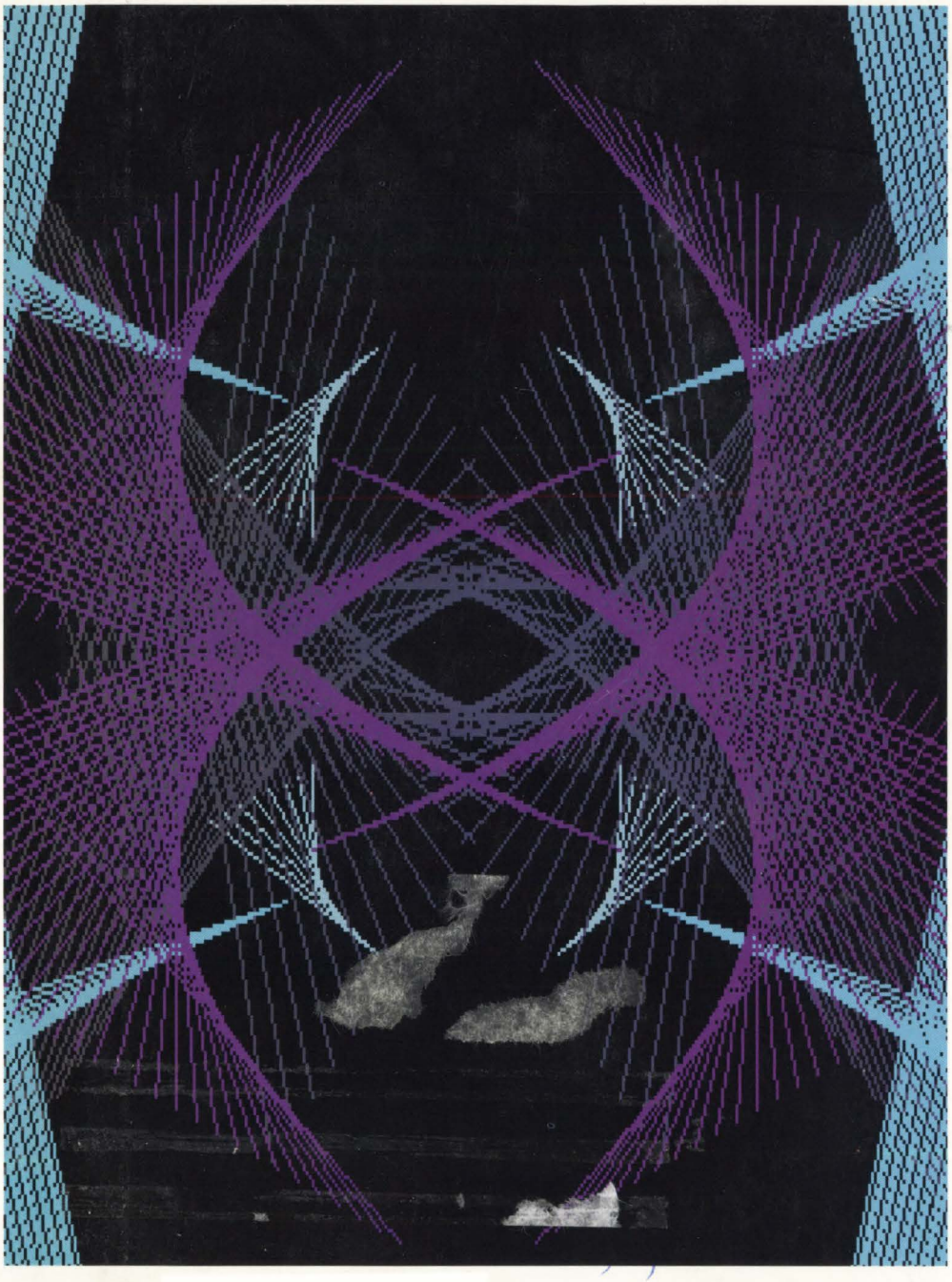
TURBO TECHNIX

THE BORLAND LANGUAGE JOURNAL • MAY/JUNE 1988 • VOLUME ONE NUMBER FOUR • $10.00



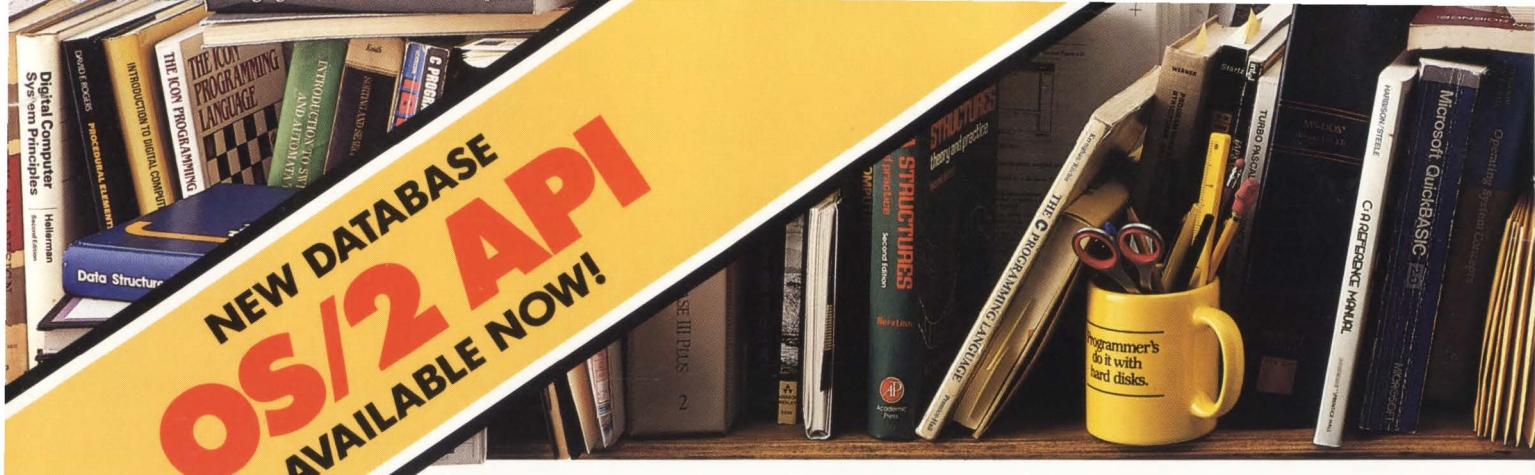
POWER GRAPHICS WITH THE BGI

A device-independent standard for Borland languages

The secrets of mouse programming

Access the DOS print spooler from Turbo Pascal

Directory searches in Turbo Basic



Finally. A pro for people who h

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 in the lower right-hand corner.

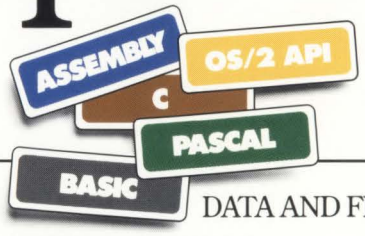
Which is designed to save you most of the time you're currently spending searching through the books and manuals on the shelf above.

It's one of a quintet of pop-up reference packages, called the Norton On-Line Programmer's Guides, that actually *gather your data for you*—on OS/2 Kernel API or your favorite programming language.

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

Not to mention a wealth of wisdom from the Norton team of top programmers. (*PC Week* used the words "massive" and "authoritatively detailed" to describe the information contained in the Guides. If you'd rather see for yourself, you might take a moment or two to examine the data box you just passed.)

You can, of course, find most of this informa-



DATA AND FEATURES

OS/2 KERNEL API (1M of data)
- Kernel API: Describes all OS/2 API services: DOSx, KBDx, MOUx and VIOx.
- Structure Tables: Lists all of the OS/2 data structures used in the Kernel API.
- Conversion Guide: DOS-to-OS/2 table shows which OS/2 calls replace DOS and ROM BIOS services.

ASSEMBLY (600K of data)
- DOS Service Calls: All INT 21h services, interrupts, error codes and more.
- ROM BIOS Calls: All ROM calls.
- Instruction Set: All 8088/86 instructions, addressing modes, flags, bytes per instruction, clock cycles and more.
- MASM: Pseudo-ops and assembler directives.

BASIC (270K each database)
- IBM BASICA, Microsoft QuickBASIC and TurboBASIC.
- Statements and Functions: Describes all statements and built-in library functions.

C (600K each database)
- Microsoft C and Turbo C: Describes the C language.
- Library Functions: Detailed descriptions of all functions.
- Preprocessor Directives: Describes commands, usage and syntax.

PASCAL—Turbo (360K of data)
- Language: Describes statements, syntax, operators, data types and records.
- Library: Describes the library procedures and functions.

FEATURES (all versions)
- Memory-resident—uses just 71K.
- Full-screen or moveable half-screen view, with pull-down menus.
- Auto lookup and searching.
- Tools for creating your own databases.
- More data: All five Norton Guides feature a variety of tables, including ASCII characters, line-drawing characters, keyboard scan codes and much more.
- Includes both OS/2 protected mode and DOS versions.

Designed for the IBM® PS/2® and PC families, and 100% compatibles. Available at most software

gramming tool ate manual labor.

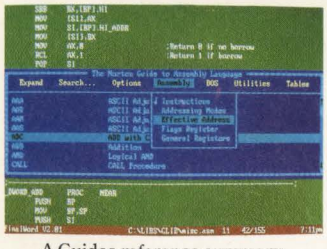
tion in the books and manuals on our shelf.

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

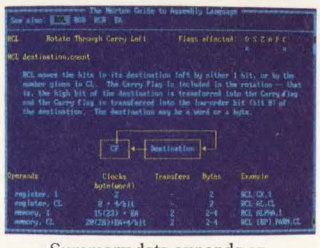
Instantly.

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

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



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.

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.

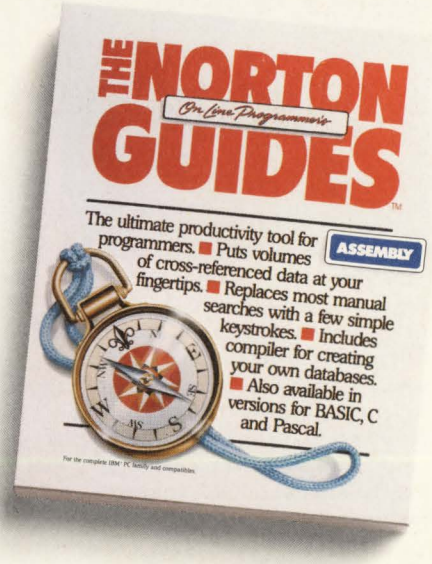
Which is why each version of the Norton Guides comes equipped with a built-in compiler—the same compiler used to develop the databases contained in the Guides.

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.

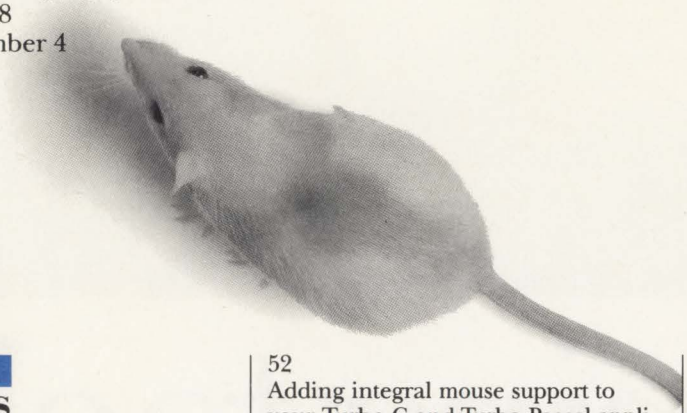
And ask for some guidance.



