 1, 4" specifies the form "July 4, 1999." A percent character in the string forces a subsequent number to two digits, and adds a leading zero if necessary. If the default string is changed to "3 %1, 4," the date appears in the form "July 04, 1999."

In order to reproduce this WordPerfect feature, Sprint must both edit and interpret the format string. Since editing the string is easier than interpreting it, we'll look at the editing process first.

TO EDIT THE STRING

Sprint stores the format string in Q-register G (QG). This one-line statement *could* perform the edit:

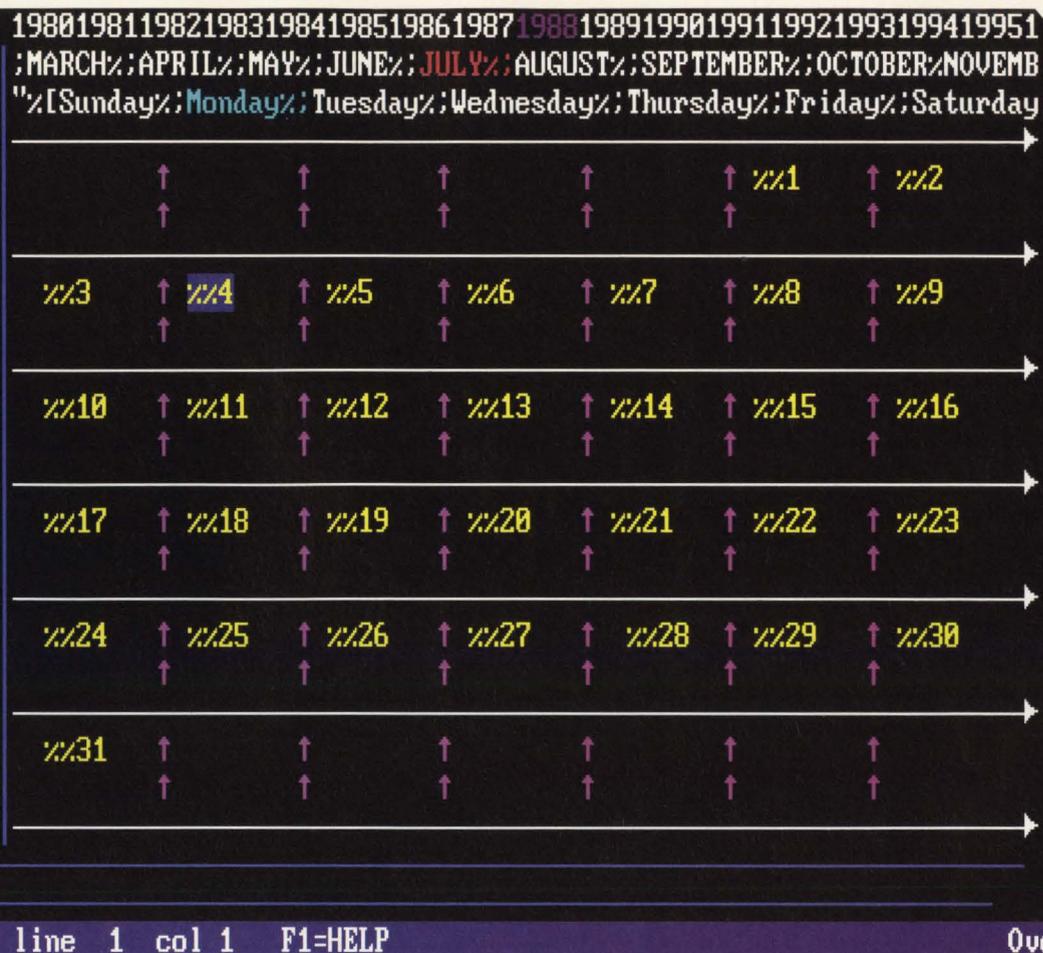
```
message "Date Format: " set QG
```

This command displays the words "Date Format:" on the status line, and makes the current contents of QG available for editing. However, we can't reasonably ask the user to haul out the manual for the list of legal codes—we need to display the available options on the screen.

The macro **DateFormat** (Figure 1) uses an "infobox" to display the codes while the user enters the new date format. An **infobox** is a noninteractive structure that is otherwise similar to a Sprint menu, and consists of a box that is just big enough to hold all of the informa-

LISTING 1: FULLDATE.SPM

```
InsertDate : ; Take date format in QG and turn into today's date
int x
0 -> posn
set Q0 "%d"
do (
  posn subchar QG -> x
  ++posn
  x case (
    0 break,
    '1' (time 3 put q0) (set q0 "%d"),
    '2' (time 4 put q0) (set q0 "%d"),
    '3' MonthName (set q0 "%d"),
    '4' (1900+time 5 put) (set q0 "%d"),
    '5' (time 5 put) (set q0 "%d"),
    '6' DayOfWeek (set q0 "%d"),
    '7' (time 2 put q0) (set q0 "%d"),
    '8' (time 2->hours ? ((hours >12)? hours-12 : hours)
      : 12 put q0) (set q0 "%d:"),
    '9' (time 1 put q0) (set q0 "%d"),
    '0' (if time 2<12 "am" else "pm") (set q0 "%d"),
    % set Q0 "%02d",
    $ (x insert) (set Q0 "%d") ;other char, pass THROUGH
  )
)
```



```

DateFormat :
  infobox "Date Format" {
    "Character  Meaning",
    " 1      Day of the month",
    " 2      Month (number)",
    " 3      Month (word)",
    " 4      Year (all four digits)",
    " 5      Year (last two digits)",
    " 6      Day of the week (word)",
    " 7      Hour (24-hour clock)",
    " 8      Hour (12-hour clock)",
    " 9      Minute",
    " 0      am / pm",
    " %%     Include leading zero for numbers less",
    "        than 10 (must directly precede number)",
    "Examples:",
    " 3 1, 4    = December 25, 1984",
    " %%2/%%1/5 (6) = 01/01/85 (Tuesday)"
  } (message "Date Format: " set QG)

```

Figure 1. This Sprint infobox explains the time and date formatting options available in the WordPerfect UI.

tion lines. The title line displays at the top of the box. When the macro that follows the infobox finishes execution, the box disappears.

CALL MY INTERPRETER

As described above, the process of editing the format string is relatively easy. The difficult task is *interpreting* the string into the appropriate date form. We'll use Sprint's macro **time**, which returns different portions of the current date and time. Table 1 shows the output of the **time** macro for different inputs.

Input	Output
0	seconds (0-59)
1	minutes (0-59)
2	hours (0-23)
3	days (1-31)
4	months (1-12)
5	year minus 1900 (0-199)
6	day of week (Sunday = 0)

Table 1. Numeric formatting values returned by Sprint's **time** macro.

continued on page 124

continued from page 123

Note that most of **time**'s outputs in Table 1 match selections in Figure 1. However, **time** does not provide names for the months and days of the week. Figure 2 lists a pair of macros that make up for the missing names. *Important note:* The entire "[...]" construct must occur on one line. The **MonthName** structure had to be broken onto three lines in order to fit on a magazine page, and the **DayofWeek** structure had to be broken onto two lines.

```
DayofWeek : ;Enter day of week at cursor
time 6 put
"%[Sunday%;Monday%;Tuesday%;Wednesday%;Thursday%;
Friday%;Saturday%]"
```

```
MonthName : ; Enter Month at cursor
time 4 -1 put
"%[January%;February%;March%;April%;May%;June%;
July%;August%;September%;October%;November%;
December%]"
```

Figure 2. These two Sprint macros return names for the month and day of the week. (Note that selection statements must exist on one line, even though both were broken to fit on the magazine page.)

THE WARMUP

Rather than attempting to interpret the format all at once, we'll start with a simplified version. Macro **SimpleDate** in Figure 3 handles all of the code characters except the "%%." **SimpleDate** does not allow you to force numeric fields to two digits.

The major structures in the **SimpleDate** macro are the **do** loop and the **case** statement within that loop. The **do** loop checks each character of the format string stored in **QG**. The **case** statement acts on those characters.

The **case** statement contains three main alternatives: Variable **x** may contain one of the code characters, zero, or neither. When **x** contains a code character, the macro interprets that code character. When **x** contains a zero, the end of the format string has been reached, so the program breaks out of the loop. If **x** is any other character, then it's passed directly into the output text.

Option 8 may be a bit hard to understand due to the use of the abbreviation characters "?" and ":" for **IF** and **ELSE**. Here is that same line with the words spelled out:

```
IF time 2->hours {
  IF hours > 12 hours-12
  ELSE hours
} ELSE 12 put "%d"
```

```
SimpleDate : ; Turn date format in QG to today's date
int x
0 -> posn
do {
  posn subchar QG -> x
  ++posn
  x case {
    0 break,
    '1' (time 3 put "%d"),
    '2' (time 4 put "%d"),
    '3' MonthName,
    '4' (1900+time 5 put),
    '5' (time 5 put),
    '6' DayOfWeek,
    '7' (time 2 put "%d"),
    '8' (time 2->hours ?
        ((hours >12)? hours-12 : hours)
        : 12 put "%d"),
    '9' (time 1 put "%d"),
    '0' (if time 2<12 "am" else "pm"),
    $ (x insert) ; other char, pass it THROUGH
  }
}
```

Figure 3. The simplified date formatting macro **SimpleDate**.

Option 8 examines the hours value and then takes one of several steps: If the hours value is both nonzero and less than 12, it's used without being changed. If the hours value is greater than 12, 12 is subtracted from that value. If the hours value is exactly zero, then the value of 12 is used.

THE MAIN EVENT

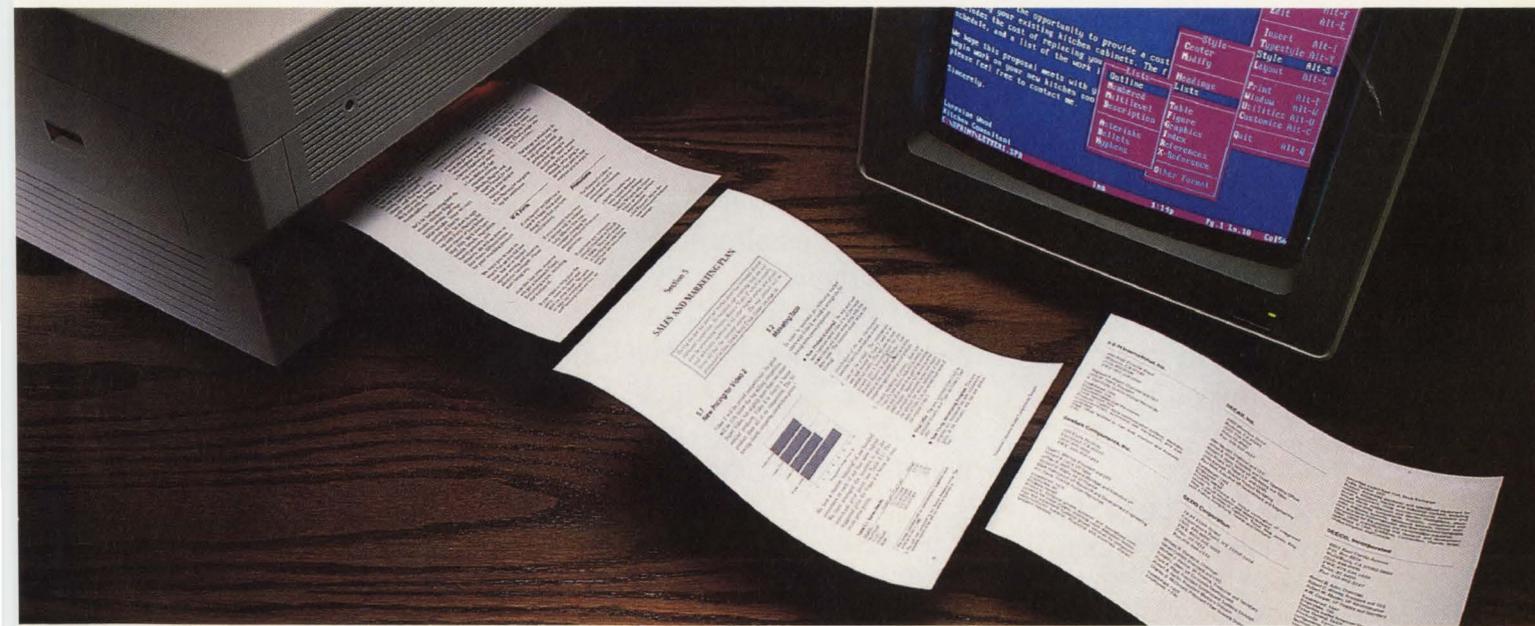
The actual **InsertDate** macro (Listing 1) is just **SimpleDate** with some added formatting. A Sprint output format string is stored in **Q**-register 0. If two digits are being forced, "%%02d" is used; otherwise, "%d" is used. Because the code character "%%" affects an immediately subsequent numeric value, **Q0** is set back to "%d" after any other character. With that step, **InsertDate** is complete.

Each word processing program takes its own unique approach to the tasks of writing and editing. With the powerful macro language in Sprint, these different approaches can be easily emulated. If you want to learn Sprint programming, study the alternate User Interface files, copy them to distinct names, and modify them to your own purposes. Soon you'll be ready to write your own ultimate UI. As Sprint's philosophy says, "If a feature you need doesn't exist—build it!" ■

Neil Rubenking is a professional Pascal programmer and writer. He is a contributing editor for PC Magazine, and can be found daily on Borland's CompuServe Forum answering Turbo Pascal questions.

Listings may be downloaded from CompuServe as SPDATE.ARC.

Introducing *Sprint*— the professional, programmable word processor!



**SPECIAL OFFER:
ONLY \$99.95!**



Why walk when



You can work on up to 24 files at once, divide your screen into as many as six windows, and never miss a beat because Sprint remembers which files you were working on last.

Because Sprint brings you the speed you're used to with Turbo Pascal® and Turbo C,® it never wastes your time and true *Turbo*-performance is finally available in a text editor.

To see just how much faster Sprint works for you, check out the comparative time tests.

Sprint gives you six optional interfaces including EMACS

The customizing you choose to do is one variation on Sprint's theme and there are six others.

We give you free (for a limited time) Alternative User Interfaces for:

- EMACS
 - WordPerfect®
 - WordStar®
 - MultiMate®
 - Microsoft®
 - SideKick®
- Word

And you also get file conversions for:

- WordStar
- Microsoft Word
- WordPerfect
- MultiMate
- DisplayWrite® 4 (DCA RFT)

The race into the Age of Customization is on—led by Sprint.® You can use Sprint *as is* and be very happy with the way everything works for you—or you can easily customize Sprint to do everything *your way*.

It's a completely *customizable* word processor that, for example, lets you re-define keys, delete menu items, make your own shortcuts, invent your own menus, and use Sprint's online facility to create your own quick reference cards.

You're given the customizing power to avoid pop-up menus altogether—if that's the way you like to work. Sprint can be completely function-key-driven, and while Sprint's function key assignments are logically defined, they're easy to alter.

Nothing goes slow when you *Sprint!*

Sprint is fast. It scrolls fast, edits fast, switches between files fast, offers fast shortcuts and proves that the slow way is no way.

*Customer satisfaction is our main concern: if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks of their respective holders.
Copyright ©1988 Borland International, Inc.

BI 1267

Prices and specifications subject to change without notice.

Minimum System Requirements:

For the IBM PS/2 and the IBM family of personal computers and all 100% compatibles. Requires PC-DOS (MS-DOS®) 2.0 or later, 256K memory (384K recommended), and two floppy drives or a hard disk.

you can Sprint?

Sprint lets you use EGA and VGA cards for 43- or 50-line displays; it directly reads ASCII files without conversion and saves files with hard carriage returns for electronic mail.

You're given a built-in compiler with a syntax similar to C; separate source files; an extensive macro language; the ability to call DOS functions and much, much more.

"Auto-Save" means you'll never lose your work when you Sprint!

Forgetting to "Save" is a fact of life as are power outages, and it used to be that a power outage could wipe

See how fast you can *Sprint!*

	Save File ¹	Top to Bottom ²	Go To Line 1500	Search & Replace ³	Find Unique Word
<i>Sprint 1.0</i>	5.9	.1	.1	1.6	3.3
WordPerfect 4.2	41.1	5.3	5.4	6.6	6.2
WordStar 4.0	4.4	4.6	4.7	17.1	13.8
MS Word 4.0	9.7	.1	N/A	4.6	7.0

Tests were performed on a Multitech 286 AT (8 MHz), 640K RAM. ¹file size 103K. ²1636 lines. ³14 occurrences. Times shown are in seconds. (Benchmark details available upon request.)

out everything you've done. Not any more. Your work is always safe when you Sprint.

Sprint's "Auto-Save" automatically saves your words as you type, so if the lights do go out, you may be in deep dark-

ness but not deep trouble. Sprint's Auto-Save is more than "insurance," it's also invisible. You know it's there, but it does its job without interrupting yours.

SHERRY L. SMYTHE
222 Fountain Avenue
Ben Lomond, CA 95005
(408) 555-5555

PROFESSIONAL OBJECTIVE:

Position with a growing company utilizing my professional e

EMPLOYMENT HISTORY:

- November 1987 to Present: Customer Service Representative SOFTWARE SERVICES, INC. Santa Cruz, California. Responsible for handling priority correspondent and CEO of company. Responsible for all written and West German customers, as well as written and v with U.S. customers, involving product orders problem solving; working with management to custom databases and reports for company; knowledge of accounting packages and spreadsht
- October 1986 to November 1987: Receptionist/Office Manager DOCTOR SERVICES, INC. Santa Cruz, California. Responsibilities included, daily appoint emphasis on production, maintaining an posting accounts receivable charges and pa bank deposits, balancing month-end controls the receivables from a manual system to an in
- August 1985 to October 1986: Accounts Receivable Clerk HARDWOOD LUMBER COMPANY Santa Cruz, California. Responsibilities included, batching A/R i computer, reconciled monthly A/R stat invoices and handled customer inquiries. Posted daily cash receipts and prepared Assisted the EDP operator and performed as required.
- April 1981 to August 1985: Receptionist HEALTH-CARE OFFICE San Jose, California. Responsibilities included, daily appoi nance claims, posting charges and statements.

Stonewall Times

The Employee Newsletter of Stonewall Brokers, Inc.
May '88

We'll Be Havin' Some Fun

This year's summer party will be held on Cowell Beach, down by the Boardwalk, on Friday, June 10th. It will start at high noon. We will have two volleyball courts, loads of beach chairs, and food and drink until well into the evening. We'll end with a bonfire and marshmallows.

We'll be barbecuing beef rib steaks, chicken thighs, salmon steaks, and vegetable kabobs. Since we can't provide all four to everybody, be sure to sign up with Party Planning for your choice of food before June 1st. We'll also have salads, breads, vegetables, baked potatoes, and desserts, as well as three or four dozen different items for your snacking pleasure.