Peter Norton
COMPUTING

dealers, or direct from Peter Norton Computing, Inc., 2210 Wilshire Blvd. #186, Santa Monica, CA 90403. 213-453-2361. Visa and MasterCard welcome.

TURBO TECHNIX
The Borland Language Journal
May/June 1988
Volume 1 Number 4



FEATURES

TURBO PASCAL

12 Meet the BGI
Tom Swan

28 Plotting the Mandelbrot Set with the BGI
Fred Robinson

36 Using Units To Hide Data Structure Details
Marshall Brain

41 Interfacing the DOS Print Spooler
Duane L. Geiger

47 Exploring the Interrupt Vector Table
Jeff Duntemann

28

The Mandelbrot Set has been called the most complex artifact to emerge from pure mathematics. The Set itself is less interesting than its boundary, which is fractal in nature and infinite in length. Plotting the fractal boundary with BGI color graphics reveals landscapes of dazzling complexity that may be enlarged and explored without limit.

52
Adding integral mouse support to your Turbo C and Turbo Pascal applications has been something of a mystery—until now. Once you understand how the mouse works and how the mouse driver communicates with your software, the mystery is solved.

TURBO C

52 Mouse Mysteries, Part 1: Text
Kent Porter

68 ++, --
Bruce F. Webster

71 A Quattro Save Translator
Bruce F. Webster

79 A Memory-Resident Clock Utility
Ron Sires

TURBO PROLOG

84 Turbo Prolog 2.0: Intelligent Evolution
Michael Floyd

90 What's In a List?
Keith Weiskamp

94 Playing Cat and Mouse in Turbo Prolog
Safaa Hashim

84

Turbo Prolog advances the quest for the ideal AI applications language by offering dynamic databases, predicates with multiple arities, device-independent graphics through the Borland Graphics Interface, and many other new features.

TURBO BASIC

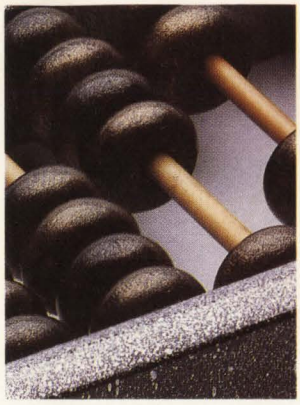
100 Variable Variations
David A. Williams

105 Instantaneous Help Screens
Ralph Roberts

106 Pick a File, Any File
Marty Franz

114 Plotter Support, Turbo-Style
William H. Murray and Chris H. Pappas

118 Background Color Magic
Mark Novisoff



120
Paradox's PAL language contains predefined functions for calculating loan payments, present value, future value, and net present value. Knowing how to use them can save a great deal of effort in creating PAL applications that handle dollars and cents.

BUSINESS LANGUAGES

120 PAL's Tools for Financial Information
Todd Freter

126 Building an Address Database in Sprint
Neil Rubenking

COLUMNS

4 BEGIN: Standardized Parts
Jeff Duntemann

131 Binary Engineering: How Loosely Are You Coupled?
Bruce Webster

136 Language Connections: A Blending of Media—and of Tools
Michael Floyd

140 Tales from the Runtime: Diving into Printf
Bill Catchings and Mark L. Van Name

160 Philippe's Turbo Talk

DEPARTMENTS

6 Dialog

150 Archimedes' Notebook: Choosing the Most Cost-Effective Lens Design
Milton C. Kurtz

154 Critique: Mach 2 for Turbo Basic
Marty Franz

155 Critique: C-scape with Look and Feel
Marty Franz

156 BookCase: Turbo C Programming for the IBM
Reviewed by Robert Alonso

157 BookCase: Artificial Intelligence Programming with Turbo Prolog
Reviewed by Sanjiva Nath

158 Turbo Resources

Cover: *At last, Borland's languages will speak graphically with one voice, through the Borland Graphics Interface (BGI). The BGI kernel detects the installed graphics display device at runtime, and loads an appropriate driver. Drawing points, lines, circles, ellipses, and polygons can be easily done without learning a completely new syntax for each Turbo language. The BGI's speed, color, and versatility will paint your graphics applications in a completely new light.*

TURBO TECHNIX

Publisher
Marcia Blake

Editor in Chief
Jeff Duntemann

EDITORIAL

Managing Editor
Michael Tighe

Technical Editor
Michael A. Floyd

Copy Editor
Pamela Dillehay

Technical Consultants
Brad Silverberg
David Intersimone
Dan Kernan
Charles Batterman
David Golden

DESIGN & PRODUCTION

Art Director
Karen Miner

Production Assistant
Annette Fullerton

Typesetting Manager
Walter Stauss

Typesetter/System Supervisor
Jeffrey Schwertley

Typesetters
Ron Foster
Jeanie Maceri

Typesetting Traffic
Charlene McCormick

Photographer
Bradley Ream

ADMINISTRATION

Purchasing
Brad Asmus

ADVERTISING

Assistant to the Publisher
Sheriann Glass

Advertising Sales Manager
John Hemsath
(408) 438-9321

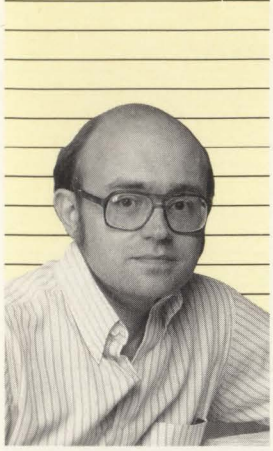
Western Region
Janet Zamucen
(714) 858-0408

New England/Mid-Atlantic Regions
Merrie Lynch
Nancy Wood
(617) 848-9306

South Region
Megan Patti
(813) 394-4963

TURBO TECHNIX (ISSN-0893-827X) is published bimonthly by Borland Communications, a division of Borland International, Inc., 4585 Scotts Valley Drive, Scotts Valley, CA 95066. *TURBO TECHNIX* is a trademark of Borland International, Inc.

Editorial and business offices: *TURBO TECHNIX*, 4585 Scotts Valley Drive, Scotts Valley, CA 95066. 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*, 4585 Scotts Valley Drive, Scotts Valley, CA 95066. POSTMASTER: Send address changes to *TURBO TECHNIX*, 4585 Scotts Valley Drive, Scotts Valley, CA 95066.



BEGIN

Standardized parts.

Jeff Duntemann

Back when I was 15 and impatient, I decided to build a car. I conned my uncle out of his broken self-propelled reel-style lawnmower that would only go backwards, scrounged some scrap 2 × 4s and pipe fittings from a Chicago vacant lot, took apart my old balloon-tire bike with the cracked frame, and I built a car. By bolting the wood frame onto the wrong side of the lawnmower, I worked around the frozen gearbox. With the frame slung *under* the balloon-tired front axle and a sawed-off bar stool as the front seat, it was, shall we say, idiosyncratic. But it ran up and down the alley at a reliable seven miles per hour until the neighbors complained.

Had I been less impatient, I could have grown up, bought a raft of machine tools, and in about 15 years hand-cloned a Porsche. In the meantime, however, I would still be taking the bus.

Most people don't build cars, and most people don't write software. Those of us who do, generally approach it as though we were hand-cloning a Porsche: starting at the top with the specification of the gleaming end-product, and then painstakingly creating a multitude of special-purpose components that, bolted together with care, produce that snazzy, Turbo-charged accounting package.

There's nothing wrong with this method. In fact, it's the *only* effective way to build competitive, commercial-quality applications. The other way to build software—by gluing together very high-level standardized software parts—has long been neglected, or else slandered under the term "prototyping," when in fact it's just the thing when all you really need to do is tool up and down the alley.

As research for a new book I'm working on, I've been building some standardized parts. One of them is an editor window that incorporates the Borland Binary Editor. The window only takes two parameters: the filename to be edited, and a number from 1 to 4 that specifies which fraction of the screen to use (whole, half, third, or a quarter). When called, the window saves the underlying text screen to the heap and then pops up as far from the cursor as it can.

Another part is a file picker window that only takes two parameters: a file spec and a number from 1 to 4. When called, it works exactly like the editor window except that it displays a bounce-bar menu of filenames that match the file spec.

Other standardized parts in the works include a date field editor, a string field editor, a phone number field editor, and (with luck) a 1200-baud telecomm window with only two parameters: a number to dial, and the 1-4 screen-portion value. Designing the parts takes some cleverness, but stringing them together takes very little. The resulting applications are "lumpy," and not especially flexible, but they work and can be created in 20 minutes flat.

The secret is this: *Keep the interfaces simple.* Choose reasonable defaults and live with them. Also, *have one part do one thing.* A universal field editor is far more trouble than separate ones for strings, dates, and integers.

What I'm reaching for here is called object-oriented programming, best known in specialized languages like C++ and Actor, but actually approachable in all of the Turbo languages. It's been a fascinating project, and I'll share my thoughts on it here from time to time, publishing some of my objects in the Turbo Pascal section of *TURBO TECHNIX*. The built-in overhead and inflexibility make object-oriented methods iffy for commercial applications, but for small or inhouse projects, these methods can save enormous amounts of work.

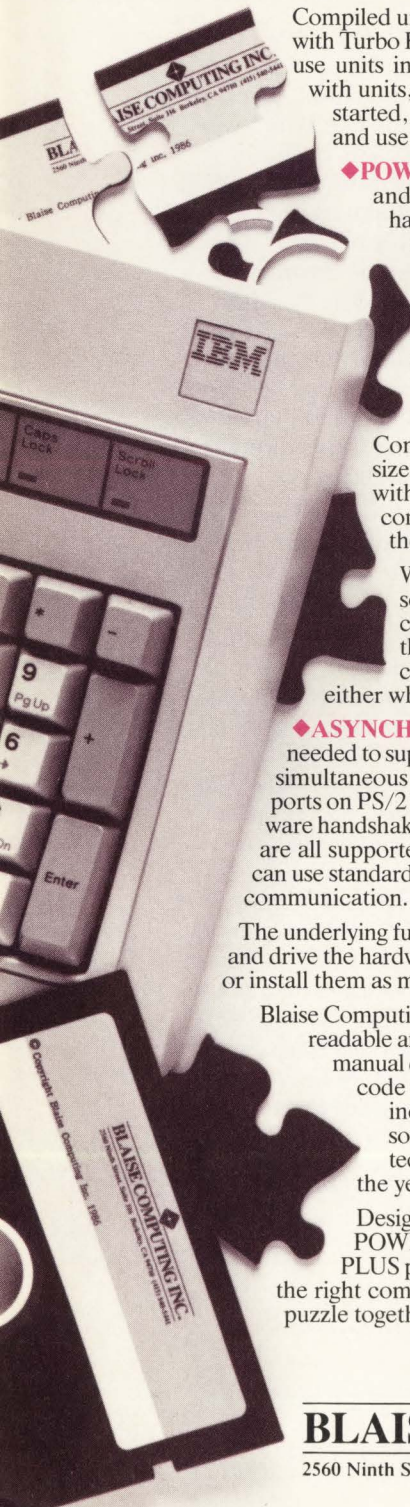
Remember, we're not talking racing stripes here. If you only need a go-kart, why build a Porsche? ■

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

Interlocking Pieces: Blaise and Turbo Pascal.

Now, for Turbo Pascal 4.0!

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.**

THE BLAISE MENU

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

YES! Send me the right pieces!
Enclosed is $_____ for _____ copies of _____.
☐ Please send me more information on your products.
CA residents add Sales Tax. Domestic orders add $4.00 for UPS shipping, $10.00 for Federal Express standard air.
Name: _____ Phone: (_____) _____-_____
Address: _____
City: _____ State: _____ Zip: _____
VISA or MC#: _____ Exp. Date: _____

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

BLAISE COMPUTING INC.
2560 Ninth Street, Suite 316 Berkeley, CA 94710 (415) 540-5441

DIALOG

Dead horses with wheels; and how do you pronounce "Dijkstra," anyway?

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
4585 Scotts Valley Dr.
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.*

INLINE TEXT

I would like to commend you on one facet of the production of your new journal. An article is started and then completed without "interrupts." Many publications give you the first page, followed by "continued on Page 500," which can be highly annoying. This is one type of **GOTO** statement that I can do without!

J. M. Anthony Danby
Raleigh, NC

When we say we're a structured magazine, we're not kidding. Once you start a TURBO TECHNIX *article, it's definitely:*

```
REPEAT
  Read(APage);
  TurnPage
UNTIL Done;
```

—Jeff Duntemann

. . . AND ONE WITH ALL FOUR LEGS

In the November/December 1987 issue of *TURBO TECHNIX*, in the article "Turbo Pascal 4.0 Arrives!" you say, "The single most important attribute of any software tool is *speed*." I say: "WRONG. The single most important attribute of any software tool is *correctness*."

Frederick D. Portoraro
Toronto, Canada

Hey, come on. When I hire a horse at a riding stable, I don't say, "Uh, could you make it a live *one, please?" When we buy tools, we expect them to be correct. Minor bugs make us complain; major bugs send the limping nag back to the vendor. You're right, of course, and it's a sad reflection on the history of our industry that correctness is not an unspoken assumption in every case. The field is littered with the corpses of companies who nailed wheels on dead horses and assumed we programmers wouldn't know the difference.*

—Jeff Duntemann

WHO'S A HYPERVISOR?

I read with interest your editorial entitled "DOS, The Understood" in the January/February 1988 issue of *TURBO TECHNIX*. You write on Page 5: "In a well-integrated 386 machine, DOS, plus a hypervisor like Windows/386 or PC-MOS 386, become pretty much everything that OS/2 is: A genuine multitasking OS with all the memory it needs."

This statement is erroneous. Unlike Windows/386, PC-MOS 386 is not a hypervisor—it is a full-fledged multiuser/multitasking operating system for 80386-, 80286-, and 8088-based machines.

Thanks for the opportunity to set the record straight.

Colleen Goidel
Public Relations Manager
The Software Link, Inc.
Atlanta, GA

Thanks, Colleen. The PC operating system scene is getting to be what the ancient Chinese philosophers would have called "interesting." Let me take a moment to sum it up: Windows/386, DesqView, and IGC's VM/386 are 386 hypervisors. All require DOS to operate, but DesqView has a mode that will operate correctly without a 386. PC-MOS/386 and Wendin-DOS are complete, DOS-compatible operating systems. PC-MOS/386 makes use of 386 memory management and virtual 86 partitioning; Wendin-DOS does not, although they claim to be working on it. And, of course, OS/2 is . . . OS/2. Everybody got all that?

—Jeff Duntemann

continued on page 8

Sophisticated User Interfaces in Minutes!

Put magic in your programs with turbo magic™

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.

continued from page 6

WHERE IT'S @

The first two issues of *TURBO TECHNIX* have been full of useful stuff. As a user of Turbo Pascal, several of the Toolboxes, SideKick, SuperKey, Reflex, and—most recently—Quattro, I think it's great to have a magazine that specializes in articles about Borland products.

The Quattro article in the second issue whetted my appetite for details of how to add features to Quattro. I hope the articles are within reach of a less-than-superstar Turbo Pascal programmer.

One thing I'd like to do with Quattro is add some @ functions that correct what I see as deficiencies in some of the existing functions. Specifically, I'd like to have versions of the @**COUNT**, @**AVG**, @**STD**, and @**VAR** functions that count only cells with numbers or formulas in them and ignore cells with text.

If these functions ignore text cells then you can include text column headings or dotted lines, etc., at the top and bottom of the blocks you want counted or averaged. That way, you can add rows of data anywhere in your current data columns and have correct counts and averages. SuperCalc 4's **COUNT** and **AV** functions ignore text cells, by the way.

If the future article that describes how to add @ functions to Quattro is not written yet, would you consider using as an example an @**COUNT** or @**AVG** function that passes over text cells? I would appreciate this very much!

Sam Baker
Columbia, SC

The idea is not to publish only one *article on building your own Quattro @ functions. We will be publishing custom @ functions regularly, along with other kinds of Quattro add-ins, in both Turbo C and Turbo Pascal. Like every other kind of programming, writing Quattro @ functions takes some close attention and practice, but much of the hard work is already done by virtue of Quattro's internals being available as callable routines. An introductory article on @ functions in our next issue will get you started, and I'll pass along your requests to our other authors who will be writing about @ functions in future issues.*

—Jeff Duntemann

DEFENDING BASIC

I really enjoyed both of Bruce Webster's articles in *TURBO TECHNIX*, January/February 1988, and they were (mostly) accurate and right on the money. His illustration showing how all of the fancy control structures are nothing more than **GOTO**s in disguise was masterful. However, I'd like to add a few comments.

In "Thinking in C," Bruce praises C for its conciseness in allowing statements such as

```
c = (a > b) ? a : b;
```

and

```
while ((line[indx++] =
  tupper(getc(infile))) != EOF)
```

as if this were some really nifty feature that BASIC does not provide. He then goes on to deride BASIC as having "very little form: it's just a collection of numbered statements." Further, in "Binary Engineering," Bruce concludes that **GOTO** should be used sparingly, and then only when necessary.

Let's face it folks—you can write terrible code in ANY language. There is nothing inherent in BASIC that would encourage poor programming, or make tracing a program's flow any more difficult than in another language. Real-world programs use procedures that call other procedures, and C or Pascal really offer no better protection against making a mess of things.

Legitimate complaints against BASIC might be that the BASICA interpreter is slow and has a clumsy line editor, or that most of the current compilers make .EXE files that are too large. But to say that BASIC encourages "spaghetti code" or that using **GOTO** is a poor practice simply isn't true. Often, attempting to avoid all use of a **GOTO** simply results in code that is harder to read.

For example, suppose you want to pause until a user presses a key. In BASIC it is often done like this:

```
WHILE INKEY$ = "": WEND
```

This is a perfect use of **WHILE..WEND**, and it avoids the need for both a **GOTO** and, more important, an extra line label. But suppose you also need to know which key was pressed. I've often seen it done this way

```
X$ = ""
  WHILE X$ = ""
    X$ = INKEY$
  WEND
PRINT "You pressed the " X$ "key"
```

where **X$** must first be cleared, just to insure that the **WHILE** will execute at least once. Here, using a **GOTO** is decidedly clearer, while creating less code in the bargain:

```
GetKey:
  X$ = INKEY$
  IF X$ = "" GOTO GetKey
```

This is a simplistic example for sure, but it illustrates what often results when a programmer attempts to shoe-horn "structure" into a situation where none is called for. With all due respect, Edsger Dijkstra seems like the programmer's equivalent of Archie Bunker.

BASIC lets you write very compact code too when you want. Similar to Bruce's examples, you might do this in BASIC

```
X = ABS(Y * (Z > 9))
```

or:

```
WHILE
  UCASE$(INPUT$(1,#1))<>CHR$(13)
WEND
```

BASIC also lets you call DOS and BIOS functions incorrectly, or overwrite the operating system—just like C does. Indeed, BASIC is a powerful and capable language, **GOTO**s and all.

Ethan Winer
Crescent Software
East Norwalk, CT

Thus flares up a debate that has raged for some years. I agree that it is possible to write atrocious code in C or Pascal; having taught an "Intro to

continued on page 10

Upgrade Your Technology

We're Programmer's Connection, the leading independent dealer of quality programmer's development tools for IBM personal computers and compatibles. We can help you upgrade your programming technology with some of the best software tools available.

Comprehensive Buyer's Guide. The CONNECTION, our new Buyers Guide, contains prices and up-to-date descriptions of *over 600 programmer's development tools by over 200 manufacturers*. Each description covers major product features as well as special requirements, version numbers, diskette sizes, and guarantees.

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. Upgrading your programming technology could be one of the wisest and most profitable decisions you'll ever make.

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.

ORDERING INFORMATION

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

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.

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.

Orders outside of Ohio are not charged sales tax. Ohio customers please add 5% Ohio tax or provide proof of tax-exemption.

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.

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

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 $10 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.

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

blaise products

	List	Ours
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 TOOLS PLUS/4.0	129	99

List $100 Ours $89

Peabody is a fast and flexible on-line reference utility with databases available for Turbo Pascal v 3 & 4, Turbo C, Microsoft C v 5,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	CALL
Paradox Network Pack *by Ansa/Borland*	995	725
Quattro: The Professional Spreadsheet	247	179
Reflex: The Analyst	150	105
Sidekick	85	65
Sidekick Plus ..... *New*	200	139
Superkey	100	68
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 Word Wizard	70	49
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 Graphix 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* ..... *New*	159	98
Essential Software Products *All Varieties*	CALL	CALL
Greenleaf Products *All Varieties*	CALL	CALL
Vitamin C *by Creative Programming*	225	149
VC Screen *Forms Designer*	100	79

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 5000 *by Microrim*	495	359
R:Base System V *by Microrim*	700	439

microsoft products

Microsoft C Compiler 5 *w/CodeView* ..... *New Version*	450	285
Microsoft COBOL Compiler *with COBOL Tools*	700	439
Microsoft FORTRAN *Optimizing Comp* ..... *New Version*	450	285
Microsoft Learning DOS	50	38
Microsoft Macro Assembler ..... *New Version*	150	99
Microsoft Mouse *Specify Serial or Bus*		
with Paint & Mouse Menus	150	99
with Microsoft Windows & Paint	200	139
with EasyCAD	175	119
Microsoft OS/2 Programmer's Toolkit ..... *New*	350	CALL
Microsoft Pascal Compiler ..... *New Version*	300	189
Microsoft QuickBASIC	99	69
Microsoft QuickC	99	69
Microsoft Windows	99	69
Microsoft Windows 386	195	129
Microsoft Windows Development Kit	500	319
Microsoft Word	450	285
Microsoft Works	195	129

List $150 Ours $129

PC/Forms is a high powered screen management package. PC/Forms takes the hassle out of screen design, screen management, and input data validation. Forms are created and maintained using the form editor; and processed at runtime via the PC/Forms runtime library. Using PC/Forms, the code required to process a complex screen can be reduced from several hundred lines to only a few.

nostradamus products

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

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	45
OPT-Tech Sort *by Opt-Tech Data Proc*	149	99
Peabody *by Copia Intl, Specify Language*	100	89
QBase *Relational Database by Crescent*	99	89
QuickPak *by Crescent Software*	69	59
Resident Expert *by Santa Rita, Specify Lang*	CALL	CALL
risC *Assembly Language by IMSI*	80	65

peter norton products

Advanced Norton Utilities	150	89
Norton Commander	75	55
Norton Editor ..... *New Version*	75	59
Norton Guides *Specify Language*	100	65
For OS/2 ..... *New*	150	109
Norton Utilities	100	59

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.

quinn-curtis products

DOS/BIOS & Mouse Tools *for Turbo Pascal*	75	67
MetraByte Data Acquisition Tools	100	89
Science & Engineering Tools	75	67

turbo pascal utilities

AZATAR DOS Toolkit *by AZATAR*	95	85
Btrieve *ISAM File Mgr by Novell*	245	184
Flash-up *by Software Bottling*	89	79
Flash-up Developer's Toolbox	49	45
MACH 2 *for Turbo Pascal by MicroHelp*	69	55
Overlay Manager *by TurboPower Software* ..... *New*	45	39
Screen Sculptor *by Software Bottling*	125	89
Speed Screen *by Software Bottling*	35	32
System Builder *by Royal American*	200	169
IMPEX *Query Utility*	130	115
Report Builder	180	159
TDEBUG 4.0 *by TurboPower Software*	45	39
Tmark *by Tangent Designs*	80	69
Turbo Analyst *by TurboPower Software* ..... *New*	75	59
Turbo Professional 4.0 *TurboPower* ..... *New Version*	99	79
TurboHALO *by IMSI, Specify Turbo C or Pascal*	95	75
TurboPower Utilities *by TurboPower*	95	78
TurboRef *by Gracon Services*	50	35

CALL for Products Not Listed Here

continued from page 8

Programming" class at a university for a few years made that all too clear. And I have seen some beautifully structured code written in BASIC. However, the proper question (to paraphrase C.S. Lewis) is how much worse the code written by the first group would be if done in BASIC, or how much better the code done by the second group would be if written in C or Pascal.