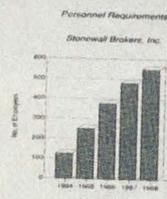
We want you to have as much fun as you did last year, but we've decided against serving and allowing alcoholic beverages. Please don't bring any.

Just like last year, everyone will get a Stonewall Towel. Everything is free, including the suntan oil.

If you want to help plan the party, come on down and give us your ideas. We need to sign up volleyball referees

Parking Problems

As you can see by the following chart, our little company isn't so little anymore...



Until the new parking structure is finished, we're going to continue having parking problems. If you can car pool with a friend, please do so (if you want names of people who live near you, contact Personnel). Whatever you do, don't take up two spaces for any reason. The visitor parking area is for visitors only. (That's people who don't work here.)

The garage is scheduled to be completed by June 1st. It will provide covered parking for 60 cars and uncovered parking for another 60. Since covered parking will be in such demand, we're going to devise a fair plan so that everyone gets to enjoy it.

Employees of the Month

Congratulations to the following Stonewall employees:

- Annette Christensen and Brad Dix for setting up the new computer system;
- Dennis Feldman for referring a new large client;
- Lora Mattos for her exquisite cooking;
- Bradley Hughes and Adam Vorwald for their record sales achievements; and
- Tom Stanley for reorganizing the warehouse.

Promotions

The President's Office is pleased and proud to announce the following promotions:

- Robert Schindler has been named Assistant Major Account Manager.
- Betty Willards will replace Robert as Senior Account Representative.
- Joy Flannery will be the new Information Systems Manager.

W-4 Form

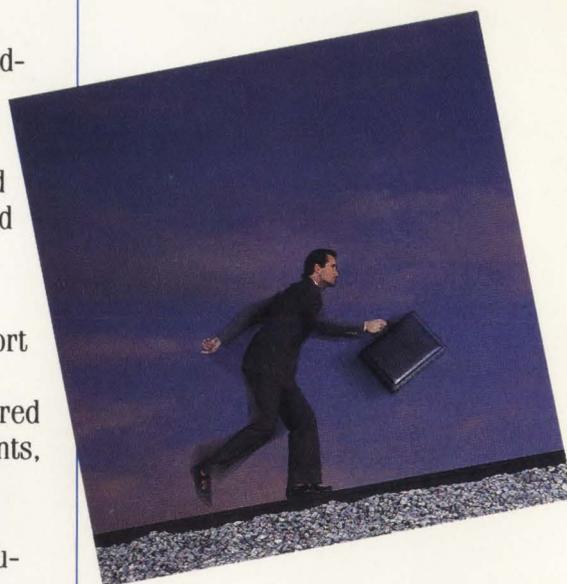
If you haven't an

You have a head start when you *Sprint!*

	Sprint 1.0	WordPerfect 4.2	MS Word 4.0	WordStar 4.0	MultiMate Adv. 1.0
Maximum file size	Disk	Disk	Disk	4MB	128K
Mail Merge	Yes	Yes	Yes	Yes	Yes
Thesaurus (integrated)	Yes	Yes	Yes	Yes	Yes
Windows Open (maximum)	6	1	8	1	1
Files Open (maximum)	24	2	8	1	1
Cross-Reference (dynamic)	Yes	No	No	No	No
Indexing Options	7	1	3	3	No
Snaking Columns (chg # on same page)	Yes	Yes	Not same pg.	No	Yes
Parallel Columns	Yes	Yes	Yes	Yes	Yes
H-P LaserJet Support	Full	Full	Full	Partial	Full
PostScript Support	Full	Text	Full	No	Text
Mouse Support (integrated)	Yes	No	Yes	No	No
AutoSave (without interruption)	Yes	No	No	No	No
User Interface					
Define Shortcuts Dynamically	Yes	No	No	No	No
Run Alternative User Interface	Yes	No	No	No	No
Verify spelling as you type	Yes	No	No	No	No
Fully programmable macro language	Yes	No	No	No	No
Suggested List Price	\$199.95	\$495.00	\$450.00	\$495.00	\$565.00

What you get when you Sprint!

- Includes Auto-Save that saves your work without interrupting it
- Sprint supports 350 popular printers including HP LaserJet,® other laser printers and typesetters plus has PostScript® support
- Supports multiple fonts, including downloadable fonts, in all sizes including scaled sizes
- Includes file conversions for Microsoft Word, WordPerfect, MultiMate, WordStar, and DisplayWrite 4 (DCA RFT)
- Includes Alternative User Interfaces for EMACS, SideKick, WordStar, WordPerfect, Microsoft Word, and MultiMate
- Comes with an integrated 100,000-word speller and 220,000-word thesaurus
- Produces highly professional output: long or short documents, cross-referencing, indexing, structured headings, tables of contents, word spacing, automatic kerning and ligatures as well as character substitution for items like typographer's quotation marks
- Can be used "as is," customized by you and/or you can use the Alternative User Interface you already know



60-Day Money-back Guarantee*

To order now,
Call (800) 543-7543

Special offer: Sprint for only \$99.95!

For registered Borland customers and for a limited time only (offer ends September 30, 1988), Sprint is all yours for only \$99.95[†] direct from Borland!

The suggested retail price for Sprint is \$199.95. We think \$100.00 off is the best way we can show our appreciation for your loyalty and support. (When you consider that many word processors are in

the \$500 to \$600 range, that \$99.95, including 6 alternative user interfaces, should start looking even better!)

Sprint works with today's hardware and will work with tomorrow's. Anywhere from an 8088 PC through a 386.

It's already a major success story in Europe; it's the #1 selling word processor in

France (and everyone knows, 50 million Frenchmen can't be wrong!)

Sprint. It's the word processor you'd expect from Borland: the value, technical excellence and programmability you'd expect from Borland. Sprint, for your eyes only, \$99.95.

[†]Plus shipping and handling

Chapter Heading

Scalable Font

Box Drawing

Section Heading

Automatic Bulleted List

PostScript® Graphics

Automatic Numbered List

Automatic Table-Referencing

Footnote Capability

Page Footing

Section 5

SALES AND MARKETING PLAN

During the last few years our market share has increased, despite increased competition. To maintain our widening lead, we will soon be announcing Oceanic Music's "Video 2" which is priced below all but the cheapest entries. The new product will be announced in New York's World Trade Center (see page 14).

5.1 New Pricing for Video 2

Video 2 will be priced competitively. Its price will be 10% below the top selling competition, Super Video, but slightly more than two other similar products. Video 2 is clearly a better product than all of its competition. The following charts compares competitive prices:¹

Product	Suggested Retail Price (\$)
Video Extra	79.95
Video Plus	89.95
Video 2	99.95
Super Video	109.95

We took a random sampling² of one hundred consumers in each of our four sales regions. We then averaged the numbers to get the maximum price point (see Table 5.1). The suggested price for Video 2 is below all maximum price points.

Region	Maximum Price Point	Margin
Northeast	\$21.99/10ape	\$ 9.50
South	\$23.49/10ape	11.00
Midwest	\$24.79/10ape	12.30
West	\$23.24/10ape	10.75

¹ The prices indicated are all manufacturer's suggested retail price as of Jan 1, 1988.
² The poll was conducted by our Market Research Group. The complete results of the poll are listed in Appendix A.

5.2 Marketing Tools

In order to maintain an increasing market share with Video 2, we need to arrange the following sales pieces/promotions:

- **New Product Collateral.** We will send new sales literature about Video 2 to all the major music chains and music stores along with free demo tapes. The literature should include the following:
 1. **Identification of the new improvements.** Describe Video 2's new features in detail.
 2. **Uses for the product.** The recommended uses and benefits of Video 2 need to be spelled out clearly. (This will also be incorporated into a featured piece for the *Monthly Bulletin to Music Chains*.) List benefits to the consumer as well as a price comparison against other similar products.
 3. **In-store marketing of product.** Show how Video 2's new display rack will fit neatly at the end of an aisle or as part of an existing shelf display. Dealers will be able to order the product on extended terms, and return the product for stock balancing if their sales of Video 2 do not exceed their sales of the original Video 1.
- **Trial Offer.** The new, improved Video 2 will be available initially as a "Take the Video 2 Test" offer.
- **New Co-op Advertising Program.** This new program will emphasize the new, collaborative policy of the company, and the new product line.

8

National Oceanic Music Corporation Report

BOUNCE AND CHOOSE IN PAL

Use Paradox's built-in menuing routines to create PAL menus that resemble Paradox menus.

Alan Zenreich

Paradox uses horizontal bounce bar menus that appear on the top two lines of the screen. I prefer to maintain this user interface in my own PAL applications because the menus don't overlap data, and the user needn't enter responses manually.

Listing 1 demonstrates three of Paradox's built-in procedures for creating bounce bar menus: **SHOWMENU**, **SHOWTABLES**, and **SHOWARRAY**. The **azSort()** procedure prompts the user to select a table from the current directory, pick a field to sort by, and choose the direction of sort. **azSort()** then creates a new, sorted table called **ANSWER**.

All of the **SHOW..** menus work similarly. The user can either move the cursor to the chosen item and press Enter, or else choose a menu item by pressing its first letter. The selection is assigned to the variable that follows the **TO** keyword. If the user presses Esc, the variable is assigned the string "Esc."

SHOWTABLES lists all of the tables for a given directory. It also searches memory for tables not yet written to disk, as well as for private tables.

SHOWARRAY uses two array variables. The first is a list of menu items; the second array variable is a list of item descriptions. **azSort()** scans a structure table and places the field names into the **zfield** array. A descriptive line for each item goes into **zdesc**.

SHOWMENU requires one or more lines; each line contains a menu item and a description in the form of two strings that are separated by a colon. Except for the last one, each item must have a comma at the end of the line to signify that another choice follows.

Obviously Listing 1 could be greatly enhanced. In its current form, however, this program demonstrates how easy it is to build menus that let the user select options in the comfortable Paradox style. ■

Alan Zenreich is a Paradox consultant and the publisher of *ParaLex*, the Paradox Documenter. He can be reached at Zenreich Systems, 78 Fifth Avenue, New York NY 10011.

Listings may be downloaded from CompuServe as PDXMNU.ARC.

LISTING 1: DEMOMENU.SC

```
PROC azTextLine(zline,ztext) ;-- centers a text at given line#
  @zline,0 ??FORMAT("W80,AC",ztext)
ENDPROC

PROC azSort()
;-- keep variables private to proc:
PRIVATE zsorttable,zsortfield,zsortorder,zfields,zdesc
CURSOR OFF ;-- turn off the cursor
CLEAR ;-- clear the screen
ClearAll ;-- remove any tables
IF IsTable("Answer") THEN
  DELETE "answer" ;-- don't want answer table in list
ENDIF
azTextLine(4,"Please select a table from the list above")
SHOWTABLES ;-- menu of available tables
DIRECTORY() "Tables available for sorting"
TO zsorttable ;-- get sort table
IF zsorttable="Esc" THEN
  RETURN false ;-- quit if user presses Esc
ENDIF
azTextLine(4,"Working, preparing list of fields for "+zsorttable)
(Tools) (Info) (Structure) SELECT zsorttable ;-- get field list
ARRAY zfields [nrows()] ;-- for menu selections
ARRAY zdesc [nrows()] ;-- for menu descriptions
SCAN ;-- scan the Struct table
  zfields [recno()]=[field name] ;-- assign field name choice
;-- assign field description:
  zdesc [recno()]="Sort table by "+zfields[!#]
ENDSCAN
DELETE "struct" ;-- remove struct table
azTextLine(4,"Select field to sort by")
SHOWARRAY ;-- menu lets user select from
zfields zdesc ;-- fields, with sort description
TO zsortfield ;-- get sort field
IF zsortfield="Esc" THEN
  RETURN false
ENDIF
@4,0 azTextLine(4,"Select the sort order desired")
SHOWMENU
"Ascending":"Sort in ascending order by "+zsortfield,
"Descending":"Sort in descending order by "+zsortfield
TO zsortorder ;-- get sort order
IF zsortorder="Esc" THEN
  RETURN false
ENDIF
azTextLine(4,"Preparing Sorted Answer table")
IF zsortorder="Ascending" THEN
  SORT zsorttable ON zsortfield TO "Answer" ;-- ascending sort
ELSE
  SORT zsorttable ON zsortfield D TO "Answer" ;-- descending sort
ENDIF
MOVETO FIELD zsortfield ;-- move to the answer table field
RETURN TRUE ;-- all done
ENDPROC

azsort() ;execute the proc (typically read in from a library)
```

C CODE FOR THE PC

source code, of course

	Bluestreak Plus Communications (two ports, programmer's interface, terminal emulation)	\$400
	CQL Query System (SQL retrievals plus windows)	\$325
	GraphiC 4.1 (high-resolution, DISSPLA-style scientific plots in color & hardcopy)	\$325
	Barcode Generator (specify Code 39 (alphanumeric), Interleaved 2 of 5 (numeric), or UPC)	\$300
NEW!	Vmem/C (virtual memory manager; least-recently used pager; dynamic expansion of swap file)	\$250
	Aspen Software PC Curses (System V compatible, extensive documentation)	\$250
	Greenleaf Data Windows (windows, menus, data entry, interactive form design)	\$250
	PforCe++ (COM, database, file, user interface, & CRT C++ classes among others)	\$345
	Vitamin C (MacWindows)	\$200
NEW!	TurboT _E X (TRIP certified; HP, PS, dot drivers; CM fonts; L _A T _E X)	\$170
	Essential resident C (TSRify C programs, DOS shared libraries)	\$165
	Essential C Utility Library (400 useful C functions)	\$160
	Essential Communications Library (C functions for RS-232-based communication systems)	\$160
	Greenleaf Communications Library (interrupt mode, modem control, XON-XOFF)	\$150
	Greenleaf Functions (296 useful C functions, all DOS services)	\$150
	OS/88 (U**x-like operating system, many tools, cross-development from MS-DOS)	\$150
	ME Version 2.0 (programmer's editor with C-like macro language by Magma Software; Version 1.31 still \$75)	\$140
	Turbo G Graphics Library (all popular adapters, hidden line removal)	\$135
	PC Curses Package (full Berkeley 4.3, menu and data entry examples)	\$120
	CBTree (B+tree ISAM driver, multiple variable-length keys)	\$115
	Minix Operating System (U**x-like operating system, includes manual)	\$105
	PC/IP (CMU/MIT TCP/IP implementation for PCs)	\$100
	B-Tree Library & ISAM Driver (file system utilities by Softfocus)	\$100
	The Profiler (program execution profile tool)	\$100
	Entelekon C Function Library (screen, graphics, keyboard, string, printer, etc.)	\$100
	Entelekon Power Windows (menus, overlays, messages, alarms, file handling, etc.)	\$100
NEW!	QC88 C compiler (ASM output, small model, no longs, floats or bit fields, 80+ function library)	\$90
	Wendin Operating System Construction Kit or PCNX, PCVMS O/S Shells	\$80
	C Windows Toolkit (pop-up, pull-down, spreadsheet, CGA/EGA/Hercules)	\$80
	Professional C Windows (windows and keyboard functions)	\$80
	JATE Async Terminal Emulator (includes file transfer and menu subsystem)	\$80
	MultiDOS Plus (DOS-based multitasking, intertask messaging, semaphores)	\$80
	WKS Library (C program interface to Lotus 1-2-3 program & files)	\$80
	Professional C Windows (lean & mean window and keyboard handler)	\$70
	Quincy (interactive C interpreter)	\$60
	EZ.ASM (assembly language macros bridging C and MASM)	\$60
	PTree (parse tree management)	\$60
	HELP! (pop-up help system builder)	\$50
	Multi-User BBS (chat, mail, menus, sysop displays; uses Galaticomm modem card)	\$50
	Make (macros, all languages, built-in rules)	\$50
	Vector-to-Raster Conversion (stroke letters & Tektronix 4010 codes to bitmaps)	\$50
	Coder's Prolog (inference engine for use with C programs)	\$45
	C-Help (pop-up help for C programmers ... add your own notes)	\$40
	Biggerstaff's System Tools (multi-tasking window manager kit)	\$40
	PC-XINU (Comer's XINU operating system for PC)	\$35
	CLIPS (rule-based expert system generator, Version 4.1)	\$35
	TELE Kernel or TELE Windows (Ken Berry's multi-tasking kernel & window package)	\$30
	Clisp (Lisp interpreter with extensive internals documentation)	\$30
	Translate Rules to C (YACC-like function generator for rule-based systems)	\$30
	6-Pack of Editors (six public domain editors for use, study & hacking)	\$30
	Crunch Pack (a dozen file compression & expansion programs)	\$30
	ICON (string and list processing language, Version 7)	\$25
NEW!	FLEX (fast lexical analyzer generator; new, improved LEX)	\$25
	LEX (lexical analyzer generator; an oldie but a goodie)	\$25
	Bison & PREP (YACC workalike parser generator & attribute grammar preprocessor)	\$25
	AutoTrace (program tracer and memory trasher catcher)	\$25
	C Compiler Torture Test (checks a C compiler against K & R)	\$20
	Benchmark Package (C compiler, PC hardware, and Unix system)	\$20
	TN3270 (remote login to IBM VM/CMS as a 3270 terminal on a 3274 controller)	\$20
	A68 (68000 cross-assembler)	\$20
	List-Pac (C functions for lists, stacks, and queues)	\$20
	XLT Macro Processor (general purpose text translator)	\$20
	Data	
	WordCruncher (text retrieval & document analysis program)	\$275
	DNA Sequences (GenBank 52.0 including fast similarity search program)	\$150
	Protein Sequences (5,415 sequences, 1,302,966 residuals, with similarity search program)	\$60
	Webster's Second Dictionary (234,932 words)	\$60
	U. S. Cities (names & longitude/latitude of 32,000 U.S. cities and 6,000 state boundary points)	\$35
	The World Digitized (100,000 longitude/latitude of world country boundaries)	\$30
	KST Fonts (13,200 characters in 139 mixed fonts: specify T _E X or bitmap format)	\$30
	USNO Floppy Almanac (high-precision moon, sun, planet & star positions)	\$20
	NBS Hershey Fonts (1,377 stroke characters in 14 fonts)	\$15
	U. S. Map (15,701 points of state boundaries)	\$15

The Austin Code Works

11100 Leafwood Lane

Austin, Texas 78750-3409 USA

acw!info@uunet.uu.net

Voice: (512) 258-0785

BBS: (512) 258-8831

FidoNet: 1:382/12

Free shipping on prepaid orders

For delivery in Texas add 7%

MasterCard/VISA



BINARY ENGINEERING

Designing data structures, part I

Bruce F. Webster

In most discussions on program design, the focus is on the design and implementation of *algorithms*, which are the actual program statements that do the work. However, algorithms are only half of the story—without data structures, it's hard to write a meaningful program of any complexity. In fact, as programs grow more complex, the data structures that they use become more important, and the design and implementation of those structures become more critical. Recognition of this process has led to a new style of software design, known as *object-oriented programming*, where a program is viewed as a collection of data structures that communicate with one another.