Ethan, I agree with your assertion that "attempting to avoid all use of a ***GOTO*** *simply results in code that is harder to read"—provided we're discussing BASIC, which has a very limited set of control structures. Given the richer and more complex set of control structures in C and Pascal, there is little need or use for* ***GOTO*** *statements, and they are not so much avoided as simply ignored.*

I will also agree that BASIC has improved vastly since its early days, mostly by shamelessly borrowing from Pascal, C, and FORTRAN such positive constructs as ***WHILE..WEND*** *statements, block* ***IF..THEN*** *statements, alphanumeric labels, unlabeled statements, and parameters to subroutines. The result is code that looks suspiciously like a mixture of Pascal, C, and FORTRAN. I just prefer to skip the new, improved BASIC and go with the real thing.*

As for Edsger Dijkstra, here's a quote (based on the old, unimproved BASIC) which should really make your day: "It is practically impossible to teach good programming to students that have had a prior experience in BASIC: as potential programmers, they are mentally mutilated beyond hope of recognition." (Selected Writings, *p. 130)*

—Bruce Webster

OS/2 ASCENDANT

Impossible for OS/2 to replace DOS, you say? Think again. Look at the history of Digital Research's CP/M. More than five years ago CP/M was *the* OS for microcomputers, mostly for Z80 machines. Then what happened? IBM rolled out its PC, initially offering two OS's: CP/M-86 and something brand new called PC-DOS.

Although CP/M was the standard of the day (supporting such staples as WordStar, SuperCalc, and dBase II, and even had a multitasking version, MP/M), IBM's practically giving DOS away at $70 per copy, along with its technical superiority and excellent documentation, rapidly made DOS the favorite OS of the PC. With its expensive pricing, its crude documentation, and user-hostile error handling, CP/M was a throwback to the Bad Old Days of computer systems that Mr. Duntemann seems to recall with distaste. For awhile, software developers supported both OS's. The final blow to CP/M-86, however, came from neither Microsoft nor IBM. It came from Lotus Development, which offered its 1-2-3 spreadsheet product in DOS format only. The success of 1-2-3 made CP/M compatibility irrelevant.

I believe that history may repeat itself. OS/2 and its successors are likely to become the PC standards that take us into the 1990s. As such, OS/2 will kill off plain old DOS, just like DOS finished off CP/M several years ago. DOS will become obsolete when software developers offer (and the PC market accepts) their next-generation blockbuster applications in OS/2 and Presentation Manager formats only.

Software developers—and PC pundits—should encourage, rather than discourage, the development of such needed standards as OS/2. At this stage, those who choose to ignore OS/2 do so at their own peril.

—Kenneth C. Kmack
Lilburn, GA

You called it, right there in your last paragraph. The Z80 industry was crippled out of the gate by its lack of hardware standards. IBM's PC blew the Z80 away because the PC provided a standard hardware environment, and with the Z80 went DRI's fortunes and CP/M in general. CP/M-86 and DOS 1.0 were just about identical. Both had awful documentation and cryptic error handling (which DOS still has). DOS was cheaper (I think more like $40, wasn't it?), but no better.

History will not repeat itself this time for several reasons. There are ten million 8088 machines out there, none of which will ever run OS/2. Tens of thousands more roll in from the Far East every day, and sell for $500 each. That's more than critical mass, that's supercritical.

OS/2 is by no means being given away. Cost does matter. Furthermore, the current DRAM shortage and price hikes could put a crimp in OS/2's early acceptance by making the necessary 2-4MB of RAM costly and hard to find. Presentation Manager is critical in many areas, especially software portability, and if PM does not find acceptance, OS/2 won't either. PM will be successful on the new generation 386/486 machines . . . but on the slower mainstream 286 boxes, it's still a very *open question.*

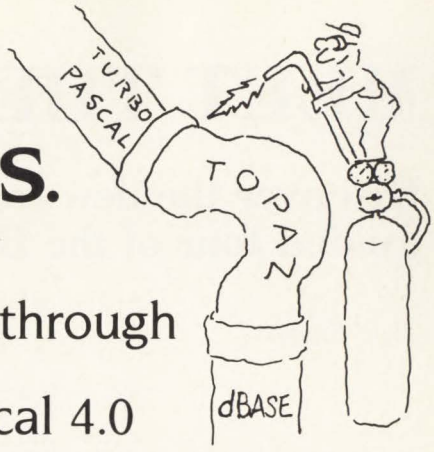
I'm flattered to see that I've finally made the rank of pundit. I haven't been so flattered since I was at a beer bust back at DePaul University, and an earnest young woman told me in disbelief, "You always talk in complete sentences*!"*

—Jeff Duntemann

Have you written a major application using one of Borland's programming products? If so, Borland Product Communications would like to hear about it. Send a letter describing your product, along with any marketing copy relating to the product, to:

Bill Burch
Borland International
4585 Scotts Valley Drive
Scotts Valley, CA 95066-0001

YOU'LL LOVE THESE UTILITIES.



SAYWHAT?! The lightning-fast screen generator

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.

TOPAZ. The breakthrough toolkit for Turbo Pascal 4.0

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 over 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 _____

THE RESEARCH GROUP

MEET THE BGI

Discover the new frontier of Turbo Graphics by taking a guided tour of the Borland Graphics Interface.

Tom Swan

SQUARE ONE

Until recently, IBM PC color graphics were about as exciting and colorful as dried mud. Graphics programmers were well advised to purchase a Macintosh II, Amiga, or Atari—anything but a PC.

Today, EGA (Enhanced Graphics Adapter) and VGA (Virtual Graphics Array) displays are putting a new face on PC graphics programming. Combined with the new Borland Graphics Interface (BGI), these display modes offer a fresh frontier for PC pioneers to conquer.

An easy way to join the graphics wagon train is to hitch Turbo Pascal 4.0 or Turbo C 1.5 to your trusty computer. These new compiler versions come equipped with interfaces to the Borland graphics kernel, a machine language wagon master that manages a herd of PC graphics display modes with ease.

This introductory guided tour of the BGI covers a wide range of terrain in order to provide you with a feeling for the scope of BGI graphics capabilities. At many of the stops, we'll just scratch the surface—further travel is left to you. Although the tour concentrates on Turbo Pascal's BGI interface, much of the following introduction to BGI graphics applies to Turbo C as well.

A KERNEL OF BEAUTY

The word *kernel* usually refers to the part of an operating system that runs in *supervisor mode*, resolving conflicts, granting access to disks and other devices, and making sure the dirty work gets done. In BGI graphics, the *kernel* is the central core that initializes the screen, draws lines and shapes, displays text, and performs other graphics-related functions.

The Borland graphics kernel is identical (or at least nearly so) in both Turbo Pascal and Turbo C. Each language provides a graphics interface, which allows you to use the items in the kernel. The *graphics interface* specifies the calling syntax of the kernel's routines, and defines the kernel's data structures. In Turbo Pascal, the graphics interface is stored in the file GRAPH.TPU; in Turbo C, the graphics interface is in the header file, GRAPHICS.H.

USING BGI GRAPHICS

To use BGI graphics in Turbo Pascal, construct your program like this:

```
PROGRAM MyGraphics;

USES Graph;

BEGIN
   { Graphics commands }
END.
```

The **USES Graph;** statement tells the compiler to load the GRAPH.TPU unit, making that unit's constants, types, variables, procedures, and functions available to your program. Turbo C programmers might want to print a copy of GRAPHIC.H for reference to the many BGI commands and structures. Turbo Pascal users can print GRAPH.DOC, a text guide to the compiled GRAPH.TPU unit. GRAPH.DOC also lists last-minute corrections to the BGI information in the *Turbo Pascal Owner's Handbook*.

THE DRIVER AND FONT FILES

BGI graphics driver code is stored in files ending in .BGI. Each .BGI file contains hardware-specific code for driving a particular kind of display device (see Table 1). When you run a graphics program, the correct .BGI file is loaded into memory; therefore, the driver file must be in a location known to the graphics program when it is run. You need the BGI interface code—either GRAPH.TPU for Turbo Pascal or GRAPHICS.H for Turbo C—only when *compiling* graphics programs. You need the .BGI driver files only when *running* these same programs.

Usually, it's best to store all .BGI files in the directory that contains the Turbo Pascal or Turbo C compiler. If you're short on disk space, though, use Table 1 as a guide while deleting all files except the file that you need for your own system.

Another kind of graphics file, ending in .CHR, contains data that allows the display of text in a variety of styles, or *fonts*. I'll explain more about fonts in a moment. For now, make sure you have the .CHR files (listed in Table 2) in your Turbo Pascal or Turbo C directory. As with .BGI files, you need .CHR files only when *running* graphics programs, not when compiling them.

INITIALIZING GRAPHICS MODES

The first step in every graphics program is to initialize a graphics mode using one of three methods: automatic, semiautomatic, or manual. The *automatic* method detects the type of graphics hardware on your system and then selects a graphics mode that produces the best results (usually a mode with the highest supported resolution). The *semiautomatic* method automatically detects the installed graphics hardware, but allows you to choose a display mode manually. The *manual* method lets you write programs that demand specific display hardware and graphics modes.