When I first learned Pascal some eight years ago, I had been using FORTRAN as my primary high-level language for about five years. At that time, FORTRAN had one data structure—the array—and I was very good at turning arrays into whatever I needed. Pascal, though, was something of a shock. I had two new data types to work with—Boolean and enumerations—plus several new data structures, including strings, records, and sets. I can still remember how uncertain I felt for several months about the best way to use all of these new tools. With time, though, I became more confident, and developed a knack for being able to quickly come up with the right data structure for a given application.

Which brings us to my first point: The best way to learn how to design data structures is to design them. That's as true—and as discomfiting—as saying that the best way to learn how to program is to program. But, as with programming, there are guidelines and rules of thumb to help you along the way.

DATA CHOICES

Pascal and C provide six basic data types: integers, reals, characters, pointers, Booleans, and enumerated (user-defined) data types. In the case of C, Booleans and enumerations are really just integer values, and are defined more by function than by actual type. In Pascal, however, Booleans and enumerations are true separate data types. Here's a brief explanation of each type and the way it's generally used.

- **Integer:** Whole numbers (both positive and negative), which are used for counting, and in situations where round-off errors must be avoided.
- **Real:** Floating point numbers, which are used for very large, very small, and fractional numbers (especially in scientific and engineering applications).
- **Character:** Individual printing and control characters, based on an extended ASCII set, which are used for text input, output, and manipulation.
- **Pointer:** Addresses of variables, procedures, functions, and key locations of system hardware, which are used to modify data, allocate data structures dynamically, and pass routines as parameters.
- **Boolean:** The logical values **False** and **True**, which are used to compare values, test relationships, and remember the outcome of such tests and comparisons. C does not have an actual Boolean type, but considers the value 0 to be **False** and any nonzero value to be **True** (though logical expressions always return 1 as **True**).
- **Enumerated:** User-defined data types that are built from a list of identifiers, such as days of the week or colors in a spectrum. Enumerated types are used to make code more self-documenting, and to avoid declaration of long lists of integer constants.

These types are the building blocks for your data structures. A *data structure* is a collection or association of data types that is constructed in such a way that the individual elements can be written to, retrieved, or tested. There are four basic types of data structures: arrays, records (or structures), files, and sets. There are also two derivative types: strings and unions. Yet another type of data structure—linked lists—is created by combining records and pointers.

ARRAYS

An *array* is a list of elements where each element is retrieved by indexing the array name. The power of an array is that the indexing can be done by an expression that is evaluated while the program runs. Indexing is possible because arrays are *homogeneous*—all the elements of the array, from 1 to *n*, are of the same data type. As a result, the compiler easily generates code to find the *n*th element of the array.

What issues are involved in designing arrays? The biggest issue is size: How large do you make the array, and how do you enforce that size limit? Once an array is allocated, its size is fixed. If it's too big, then you're wasting data space. If the array is too small, problems may occur while your program executes. And finally, since compilers have limits on array size, the array may not ever be big enough, period.

For many applications, the array size has a natural limit that is defined by the problem itself. If you're using an array to count how many times each letter of the alphabet occurs in a given text, then you know that you only need 26 elements. If you're counting upper- and lowercase letters separately, you need 52 elements. If you're counting occurrences of all characters within a file, including punctuation, digits, and special characters, then you need 256 elements (one for each possible character).

However, there are times when the number of items varies from one run of the program to another, sometimes by a great deal. In that case, you have two choices: set an arbitrary limit, or use dynamic array allocation.

Setting an arbitrary limit is easy, but it has the drawbacks mentioned earlier of either wasting data space or else not having enough data space. Actually, "wasted space" isn't really wasted unless you have to eliminate some other data structures in order to make room for the array. Not having enough space is more serious. If the array runs out of room while the program is executing, then the program has to handle that error condition gracefully. If possible, you should let the user know about any limits ahead of time, as well as when he or she might encounter them.

One technique for managing fixed array size within a program is to declare constants that define the array's limits, and then reference those constants when the array is used in the program. A better solution, however, is dynamic array allocation. This can be accomplished in either C or Pascal (although it's a bit more complicated in Pascal). To do so, first find out how much space is required, then allocate that amount of space on the heap for the array. This process is handled by the C code shown in Figure 1, which takes advantage of the fact that arrays and pointers in C have the following relationship:

```
{
  int  *list,i,count;

  do {
    printf("Enter # of values: ");
    scanf("%d",&count);
  } while (count < 0);
  if (count == 0)
    exit();
  list = (* int) calloc(sizeof(int),count);
  if (list == NULL)
    exit();
  for (i=0; i<count; i++) {
    printf("Enter item #%d: ",i);
    scanf("%d",list[i]);
  }
  ...
  free(list);
}
```

Figure 1. Dynamic array allocation in C.

```
type
  NumList = array[1..2] of integer;
  NLPtr   = ^NumList;
var
  List    : NLPtr;
  I,Count : integer;
begin
  repeat
    Write('Enter # of values: ');
    Readln(Count)
  until (0 <= Count);
  if Count = 0
    then Halt;
  GetMem(List,SizeOf(integer)*Count);
  for I := 1 to Count do begin
    Write('Enter value #',I:3,': ');
    Readln(List[I]);
  end;
  ...
  FreeMem(List);
end.
```

Figure 2. Dynamic array allocation in Pascal.

*(a + i) == a[i]

With this relationship, even though **list** is declared as a pointer to type **int**, it can be indexed just like an array.

The Pascal solution, which is shown in Figure 2, isn't quite as tidy. It requires that we disable range checking and then declare a dummy array type and a pointer to that type. The number of elements to the array can be allocated by using **GetMem** and **SizeOf**. However, note that the pointer dereferencing operator (^) must be used to access the array itself.

With any solution, you need to be aware of the overall size of the array. There is a limit on how

continued on page 134

continued from page 133

large an array can be, even if it's allocated dynamically. Typically, an array can be no larger than 64K, and it usually must be somewhat smaller. The limit on the number of elements then becomes a function of the size of the element. If the array contains bytes or integers, then element size probably isn't a problem. On the other hand, if the array contains records that are each several hundred bytes long, then you can run out of array space very quickly. In this situation, you're probably better off with a linked list.

Multidimensional arrays can also chew up memory in a big hurry. Consider the following Pascal array declaration:

```
VAR
  A : ARRAY [1..100] [1..20] [1..20]
      OF real;
```

What we have here is an array of one hundred 20 × 20 floating point matrices. No problem, right? Wrong. The entire array holds 40,000 six-byte real numbers, and requires 240,000 bytes—definitely more than most compilers allow for any single array.

RECORDS

A *record* (in C terminology, a *struct*) is the data structure that complements the array. The elements of a record can be of different data types, making the record *heterogeneous*. Each element, known as a *field*, has a name. An element is referenced by appending the field name to the record variable's name. This means, of course, that you can't index through the elements of a record in the same way that you can index the elements of an array.

The strength of a record is its ability to associate data of different types. For example, suppose that you're writing a program to keep track of students in a school, and you design a record type to hold information for each student. This information would include name, age, sex, Social Security Number, grade point average (GPA), and class standing. The Pascal result might look like that shown in Figure 3; the C equivalent is shown in Figure 4.

```
type
  NameStr   = string[20];
  Gender    = (unknown, female, male);
  Grade     = (freshman, sophomore, junior, senior);
  Students  = record
    Last,First,Middle : NameStr;
    Age                : byte;
    Sex                : Gender;
    GPA                : real;
    Standing           : Grade;
  end;
```

Figure 3. A simple record type in Pascal.

```
typedef unsigned char byte;
typedef char nameStr[21];
typedef enum { unknown, male, female } gender;
typedef enum { freshman, sophomore, junior, senior} grade;

typedef struct {
  nameStr   last,first,middle;
  byte      age;
  gender     sex;
  float     gpa;
  grade     standing;
} students;
```

Figure 4. A simple record type in C.

To reference a field in a Pascal record, write the record variable name, followed by a period (“.”), followed by the field name. For example, the following code contains a variable **who** of type **student**, and assigns the value of 4.0 to the field **GPA**:

```
Who.GPA := 4.0;
```

How do you design a record? The first step, of course, is to list the information that belongs in the record. As you do so, give a name to each item and note which type of data or range of values are needed. For the example above, you might have created a list similar to the one shown in Figure 5.

last name	Last	string
first name	First	string
middle name	Middle	string
age	Age	0 to 150
sex	Sex	male, female, unknown
grade point average	GPA	0.0 to 4.0
current class	Standing	frosh, soph, jun, senior

Figure 5. The first step in designing a record: List your data items.

The second step is to define any auxiliary data types that you might need (or want) in order to create the record. In doing so, there are certain tradeoffs to bear in mind. For example, three strings are needed for the first, middle, and last names. Since strings are just a form of array, you face the usual array tradeoff of making the array too large and wasting space, or else making it too small to hold any reasonable instance of its data. In this case, I chose 20 characters per name as a nice compromise. That's large enough to handle 99+ percent of the names, but doesn't waste a tremendous amount of space. Still, those 63 bytes (each string has an additional byte to hold its current length) represent most of the space occupied by a **Students** record; if the overall record size is too big, that's the first place to cut.

Similarly, fields that are described with lists of words (such as “male, female, unknown”) are good

candidates for enumerated data types. They provide a level of self-documentation to your code that can be very useful, especially when using a debugger that recognizes enumerated types. Since enumerated types occupy only a single byte, they are always a more efficient choice than that of storing the equivalent lists of words as strings.

Finally, numeric fields should use the appropriate numeric type. When integer values are involved, use the appropriate integer type to enhance error checking and to minimize space. A list of the integer types in Turbo C and Turbo Pascal, according to ranges of values, is given in Figure 6.

Range	Turbo Pascal	Turbo C
-128..127	shortint	signed char
0..255	byte	unsigned char
-32768..32767	integer	int
0..65535	word	unsigned int
-2147483648..2147483647	longint	long
0..4294967295	<none>	unsigned long

Figure 6. Integer data types available in Turbo Pascal and Turbo C.

Likewise, use the appropriate floating point type for real numbers. In Turbo C, use **float** or **double**, depending upon how much precision and exponent range is needed. In Turbo Pascal, the usual choice is **real**; however, if you have an 8087/80287 math coprocessor, you can use one of the standard IEEE types (**single**, **double**, **extended**) to better fit your needs.

Since a record can have data structures as its fields, you might want to also declare those data structures as distinct types. Unless you run into a serious problem with confusing identifier names, you're usually better off declaring each data structure as a separate type, then using that type name in declaring the record. This approach has three advantages. First, it produces cleaner and more readable code. Second, it makes it easier to pass record, set, and array fields as separate parameters to subprograms, since you now have predefined types for the subprogram's formal parameters. Third, it allows you to quickly change the underlying data structure of a given field by redeclaring that type and then modifying the program wherever it references that data structure. Having defined any needed types, you can then declare the record type itself, listing each field name and its corresponding type. A more extended example of this process can be seen in Figure 7, which I'll discuss in more detail later.

ARRAYS OF RECORDS

One of the most useful data structures is an *array of records*. This is just what it sounds like: An array of

some size, where each element of the array is a record. As an indexed list with each element containing a mixture of data types, this array combines the best of both data structures

The process of setting up an array of records is just like that of setting up any other array. For example, an array of **Student** records might be declared as:

```
VAR
  SList : ARRAY[1..100] OF Student;
```

You can now set the **GPA** field of the 20th record with the following code:

```
SList[20].GPA := 4.0;
```

Arrays of records fall prey to the same problems that befall other arrays: wasted space, not enough space, or being too large for a single data segment. Since it's easy to create records that are several dozen (or even several hundred) bytes, the problems are all magnified. An alternative to an array of records is a linked list of records, but linked lists are beyond the scope of this month's column.

NESTED RECORDS

Both C and Pascal allow you to *nest* records—to define record fields that are themselves records. This approach lets you “modularize” your records by replacing several fields with a single field; this single field consists of a record containing the replaced fields. In other words, just as you can break a large section of code into several smaller subroutines, you can take a record with a long list of fields and break it down into several “subrecords.” This method offers several advantages. It makes the overall record structure easier to follow, and groups related information together into a single field. It allows those groups of information to be manipulated as single entities (assigning, passing as parameters, and so on).

Figure 7 shows an extended version of the type **Student**. Several new record types (**Time**, **Periods**, **Classes**) have been created to add the student's class schedule. Arrays of type **Periods** have been added to let each class have up to six periods per week. Arrays of type **Classes** let each student have up to 12 classes.

If you have a nested record and want to reference a field of a subrecord's field, take this approach: Use another period, followed by the field name. For example, if you have a variable **Temp** of type **Period**,

continued on page 136

```

const
  PMax = 6;
  CMax = 12;

type
  NameStr = string[20];
  Gender = (unknown, female, male);
  Grade = (freshman, sophomore, junior, senior,
graduate);
  Days = (sun, mon, tues, wed, thur, fri, sat);
  Time = record
    Hour : 0..23;
    Min : 0..59
  end;

  Periods = record
    Day : Days;
    Start, Finish : Time
  end;
  PList = array[1..PMax] of Periods;

  Classes = record
    Title, Instructor : NameStr;
    Period : PList;
    PNum : 0..PMax;
    Score : real
  end;
  CList = array[1..CMax] of Classes;

  Students = record
    Last, First, Middle : NameStr;
    Age : byte;
    Sex : Gender;
    GPA : real;
    Standing : Grade;
    Class : CList;
    CNum : 0..CMax
  end;

```

Figure 7. A more complex record type in Pascal.

BINARY ENGINEERING

continued from page 135

you can set it to "Wednesday, from 11:00 am to 12:20 pm" with the following statements:

```

Temp.Day := wed;
Temp.Start.Hour := 11;
Temp.Start.Min := 00;
Temp.Finish.Hour := 12;
Temp.Finish.Min := 20;

```

There is one disadvantage to nested records: the need to reference an individual field through many levels. Suppose you want to find the closing time of the last period of the first class for the 17th student in **SList**. Assuming that **LHour** and **LMin** are of type **integer**, the statements would look like this:

```

LHour := SList[17].Class[1].
         Period[PNum].Finish.Hour;
LMin := SList[17].Class[1].
        Period[PNum].Finish.Min;

```

As you can see, it can get a bit unwieldy at times. Pascal's solution is the **WITH** statement, which lets you drop the record name and period, and simply reference the field. **WITH** statements can be nested, so that you can work your way inward. This is demonstrated in Figure 8, which shows a section of code that adds up to all of the class time that a student should be spending.

```

const
  SMax : 100;
var
  SList : array[1..SMax] of Students;
  SCount : 0..SMax;
  I, J, K : byte;
  DM, DH : integer;
  Minutes, Hours : word;
begin
  ... { get data in somehow }
  for I := 1 to SCount do with SList[I] do begin
    Minutes := 0;
    for J := 1 to CNum do with Class[J] do begin
      for K := 1 to PNum with Period[K] do begin
        DM := Finish.Min - Start.Min;
        DH := Finish.Hour - Start.Hour;
        Minutes := Minutes + DM + 60*DH
      end
    end;
    Hours := Minutes div 60;
    Minutes := Minutes mod 60;
    Writeln>Last, ' ', Hours:2, ':', Minutes:2)
  end;
  ...
end.

```

Figure 8. Tallying the total time spent in class by each student.

Since C has no analogous statement to Pascal's **WITH**, explicit referencing is required. The best solution is to limit how deeply records are nested. Barring that, you can assign a pointer to the subrecord and reference that subrecord, but it's not quite as clean and easy, and sometimes it doesn't help all that much.

How deeply should records be nested? As deeply as makes sense. The example in Figure 7 is justifiable by a simple test: How would each record type look if there were no nesting? In fact, could you even implement the **Student** data type without nesting records? Yes, but it would require the use of lots of arrays, including some two-dimensional arrays.

That's as much as I can cover in this issue. As you can guess, we've just begun to touch on some of the aspects of data structure design. In the next issue, I'll talk more about data structures, and will explore some general design issues. ■

Bruce Webster is a computer mercenary living in California. He can be reached via MCI MAIL (as Bruce Webster) or on BIX (as bwebster).

“Behind the beauty of the Turbo C environment stands the brawn of a full-fledged compiler”

Marty Franz, *PC Tech Journal*

“ Taking compilers and program development tools into the next generation is Borland International's Turbo C, a \$99.95 package that will stun you with in-RAM compilations that operate at warp speed.

... a 21st century compiler at a preinflation 1967 price. Is it any wonder that Turbo C was included in the Best of 1987?

Richard Hale Shaw, PC Magazine

Turbo C represents an all-new price-performance level—one that will be hard to match, much less beat.

Stephen Randy Davis, PC Magazine

Turbo C showed excellent compiler speeds, good overall benchmark scores, and extraordinary floating-point performance. *Scott Robert Ladd, Micro Cornucopia* ”

Our new Turbo C 1.5 is a technological tour de force

At Borland we believe the slow way is no way, so Turbo C® is a racer. And as well as white-knuckle speed, Turbo C also gives you spectacular graphics.

Minimum system requirements: For the IBM PS/2™ and the IBM® family of personal computers and all 100% compatibles. PC-DOS (MS-DOS®) 2.0 or later. 384K. *Artwork metallic courtesy of Genigraphics® Corporation

**Customer satisfaction is our main concern; if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Copyright ©1988 Borland International, Inc. BI-1232



Actual photograph of Turbo C graphics displayed on IBM 8514 screen.*

Some of the reasons why the critics are so enthusiastic about Turbo C 1.5

Turbo C now includes:

- A professional-quality graphics library of over 70 functions
- A librarian that allows you to build your own object module libraries
- Context-sensitive help for the language and the library routines
- Text/video functions, including windows
- 43- and 50-line mode support
- VGA, CGA, EGA, Hercules, and IBM 8514 support
- File search utility (GREP)
- Sample graphics applications
- More than 100 new functions

The professional optimizing compiler for less than \$100.00

For professional-quality C at a sane price, nothing comes close to Turbo C. It's super-fast and super-graphic. (We used it ourselves to write Eureka.™ The Solver and to develop the presentation-quality graphics in Quattro,™ our new and highly successful professional spreadsheet.) No one can deliver technical superiority like Borland.

*60-Day Money-back Guarantee***

To order from Programmer's Connection, call,

USA: (800) 336-1166

CANADA: (800) 225-1166

OHIO & ALASKA (collect)

(216) 494-3781





LANGUAGE CONNECTIONS

Turbo Prolog to Turbo C is now a two-way bridge!

Gary Entsminger

The new Turbo Prolog (version 2.0) is the BMW of programming languages, the Prince of Prologs. Michael Floyd introduced Turbo Prolog 2.0 and its new features in the May/June issue of *TURBO TECHNIX*. In this column, I'll focus on Turbo Prolog's exciting new language interface, which not only allows Turbo Prolog programmers to call functions written in other languages, but also allows programmers using other languages (such as Turbo C) to call predicates written in Turbo Prolog. In addition, Turbo C programmers can call the Turbo Prolog library predicates. Now you can have the best of worlds—Turbo C's speed and efficiency, Turbo Prolog's logic, and the Turbo Prolog Runtime Library—at your fingertips.

To see how the interface works, we'll examine three examples. The first example demonstrates the basic interfacing process, and points out a few of the differences between Turbo Prolog 2.0 and earlier versions. The second example shows how to call Turbo Prolog built-in predicates, and other Turbo Prolog predicates, from Turbo C. Finally, the last example is a variation of the last "Language Connections" column (*TURBO TECHNIX*, May/June, 1988) in which Michael Floyd converted a Turbo Prolog list into a Turbo C array and back again. This third example highlights some of the new memory management features of Turbo Prolog.

DECLARING EXTERNAL FUNCTIONS

Recall that to use external functions with Turbo Prolog, we must declare them as **global predicates** in the Turbo Prolog module. To declare a global predicate for an external procedure, we must specify the global predicate's name, the data types of its arguments, its flow variants, and the language that the global predicate will interface with. For instance, to declare a global predicate called **add** that is written in Turbo C, we make a declaration similar to:

```
global predicates
  add(integer, integer, integer)
  - (i,i,o) language c
```

This statement declares the external procedure **add**, which takes two integers and returns a third (as specified by the **(i,i,o)** flow pattern). Remember that Turbo Prolog requires the programmer to specify input and output parameters (or flow patterns) when declaring a function written in another language. Known parameters in the flow pattern are *input* parameters, and unknown parameters are *output* parameters. This in-and-out pattern indicates how the function behaves.

Turbo Prolog lets us declare more than one flow pattern per function. This enables us to "get more out of a function" by calling it under different circumstances, with different known (or *bound*) and unknown (or *free*) variables. To allow **add** to handle the different cases, simply add additional flow patterns to the declaration:

```
global predicates
  add(integer, integer, integer) -
    (i,i,o) (i,o,i) (o,i,i)
    (i,i,i) language c
```

Of course, we must have a separate function to handle each of the possible flow patterns.

Internally, Turbo Prolog generates a different call for each of the global predicate's possible flow patterns. Each subsequently generated call consists of the predicate's name, followed by an underscore character, and then a number. The number for the first flow pattern is **0**. For each additional flow pattern, the number increments by one. In this case, the predicate **add** has four flow patterns, and Turbo Prolog generates four calls to functions named **add_0**, **add_1**, **add_2**, and **add_3**. You must keep this naming convention in mind when creating your Turbo C functions.

In Listing 1, **add** describes the relationship between three integer variables: **x**, **y**, and **z** (where **z** is the sum of **x** and **y**). Flow pattern 1 says, "We know **x** and **y**, so find the value of **z**." The Turbo C function **add_0** looks like this:

```
add_0(int x, int y, int *z)
{
  *z = x + y;
}
```

Turbo Prolog passes the values that are bound to **x** and **y** to the Turbo C function, which adds them together and passes the sum (**z**) back.

Flow pattern 2 says, "We know values for **x** and **z**. Now find the value of **y**." The Turbo C function **add_1** takes the difference of **z** and **x** and returns a pointer to **y**:

```
add_1(int x, int *y, int z)
{
    *y = z - x;
}
```

Likewise, Listing 1 contains functions **add_2** and **add_3** to handle the additional flow patterns.

THE "AS" KEYWORD

Interfacing Turbo Prolog 1.1 with routines already developed in Turbo C means that you must rename all of your Turbo C routines to handle the **_0** suffix generated by Turbo Prolog. This may be a problem if these same routines are also used by other Turbo C modules. Turbo Prolog 2.0 overcomes this problem by allowing you to *alias* the internal call generated by Turbo Prolog. For instance, if a Turbo C function called **bin_search** is to be called by Turbo Prolog, make the following declaration in Turbo Prolog:

```
global predicates
    binary_search(integer, integer) -
        (i,o) language c as bin_search
```

The **as** keyword in this declaration tells Turbo Prolog to generate a call to **bin_search** instead of **binary_search_0**. Therefore, you can call Turbo C routines without having to rename them.

There is one word of caution when using the **as** keyword. If the global predicate is declared in such a way that Turbo Prolog must generate more than a single call, you will not be able to take advantage of the **as** keyword because **as** only allows a single alias name to be generated. In such cases, Turbo Prolog adds the **_0** suffix to the alias name specified by the **as** keyword.

This situation arises in two cases. The first case is when more than one flow pattern is specified in the **global predicates** declaration (as I discussed earlier). The second case arises when a global predicate has multiple arities; this situation is discussed next.

MULTIPLE ARITIES

As you may already know, Turbo Prolog 2.0 now allows you to declare predicates to have multiple arities. This capability extends to global predicates as well. Turbo Prolog generates an additional call for each arity and uses the same naming conventions that I described for multiple flow patterns. In the case of the global predicate **match**, which takes either one or two arguments, we could make the following declaration:

```
global predicates
    match(integer) - (i) language c
    match(integer,real) - (i,o)
    language c
```

Turbo Prolog generates two calls, **match_0** and **match_1**.

You may now be wondering what happens when each arity has several flow patterns. In this situation, Turbo Prolog generates names for each flow pattern within a given arity first, then moves on to the next arity. For example, consider the following declarations:

```
global predicates
    match(integer)
        - (i) (o) language c

    match(integer,real)
        - (i,o) (i,i) language c
```

Here, Turbo Prolog generates two calls, **match_0** and **match_1**, for the first **match** predicate. Two additional calls, **match_2** and **match_3**, are created for the second **match** predicate.

As mentioned earlier, the **as** keyword cannot be used to alias the generated calls, since multiple calls are generated.

CALLING TURBO PROLOG FROM TURBO C

Although Turbo Prolog version 1.1 permitted programmers to call external functions written in other languages, it had no hooks to allow Turbo Prolog predicates to be called from another language. With Turbo Prolog 2.0, this has changed—now if you declare a global predicate in another language, and Turbo Prolog clauses exist for that predicate, Turbo Prolog 2.0 generates a callable routine for that language. The key restriction is that Turbo Prolog *must* be the main program. This enables it to control memory and to set up its own heap and stacks.

Listings 3 and 4 represent a simple example that uses Turbo C to call several user-defined predicates. In this example, the Turbo Prolog module is the "main" program as designated by the **goal**. The **goal** immediately calls **extprog**, which is the Turbo C function. **extprog** then makes calls back to the Turbo Prolog module to create a window and display the current directory. At this point, the user can highlight and select a given file which is then displayed in another window using Turbo Prolog's built-in editor.

Note in Listing 3 that, even though the **language** specified for all **global predicates** is **c**, the only external function that is actually written in Turbo C is **extprog**. Remember, the **language** specifier merely designates the calling conventions to be used (not necessarily the language that the external function is written in).

Also, note in Listing 3 that I've made ample use of the **as** keyword. Of course, this would not have been possible had I specified multiple flow patterns or arities for a given predicate.

continued on page 141

LISTING 1: PADD.PRO

```
GLOBAL PREDICATES
add(integer, integer, integer) -
  (i,i,o),(i,o,i),(o,i,i),(i,i,i) language c

GOAL
add(2,3,Z), write("2 + 3 = ",Z), nl,
add(2,Y,5), write("5 - 2 = ",Y), nl,
add(X,3,5), write("5 - 3 = ",X), nl,
add(2,3,5), write("2 + 3 = 5").
```

LISTING 2: CADD.C

```
void add_0(int x, int y, int *z) /* (i,i,o) flow pattern */
{
  *z = x + y;
}

void add_1(int x, int *y, int z) /* (i,o,i) flow pattern */
{
  *y = z - x;
}

void add_2(int *x, int y) /* (o,i) flow pattern */
{
  *x = 1 - y;
}

void add_3(int x, int *y) /* (i,o) flow pattern */
{
  *y = x+x;
}
```

LISTING 3: PEDIT.PRO

```
GLOBAL PREDICATES
mymakewindow(integer, integer, integer, string, integer, integer,
integer, integer) - (i,i,i,i,i,i,i,i)
  language c as "makewindow"
myremovewindow - language c as "removewindow"
clrscr - language c as "clrscr"
myreadchar(char) - (o) language c
myreadline(string) - (o) language c
my_edit(string) - (i) language c as "edit"
my_dir(string) - (o) language c as "dir"
myfile_str(string, string) - (i,o) language c as "getfile"

extprog - language c

GOAL
extprog.

CLAUSES

mymakewindow(WNO,WATTR,FATTR,TEXT,SROW,SCOL,ROWS,COLS):-
makewindow(WNO,WATTR,FATTR,TEXT,SROW,SCOL,ROWS,COLS).

myremovewindow:- removewindow.

clrscr:- clrscr.

myfile_str(File,Str):-
file_str(File,Str).

my_dir(Filename):-
dir("","*.","*",Filename).

my_edit(Str):-
edit(Str,_).
```

LISTING 4: CEDIT.C

```
extprog_0()
{ char dummychar; char *Str; char *Filename;
makewindow(1,7,7,"Directory Window",5,5,15,60);
dir(&Filename);
removewindow();
getfile(Filename,&Str);
makewindow(2,7,7,"Edit Window",0,0,25,80);
edit(Str);
removewindow();
clrscr();
}
```

On the Turbo C side (see Listing 4), a Turbo Prolog external predicate is effectively identical to any other external function. Turbo C calls the Turbo Prolog predicate just as it would call another Turbo C function. The only difference is that the predicate is actually coded in Turbo Prolog.

DYNAMIC MEMORY ALLOCATION

In order to create a dynamic structure of unspecified size in Turbo C, we must take care of a few low-level details ourselves (such as memory allocation). Turbo C supplies the standard functions: **malloc** and **free** for handling the heap, and **calloc** for allocating space on the stack. However, we must handle the heap and stack by Turbo Prolog rules. Thus, in order to pass structures such as Turbo Prolog lists, we must allocate memory for the list using Turbo Prolog's library routines.

Turbo Prolog provides two library routines to handle the heap: **_malloc** and **_free**. These routines are declared in C as:

```
void *_malloc(unsigned size);
void *_free(void *);
```

In addition, **alloc_gstack** is provided to handle the stack:

```
void *alloc_gstack(unsigned size)
```

This routine returns a pointer to a memory block of length **size**. When you use **alloc_gstack**, any memory that was previously allocated is freed when a fail occurs, which causes Turbo Prolog to backtrack across the memory allocation.

I should mention that these memory management routines were previously provided by CPINIT.OBJ in the form of **malloc_heap**, **release_heap**, and **palloc**, respectively. Since Turbo Prolog's library routines are now directly callable from Turbo C, however, CPINIT is no longer required.

The example in Listings 5 and 6 uses **alloc_gstack** to allocate memory on the Turbo Prolog global stack for a Turbo Prolog list. By comparing this example with the memory allocation example in the May/June "Language Connections" column, you'll be able to easily identify the differences in connecting the different versions of Turbo Prolog. In addition, you'll be able to see some of the new features that are added with the 2.0 connection.

In both memory allocation examples, we begin in Turbo Prolog by asking for a list of integers. The user enters integers one at a time, terminating each entry with a carriage return, and ending the input by an arbitrarily chosen number (in this case, -999). The entries are stored temporarily in a database and then collected into a list using **findall**.

continued on page 142

LISTING 5: PSORT.PRO

```
GLOBAL DOMAINS
list = integer*

GLOBAL PREDICATES
mymakewindow(integer, integer, integer, string, integer,
integer, integer, integer) -
(i, i, i, i, i, i, i) language c
myremovewindow language c
sortlist(list, list) - (i, o) language c
write_string(string) - (i) language c
pause - language c

DATABASE
db(integer)

PREDICATES
run
repeat
test_input(integer)

GOAL
run.

CLAUSES
run:- /* Get items. */
clearwindow,
repeat,
write("Enter list (-999 to quit): "),
readint(S),
test_input(S),
findall(N, db(N), List),
sortlist(List, L), /* Call C function. */
makewindow(2, 7, 7, " In Turbo Prolog ", 7, 10, 7, 50),
write(L), nl,
pause, removewindow, clearwindow.

test_input(S):-
S = -999. /* End of list, so succeed & process. */
test_input(S):- /* If list hasn't been terminated,
assert new member, and fail to force
backtracking. */
S <> -999,
assert(db(S)), fail.

repeat.
repeat:- repeat.

/* predicates called from Turbo C */
write_string(I):- write(I).

pause:-
write("\n\n\n Press any Key To Conitnue..."),
readchar(_).

mymakewindow(WND, WATTR, FATTR, TEXT, SROW, SCOL, ROWS, COLS):-
makewindow(WND, WATTR, FATTR, TEXT, SROW, SCOL, ROWS, COLS).

myremovewindow:-
removewindow.
```

```

#define alloc_gstack walloc
#define listfno 1
#define nilfno 2

void *alloc_gstack(unsigned);
/* Declare a Prolog list in C */
typedef struct ilist {
    char Functor;
    int Value;
    struct ilist *Next;
} IntList;

int ListToArray(IntList *List, int **ResultArray)
{ IntList *SaveList = List;
  int *Array;
  int i = 0;
  /* Count list items. */
  for(i=0; List->Functor !=listfno;
      List = List->Next)
      i++;
  /* Allocate stack space. */
  Array = alloc_gstack(i*sizeof(int));

  List = SaveList;
  /* Copy list to array. */
  for(i=0; List->Functor !=listfno; List=List->Next)
      Array[i]=List->Value;

  *ResultArray=Array;
  return(i);
}

ArrayToList(int Array[],int n,IntList **List)
{ int i;
  /* Allocate a record for each element. */
  for (i=0; i<n; i++)
  { IntList *p = *List = alloc_gstack(sizeof(IntList));
    p->Functor = listfno;
    p->Value = Array[i];
    List = &(*List)->Next;
  }
  /* Allocate th last record in the list. */
  { IntList *p = *List = alloc_gstack(sizeof(char));
    p->Functor = nilfno;
  }
}

/* Increment all values in the list. */
sortlist_0(IntList *InList, IntList **OutList)
{ int i, j, n, temp, *Array;
  n = ListToArray(InList, &Array);
  for(i=0; i<n-1; i++)
      for(j=i+1; j<n; j++)
          if (Array[i] > Array[j])
          {
              temp = Array[j];
              Array[j] = Array[i];
              Array[i] = temp;
          }
  /* Call Prolog predicates. */
  mymakewindow_0(1,7,7," In Turbo C ",5,5,15,60);
  write_string_0("\n\n Sort Completed\n\n");
  pause_0();
  myremovewindow_0();
  ArrayToList(Array,n,OutList);
}

```

LANGUAGE

continued from page 141

Turbo Prolog then passes this list to the Turbo C function **ListToArray**, which converts the list into an array and sorts the elements. Once the array is sorted, a window is created using **mymakewindow**. Next, a message is displayed (**write_string**) to indicate that the sort is complete, and the system pauses (**pause**) until the user hits a key. (**mymakewindow**, **write_string**, and **pause** are all written in Turbo Prolog.) An interesting point is that **pause** is called in both the **sortlist** routine of the Turbo C module, and in the **run** clause of the Turbo Prolog program. This shows how a set of tools developed in Turbo Prolog can be directly callable from either Turbo Prolog or Turbo C without modification.

Finally, the Turbo C **sortlist** routine converts the array back to a list using **ArrayToList**, and passes the list back to Turbo Prolog.

COMPILING AND LINKING

Much of the compile and link process is unchanged, so I will not reiterate the entire process. (For a complete discussion, refer either to the *Turbo Prolog Owner's Handbook*, or to any of the earlier "Language Connections" columns.) On the Turbo C side, the actual compile has not changed. You must still use the Large memory model, and set **Generate underbars OFF** and **Jump optimization ON**.

The compile process has changed slightly on the Turbo Prolog side when you compile as a project. In this case, you now have the ability to link in other libraries in addition to the Turbo Prolog Runtime Library. This option is available in the **Options** pull-down menu.

The command line link process has a minor change. Turbo Prolog 2.0 no longer requires **CPINIT.OBJ** to set up the calling conventions, so **CPINIT** does not appear in the command line. Otherwise, the compile and link process has not changed.

The new features in Turbo Prolog 2.0 make it a powerful programming tool, not only in its own right as a standalone programming language, but also as a utility for other languages such as Turbo C. Turbo C programmers who love to muck around in the depths, but hate learning high-level details (some of my favorite C programming friends talk this way) should look into this connection—you won't even have to clean up to do it. ■

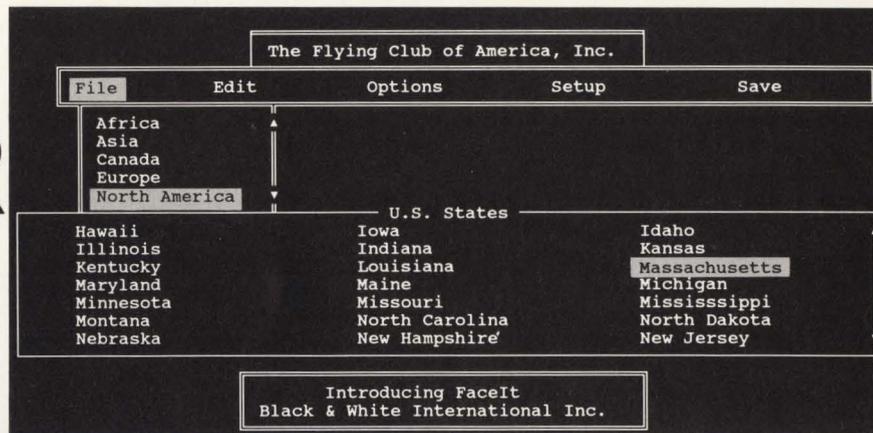
Gary Entsminger is a freelance writer, and an associate editor of Micro Cornucopia magazine.

Listings may be downloaded from CompuServe as LCVIN5.ARC.

TURBO
PROGRAMMERS

PUT YOUR BEST FACE FORWARD

With Our New Interface Manager



Introducing FaceIt™

A powerful interface design tool for Turbo Basic®, C®, Pascal® and Prolog®.

Just think of some of the best text-based menu driven interfaces you have encountered or even written. Now imagine being able to get those same pop-up, pull-down, horizontal, and multiple column menus in minutes, without writing a single line of code. FaceIt lets you add state-of-the-art interfaces to all your Turbo programs. Use FaceIt to create front-ends to systems, build online help systems or as a quick prototyping tool.

A New Way To Create Interfaces

We know how hard it is to create that special "look and feel". Menus used to take hours and even days to design, code and tweak just to get them right. But now there's FaceIt. FaceIt does all the work of menu and interface creation for you. You specify the contents of the menu and FaceIt does the rest. It designs perfect menus and layouts every time.

How FaceIt Works

1. You define the contents of your menus using any editor. Or, you can import data directly from a dBASE™ DBF file.
2. Then FaceIt, using this data, designs a single menu or multiple-menu interface.
3. Use the interface as is or polish it up using FaceIt's interactive mode.
4. You're done.