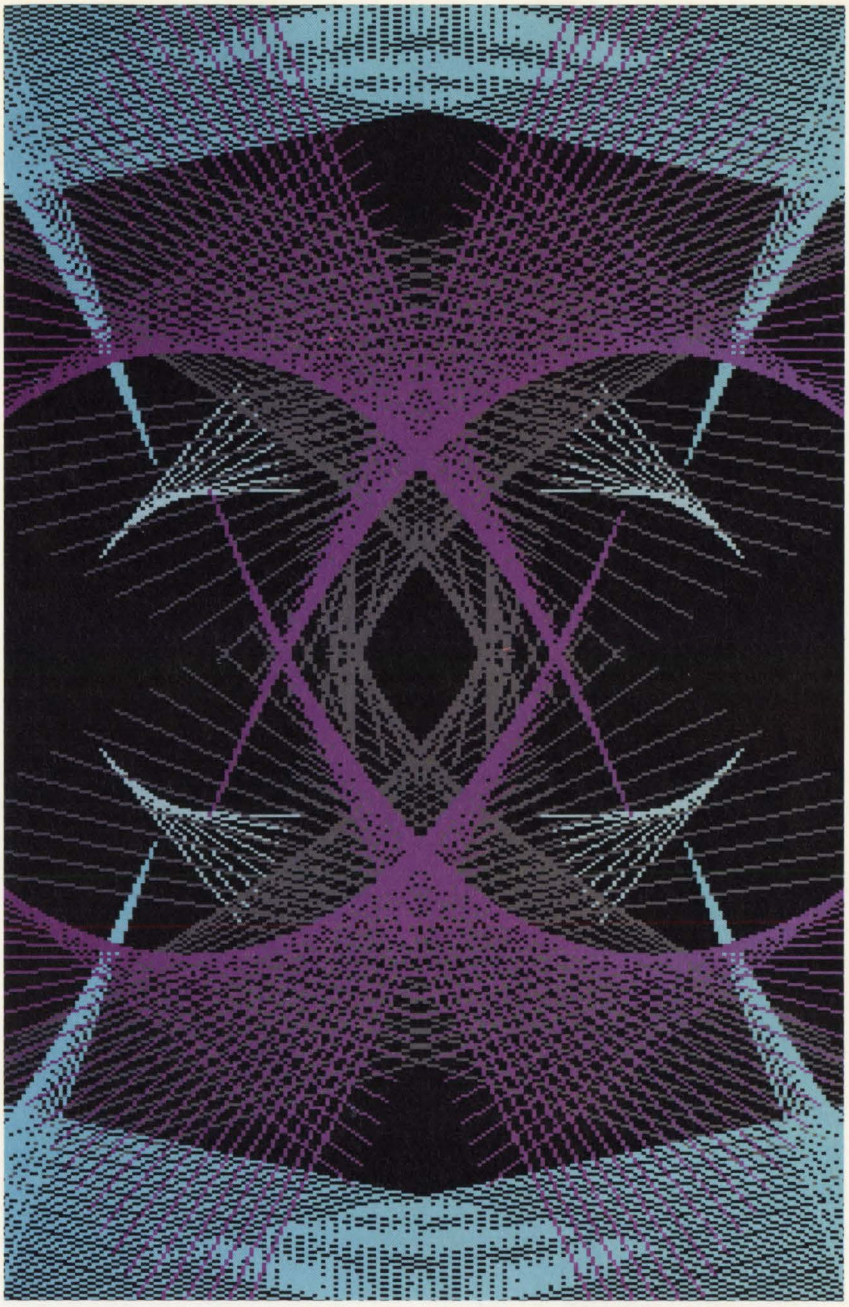
All three initialization methods require three global integer variables:

```
VAR
  grDriver,
  grMode,
  grError : Integer;
```

Integer **grDriver** represents a .BGI driver by number (see Table 1). The **grMode** variable represents the mode for this driver, thereby selecting a display resolution and, for some drivers, a set of colors. (Most drivers support several different modes. Refer to your Turbo language reference manuals and the files GRAPH.DOC and GRAPHICS.H for the mode constants you can assign to **grMode**.) The **grError** variable holds error codes returned by several graphics routines.

Automatic initialization. To automatically detect and initialize a graphics display, first assign the constant **Detect** to **grDriver**. Then call **InitGraph**:

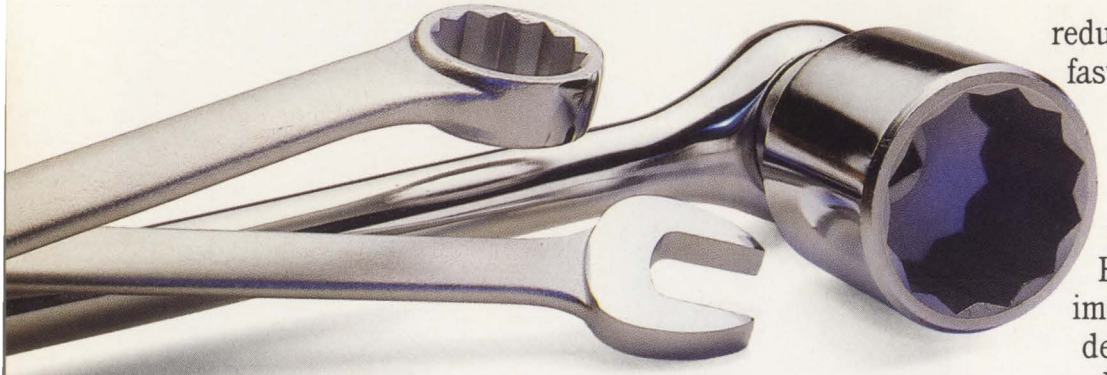
continued on page 16



DRIVER	NUMBER	DESCRIPTION	DISK FILE
CGA	1	Color Graphics Adapter	CGA.BGI
MCGA	2	Multicolor Graphics Array	CGA.BGI
EGA	3	Enhanced Graphics Adapter	EGAVGA.BGI
EGA64	4	64K EGA	EGAVGA.BGI
EGAMono	5	Monochrome EGA	EGAVGA.BGI
Reserved	6	none	none
HercMono	7	Hercules Monochrome Graphics	HERC.BGI
ATT400	8	AT&T 400-line graphics	ATT.BGI
VGA	9	Video Graphics Array	EGAVGA.BGI
PC3270	10	IBM PC 3270 Graphics	PC3270.BGI

Table 1. BGI graphics drivers.

Buy Our Tools, And We'



Introducing Emerald Bay. The breakthrough database server technology for developing single and multi-user applications. Emerald Bay provides your programs a common data storage and retrieval method which allows data to be transparently shared across multiple and diverse applications.

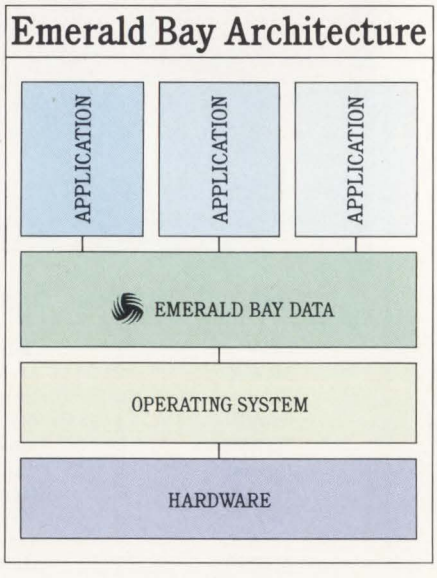
And when you buy one of our tools for "C," dBASE™ or Lotus® developers, we'll give you the personal engine—free. No royalties to pay, no licenses to sign.

Developed by Wayne Ratliff, the creator of dBASE, Emerald Bay is much more than just another DBMS product, it's an entirely new way to manage data. It's designed to provide an open platform for developing applications in several languages and environments, while Emerald Bay maintains data security, concurrency and integrity.

How The Engine Works

Before, data couldn't be readily shared between applications. But with Emerald Bay, PC applications each share a common data storage and retrieval method. And although the functions of the applications may vary widely, any one application can share another's data transparently; there is no data conversion or translation necessary.

When a PC is an intelligent workstation on a LAN, the Emerald Bay database server technology controls all data



security and integrity, including transaction logging with rollback. An application simply makes a request, which is sent to the engine. There, only the essential data is sent back to the workstation. The result is vastly reduced network traffic and faster data access times.

How You Work With The Tools

With the tools we provide, you can easily develop Emerald Bay applications immediately in your familiar development environment.

Emerald Bay technology handles the usually code-intensive management of data, so you can concentrate on what you do best—developing applications.

The ***Developers Toolkit for "C"*** includes well-documented, easy to use "C" libraries that give you the power to create advanced applications, without the effort usually associated with designing and coding a database "backend."

Eagle is Emerald Bay's sophisticated dBASE-like programming language. As the logical evolution of database language, Eagle introduces advanced features, routines and language components, including a compiler, network commands, user-defined functions in "C" and Assembly and automatic index maintenance.

Summit is an "add-in" database management system for Lotus 1-2-3, which gives you sophisticated data manipulation and analysis commands. All three of Emerald Bay's development tools come with the Core Components which include Report Writer, Forms Generator,

ll Give You The Engine.

an Import/Export facility and the Database Administrator.

The ***Emerald Bay Database Server*** is the heart of the multi-user Emerald Bay technology. Its client/server architecture is superior to current implementations of LAN/DBMS products, and increases total system throughput, while reducing network traffic and access times.

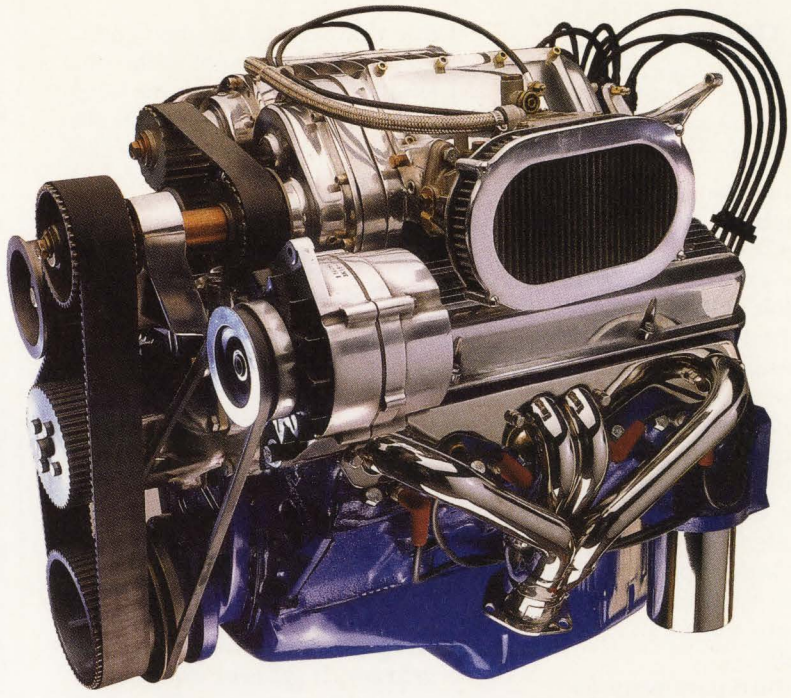
Finally, while providing a path to other operating systems such as OS/2, Macintosh and UNIX, Emerald Bay is a microcomputer-based technology that optimizes your *current* hardware investment.

Free Technical Seminars

We're hosting a series of free Emerald Bay Technical Seminars during April and May in cities across the country. It's your chance to see Wayne Ratliff demonstrate the capabilities of Emerald Bay in person, as well as get some practical experience with the technology yourself.

Call us toll-free at 1-800-777-2027 (and ask for Sandra) for the date and location of the seminar nearest you. Space is limited, so be sure to reserve your seat today.

Emerald Bay. Advanced database server technology. Available *now.*



Emerald Bay Engine Specifications

Data Storage

- Max. databases: No limit
- Max. tables per database: 1000
- Max. fields per table: 800
- Max. field width: 512 characters
- Max. records per table: No limit
- Max. width of records: 10,000 bytes (no limit on ext. fields)
- Max. open databases: 7 (MS-DOS limitation)

Index Storage

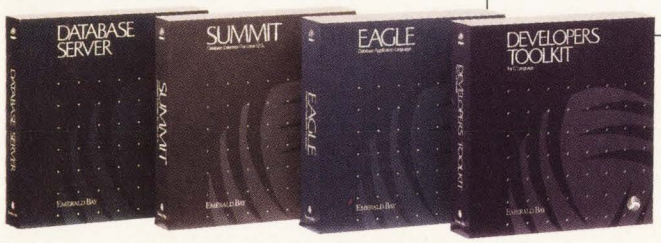
- Composite keys supported
- Mixed data type keys allowed
- Keys of up to 100 bytes in length
- Automatic index maintenance
- Ascending and descending keys
- Case independent keys
- Automatic table indexing on record number

Security And Integrity Features

- Access permissions by Read, Write, Delete, Add and Grant
- All five access permissions work on tables and objects
- Read, Write and Grant access permissions operate at field level
- All data other than binary fields can be encrypted
- Transaction logging, with commit and rollback functions
- Full security functions at field and table level
- Optional data encryption at field level

System Requirements

- MS-DOS 3.1 or greater
- Network database server or Single-user computer: PC XT, AT, PS/2 or 386 compatible, 640K, Hard Disk
- Workstation on LAN: PC, XT, AT, PS/2 or 386 compatible, 640K
- NetBIOS compatible networks supported



EMERALD BAY

MIGENT™

865 Tahoe Blvd., Call Box 6, Incline Village, NV 89450

Trademarks pending: Emerald Bay, Eagle, Summit (Migent, Inc.)