FaceIt Features

- | | |
|--|--|
| <input checked="" type="checkbox"/> Scrolling menus with scroll bars | <input checked="" type="checkbox"/> Return Strings |
| <input checked="" type="checkbox"/> Headers and Footers | <input checked="" type="checkbox"/> Runtime Module Uses Only 19K |
| <input checked="" type="checkbox"/> Onscreen Menu Customization | <input checked="" type="checkbox"/> Supports |
| <input checked="" type="checkbox"/> Full Color Support | EMS 3.2 and above |
| <input checked="" type="checkbox"/> Separators/Blank Items | 43 Line EGA Mode |
| <input checked="" type="checkbox"/> Initial Character Selection | 50 Line VGA Mode |
| <input checked="" type="checkbox"/> Item/Menu Level Help | 40 ColumnMode |
| <input checked="" type="checkbox"/> Default/Manual Placement | Microsoft Compatible Mice |
| <input checked="" type="checkbox"/> Unavailable Items | <input checked="" type="checkbox"/> Not Copy Protected |

Build Interfaces Right On The Screen

Design sophisticated multi-menu systems with FaceIt's interactive mode. Start with a single main menu. Link sub-menus to the main menu. Annotate each item with status lines. Then add context-sensitive help.

Total Menu Customization

Change window shapes, border styles and color every element of the menu right down to the individual menu item.

Multiple-Language Support

FaceIt includes language specific modules (LSMS) for all Borland and Microsoft languages, a dBASE™, FoxBASE™, Clipper™ and Quicksilver™. These LSMS provide the two way communication between your application and the FaceIt engine. FaceIt can return to your program the face of the menu or a return string from another menu or the name of the menu and the number of the item selected.

The Perfect Turbo Companion

FaceIt includes a royalty-free runtime module that you may distribute with your applications. So, whether you program for yourself, your company or other people, FaceIt will create the right face and give your application the look it deserves.

FaceIt™

and put your best face forward.

Only \$99

Call Today 212-787-6633

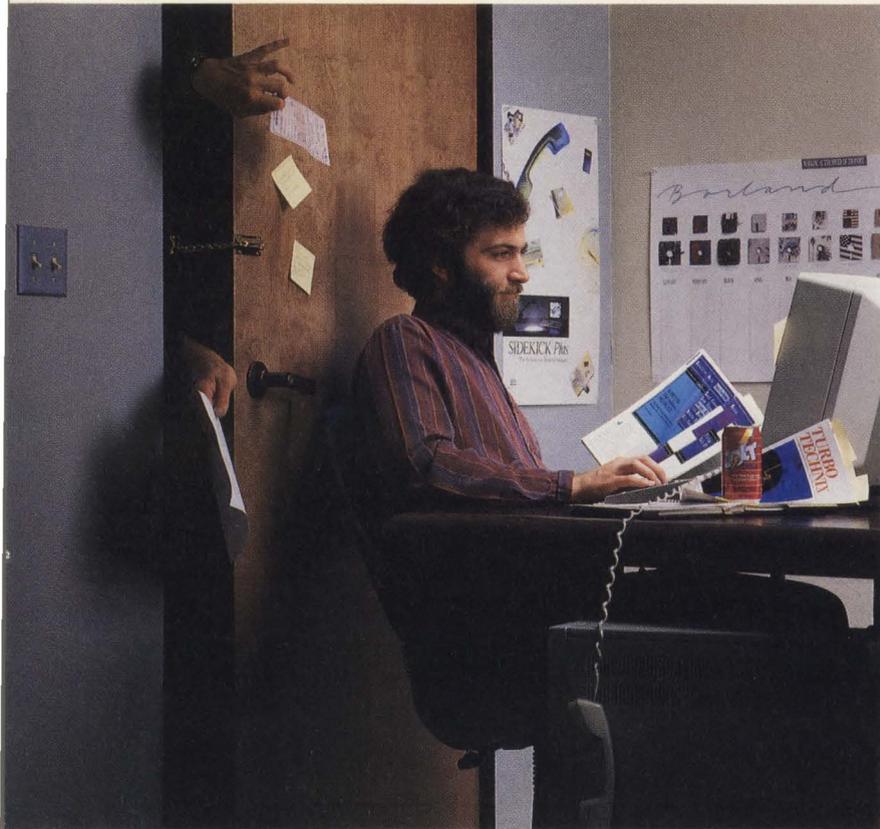
Black & White International, Inc.

P.O. Box 21108
NY, NY 10129



Try FaceIt risk free! FaceIt is backed by our 30 day unconditional money back guarantee.

FaceIt is a trademark of Black & White International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Requires DOS 2.1 or higher, for the IBC PC, PS/2 and true compatibles



*There's only
one way
to reach
a programmer—
Use the
programmers'
magazine:
**TURBO
TECHNIX**
THE BORLAND LANGUAGE JOURNAL*

Our readers know that *TURBO TECHNIX* is the place to be when the focus is on development. They watch us for the tips and techniques that help them utilize the speed and power of Borland's programming languages. And they spend a lot of time in these pages. Your ad should be here.

NOVEMBER/DECEMBER 1988
ISSUE CLOSING DATE: SEPTEMBER 8

Multitask Turbo Pascal applications under DOS ... understand and circumvent the DOS reentrancy problem with TSRs ... write code-generating scripts in PAL ... take the mystery out of structures and unions in Turbo C ... store data in 286 extended memory ... discover definite clause grammars in Turbo Prolog ... plus our columnists, Dialog, and Philippe at his provocative best.

JANUARY/FEBRUARY 1989
ISSUE CLOSING DATE: NOVEMBER 2

Implement a "poor man's LAN" using PC parallel ports ... learn about Turbo Pascal's enumerated types ... interpret HPGL plotter command files to the screen ... understand state space through Turbo Prolog ... turn data files to .OBJ files for linking as externals to Turbo Pascal ... study Paradox memory management ... and enjoy all our regular columnists and features.

**CALL NOW
RESERVE YOUR
TURBO TECHNIX
SPACE TODAY!**

Publisher
John Hemsath

Home Office
(408) 438-9321

Western Office
(714) 858-0408
Janet Zamucen

New England Office
Mid-Atlantic Office
(617) 848-9306
Merrie Lynch
Nancy Wood

Southeastern Office
(813) 394-4963
Megan Patti



TALES FROM THE RUNTIME

Organization and optimization

Mark L. Van Name and Bill Catchings

In the past few columns, we've focused on adding capabilities to the Turbo C Runtime Library. In this column, we discuss two primarily procedural topics: How to use the Runtime's source code organization and batch files (see accompanying sidebar, "Runtime Maintenance"), and how to optimize Runtime C routines by translating them into assembly language.

The motive for such translations is clear: no matter what you want your code to do, you usually want the code to do it faster. You get the biggest speed improvements, of course, with better algorithms, but you can often get substantial gains by recoding a few key routines in assembler.

If you program regularly in assembler, you might not find this kind of recoding much trouble. For most of us, however, it's a slow process that requires a great deal of attention to detail. Fortunately, you can skip a great deal of work by having Turbo C generate the initial assembly language version of the routine—then you only have to improve that version and integrate it with the rest of the Library. (While we'll look at an example from the Runtime, you can also use this technique for other routines.)

We feel obliged to point out that you probably should avoid this kind of optimization except for heavily used routines, or routines where performance is crucial. One of the main points of using C is to work with a high-level, structured language. Furthermore, much of the Runtime is already in assembler, in keeping with Borland's emphasis on performance. Finally, Turbo C generates basically good code, so you'll often find (as we did in the example below) that there is little to improve.

OPTIMIZING THE `rand` FUNCTION

Before you translate an entire routine to assembler, consider replacing only small parts of its C code with some inline assembly code. If you do include assembler in a C routine, remember to put the following directive early in the file:

```
#pragma inline
```

Also, each line of assembly code must begin with the Turbo C reserved word `asm`.

Assuming, however, that you want to go for the whole enchilada, let's look at a procedure that you can use. We chose the Runtime's `rand` function (see Listing 1) because it provides a simple starting point, and we'll make only a very minor improvement to this routine. We want to examine the procedure itself more than the particular changes.

First, compile the routine with TCC using the `-S` option to produce a listing of the assembler routine that was generated. TCC names the file `<filename>.ASM` (in our case, `RAND.ASM`). Notice that Turbo C adds line numbers as comments—these line numbers identify the source lines that correspond to each section of the assembler code, and are handy if you want to replace specific lines with more efficient inline assembler fragments.

We made a copy of the assembler output file to use later for discussion and comparison, and named that copy `RAND.OUT` (see Listing 2). We then worked directly on `RAND.ASM`; Listing 3 shows the final version of that code.

Notice first that `RAND.OUT` contains many setup and cleanup instructions (these instructions are common to most routines, so there's not much new here). Furthermore, these instructions can make it difficult to see what the code is really doing. To correct these problems, we cut away the code containing the instructions, and replaced it with the Runtime assembler macros that surround most Runtime assembler routines. We lifted this boilerplate code directly from `spawn`, in the file `SPAWN.ASM` (but nearly any other Runtime assembler routine would do as well).

As we described in a previous column, most of these macros come from the file `RULES.ASI`, so you need to include that file in the code. The macro `Header@` replaces most of the setup code that Turbo C generates. The `CSeg@` and `CSegEnd@` macros replace most of the other setup and cleanup instructions. These macros signal the start and end, respectively, of the code segment.

HANDLING VARIOUS MEMORY MODELS

This approach does have one limitation—the assembler code that Turbo C generates only works for the memory model that is used for the compilation. If you're not sure whether your code should vary

continued on page 146

FROM THE RUNTIME

continued from page 145

from one memory model to another, compile the code with several different memory models and compare the resulting .ASM files. If they are all the same, you have no problem.

If they are different, or if you already know that you need to make some changes to force your code to work with all of the memory models, then you have a little more work to do. The memory model that we use in RAND.ASM could affect the code in three areas.

First, we have to decide whether to declare the procedures as near (for Small-code memory models) or as far (for Large-code memory models). Fortunately, the standard Runtime macro **PubProc@** handles this problem for us. By replacing the procedure declarations with **PubProc@**, we make those procedures *public* (callable from anywhere in a program). **PubProc@** also takes the memory model switch from the MASM command line, and uses the switch to determine if the routines should be near or far. We use the second **PubProc@** argument, **_CDECL_**, to indicate that the routines should use calling and naming conventions in C rather than in Pascal.

We also must deal with the different stack offsets of the seed argument to **srand**. We use a defined constant, **SEEDOFF**, in **srand** to retrieve the argument correctly. Conditional assembly defines **SEEDOFF** appropriately for the routine's memory model.

The third difference arises only in the case of the Huge memory model, where each file places its static variables into its own data segment. This feature requires us to manipulate the DS segment register in order to access the seed variable in the Huge memory model. To eliminate this potential problem, we put the seed variable in the code segment after the **CSeg@**.

OTHER CONSIDERATIONS

Once the code works for all of the memory models that you might use, make a few more changes before you begin to speed up the code. For one thing, Turbo C expands all of the C **#define** statements, so that the assembler output shows no constant variables. To make the code easier to read and maintain, put back these constants as we did with **INCREMENT**, **MULTHIGH**, and **MULTLOW**. **MULTHIGH** and **MULTLOW** are the two 16-bit halves of **MULTIPLIER** (a 32-bit constant). We had to declare two 16-bit constants because the assembler works with 16-bit quantities.

A few other changes also make the routine easier to read. We removed unnecessary labels, such as **@1** and **@2**. More importantly, we commented the code.

continued on page 149

LISTING 1: RAND.C

```
/*-----*/
* filename - rand.c
*
* function(s)
*   srand - initializes random number generator
*   rand - random number generator
*-----*/

/*[]-----[]*/
/*
/*   Turbo C Run Time Library - Version 1.5
/*
/*
/*   Copyright (c) 1987 by Borland International
/*   All Rights Reserved.
/*
/*[]-----[]*/

#include <stdlib.h>

#define MULTIPLIER    0x015a4e35L
#define INCREMENT     1

static long   Seed = 1;

/*-----*/
Name          srand - initializes random number generator
Usage         void srand(unsigned seed);
Prototype in  stdlib.h
Description   see rand below
Return value  Nothing

*-----*/
void srand(unsigned seed)
{
    Seed = seed;
}

/*-----*/
Name          rand - random number generator
Usage         int rand(void);

Related
functions usage void srand(unsigned seed);
Prototype in  stdlib.h
Description   rand uses a multiplicative congruential random number
              generator with period 232 to return successive
              pseudo-random numbers in the range from
              0 to 215 - 1.

              The generator is reinitialized by calling srand with
              an argument value of 1. It can be set to a new
              starting point by calling srand with a given seed
              number.

*-----*/
int rand(void)
{
    Seed = MULTIPLIER * Seed + INCREMENT;
    return((int)(Seed >> 16) & 0x7fff);
}
```

RUNTIME MAINTENANCE

When we started this column nearly a year ago, the Runtime Source that we used did not include any source code maintenance batch files or support directories. In our first column, therefore, we showed our own batch files and directories. The Turbo C 1.5 Runtime Source (now shipping) does include batch files and a good directory structure, so we'll use them in future columns.

This newer Runtime also contains many new and changed source files. Don't worry, though—all of the modifications presented in previous columns are still valid. We suggest that you take the changes from those columns and fit them into the new source, rather than entirely replacing the newer routines with the changed ones. In this way, you don't lose any improvements in the new routines.

The version 1.5 Runtime uses the directory structure shown in Figure 1. Both the CLIB and MATH directories contain six object file subdirectories: SMALL, COMPACT, MEDIUM, LARGE, HUGE, and OBJ. The first five hold the objects for their respective memory models. The OBJ directory contains objects that are independent of any memory model. The libraries themselves go into the CLIB and MATH directories.

Be aware that while this new structure is easy to use, it consumes a great deal of disk space because it retains one version of every object file for each memory model.

```

\TURBOC\LIBRARY\INCLUDE (include files)
\TURBOC\LIBRARY\CLIB (C library sources)
\TURBOC\LIBRARY\MATH (Math library sources)
\TURBOC\LIBRARY\EMU (Floating point emulation sources)

```

Figure 1. The directory structure used for the Turbo C Runtime Library Source.

```

NEW_CLIB.BAT (Creates a new set of general C libraries)
NEW_MATH.BAT (Creates a new set of math libraries)
UPDASM.BAT (Assembles a routine and adds it to the library)
UPDC.BAT (Compiles a routine and adds it to the library)

```

Figure 2. A list of batch files mentioned in earlier columns that are now replaced by batch files provided with the Turbo C Runtime Library Source.

The Runtime also includes some batch files that help you manage your code. They replace the four batch files (in Figure 2) presented in our first column.

One of the main new batch files is CLIB.BAT, which recompiles all of the CLIB library source files for any memory model or for ALL of them, and then builds the specified library or libraries. CLIB does not update those libraries, however; instead, it builds them from scratch. It also lets you specify additional TCC options. For example, you can build the Large memory model library with the following command:

```
CLIB LARGE
```

To build all of the memory model versions, and at the same time define DEBUG for conditional compilations, use the command:

```
CLIB ALL -DDEBUG
```

CLIBRLIB.BAT also builds the libraries from scratch, but it assumes that you have already compiled all of the routines. It skips the compilation

and assembly stages and uses the existing objects. It also lets you specify a memory model or ALL.

CLIBRCMP.BAT is almost "the other half" of CLIBRLIB.BAT. It only compiles or assembles source files for a given memory model or ALL, and it does not touch the libraries. You can use it for both C and assembler files by specifying the file's extension separately in this command line format:

```

CLIBRCMP <model> <filename>
          <extension>
          <MASM or TCC switches>

```

To compile a single source file and update it in the library without rebuilding the entire library, use the new batch file CLIBREPL.BAT:

```

CLIBREPL <model> <filename>
          <extension> <library dir>
          <MASM or TCC switches>

```

The Runtime also includes batch files that perform similar functions for the MATH library. These new batch files and directories, along with all of the new version 1.5 source code, are valuable additions to the Runtime. We're glad to have it shipping. ■

—Mark L. Van Name
and Bill Catchings

LISTING 2: RAND.OUT

```

name rand
_text segment byte public 'code'
dgroup group _data,_bss
assume cs:_text,ds:dgroup,ss:dgroup
_text ends
_data segment word public 'data'
_data label byte
_data ends
_bss segment word public 'bss'
_bss label byte
_bss ends
_data segment word public 'data'
_Seed label word
dw 1
dw 0
_data ends
_text segment byte public 'code'
; Line 40
_srand proc near
push bp
mov bp,sp
; Line 41
mov ax,word ptr [bp+4]
xor dx,dx
mov word ptr dgroup:_Seed+2,dx
mov word ptr dgroup:_Seed,ax
; Line 42
a1:
pop bp
ret
_srand endp
; Line 66
_rand proc near
; Line 67
mov dx,346
mov ax,20021
mov cx,word ptr dgroup:_Seed+2
mov bx,word ptr dgroup:_Seed
call far ptr lxmula
add ax,1
adc dx,0
mov word ptr dgroup:_Seed+2,dx
mov word ptr dgroup:_Seed,ax
; Line 68
mov ax,word ptr dgroup:_Seed+2
and ax,32767
a2:
; Line 69
ret
_rand endp
_text ends
_data segment word public 'data'
_sa label byte
_data ends
_text segment byte public 'code'
public _srand
public _rand
_text ends
extrn LXMULA:far
end
)

```

LISTING 3: RAND.ASM

```

NAME RAND
PAGE 60,132

INCLUDE RULES.ASI ; Get the standard assembler macros

; Segment definitions

```

```

Headera ; Set the world up nicely
EXTRN LXMULA: FAR ; LXMUL is always a far procedure

; Declare some constants
INCREMENT equ 1
MULTHIGH equ 015AH ; High order word of the multiplier
MULTLOW equ 4E35H ; Low order word of the multiplier

IFDEF _HUGE_
SEEDOFF equ 8 ; Pick the appropriate argument
offset
ELSE
IFDEF LPROG
SEEDOFF equ 6
ELSE
SEEDOFF equ 4
ENDIF
ENDIF

CSega ; Start the code segment
_Seed dw 1 ; The seed value (initially 1)
dw 0

PubProc@ srand, __CDECL__ ; Declare a global C function
push bp ; Save the stack frame pointer
mov bp, sp ; Get a new one
mov ax, word ptr [bp+SEEDOFF] ; Get the requested seed
xor dx, dx ; Cast to a long
mov _Seed+2, dx ; Save it as the seed
mov _Seed, ax
pop bp ; Restore the old pointer
ret

EndProc@ srand, __CDECL__

PubProc@ rand, __CDECL__ ; Declare a global C function
mov dx, MULTHIGH
mov ax, MULTLOW
mov cx, _Seed+2 ; Get the seed
mov bx, _Seed
call far ptr lxmula ; Multiply the two 32 bit numbers
add ax, INCREMENT ; Add in the increment
adc dx, 0 ; Make it a 32-bit add
mov _Seed+2, dx ; Save the answer as the new seed
mov _Seed, ax

mov ax, dx ; Get the answer to return
and ax, 7FFFH ; Remove the sign bit
ret

EndProc@ rand, __CDECL__

CSegEnda

END

```

RUNTIME

continued from page 146

If you plan to keep the assembler version of a routine, this task, while not the most pleasant, is very important.