FONT NAME	NUMBER	TYPE	DISK FILE
DefaultFont	0	Bit map	none
TriplexFont	1	Stroked	TRIP.CHR
SmallFont	2	Stroked	LITT.CHR
SansSerifFont	3	Stroked	SANS.CHR
GothicFont	4	Stroked	GOTH.CHR

Table 2. BGI fonts.

MEET THE BGI

continued from page 13

```
grDriver := Detect;
InitGraph( grDriver, grMode, '' );
```

The first two parameters to **InitGraph** are the integer variables, **grDriver** and **grMode**. Because **grDriver** equals **Detect**, **InitGraph** automatically determines the type of graphics hardware installed on your computer, loads the appropriate .BGI driver, and selects a default display mode. **InitGraph** returns values in **grDriver** and **grMode** that represent the selected display driver and mode. You'll want to preserve these values, especially if you plan to switch between text and graphics screens during program execution.

The third parameter to **InitGraph** is a string specifying the directory path where you store your .BGI driver files. Use a null string (two single quotes with no space between them) if your .BGI files are in the same directory as the running program. If you store your .BGI files elsewhere (for example, in C:\TPAS\GRAPHICS), pass the pathname to **InitGraph**:

```
grDriver := Detect;
InitGraph( grDriver, grMode,
 'C:\TPAS\GRAPHICS' );
```

If all goes well, the screen should now be clear and ready for graphics commands. To be certain of this, check function **GraphResult**, which returns an error code for the most recent graphics operation (see Table 3). All error values are negative except for 0, thus indicating that no error has occurred.

GraphResult resets itself and, therefore, returns a valid number only *once* after a graphics operation that returns an error code. To preserve the error code value, assign **GraphResult** to the integer variable **grError** *immediately* after a graphics operation. To display an English language error message rather than a cryptic error code number, pass **grError** to string function **GraphErrorMsg** in a **Writeln** statement. **GraphErrorMsg** inserts filenames in the empty parentheses for error codes such as –3 and –4.

Figure 1 lists the full automatic initialization sequence, complete with error detection. Because **grDriver** equals **Detect**, **InitGraph** automatically detects and initializes a graphics display mode, loading the appropriate driver into memory. After **InitGraph**, assign **GraphResult** to integer variable **grError**. If **grError** does not equal the constant **grOk** (see Table 3), then an error occurred during initialization. If this is the case, pass **grError** to **GraphErrorMsg** in a **Writeln** statement to display an error message along with your heartfelt regrets. If no error is detected, you're ready to roll.

Semiautomatic initialization. The semiautomatic initialization sequence is a variation on the automatic initialization theme. First, call procedure **DetectGraph** with two integer variable parameters:

```
DetectGraph(grDriver, grMode);
```

After **DetectGraph** executes, **grDriver** and **grMode** contain values representing the driver and display mode values that **InitGraph** returns when automatically detecting and initializing graphics modes. After inspecting **grDriver**, you can change **grMode** to a different value, thereby selecting an alternate mode for the detected driver by calling **InitGraph**.

For example, let's use the initialization sequence in Figure 2 to select 640 × 200-resolution CGA or MCGA modes. After executing **DetectGraph**, assign an appropriate mode constant to **grMode**. In this case, assign either **CGAHi** or **MCGAMed**, which selects an alternate mode instead of the usual defaults. Now, when **grDriver** does not equal either **CGA** or **MCGA**, the program halts with an error message. Otherwise, **grDriver** and the modified **grMode** are passed to **InitGraph**, completing the semiautomatic initialization.

Manual initialization. The third and final way to initialize graphics is to forego automatic device detection altogether and to assign values directly to **grDriver** and **grMode** for specific graphics hardware. To initialize EGA 640 × 350 16-color, 2-page graphics, write the following code sequence:

CONSTANT	ERROR CODE	MESSAGE
grOk	0	No error
grNoInitGraph	–1	(BGI) graphics not installed
grNotDetected	–2	Graphics hardware not detected
grFileNotFound	–3	Device driver file not found ()
grInvalidDriver	–4	Invalid device driver file ()
grNoLoadMem	–5	Not enough memory to load driver
grNoScanMem	–6	Out of memory in scan fill
grNoFloodMem	–7	Out of memory in flood fill
grFontNotFound	–8	Font file not found ()
grNoFontMem	–9	Not enough memory to load font
grInvalidMode	–10	Invalid graphics mode for selected driver
grError	–11	Graphics error
grIOerror	–12	Graphics I/O error
grInvalidFont	–13	Invalid font file ()
grInvalidFontNum	–14	Invalid font number
grInvalidDeviceNum	–15	Invalid device number

Table 3. BGI error codes and messages.

```
grDriver := EGA;
grMode   := EGAHi;
InitGraph(grDriver, grMode,'');
```

With this kind of initialization, the program runs only on hardware that supports this particular EGA multipage mode. (One page equals the amount of memory required to hold a single graphics screen. Some drivers and modes support multiple pages; others support only one.)

Be careful when manually initializing graphics. On some systems, selecting nonexistent modes can lock up the computer, forcing you to reboot. For these reasons, use automatic and semiautomatic initialization methods whenever you can—they help you write programs that run on the widest possible range of PCs.

TOOLS OF THE BGI ARTISAN

With initialization out of the way, you're ready to begin using all of the BGI's many graphics routines. There's more to BGI graphics than I can possibly cover in a single article, but the following will give you a running start.

Listing 1 (RANDOMGR.PAS) lets you experiment with BGI graphics by using a "replaceable" procedure to perform the graphics drawing activity. Procedure **DoGraphics** contains a **REPEAT..UNTIL** loop, which cycles until you press a key. The **ReadKey** statement just before the end of **DoGraphics** clears this keystroke from memory.

The actual drawing commands are inside the **REPEAT..UNTIL** loop. In the version of RANDOMGR shown as Listing 1, two commands select colors and then draw colored lines at random, quickly filling the screen with an assortment of colored lines. To slow the action, insert the statement **Delay(150);** between **REPEAT** and **SetColor**. For debugging, you can use similar delays in order to watch complex graphics sequences display themselves to the screen in slow motion.

Throughout this article, you'll encounter replacement **DoGraphics** procedures consisting of article figures that draw graphics other than lines. By replacing the original **DoGraphics** with one of these other procedures, you can see a broad sampling of BGI graphics in action with minimal fuss. The replacement procedures are collected in a file called GPROCS.SRC, which is contained in the CompuServe download file PASBGI.ARC.

Ending graphics programs. Before ending your graphics programs, always call **CloseGraph** in the manner shown near the bottom of Listing 1. **CloseGraph** restores the display mode that was in effect before the call to **InitGraph**, and removes the graphics kernel from memory. This prevents leaving the display in graphics mode.

If you want to return to graphics mode after you've called **CloseGraph**, but you haven't ended execution of your program, you must again call **InitGraph** to reload and initialize the graphics kernel.

```
grDriver := Detect;
InitGraph( grDriver, grMode, '' );
grError := GraphResult;
IF grError <> GrOk
   THEN Writeln( 'Graphics error : ', GraphErrorMsg( grError ) )
   ELSE DoGraphics;
```

Figure 1. Automatic BGI initialization.

```
DetectGraph( grDriver, grMode );
IF grDriver = CGA  THEN grMode := CGAHi   ELSE
IF grDriver = MCGA THEN grMode := MCGAMed ELSE
   BEGIN
      Writeln( 'Requires CGA or MCGA graphics' );
      Halt
    END; { if }
InitGraph( grDriver, grMode, '' );
```

Figure 2. Semiautomatic BGI initialization.

Colors and lines. SetColor takes a value from **0** to **n**, selecting from among **n+1** colors for subsequent drawing commands. The maximum color value **n** is different for various display modes, but is always within the range **0..15**. To find the maximum value for your system, call function **GetMaxColor**. Listing 1 uses this technique to generate random values from **1** to **GetMaxColor**, selecting among all possible hues except **0**, which is the background color (usually black).

LineTo takes two integer parameters that represent the (x,y) coordinate of one display pixel. A pixel is the smallest graphics element you can display—a single dot, the quark of computer graphics. **LineTo** connects two pixel locations anywhere in the current *viewport*, which is the currently usable display area. The first location is called the *Current Point* (CP); the kernel remembers this location at all times. The second location is given by the coordinate pair passed to **LineTo** as its two integer parameters. Many graphics commands use CP as a starting point, initialized by **InitGraph** to (0,0) in the upper left corner. **LineTo** connects CP to the passed (x,y) parameters. After drawing, the line's end becomes the new CP; each subsequent line drawn with **LineTo** begins where the previous line ends.

Two other functions in Listing 1 help the program run correctly in all display modes. Function **GetMaxX** returns the maximum **x** (horizontal) coordinate value for the current graphics mode. Function **GetMaxY** returns the maximum **y** (vertical) value. Together, these two integer functions let you write programs that automatically adjust for different display resolutions.

More about lines and pixels. By modifying Listing 1, you can experiment with many other BGI commands such as **Line**, which connects two coordinates **(x1,y1)**

continued on page 18

```
PROCEDURE DoGraphics;

VAR     ch : Char;
        xmax, ymax : Integer;

BEGIN
    xmax := GetMaxX + 1;
    ymax := GetMaxY + 1;
    REPEAT
        Delay(150);
        SetColor( 1 + Random( GetMaxColor ) );  { Select color }
        Line( Random( xmax ), Random( ymax ),   { x1, y1 }
              Random( xmax ), Random( ymax ) ); { x2, y2 }
    UNTIL Keypressed;
    ch := Readkey
END; { DoGraphics }
```

Figure 3. Line demonstration procedure replacing ***DoGraphics*** *in Listing 1.*

```
PROCEDURE DoGraphics;

VAR     ch : Char;
        xmax, ymax : Integer;

BEGIN
    xmax := GetMaxX + 1;
    ymax := GetMaxY + 1;
    REPEAT
        SetFillStyle( 1 + Random( 11 ),              { pattern }
                      1 + Random( GetMaxColor ) );  { color }
        Bar( Random( xmax ), Random( ymax ),   { x1, y1 }
             Random( xmax ), Random( ymax ) ); { x2, y2 }
    UNTIL Keypressed;
    ch := Readkey
END; { DoGraphics }
```

Figure 4. Filled bar demonstration procedure replacing ***DoGraphics*** *in Listing 1.*

```
PROCEDURE DoGraphics;

VAR     ch : Char;
        xmax, ymax : Integer;

BEGIN
    xmax := GetMaxX + 1;
    ymax := GetMaxY + 1;
    REPEAT
        SetColor( 1 + Random( GetMaxColor ) );
        Arc( Random( xmax ),    { X coordinate value }
             Random( ymax ),    { Y coordinate value }
             Random( 361 ),     { Start angle }
             Random( 361 ),     { End angle }
             Random( 50 )  );   { Radius }
    UNTIL Keypressed;
    ch := Readkey
END; { DoGraphics }
```

Figure 5. Arc demonstration procedure replacing ***DoGraphics*** *in Listing 1.*

MEET THE BGI

continued from page 17

and **(x2,y2)**. Unlike **LineTo**, **Line** ignores CP. To see how **Line** works, replace **DoGraphics** in Listing 1 with the procedure in Figure 3. When you run the modified program, random lines are no longer connected end for end. Take out the **Delay** statement to shift the program into high gear.