OPTIMIZING FOR SPEED

Our lone optimization is a minor one. In the following statement (which appears near the end of Listing 2), Turbo C fetches the return value from memory:

```
mov ax, _Seed+2
```

Since the value is already in the register DX, we can avoid that memory fetch. We replaced that statement with this faster one:

```
mov ax, dx
```

While this improvement is only a tiny one, you'll often have opportunities to improve a routine. For example, if you are writing code for an 80386-based machine, you can speed up this routine a great deal by replacing the call to the 32-bit multiply routine with the 80386's 32-bit multiply instruction. You can use this instruction, as well as many others, even if you are *not* running the 80386 in protected mode.

When you are done translating this routine, or any other routine, add your new version to the Runtime Library. The Turbo C 1.5 Runtime Library Source includes source code maintenance batch files (see the accompanying sidebar, "Runtime Maintenance"). Once you get used to them, you're off to the races with your new, faster routines. ■

Mark L. Van Name is a freelance writer. Bill Catchings is a freelance writer and a software engineer at Data General Corp.

Listings may be downloaded from CompuServe as OPTMIZ.ARC.

ADD TO THE POWER OF YOUR PROGRAMS WHILE YOU SAVE TIME AND MONEY!

CBTREE does it all! Your best value in a B+tree source!

Save programming time and effort.

You can develop exciting file access programs quickly and easily because CBTREE provides a simple but powerful program interface to all B+tree operations. Every aspect of CBTREE is covered thoroughly in the 80 page Users Manual with complete examples. Sample programs are provided on disk.

Gain flexibility in designing your applications.

CBTREE lets you use multiple keys, variable key lengths, concatenated keys, and any data record size and record length. You can customize the B+tree parameters using utilities provided.

Your programs will be using the most efficient searching techniques.

CBTREE provides the fastest keyed file access performance, with multiple indexes in a single file and crash recovery utilities. CBTREE is a full function implementation of the industry standard B+tree access method and is proven in applications since 1984.

Access any record or group of records by:

- Get first
- Get previous
- Get less than
- Get greater than
- Get sequential block
- Get all partial matches
- Insert key and record
- Delete key and record
- Change record location
- Get last
- Get next
- Get less than or equal
- Get greater than or equal
- Get partial key match
- Get all keys and locations
- Insert key
- Delete key

Increase your implementation productivity.

CBTREE is over 8,000 lines of tightly written, commented C source code. The driver module is only 20K and links into your programs.

Port your applications to other machine environments.

The C source code that you receive can be compiled on all popular C compilers for the IBM PC and also under Unix, Xenix, and AmigaDos! No royalties on your applications that use CBTREE. CBTREE supports multi-user and network applications.

CBTREE IS TROUBLE-FREE, BUT IF YOU NEED HELP WE PROVIDE FREE PHONE SUPPORT.

ONE CALL GETS YOU THE ANSWER TO ANY QUESTION!

CBTREE compares favorably with other software selling at 2,3 and 4 times our price.

Sold on unconditional money-back guarantee.

YOU PAY ONLY \$159 - A MONEY-SAVING PRICE!

TO ORDER OR FOR ADDITIONAL INFORMATION

CALL 1-800-346-8038 or (703) 847-1743

OR WRITE

NOW! Variable length records.

**NEW! --- Limited Time Offer.
Object Library for Only \$49!**



Peacock Systems, Inc., 2108-C Gallows Road, Vienna, VA 22180

ARCHIMEDES' NOTEBOOK

Rocketry with Eureka

David Eagle

Modeling is the process of performing experiments without really performing them. This can be critical in cases where the governing physical laws are well known, but the costs of "cut-and-try" analysis are prohibitive. Rocketry is a good example. Apart from using very small models, the costs and dangers of experimentation make it necessary to get as much information from analysis and modeling as possible.

The ability to quickly and accurately predict vertical rocket flight allows the analyst to assess the effect of different aerodynamic configurations, rocket engines, and launch site conditions on the "best" altitude performance. With Eureka, the analyst need only be concerned with formulating the equations that define a problem, *not* with the numerical methods required to solve the problem.

ROCKET.EKA is an equation file that demonstrates the ability of Eureka: The Solver to perform optimization with an inequality constraint. ROCKET determines the optimum launch mass of a single-stage rocket in order to maximize the total altitude attained by the rocket. This fundamental and important problem in aerospace engineering illustrates the unique interaction between aerodynamics, flight mechanics, and propulsion.

DENSITY MODEL

The atmospheric density model used in ROCKET compensates for "nonstandard" launch sites. These are sites that may not be at sea level, or rocket launchings that occur anywhere on unusually hot or cold days when variations exist in atmospheric density due to temperature. The user must specify the altitude of the launch site and the ambient temperature on the launch day. Launch site altitudes are positive for sites above sea level, and negative for sites below sea level. The density model is valid for rocket flight to altitudes of 11,000 meters (11 kilometers). Above that threshold altitude, a different model (not given here) must be used.

The atmospheric density at the launch site (as a function of the altitude and temperature) is computed with Equation 1, as shown in Figure 1.

FLIGHT MECHANICS

The rocket's altitude and velocity at the moment of rocket engine burnout are computed with Equations 2 and 3, as shown in Figure 2.

The altitude gained by the rocket during the coasting portion of the flight, and the coasting time, are determined using Equations 4 and 5, as shown in Figure 3.

Finally, the maximum altitude and total flight time are shown below in Equations 6 and 7.

$$X_{max} = X_{bo} + X_c \quad (6)$$

$$T_{total} = T_d + T_c \quad (7)$$

X_{max} is the value we are trying to maximize with Eureka: The Solver.

WORKING WITH ROCKET

ROCKET.RPT (Listing 1) is a Eureka report file containing both the equation file ROCKET.EKA and its solution file. The equation file begins with the inputs it requires and the outputs to be generated as part of the solution. The equation file specifies both the variable names used in the program and their units of measure (note the use of the metric system). After several unit conversions and constants are defined, the equation file lists nine items that must be defined by the user in order for Eureka to solve the equation file. Eureka recognizes these user-defined items as the input items specified earlier in the file. You can model different rockets and different launch circumstances by varying the values of these items.

It's very important that the user set the inequality constraint, and the initial guess for **massi**, equal to the propellant mass (in kilograms) of the rocket engine. This is shown in the following lines from the equation file:

```
massi > .0083
```

```
massi := .00833 [ kilograms ]
```

After all the input items are set to reasonable values, Eureka calculates the atmospheric density at the launch site and performs several intermediate calculations before finally solving for the maximum altitude reached by the rocket. Eureka iterates for the initial launch mass of the rocket **massi** through the following directive, which is defined at the beginning of the equation file:

```
$ max(alt.max)
```

The equation file can be easily modified to calculate a rocket's maximum altitude for a fixed value of initial mass by removing the **max** directive and inequality lines, and hardwiring the value for **massi**.

$$\rho = \frac{1.22557(1 - 2.2556913E-5h)^{4.256116}}{1 + \frac{(T - 59)}{518.67}} \quad (1)$$

where

- ρ = launch site density (kilograms per cubic meter)
- h = launch site altitude (meters)
- T = launch site temperature (degrees Fahrenheit)

Figure 1. The atmospheric density model used in ROCKET.EKA.

$$X_{bo} = (m \div k) \ln \left\{ \cosh \left[T_d \sqrt{k(F - mg)} \div m \right] \right\} \quad (2)$$

$$V_{bo} = \sqrt{(F - mg) \div k} \tanh \left\{ T_d \sqrt{k(F - mg)} \div m \right\} \quad (3)$$

where

- X_{bo} = burnout altitude (meters)
- V_{bo} = burnout velocity (meters)

Figure 2. The equations defining rocket altitude and velocity at burnout.

$$X_c = (m \div 2k) \ln(kV_{bo}^2 \div mg + 1) \quad (4)$$

$$T_c = \sqrt{(m \div kg)} \operatorname{atan} \left\{ V_{bo} \sqrt{k \div mg} \right\} \quad (5)$$

where

- X_c = coast altitude (meters)
- T_c = coast time (seconds)

and

- m = average rocket mass (kilograms)
- $k = \frac{1}{2} \rho C_d A$ (kilograms/meter)
- r = atmospheric density (kilograms/meter³)
- A = cross-sectional area (meters²)
- $F = \text{average thrust} = \frac{I_t}{T_d}$ (newtons)
- I_t = rocket engine total impulse (newton seconds)
- T_d = thrust duration (seconds)

Figure 3. The equations defining the altitude and velocity during coast time.

At the end of Listing 1 is Eureka's solution of the equation file. The message "Warning: root or log of negative number" indicates that Eureka encountered an undefined quantity (either the root or the log of a negative number) during its iterative solution of the equation file. This in no way corrupts Eureka's final solution of the equation file, as I have proven by duplicating the logic of the equation file in BASIC and noting that the resulting values are identical to those of Eureka. With the provided solution as a general guide for what is reasonable, vary the model and see how the outputs change. No programming is required—such is the power and purpose of Eureka: The Solver. ■

David Eagle is an aerospace engineer in Texas.

Listings may be downloaded from CompuServe as ROCKET.ARC.

LISTING 1: ROCKET.RPT

```

*****
Eureka: The Solver, Version 1.0
Friday April 15, 1988, 11:40 am.
Name of input file: C:\EUREKA\ROCKET.EKA
*****

; program "ROCKET.EKA"                      April 15, 1988

; For a given rocket engine and aerodynamic characteristics,
; this Eureka program determines the optimum launch mass of
; a single stage rocket which maximizes total altitude.

; Copyright (c) 1988 by David Eagle

; Input

; alt.site = altitude at launch site          ( meters )
; temp.site = temperature at launch site      ( degrees F )
; tburn = thrust duration of rocket engine    ( seconds )
; impulse = total impulse of rocket engine    ( newtons )
; mprop = rocket engine propellant mass      ( kilograms )
; diameter = frontal diameter of rocket      ( millimeters )
; cd = drag coefficient of rocket             ( non-dimensional )

; Output

; alt.bo = burnout altitude                   ( meters )
; vel.bo = burnout velocity                   ( meters per second )
; mass.bo = burnout mass                      ( kilograms )
; tcoast = coast time                         ( seconds )
; tflight = total flight time                 ( seconds )
; alt.max = maximum altitude                  ( meters )
; massi = optimum initial rocket mass         ( kilograms )

; maximize the total altitude with Eureka's "max" directive

$ max ( alt.max )

; define unit conversions

$ units
kilograms -> grams : x * 1000
meters -> kilometers : x / 1000
degF -> degC : 5 / 9 * ( degF - 32 )
$ end

; define the acceleration of gravity
; ( meters per second per second )

gravity = 9.80665

gravity := gravity [ meters/sec^2 ]

; -----
; NOTE: the next nine items are user inputs

; launch site altitude ( meters ) ,

alt.site = 100

alt.site := alt.site [ meters ]

; launch site temperature ( degrees F )

temp.site = 70

temp.site := temp.site [ degF ]

; rocket engine thrust duration ( seconds )

tburn = 1.2

tburn := tburn [ seconds ]

; rocket engine total impulse ( newton-seconds )

impulse = 5

impulse := impulse [ newtons ]

; rocket engine propellant mass ( kilograms )

mprop = .00833

mprop := mprop [ kilograms ]

; rocket frontal diameter ( millimeters )

diameter = 18

```

```

diameter := diameter [ millimeters ]

; drag coefficient ( non-dimensional )
cd = .321

; constrain the initial launch mass to a value greater than
; the rocket's propellant mass ( kilograms )
massi > .0083

; provide Eureka with an initial guess for the rocket's
; initial launch mass ( kilograms )
; "massi" is initially set equal to the propellant mass
massi := .00833 [ kilograms ]

; -----
; compute the atmospheric density at the launch site
; ( kilograms per cubic meter )
density = 1.22557 * ( 1 - 2.2556913E-5 * alt.site ) ^ 4.256116 / _
( 1 + ( temp.site - 59 ) / 518.67 )

density := density [ kg/m^3 ]

; calculate the rocket's average thrust ( newtons )
thrust = impulse / tburn
thrust := thrust [ newtons ]

; calculate the rocket's average mass ( kilograms )
mass = ( massi - .5 * mprop )
mass := mass [ kilograms ]

; calculate the rocket's average weight ( newtons )
weight = mass * gravity
weight := weight [ newtons ]

; determine the rocket's "total" aerodynamic coefficient
k = .5 * density * cd * pi() * diameter ^ 2 / 4e6

; compute burnout altitude ( meters )
alt.bo = ( mass / k ) * ln ( cosh ( tburn * _
sqrt ( k * ( thrust - weight ) ) / mass ) )
alt.bo := alt.bo [ meters ]

; compute burnout velocity ( meters per second )
vel.bo = sqrt ( ( thrust - weight ) / k ) * _
tanh ( tburn * sqrt ( k * ( thrust - weight ) ) / mass )
vel.bo := vel.bo [ meters/second ]

; compute burnout mass ( kilograms )
mass.bo = massi - mprop
mass.bo := mass.bo [ kilograms ]

; compute burnout weight ( newtons )
weight.bo = mass.bo * gravity
weight.bo := weight.bo [ newtons ]

; compute coast time ( seconds )
tcoast = sqrt ( mass.bo / ( k * gravity ) ) * _
atan2 ( vel.bo * sqrt ( k / weight.bo ) , 1 )
tcoast := tcoast [ seconds ]

; compute the total flight time of the rocket ( seconds )
tflight = tburn + tcoast
tflight := tflight [ seconds ]

```

```

; solve for the rocket's maximum altitude ( meters )
alt.max = alt.bo + ( .5 * mass.bo / k ) * _
ln ( k * vel.bo ^ 2 / weight.bo + 1 )
alt.max := alt.max [ meters ]

```

Solution:

Variables	Values
alt.bo	= 125.11504 meters = .12511504 kilometers
alt.max	= 546.24549 meters = .54624549 kilometers
alt.site	= 100.00000 meters = .10000000 kilometers
cd	= .32100000
density	= 1.1886383 kg/m ³
diameter	= 18.000000 millimeters
gravity	= 9.8066500 meters/sec ²
impulse	= 5.0000000 newtons
k	= .000048546694
mass	= .020707348 kilograms = 20.707348 grams
mass.bo	= .016542348 kilograms = 16.542348 grams
massi	= .024872348 kilograms = 24.872348 grams
mprop	= .0083300000 kilograms = 8.3300000 grams
tburn	= 1.2000000 seconds
tcoast	= 7.5213853 seconds
temp.site	= 70.000000 degF = 21.111111 degC
tflight	= 8.7213853 seconds
thrust	= 4.1666667 newtons
vel.bo	= 190.35449 meters/second
weight	= .20306971 newtons
weight.bo	= .16222501 newtons

Confidence level = 96.3%
All constraints satisfied.
Warning: root or log of negative number.

CRITIQUE

DESKTOP FOR PARADOX

Kallista, Inc.

600 South Dearborn Street, Suite 1611

Chicago, Illinois 60605

(312) 663-0101

\$45.00

Many Paradox users never experience programming with PAL, the Paradox Applications Language. This is a shame, because understanding PAL can dramatically increase your productivity as a Paradox user. Almost anything that you do repetitively might be better left to a small PAL program that can be invoked by a single keystroke. Because utility programs like this take time and inspiration to develop, they often go undone. Kallista, Inc.'s Desktop for Paradox is a collection of utility programs and help screens that are coded entirely in PAL—and they're only a keystroke away.

Desktop includes about 30 routines that are invoked by SETKEY assignments in INIT.SC, which is the script that Paradox looks for when first loaded. The Desktop scripts can also be incorporated into your own scripts. Desktop requires a hard disk, and Kallista recommends at least 640K RAM.

The well-organized 40-page manual is conversational in tone and includes many asides directed at the reader. The documentation includes an overview, a discussion of programming style, installation notes, and a reference section that describes each routine in detail. The author points out some anomalies in the Paradox keyboard map-

ping, as well as other minor Paradox inconsistencies that might otherwise go unnoticed.

Program installation is straightforward and well-documented, even to the point of showing how to place the files on a RAM disk for faster access. Paradox reassigns keys per INIT.SC; thus, when one of the reassigned keys (Ctrl- and Shift-function keys, or Alt-number keys) is pressed later, the desired utility activates. Pressing Alt-F1 brings up Desktop Help, which is a master menu of Desktop's available features.

The help screens give a full-screen display of commonly referenced pages in the PAL manual. These screens include syntax for **ValCheck** pictures, field assignments, **FORMAT** parameters, PAL keystroke assignments, and abbreviated menu commands. **PAL ERRORCODES**, automatic record-locking constraints, Paradox data types and expressions, and ASCII codes for commonly used keys are all available. These help screens can save innumerable trips to the massive PAL manual. However, they pop up only while you're using Paradox. As a result, some of their usefulness is lost if you EXEC out to an external editor (as many of us do).

Kallista suggests that you retain the use of the PAL editor and then hot-key to your own editor for heavy-duty work. However, I know very few people who feel that retaining the PAL editor is worthwhile. Although it's provided with basic cut-and-paste block operations by Desktop, the PAL script editor is probably more useful for designing forms and reports than for design-

ing PAL scripts. Another script-oriented routine, called **PALKEY-WORDS()**, generates the proper syntax for often-used expressions like **SHOWMENU**, **WAIT**, **IF..THEN..ELSE**, and so forth. This routine also places that syntax at the cursor position, which can be helpful for the novice who is not familiar with the PAL syntax. A box-draw program assigns border characters to the cursor keypad, and can be used in script, form, or report design to speed up otherwise tedious screen drawing. A keyboard-to-ASCII converter types the ASCII code for a pressed key directly into a script.

Desktop for Paradox contains several SideKick-like functions (although they are not memory-resident) that include pop-up calculators, a calendar, a filecard viewer, ASCII tables, and a mini-notepad. More unique to Paradox, and perhaps more helpful among the pop-up utilities, are a screen attributes table, a system information screen, a Paradox table status display, and a program that lets you change a Paradox directory by selecting from a table of available directories.

The programs in Desktop are more interesting from a coding standpoint than from a performance standpoint, because their performance can be sluggish. The code appears to run smoothly, and is professionally presented and commented. Since all of the source code is provided (along with permission to use portions in your own applications), Desktop's well-documented scripts go a long way toward educating new PAL pro-

continued on page 154

grammers. Kallista even acknowledges that the programs might not be the most efficient, and points to PAL's flexible coding style. If you like the way the code works, you can use it as is; if not, you're encouraged to improve on it and to contact the author.

Desktop's value will depend upon your experience level and programming style. It can offer something for everyone, from novice to Paradox power user. At \$45.00 and with a 30-day, no-questions-asked, money-back guarantee, Desktop is certainly a bargain. All else aside, having an opportunity to take a close look at someone else's PAL code is more than enough reason to purchase Desktop.

—Alan Zenreich

PEABODY 1.02

Copia International, Ltd.
1964 Richton Drive
Wheaton, Illinois 60187
(312) 665-9850
\$50.00 per database

Peabody, an online reference to several popular programming languages, contains one or more disk-based databases, plus an "engine" that accesses the databases. Peabody runs either in standalone mode or as a memory-resident "pop-up." Although the 550K database must remain on disk, an efficient indexing scheme makes information retrieval very fast.

Peabody displays information in overlapping windows, called *frames*. The program is invoked by pressing either of two possible hot-key sequences, or by using a "hyper-key" sequence. The Ctrl-tab hot-key sequence pops up Peabody's table of contents and allows you to select information by topic. An alternate hot-key sequence (leftshift-tab) re-displays the most-recently displayed frame. The "hyper-key" sequence (Alt-leftshift) reads the word that is located nearest to the screen cursor. If that word is one of Peabody's keywords, which number over 400, then the program displays the related information; otherwise, Peabody displays the table of contents.

Three different methods allow you to exit from Peabody. Pressing the Esc key backs Peabody out one frame at a time, and eventually returns you to your application. When the Ctrl-Esc sequence is pressed, Peabody erases all frames and exits immediately. The so-called "sticky key" sequence (Ctrl-backspace) returns you to your application, but leaves the most recent Peabody frame displayed on the screen (this makes the transfer of information from Peabody to your program much easier). With a second tap of the sticky key, that frame is erased.

The Peabody databases are quite complete. For example, the Turbo C database includes clear explanations of all Turbo C statements, operators, functions, and data types, plus information about the Turbo C compiler. In addition, Peabody can keep more than one database on-call. A different hyper-key can be assigned to each database, or a single hyper-key can initiate a search through all databases. Other nice features include the ability to list disk directories and to display the contents of disk files and blocks of memory. Peabody also displays several useful tables of ASCII characters, extended keyboard codes, and a list of C data types.

Peabody comes with several utility programs. One utility lets you change aspects of Peabody's setup, such as key assignment, display colors, and frame size and position. Another utility allows information to be added to a Peabody database. For example, if you've purchased a specialized function library for Turbo C, you can add information about that library to the Peabody database.

At this writing, Peabody databases are available for Turbo Pascal 3.0 and 4.0, Turbo C 1.5, and Microsoft C 5.0. By the time this article is published, databases for MS-DOS and the Microsoft Macro Assembler 5.0 should be available. Peabody requires a hard disk plus one floppy disk, DOS 2.1 or higher, and at least 256K of memory. In memory-resident mode, Peabody uses a minimum of 89K.

Despite Peabody's many strong points, some aspects of its operation could stand improvement. For example, frames pop up in a fixed screen location, and often obscure the section of source code on which you're currently working. Although a frame can be moved in order to uncover the display that lies below, this is a tedious procedure that requires pressing Scroll Lock and then using the arrow keys to move the frame a character at a time.

Peabody's video display is slow and uses screen space very inefficiently. When a topic includes more information than can be presented in a single frame, the program uses multiple frames that partially overlap. When several frames are displayed at the same time (**printf**, for example, uses 11 frames), much of the screen area is taken up with partially covered frames that obscure the underlying application information. It would be much better to use a single frame that allows data to be scrolled.

Another problem with Peabody is the lack of a cross-referencing system. When you're reading about one topic, you may want to quickly move to a related topic and then return to the original topic. For example, the Peabody section on the **fprintf()** function refers the user to the section on **printf()** for details on format strings; however, Peabody offers no easy way to move between these related topics. The hyper-key lets you move between topics only if the desired keyword appears in the current frame (and the cursor has to be moved, one character at a time, to that word). Otherwise, the main keyword index must be called and then scrolled through until the desired keyword is found.

Online databases of programming information are a great idea, and can be extremely useful. As it stands, Peabody is good. If the above-mentioned problems were corrected, it would be terrific. ■

—Peter Aitken

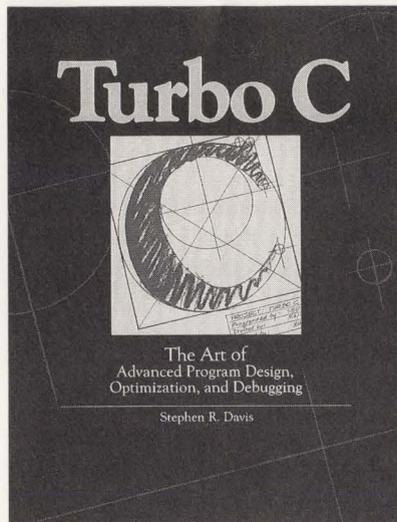
TURBO C: THE ART OF ADVANCED PROGRAM DESIGN, OPTIMIZATION, AND DEBUGGING.

Stephen R. Davis, M & T Books, Redwood City, CA: 1988, ISBN 0-934375-38-0, 439 pages, softcover, \$24.95, book and listings disk \$39.95.

Stephen R. Davis' book, *Turbo C: The Art of Advanced Program Design, Optimization, and Debugging*, is intended for programmers who use Turbo C on the IBM PC. Davis covers the additional topics that a Turbo C programmer needs to know in order to write programs that use the PC's facilities (MS-DOS, the BIOS, and the underlying hardware).

The book begins with an overview of the C language, then quickly moves into a comparison of Turbo C and Kernighan and Ritchie's C. These two sections are intentionally skimpy, since they cover material already presented in the Turbo C documentation. Even so, more material covering the Turbo C Integrated Development Environment (IDE) and command line utilities would be welcome. In particular, a detailed example of the command-line version of MAKE would be helpful.

After this overview, the book turns next to pointers and linked lists. These are glossed over by both Kernighan and Ritchie and the Turbo C manual, so their inclusion in Davis' book is worthwhile. Davis also covers pointers to functions (another useful programming feature that is unique to C) and linked-list integrity (which most C pro-



grammers learn about the hard way, after trying to debug a linked-list program). This section is well-written, and goes beyond the basics of the code in explaining the philosophy behind writing these types of programs in C.

Davis then covers the specifics of Turbo C programming on the PC. Three chapters peel away the PC's architecture layer by layer, beginning with MS-DOS, then moving to the BIOS, and finally going over the hardware. Rather than exhaustively covering each layer, the intent of these chapters is to provide suggestions and background information about using the large Turbo C library—but more information should have been included. Nonetheless, some good tips can be gleaned from the material. In particular, the chapter on PC hardware presents windowing functions that work directly with screen memory and do not cause flicker on CGA adapters. The MS-DOS chapter includes a good discussion of the `exec()` and `spawn()` functions.

This book explores two areas of keen interest to PC programmers who write commercial applications: optimizing Turbo C programs for maximum performance, and writing Terminate and Stay Resident (TSR) or "pop-up" programs. The optimization chapter offers good advice, and includes a section on rewriting the time-sensitive parts of a program in inline assembly language. This chapter also includes information about the optimizations already available within Turbo C, including the size, speed, and auto-enregistering compilation options available from the IDE.

Davis does a good job of explaining the basics of TSR programming, but he leaves out a few details. For example, he briefly covers how to write an interrupt 0x21 sentry that knows if DOS is active when the TSR program is invoked. (Since DOS is not reentrant, the programmer needs to know this in order to avoid making any DOS calls and crashing the program.) Unfortunately, Davis does not present any C or assembler code to perform this valuable function.

Davis pays a lot of attention to detail in his sample code. For instance, the function that converts a number to ASCII outputs octal and hex as well as decimal. The code is thoroughly commented and can be readily incorporated into a C programmer's toolbox.

On the whole, the book is well-written and informative, but it has one problem—it should include more detailed technical information on some topics. For example,

continued on page 156

continued from page 155

even though "debugging" is mentioned in the book's title, no mention is made about the use of popular debuggers such as CodeView or Periscope with Turbo C programs. There is no coverage of MS-DOS critical error and Control-Break interrupt handlers, which is an advanced topic that programmers need to know about.

But these are minor criticisms. If you crave more information than the Turbo C manuals provide, then by all means check out *Turbo C: The Art of Advanced Program Design, Optimization, and Debugging*. ■

—Marty Franz

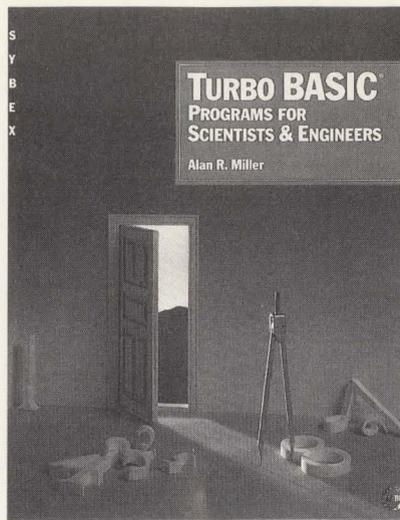
TURBO BASIC PROGRAMS FOR SCIENTISTS & ENGINEERS

Alan R. Miller, Sybex Books, Berkeley, CA: 1987, ISBN 0-89588-4291, 276 pages, softcover, \$19.95.

Mathematical analysis is an integral part of almost every area of science and engineering. While applications for some mathematical techniques are limited to one or two scientific specialties, other techniques are widely used across many scientific disciplines. Alan R. Miller's *Turbo Basic Programs for Scientists & Engineers* provides a library of Turbo Basic routines for solving some of these commonly encountered mathematical problems.

The first chapter examines the suitability of Turbo Basic for use with the types of numerical computations that are involved in scientific and engineering problems. Specifically, the precision and range of floating point operations, along with the accuracy of some functions, are evaluated and found to be acceptable. This chapter also briefly describes some features of Turbo Basic (the availability of sub-routines, multiline functions, and local variables) that make it more suitable than earlier BASICs for scientific computation.

The next two chapters introduce some fundamental statistical, vector, matrix, and random number operations. One chapter—the longest



one in the book—deals with the simultaneous solution of linear equations. Cramer's Rule, Gauss and Gauss-Jordan elimination, and the Gauss-Seidel iterative method are covered. This chapter also discusses ill-conditioned equations (using the Hilbert Matrix as an example), and includes sections on a simultaneous equation best-fit program, and on equations with complex coefficients.

Three chapters are devoted to curve fitting. The first is a general introduction, which presents the least-squares method for linear fits. The second chapter in this group presents a generalized least-squares program, dealing first with second-order polynomial equations that are fit by a parabola, then fitting to higher-order polynomials and non-polynomial equations. The third chapter in this section deals with nonlinear curve fitting. Sorting routines for the curve-fitting programs are developed as a separate chapter on data sorting methods that covers bubble, shell, and quick sorts.

The book includes a chapter on the solution of equations by Newton's method, which is an important technique with many applications. Another chapter deals with numerical integration and covers the trapezoid, Simpson's, and Romberg's methods. Finally, a chapter on "Advanced Applications" discusses the Gaussian function (and its complement), the gamma function, and the Bessel function.

The author's teaching background is evident in the pedantic style of the book. Each chapter

ends with exercises on that chapter's topics; the answers are given at the end of the book. For some of the numerical techniques, real-world examples are presented and solved. I tested most of the book's routines and sample programs, and found no flaws in either the programming or the algorithms.

It's difficult to determine the target audience at which this book is aimed. It is certainly not merely a "cookbook" of analysis routines intended for those who are already familiar with the numerical techniques and simply want a collection of tested BASIC routines to plug into their own programs. Because earlier chapters develop techniques and code that are used in later chapters, the text is intended to be read from beginning to end. The book resembles a textbook in that mathematical algorithms are described and explained before the BASIC code is developed. Explanations are somewhat superficial, however; while they help in code development, they come nowhere near a rigorous theoretical foundation for the described techniques.

The book is also inconsistent in terms of the knowledge that it expects on the part of the reader. The author sometimes assumes that the reader is familiar with differential and integral calculus; other times, however, much simpler mathematical concepts, such as the standard deviation, are explained in great detail.

Although it may not sound as though I like the book, my overall impression is actually positive. The weaknesses that I've mentioned may actually be the book's greatest strength. If *Turbo Basic Programs for Scientists & Engineers* were a pure cookbook of numerical routines, or if it included a complete theoretical exposition on the algorithms, it would better suit a much smaller audience. As it stands, this book may not meet anyone's needs perfectly, but it will be of some value to a relatively wide audience. Unfortunately, a program disk is not available for purchase (although the author provided one for this review), so you'll be in for a bit of typing. But even if you use only a couple of the routines in this book, your purchase cost will be justified. ■

—Peter Aithen

COMPUSERVE

The best online information about the Borland languages can be found on CompuServe's three Borland forums. Quite apart from providing the listings appearing in *TURBO TECHNIX*, the Borland forums contain many megabytes of useful utilities and source code in all Borland languages. Furthermore, some of the most interesting and knowledgeable people in the programming subculture hang out on CompuServe, providing an informal, online user group that is always in session. If you have a question, leave a message in the appropriate forum, and in almost every case someone will jump in with an answer.

Subscribing to CompuServe can be done through the coupon enclosed with every Borland product (which also includes \$15 worth of online time for your first month) or by calling CompuServe at (800) 848-8199. You'll need a modem and some sort of communications software that supports the XMODEM file transfer protocol.

How to access the Borland Forums on CompuServe:

TURBO TECHNIX listings for Turbo Pascal and Turbo Basic are available in DL 1 (Data Library 1) of the BPROGA Borland Programming Forum (**GO BPROGA**). Turbo C and Turbo Prolog listings are stored in DL 1 of the BPROGB Forum (**GO BPROGB**). Listings for Business Language articles are also available in DL 1 of the Borland Applications Forum (**GO BORAPP**). From the initial CompuServe prompt, type **GO <forum name>** or follow the menus. If you're not already a member of a forum, you must join by following the menus before you can download the listing files.

How to download TURBO TECHNIX code listings from CompuServe:

At the Functions prompt, type: **DL 1**. This will take you to the *TURBO TECHNIX* data library, where all listing files are stored. Listing files are archived using the ARC52 archiving scheme. You will need the ARC-E.COM program (available in DL 0 of BPROGA, BPROGB, and BORAPP) or one compatible with it to extract listing files from downloaded archives.

Magazine archive files are organized two ways: by article and by issue. In other words, there will be one .ARC file for every article that includes listings; and a single, larger .ARC file for each issue that contains all of the individual .ARC files for that issue. You can therefore download listings for individual articles, or download the entire issue's listings in one operation.

The all-issue files follow a naming convention such that NVDC87.ARC (which contains all listing archives from the November/December, 1987 issue), JNFB88.ARC (for the January/February, 1988 issue), and so on. The name of an article's individual listings archive file is given at the end of the article.

To download an archive file, bring up the DL 1 prompt and type:

DOW <filename>/PROTO: XMO

After pressing Enter, start your own communications program's XMODEM receive function. After you have completely received the file, you must press Enter once to inform CompuServe that the download has been completed. Once you have downloaded an archive file, you can "extract" its component files by invoking ARC-E.COM at the DOS prompt with:

C>ARC-E <filename> ■

YOUR SUBSCRIPTION

A free trial subscription to *TURBO TECHNIX* is yours for the asking when you register any of the Borland languages, language toolboxes, or Paradox. A subscription request card is packaged with each of those products—do fill it out and return it to be sure you get every issue. Don't forget your signature and the serial number of a qualifying Borland product—we need them to grant your free subscription.

If you have moved or changed your name, please use the card in this issue to update the information. If possible, attach the old mailing label to the card.

ONLINE AND BETWEEN COVERS

The following information describes two sources where you can learn more about Borland language products: On the Borland CompuServe forums, and in books now or soon to be in print. This issue, the CompuServe highlights are from the Turbo C/Turbo Prolog Forum, BPROGB. The files shown in this section are *not* related to articles in *TURBO TECHNIX*, but are of general interest to Turbo programmers. All files for Turbo C are stored in DL 4; all Turbo Prolog files are stored in DL 2. The books presented here are only a sampling. (We can't possibly list all published Borland-related books. Also, this listing reflects no judgment about the quality of any book.) For more information on these and other Borland-related books, contact the publishers or your local bookstore.

TURBO C:

INTER.ARC Uploaded: 04/14/88
Size: 73,728 bytes

A cornucopia of descriptions of interrupt and function calls on the IBM PC. A great thing to have on your hard disk if you don't have a Ray Duncan or Peter Norton book handy.

continued on page 158

TURBO RESOURCES

continued from page 157

CRT43.ARC Uploaded: 01/17/88

Size: 5,354 bytes

Replacement for the CRTINIT module in each of the five Turbo C Libraries. This enables Turbo C 1.5's console I/O functions to take advantage of the full screen in EGA 43-line and VGA 50-line modes.

3DLIB.ART Uploaded 02/29/88

Size: 62,782 bytes

Displaying solid objects in three dimensions. Requires Turbo C 1.5 to use all of the functions. You may obtain permission to use 3D TRANSFORMS commercially, along with complete source code for \$25.00.

TCPOPU.DEF Uploaded: 11/25/87

Size: 2,431 bytes

Mouse menu definition file that enables you to use a Logitech or Microsoft Mouse to drive Turbo C's Integrated Development Environment.

CHRLSTEXE Uploaded: 03/29/88

Size: 13,106 bytes

This program "decompiles" BGI font files (*.chr) into the Turbo C 1.5 BGI function calls that are necessary to draw the characters. This is an interesting file for BGI enthusiasts.

CHAIN.C Uploaded: 6/21/87

Size: 4,297 bytes

Outline of the techniques involved in trapping an interrupt in C, chaining to another interrupt handler, and setting the stack properly.

TURBO PROLOG:

STRING.PRO Uploaded: 09/11/87

Size: 3,823 bytes

This will help you interface Turbo C routines with Turbo Prolog. The file shows how to pass strings between the two languages.

LIGHT.ARC Uploaded: 03/23/88

Size: 5,716 bytes

Much like the Word Wizard package for Turbo Pascal, this file summarizes the Turbo Prolog calls that access the Turbo Lightning dictionaries and thesaurus. With the power of Turbo Prolog, very interesting games and/or applications can be built with this collection of Lightning calls.

CPINIT.ARC Uploaded: 3/1/88

Size: 2,688 bytes

The latest versions of CPINIT.OBJ, CPINIT.LIB, and CPINIT.DOC, which allow even greater flexibility in the interface between Turbo Prolog and Turbo C 1.5.

PROTUN.ARC Uploaded: 06/25/87

Size: 19,840 bytes

Contains files to assist you in linking Turbo Prolog programs with assembly language. Areas covered include compound objects, strings, symbols, and

lists. The files are well written, and will help any programmer who wishes to interface Turbo Prolog with either assembler or Turbo C.

TURBO PASCAL BOOKS

Turbo Pascal Express; Robert Jourdain; Brady Utilities.

Advanced Turbo Pascal; Herbert Schildt; Osborne/McGraw-Hill.

Turbo Pascal: The Complete Reference; Stephen O'Brien; Osborne/McGraw-Hill.

Turbo Pascal for BASIC Programmers; Paul Garrison; Que Corp.

Mastering Turbo Pascal 4.0; Tom Swan; Howard W. Sams & Co.