Figure 3 adds two new integer variables, **xmax** and **ymax**. Before the **REPEAT..UNTIL** loop begins, the program assigns these variables the maximum coordinate values of the current graphics mode, plus 1. Because the display resolution is constant, the program runs faster by performing these additions outside the loop. Also, assigning values to variables avoids repeated calls to **GetMaxX** and **GetMaxY**. Small tricks such as these contribute to making graphics programs (and other kinds of programs, too!) run as fast as possible.

To display individual pixels, use **PutPixel**. Remove the statements between **REPEAT** and **UNTIL** from Figure 3 and insert the following for a colored confetti display:

```
PutPixel(Random(xmax),
 Random(ymax ),
 Random(GetMaxColor + 1 ))
```

PutPixel takes three parameters: integer **x** and **y** coordinate values, and a color value from **0** to **GetMaxColor**. Unlike **Line** and **LineTo**, **PutPixel** does not use the color value passed to **SetColor**. For the reasons mentioned above, you'd be smart to assign **GetMaxColor+1** to an integer variable and then move this addition outside the **REPEAT..UNTIL** loop.

Rectangles. Procedure **Rectangle**, as its name suggests, draws rectangles. Pass four coordinate values to locate the upper left and lower

continued on page 20

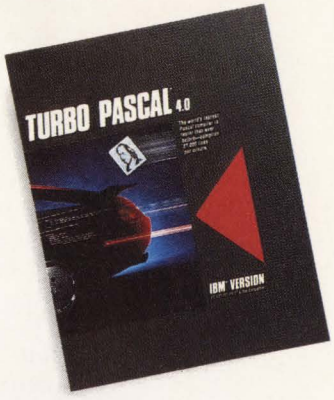
Program in the fast lane with Borland's new Turbo Pascal 4.0!

Our new Turbo Pascal® 4.0 is so fast, it's almost reckless. How fast? Better than 27,000 lines of code per minute.* That's more than twice as fast as Turbo Pascal 3.0.

4.0 Technical Highlights:

- Compiles 27,000 lines per minute
- Includes automatic project Make
- Supports > 64K programs
- Uses units for separate compilation
- Integrated development environment
- Interactive error detection/location
- Includes a command line version of the compiler
- Highly compatible with 3.0

For the IBM PS/2™ and the IBM® and Compaq® families of personal computers and all 100% compatibles



4.0 breaks the code barrier

No more swapping code in and out to beat the 64K code barrier. Designed for large programs, Turbo Pascal 4.0 lets you use all 640K of memory in your computer.

4.0 uses logical units for separate compilation

Pascal 4.0 lets you break up the code gang into "units," or "chunks." These logical modules can be worked with swiftly and separately. 4.0 also includes an automatic project Make.

4.0's cursor automatically lands on any trouble spot

4.0's interactive error detection and location means that the cursor automatically lands where the error is. While you're compiling or running a program, you get an error message *and* the cursor flags the error's location for you.

Sieve (25 iterations)

	Turbo Pascal 4.0	Turbo Pascal 3.0
Size of Executable File	2224 bytes	11682 bytes
Execution speed	9.3 seconds	9.7 seconds

Sieve of Eratosthenes, run on an 8MHz IBM AT

Since the source file above is too small to indicate a difference in compilation speed we compiled our CHESS program from Turbo Gameworks to give you a true sense of how much faster 4.0 really is!

Compilation of CHESS.PAS (5469 lines)

	Turbo Pascal 4.0	Turbo Pascal 3.0
Compilation speed	12.1 seconds	35.5 seconds
Lines per minute	27,119	9,243

CHESS.PAS compiled on an 8 MHz IBM AT

Only $99.95

*60-Day Money-back Guarantee***

For the dealer nearest you, or to order now,

Call (800) 543-7543

*Run on an 8MHz IBM AT.

**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.

BORLAND

YES! I want to upgrade to Turbo Pascal 4.0 and the 4.0 Toolboxes

If you are a registered Turbo Pascal user and have not been notified of Version 4.0 by mail, please call us at (800) 543-7543. To upgrade if you have not registered your product, just send the original registration form from your manual and payment with this completed coupon to:

Turbo Pascal 4.0 Upgrade Dept., Borland International
4585 Scotts Valley Drive, Scotts Valley, CA 95066

Name _____

Ship Address _____

City _____ State _____

Zip _____ Telephone ( ) _____

This offer is limited to one upgrade per valid registered product. It is good until June 30, 1988. Not good with any other offer from Borland. Outside U.S. make payments by bank draft payable in U.S. dollars drawn on a U.S. bank. CODs and purchase orders will not be accepted by Borland.

For the IBM PS/2™ and the IBM® and Compaq® families of personal computers and all 100% compatibles

†To qualify for the upgrade price you must give the serial number of the equivalent product you are upgrading.

Please check box(es)	Suggested Retail	Upgrade Price†	Serial No.
☐ Turbo Pascal 4.0 Compiler	$ 99.95	$ 39.95	_____
☐ Turbo Pascal Tutor	69.95	19.95	_____
☐ Turbo Pascal Database Toolbox	99.95	29.95	_____
☐ Turbo Pascal Graphix Toolbox	99.95	29.95	_____
☐ Turbo Pascal Editor Toolbox	99.95	29.95	_____
☐ Turbo Pascal Numerical Methods Toolbox	99.95	29.95	_____
☐ Turbo Pascal Gameworks	99.95	29.95	_____

Total product amount $ _____
CA and MA residents add sales tax $ _____
In US please add $5 shipping and handling for each product
Outside US please add $10 shipping & handling for each product $ _____
Total amount enclosed $ _____

Please specify diskette size ☐ 5¼" ☐ 3½"

Payment: ☐ VISA ☐ MC ☐ Check ☐ Bank Draft

Credit card expiration date: _____/_____

Card # |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|

TT/M-J/88

```
PROCEDURE DoGraphics;

VAR     ch : Char;
        xmax, ymax : Integer;
        xrad, yrad : Integer;
        x, y       : Integer;

BEGIN
    xmax := GetMaxX + 1;
    ymax := GetMaxY + 1;
    xrad := xmax DIV 8;  { X radius limit }
    yrad := ymax DIV 8;  { Y radius limit }
    SetColor( White );
    REPEAT
        x := Random( xmax );
        y := Random( ymax );
        Ellipse( x,                 { X coordinate value }
                 y,                 { Y coordinate value }
                 0,                 { Starting angle }
                 360,               { Ending angle }
                 Random( xrad ),    { X radius }
                 Random( yrad ) );  { Y radius }

        SetFillStyle( 1 + Random( 11 ),   { pattern }
            1 + Random( GetMaxColor )  ); { color }

        FloodFill( x, y, White )     { Fill ellipse }

    UNTIL Keypressed;
    ch := Readkey
END; { DoGraphics }
```

*Figure 6. Filled ellipse demonstration procedure replacing **DoGraphics** in Listing 1.*

CONSTANT	VALUE	FILL EFFECT
EmptyFill	0	Background color
SolidFill	1	Solid color
LineFill	2	Lines (---)
LtSlashFill	3	Thin slashes (///)
SlashFill	4	Thick slashes (///)
BkSlashFill	5	Thick backslashes (\\\)
LtBkSlashFill	6	Thin backslashes (\\\)
HatchFill	7	Light hatch marks
XHatchFill	8	Heavy hatch marks
InterleaveFill	9	Interleaved lines
WideDotFill	10	Sparse dots
CloseDotFill	11	Dense dots
UserFill	12	Previous SetFillPattern

*Table 4. **SetFillStyle** patterns.*

MEET THE BGI

continued from page 18

right corners of a rectangular area on screen. To draw rectangles at random in the color passed to **SetColor**, change **Line** to **Rectangle** in Figure 3.

But, you say, you want boxes filled with color, not just the outlines that **Rectangle** gives? Okay, replace **Rectangle** with **Bar.** Notice that **Bar** does not fill the insides of boxes with the colors passed to **SetColor**. To specify a fill color for **Bar**, pass two word parameters—**pattern** and **color**—to **SetFillStyle**.

Table 4 lists the constants you may assign to **pattern** in order to select different fill styles for **Bar**. As always, the **color** parameter is a value from **0** to **GetMaxColor**. The complete procedure for displaying filled boxes in a variety of colors and patterns is in Figure 4, which replaces **DoGraphics** in Listing 1.

Circles and ellipses. Three basic procedures draw curves in BGI graphics: **Arc**, **Circle**, and **Ellipse**. **Arc** draws a partial circle. **Circle** draws a complete circle. **Ellipse** draws an oval, which may or may not be circular.

To experiment with **Arc**, replace **DoGraphics** in Listing 1 with Figure 5. **Arc** takes five parameters to draw a semicircle. The first two values specify the (x,y) coordinate of the semicircle's center of radius. The next two values specify starting and ending angles, forming imaginary spokes joining the semicircle center with the two ends of the arc. Angles may have any values from 0 to 360; an angle with 0 degrees intersects the arc at 3 o'clock, and greater angles move counterclockwise. The final **Arc** parameter specifies the semicircle's radius (i.e., the distance of the drawn arc from its center).

The **Circle** procedure is easier to use and takes only three parameters: **x** and **y** values that specify the circle's center, and a radius value. To experiment with **Circle**, replace **Arc** (all five lines) in Figure 5 with:

```
Circle(
 Random( xmax ),  {X}
 Random( ymax ),  {Y}
 Random( 50 ) );  {radius}
```

Circle always draws a round circle—not such an easy feat when you consider that pixels on most PC displays are not square or even close to it. Because of this, the kernel has to adjust the circle's width and height according to the display's *aspect ratio*. Otherwise, circles would be egg-shaped. BGI graphics makes drawing round circles easy—you don't have to deal explicitly with the display's aspect ratio.

This advantage becomes more apparent when you use procedures like **Ellipse**. Unlike **Circle**, this procedure does not adjust for the screen's aspect ratio. **Ellipse** takes six parameters: **x** and **y**

coordinate values, starting and ending angles (like **Arc**), and two radii that represent invisible horizontal and vertical axes within the ellipse's boundaries. To display round circles with **Ellipse**, you have to adjust the axes yourself in order to compensate for the current display's aspect ratio.

Just for fun, let's fill ellipses with patterns and colors, as we did earlier with rectangles using procedure **Bar**. Again, **SetFillStyle** selects a fill pattern and color. This time, however, there's no filled-oval procedure similar to **Bar**. Instead, we need the procedure **FloodFill**, which fills enclosed shapes with the pattern and color passed to **SetFillStyle**. The complete **FloodFill** procedure, which replaces **DoGraphics** in Listing 1, is in Figure 6.

Figure 6 demonstrates how to fill shapes with **FloodFill**. The shape must be completely enclosed with pixels of a single color that is different from the background color. Any gaps in the shape's outline allow the fill color to leak outside of the shape, like paint through a sieve. Figure 6 prevents this disaster by calling **SetColor** with the constant **White**, which is the color that **Ellipse** uses for the drawn figure's border.