Complete Turbo Pascal 4.0; Jeff Dunte-mann; Scott, Foresman & Co.

Stretching Turbo Pascal; Kent Porter; Brady Books.

Turbo Pascal Programs for Scientists & Engineers; Alan R. Miller; Sybex, Inc.

Using Turbo Pascal Library Units; Namir Shammas; Wiley & Sons, Inc.

The Complete Turbo Programmer's Reference; Keith Weiskamp; Wiley & Sons, Inc.

TURBO C BOOKS

Using Turbo C; Herbert Schildt; Osborne/McGraw-Hill.

Advanced Turbo C; Herbert Schildt; Osborne/McGraw-Hill.

Turbo C: Memory Resident Utilities, Screen I/O and Programming Techniques; Al Stephens; MIS Press.

Turbo C, The Art of Program Design, Optimization and Debugging; Stephen Randy Davis; M&T Books.

Turbo C: The Complete Reference; Stephen O'Brien; Osborne/McGraw-Hill.

Turbo C Programming for the IBM; Robert LaFore; Howard W. Sams & Co.

Complete Turbo C; Strawberry Software; Scott, Foresman & Co.

Programming with Turbo C; Beverly and Scott Zimmerman; Scott, Foresman & Co.

Systems Programming in Turbo C; Michael Young; Sybex, Inc.

Turbo C At Any Speed; Richard Wiener; Wiley & Sons, Inc.

TURBO PROLOG BOOKS

Turbo Prolog Features for Programmers; Sanjiva Nath; MIS Press.

Using Turbo Prolog; Khin Yin; Que Corp.

Mastering Expert Systems with Turbo Prolog; Carl Townsend; Howard W. Sams & Co.

Turbo Prolog Primer; Dan Shafer; Howard W. Sams & Co.

Advanced Techniques in Turbo Prolog; Carl Townsend; Sybex Inc.

Introduction to Turbo Prolog; Carl Townsend; Sybex Inc.

Turbo Prolog Advanced Programming Techniques; Philip Seyer, Safaa Hashim; Tab Books.

AI Programming with Turbo Prolog; Keith Weiskamp, Terry Hengli; Wiley & Sons, Inc.

TURBO BASIC BOOKS

Using Turbo Basic; David Schneider, Frederick Moser; Osborne/McGraw-Hill.

Advanced Turbo Basic; Ken Knecht; Scott, Foresman & Co.

Turbo Basic, Programs for Scientists & Engineers; Alan R. Miller; Sybex Inc.

Introduction to Turbo Basic; Douglas Hergent; Sybex Inc.

The Power of Turbo Basic; Leon Wortman; Tab Books Inc.

PARADOX BOOKS

The Paradox Companion; Douglas Cobb, Steven S. Cobb, Ken E. Richardson; Bantam Books.

Paradox: The Complete Reference; James Keogh; Osborne/McGraw-Hill.

Using Paradox; George T. Chou; Que Corp.

Paradox for the Programmer; Nelson T. Dinerstein; Scott, Foresman & Co.

TUG

The national user group for Turbo languages is TUG, the Turbo User Group. TUG publishes a bimonthly journal called *Tug Lines* that contains bug reports, programming how-to's, and product reviews. Extensive public-domain utility and source code libraries are available to members. An optional multi-user BBS with file uploading/downloading, messaging, and teleconferencing is available to the public. Membership dues are \$24.00 US/year (including Washington State); \$28.00 Canada and Mexico; \$39.00 overseas.

TUG

PO Box 1510

Poulsbo, WA 98370

BBS: (206) 697-1151

LOCAL USER GROUPS

One of the best places to look for advice and face-to-face assistance with your programming problems is at a local user group meeting. Most user groups in the larger cities have special interest groups (SIGs) devoted to the most popular programming languages, usually with strong Turbo presences. We will be listing some of the largest and most active user groups in major urban areas across the country; obviously, there are thousands of user groups that we cannot list due to space limitations. If no listed group is conve-

nient to you, ask about local user groups at a local computer store or check with a faculty member at a high school or college with a computer curriculum.

BOSTON COMPUTER SOCIETY

Information: (617) 367-8080
BBS: (617) 227-7986
One Center Plaza
Boston, MA 02108

CAPITAL PC USER GROUP (DC)

4520 East-West Highway, Suite 550
Bethesda, MD 20814

CHICAGO COMPUTER SOCIETY

Information: (312) 942-0705
BBS: (312) 942-0706

HAL/PC (HOUSTON)

Information: (713) 524-8383
BBS: (713) 847-3200 or (713) 442-6704

NEW YORK PC USER GROUP, INC.

Information: (212) 533-6972
BBS: (212) 697-1809
40 Wall Street, Suite 2124
New York, NY 10005

PACS (PHILADELPHIA)

Information: (215) 951-1255
BBS: (215) 951-1863
PACS, c/o Lasalle University
Philadelphia, PA 19141

SAN FRANCISCO PC USERS GROUP

Information: (415) 221-9166
444 Geary Blvd, Suite 33
San Francisco, CA 94118

ST. LOUIS USERS GROUP

Information: (314) 968-0992
BBS: (314) 361-8662

TWIN CITIES PC USER GROUP

Information: (612) 888-0557
BBS: (612) 888-0468
PO Box 3163
Minneapolis, MN 55403

C:>CLASS.ADS

C:>CLASS.ADS is *TURBO TECHNIQ* magazine's display classified advertising section. We welcome to these pages all those who would like to take advantage of the special sizes and rates available for C:>CLASS.ADS—\$150 per column inch, with a 2-inch minimum. (A minimum ad, for example, measures exactly 2 1/16" wide by 2" long.) All C:>CLASS.ADS must be pre-paid and submitted in camera-ready form (black and white PMT or Velox) to:

C:>CLASS.ADS
TURBO TECHNIQ
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001

For additional information, please call Production Assistant Annette Fullerton at (408) 438-9321.

**Turbo C or TPascal 4.0
Complete data base
code in just 10 minutes!**

Draw & paint your screens, point out indexes & that's it! Generator has: B-tree file manager, Automatic indexing, Context sensitive help, Automatic programmer documentation.

Unlimited free support

\$289 - TP or TC / \$475 for both

30 day money - back guarantee

**Turbo Programmer
ASCII - (800) 227-7681**

PASCAL

- ▲ Convert Turbo Pascal (V3.X) to Turbo C!
- ▲ Saves You Hundreds of Hours!
- ▲ \$49 + S&H (US/Canada=\$5, Foreign=\$20)
- ▲ Foreign Bank Check, add \$30
P.O./C.O.D., add \$10
- ▲ Demo Disk = \$5

CHEN & ASSOCIATES, INC.

4884 Constitution Ave., Ste. 1E
Baton Rouge, Louisiana 70808
(504) 928-5765 (Inquiries) / 1-800-448-CHEN (Orders)

TURBO SOFTWARE

We have a large collection of the best Shareware & Public Domain for the Turbo Languages!

- Turbo Pascal 3.0 6 disks for \$25
- 4.0 4 disks for \$18
- Turbo Prolog 3 disks for \$14
- Turbo C 5 disks for \$21
- Turbo Basic 3 disks for \$14

3 1/2 disk format \$1 per disk extra.
All disks completely filled! Windowing Packages, Utilities, Examples, Tutorials, Enhancements, and more. Free 32 pg. catalogue with over 200 disks described. Each disk only \$4.50 or less. Free shipping! Visa/Master Card, C.O.D.

Computer Solutions

P.O. Box 354 • Mason, MI 48854
1-800-874-9375 to order
1-517-628-2943 for info & MI

**JAKE: A BREAKTHROUGH IN
NATURAL LANGUAGE SOFTWARE**

Create a natural language front end to your database, game, or graphics program! **JAKE** is a library usable with Turbo C for translating English queries and commands into function calls and data structures. **JAKE** offers context-sensitive semantic processing, while interfacing easily to any application and using <64K of memory. \$495 complete.

Sound too good to be true? Get our interactive demo for only \$10 and see.

CALL (408) 438-6922 **VISA, MC**

EKS English Knowledge Systems, Inc.
5525 Scotts Valley Dr. Suite 22
Scotts Valley, CA 95066

OPT-TECH SORT™

**The High Performance Sort/
Merge utility. Use stand-alone or
Call as a subroutine. Unlimited
filesize, multiple keys, record
selection & much more!**

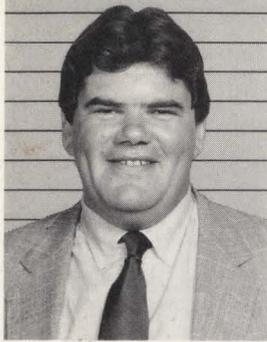
for MS-DOS \$149.

Call or write for more info.

Opt-Tech Data Processing
P.O. Box 678/Zephyr Cove, NV 89448
(702) 588-3737

ADVERTISERS' INDEX

Advertiser	Page No.	
Aker Corp.	19	Nostradamus C3
American Cybernetics	67	Opt-Tech Data Processing 159
ASCII	159	Osborne/McGraw-Hill 55, C4
Austin Code Works	131	Paradigm Systems 33
Black and White International	143	Peacock Systems 149
Blaise Computing	5, 7	Chen and Associates 159
Borland International 44-45, 63, 68-69, 77, 79, 95-97, 113, 119, 125-129, 137		Peter Norton Computing 15
Computer Solutions	159	Polytron Corp. 37
Disk Software	57	Powerline Software 29
English Knowledge Systems, Inc.	159	Programmer's Connection 9
Entelekon Software Systems	47	Programmer's Paradise 38
Lahey Computer Systems, Inc.	105	Quarterdeck Office Systems C2-1
Matrix Software	17	Software Artistry 11
MicroWay, Inc.	101	Softway, Inc. 117
		Sophisticated Software 21
		The Research Group 23
		Turbo Power Software 14, 103
		Vertical Horizons 65



PHILIPPE'S TURBO TALK

The technology of success

Philippe Kahn

Editor's note: The following was excerpted from an interview conducted by Odile Conscil of VSD France, a weekly newspaper in Paris, and is translated from the French.

VSD: Philippe, what sort of skills do you feel will be particularly rewarding in the 90s, in view of emerging technologies such as fast telecommunications, artificial intelligence, biotechnology, and so on?

PHILIPPE: That's difficult to say. All technologically competent people will probably do very well, but you can be sure that world-class programmers will never find themselves on unemployment. In fact, I believe that they will see their salaries rise far more quickly than the average. More and more, they are going to have to write quality programs, and that's a *human* problem rather than a simple matter of getting some predictable technology to work predictably.

VSD: What will be the most sought-after skills in the future?

PHILIPPE: To be considered the best, people will have to acquire highly specialized technological skills. Expertise in computer programming, biotechnology, superconductivity, etc., will all be very much in demand. Keep in mind that such specialized skills will be necessary to make this highly advanced technology accessible to less technical or non-technical users.

VSD: Can one still make a fortune in developing software? And if so, under what conditions?

PHILIPPE: Of course. The essential factor is innovation. But in practical terms, you have to have a first-class product. Then you need a workable strategy for bringing the product to market—and that is a challenge that *cannot* be underestimated, either. It is certainly more difficult today than it was five years ago. But, with talent, creativity, and a lot of work, anything is possible.

VSD: Will the current feverish interest in artificial intelligence be maintained through the next six or seven years, and if so, by whom?

PHILIPPE: I believe that you have to look at artificial intelligence as a part of the general culture of computer science. Artificial intelligence is not a standalone field, but one of the basic technical skills that all high-end software engineers must master. AI methods should be transparently integrated into the software of the future, rather than studied in isolation as they often are today.

VSD: To make it in the computer industry today, is it enough to be a clever engineer... or do you have to be a sort of generalist savant, comfortable in any field?

PHILIPPE: That depends on what you mean by "making it." If success means making a fortune, I believe that you had better be able to handle engineering problems and anything else that comes along. On the other hand, if success means simply making a good living doing interesting work, then technical competence alone may be enough. Let me emphasize that financial success should never be

a goal. It should be a consequence of getting a great job done. And no matter what, there is an incredible satisfaction in doing a great job.

VSD: Being that sort of financially secure Renaissance man, what kind of services are you yourself willing to pay for?

PHILIPPE: As I said, at Borland we are always willing to pay, and pay well, for the most competent and motivated people we can find. My collaborators are stars in their fields. To succeed, it is essential to surround yourself with brilliant people. You might say that we're always ready to make an exception for the exceptions.

VSD: What insights have your life in the United States and your travels around the world brought to mind concerning the evolution of businesses over time?

PHILIPPE: More than anything else, I have noticed that one essential quality tends to disappear: courage. The courage to work from 12 to 18 hours per day for months on end. The courage to keep from quitting the first time things go sour. We see these qualities in Japan, in Korea, and all around the Pacific Rim. Unfortunately, I meet too many people in our Western culture who want to be "lounge chair entrepreneurs." We need to redefine a real ethic of business and work. It's essential! ■

Turbo-Plus 5.0

\$99.⁹⁵

**SOFTWARE
AHEAD
OF ITS TIME**

Turbo-Plus™ Features:

- **Screen Painter**
- **Unit Libraries**
 - **Assembler Efficiency**
 - **I/O Code Generation**
 - **Window Code Generation**
 - **Window Management**
 - **Special Effects**
 - **Screen Support**
 - **System Integration**
 - **User Color Selection**
 - **Run-Time Dynamic Menus**
 - **Universal Menus**
 - **Window & Menu Compression**
 - **Transparents and Shadows**
 - **Keyboard Support**
 - **Cursor Support**
 - **Field I/O Routines**
 - **Reentrant Routines**
 - **Diagnostic Tools**
 - **File Handling**
 - **System Resources**
 - **Sound Effects**
 - **Critical Error Handlers**
 - **Automatic Directories**
 - **Sample Programs**
 - **Complete Pop-Up Help**
- **280 Page Illustrated Manual**

Nostradamus Inc.

3191 South Valley Street (Suite 252)
Salt Lake City, Utah 84109

(801) 487-9662

Data/BBS 801-487-9715 1200/2400,n,8,1

Visa, Amex, C.O.D., Check or P.O.

60-day satisfaction, money-back guarantee

Demo Diskettes and brochures available

Out of U.S. add postage

Nostradamus®

The Language Standard is **TURBO PASCAL 4.0**
The Enhancement Standard is **TURBO PLUS 5.0**

"Turbo Plus 5.0 gives every Program the professional touch. . . saves hours of coding. A must in my programming."

Mike Cushman • Former Editor, "PC World"

"After spending hundreds upon hundreds of dollars searching through many utilities and libraries, I must say that Nostradamus is my choice!"

Mr. Paul Mayer • ZPAY Payroll Systems • Franklin Park, IL

"I've tried most similar products on the market, Turbo-Plus with Screen Genie is clearly superior."

Dr. David Williamson • Chiropractic Health Services • Durnam, NC

"This is, without a doubt, the most powerful and easy to use programming toolbox that I have seen for the PC environment, and Turbo Pascal in particular."

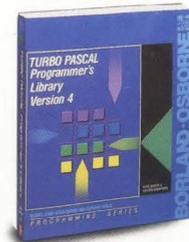
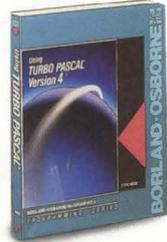
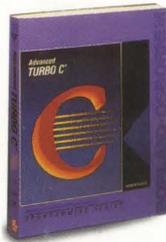
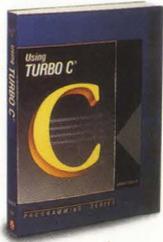
Mr. John Drabik • Geotron International, Inc. • Salt Lake City, UT

"Your products are first rate. Your Turbo-Plus products give my humble efforts a touch of class and speed that I would never have achieved otherwise. Obviously your products make Turbo Pascal a much better product."

Mr. L.M. Johnson • Saguro Technical Services • Cave Creek, AZ

SPECIAL TURBO SALE

Get \$5.00 Off Every Turbo Pascal 4 Book
Get \$3.00 Off Every Turbo C & Turbo Basic Book



Using Turbo C®

by Herbert Schildt

For all C programmers, beginners to pros, this excellent guide helps you write Turbo C programs that get professional results.

~~\$19.95~~ Paperback, ISBN: 0-07-881279-8, 431 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$16.95

Advanced Turbo C®

by Herbert Schildt

Unveils Turbo C power programming techniques to serious programmers. Covers Turbo Pascal conversion to Turbo C and Turbo C graphics.

~~\$22.95~~ Paperback, ISBN: 0-07-881280-1, 397 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$19.95

Turbo C®: THE COMPLETE REFERENCE

By Herbert Schildt

Covers Version 1.5

Programmers at every level of Turbo C expertise can quickly locate information on Turbo C functions, commands, codes, and applications — all in this handy encyclopedia.

~~\$24.95~~ Paperback, ISBN: 0-07-881346-8, 850 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$21.95



Turbo Pascal® THE COMPLETE REFERENCE

Covers Version 4
by Stephen O'Brien

The first single resource that lists every Turbo Pascal command, function, and feature, all illustrated in short examples and applications. Ideal for every Turbo Pascal programmer.

~~\$24.95~~ Paperback, ISBN: 0-07-881290-9, 814 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$19.95



Using Turbo Basic®

by Frederick E. Mosher
and David I. Schneider

Introduces Turbo Basic to novices and seasoned pros alike. Learn about the Turbo Basic operating environment and the interactive editor.

~~\$19.95~~ Paperback,
ISBN: 0-07-881282-8,
457 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$16.95

Using Turbo Pascal® VERSION 4

by Steve Wood

Build the skills you need to become a productive Turbo Pascal 4 programmer. Covers beginning concepts to full-scale applications.

~~\$19.95~~ Paperback, ISBN: 0-07-881356-5, 546 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$14.95

Advanced Turbo Pascal® VERSION 4

by Herbert Schildt

The power of Turbo Pascal 4 will be at your fingertips when you learn the top-performance techniques from expert Herb Schildt.

~~\$21.95~~ Paperback, ISBN: 0-07-881355-7, 416 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$16.95

Turbo Pascal® PROGRAMMER'S LIBRARY, SECOND EDITION

by Kris Jamsa and Steven Nameroff

Take full advantage of Turbo Pascal, and the newest versions of Turbo Pascal, with this outstanding collection of programming routines. Includes routines for the Turbo Pascal toolboxes.

~~\$22.95~~ Paperback, ISBN: 0-07-881368-9, 600 pp., 7 $\frac{3}{8}$ x 9 $\frac{1}{4}$

Borland-Osborne/McGraw-Hill Programming Series

\$17.95

For A Limited Time Only

ORDER TODAY! CALL TOLL-FREE 800-227-0900

Use Your Visa, MasterCard,
or American Express

McGraw-Hill Osborne McGraw-Hill
2600 Tenth Street
Berkeley, California 94710

Turbo Basic, Turbo C, and Turbo Pascal are registered trademarks
of Borland International. Copyright © 1988 McGraw-Hill, Inc.

McGraw-Hill
BORLAND·OSBORNE