Notice that the starting and ending angles in **Ellipse** are 0 and 360. These are the correct values to use when drawing ellipses with no gaps in the border. If you want to draw a semiellipse—like a semicircle, but not necessarily round—use values in the range of 0 to 360, as you did with **Arc**.

The two radii values, limited to **0..xrad-1** and **0..yrad-1** in Figure 6, control the width and height of the ellipse. Before the **REPEAT..UNTIL** loop, **xrad** and **yrad** are limited to one-eighth the display width and height. Thus, ovals appear in relatively equal sizes in all display resolutions.

I can't stress enough the importance of writing graphics programs in this way to ensure that

continued on page 22

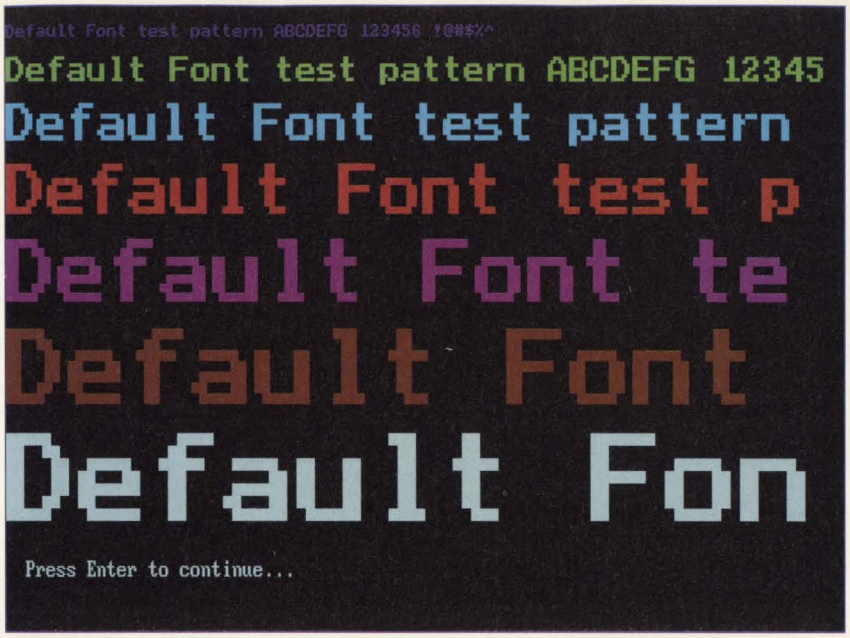


Figure 7. A BGI bit-mapped font in increasing sizes. Note how aliasing (the stairstep effect on diagonal character segments) becomes more objectionable as the characters are enlarged.

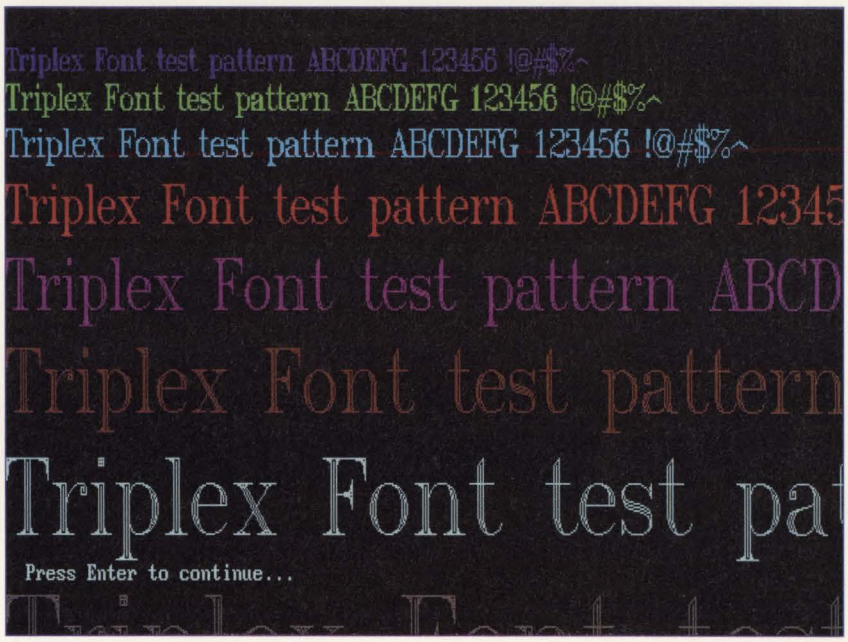


Figure 8. A BGI stroked font in increasing sizes. Stroked fonts often improve in appearance as they are enlarged. Note the clipping of drawn characters at the bottom edge of the screen.

LISTING 1: RANDOMGR.PAS

```
PROGRAM RandomGraphics;

(* Demonstrates BGI graphics--Turbo Pascal 4.0--by Tom Swan *)

USES    Crt, Graph;

VAR     grDriver, grMode, grError : Integer;

PROCEDURE DoGraphics;

VAR    ch : Char;

BEGIN
    REPEAT
        SetColor( 1 + Random( GetMaxColor ) );  { Select color }
        LineTo( Random( GetMaxX + 1 ),     { x coordinate value }
                Random( GetMaxY + 1 ) );   { y coordinate value }
    UNTIL Keypressed;
    ch := Readkey
END; { DoGraphics }


BEGIN
    grDriver := Detect;
    InitGraph( grDriver, grMode, '' );
    grError := GraphResult;
    IF grError <> GrOk
    THEN Writeln( 'Graphics error : ', GraphErrorMsg( grError ) )
    ELSE BEGIN
             DoGraphics;
             CloseGraph
         END
END.
```

LISTING 2: FONTDEMO.PAS

```
PROGRAM FontDemo;

(* Demonstrates BGI text fonts--Turbo Pascal 4.0--by Tom Swan *)

USES    Crt, Graph;

VAR     grDriver, grMode, grError : Integer;

{ Display message, wait for Enter key, clear screen. }
PROCEDURE Pause;

BEGIN
    GotoXY( 2, 23 );
    Write( ' Press Enter to continue...' );
    Readln;
    ClearViewPort
END; { Pause }

{ Display test text for font number fn }
PROCEDURE TestFont( fn : Integer );

VAR     size : Integer;     { Font size }
        name : String[10];  { Font name }

BEGIN
   CASE fn OF
      DefaultFont   : name := 'Default';
      TriplexFont   : name := 'Triplex';
      SmallFont     : name := 'Small';
      SansSerifFont : name := 'Sans Serif';
      GothicFont    : name := 'Gothic'
   END; { case }
```

MEET THE BGI

continued from page 21

they run correctly in different display modes. Never use fixed constants for coordinate limits in your programs; instead, call **GetMaxX**, **GetMaxY**, and **GetMaxColor** early in your program and then assign their return values to variables. Later in your program, use the values in those variables to adjust other variables accordingly so that the graphics conform to the size and color palette of the current graphics mode.

The final steps in Figure 6 call **SetFillStyle** to select a fill pattern and color at random; **FloodFill** then paints the oval. **FloodFill**'s **x** and **y** parameters locate a pixel somewhere inside the border of the shape to be filled. The third parameter, **White** in this example, specifies the shape's border color.

INTERMISSION

So far, we've covered initialization techniques, pixels, lines, rectangles, filled bars, arcs, circles, ellipses, and flood filling. These are merely the fundamentals of BGI graphics, which has many more routines for drawing polygons (filled and unfilled), moving the CP, customizing fill and line patterns, adjusting line widths, drawing pie chart wedges, displaying 3-D bars like those in a fancy bar chart, playing around with color palettes, inspecting the display's aspect ratio, using bit-map images, animating with multiple display pages, and more.

Let me take a moment to go over a few special routines that you'll undoubtedly use from time to time.

Clearing the screen. To clear the display call **ClearDevice**, which is faster than **ClearViewport**. (Clear the display with **ClearViewport**

only if you call **SetViewPort** to restrict the viewport—the visible window in which the drawing appears— to be less than the full screen width and height.) To change the background color used by both **ClearDevice** and **ClearViewPort**, call **SetBkColor** with a value from **0** to **GetMaxColor**.

Moving and inspecting CP. MoveTo moves CP to any (x,y) coordinate. This step is useful when you're preparing to use commands such as **Line**, which starts drawing at CP. **MoveRel** moves CP relative to the current position of CP, according to the amounts specified by the integer parameters **DX** and **DY**. Use negative **DX** values to move CP to the left, and positive values to move it to the right. Negative **DY** values move CP up, and positive values move it down. Integer functions **GetX** and **GetY** return the CP's current **x** and **y** values.

Restricted views. SetViewPort takes five parameters. As in **Rectangle**, the first four values represent the two coordinates that depict the upper left and lower right corners of a rectangle. **SetViewPort** restricts future drawing to the area within this rectangle's borders only if the fifth parameter, which is **Clipping** of type Boolean, equals **True**. If **Clipping** is **False**, then the kernel allows drawing to occur outside of the viewport boundaries. This technique is useful for centering the coordinate origin (i.e., (0,0)), which is normally located in the upper left corner of the screen. This advanced technique won't be detailed here, but it shows why you might want to turn **Clipping** off.

continued on page 24

```
   FOR size := 1 TO 8 DO
   BEGIN
      SetColor( size );
      SetTextStyle( fn, HorizDir, size );
      MoveTo( 0, GetY + TextHeight('M') + 2 );
      OutText( name + ' Font test pattern ABCDEFG 123456 !@#$%^' )
   END { for }

END; { TestFont }

PROCEDURE WritelnTest; { Display text via Writeln }

BEGIN
    GotoXY( 2, 2 );
    Writeln( ' This text is displayed by Writeln' );
    Pause
END; { WritelnTest }


PROCEDURE DoGraphics;  { Display graphics }

VAR     fontNumber : Integer;
        ch : Char;

BEGIN
    DirectVideo := False;  { So Write, Writeln work in graphics
                             when using the Crt unit. }
    { Draw blue border and restrict viewport to protect
    border from erasure: }
    SetColor( Blue );
    Rectangle( 0, 0, GetMaxX, GetMaxY );
    SetViewPort( 1, 1, GetMaxX-1, GetMaxY-1, ClipOn );
    WritelnTest;  { Demonstrate Writeln in graphics mode }
    FOR fontNumber := DefaultFont TO GothicFont DO
    BEGIN
        TestFont( fontNumber );
        Pause


    END { for }
END; { DoGraphics }

BEGIN
    grDriver := Detect;
    InitGraph( grDriver, grMode, '' );
    grError := GraphResult;
    IF grError <> GrOk
    THEN Writeln( 'Graphics error : ', GraphErrorMsg( grError ) )
    ELSE BEGIN
            DoGraphics;
            CloseGraph
         END
END.
```

LISTING 3: COLORS.PAS

```
PROGRAM Colors;

(* Displays Color Chart--Turbo Pascal 4.0--by Tom Swan *)

USES    Crt, Graph;

VAR     grDriver : Integer;     { Graphics driver number }
        grMode   : Integer;     { Graphics driver mode }
        grError  : Integer;     { Graphics error code }

        barWidth : Word;        { Pixel width of bars }
        labelHeight : Word;     { Pixel height of bar labels }
        bwd2  : Word;           { barWidth DIV 2 }
        bwt2  : Word;           { barWidth times 2 }
        lhd2  : Word;           { labelHeight DIV 2 }
```

```
{ Initialize global variables, load text font, and display title. }
PROCEDURE Initialize;

BEGIN
  { Select text style, direction, size, and justification.
    Then, display title at top center of screen. Reduce text
    size for displaying labels under bars: }
    SetTextStyle( SansSerifFont, HorizDir, 5 ); { Title }
    SetTextJustify( CenterText, CenterText );
    MoveTo( GetMaxX DIV 2, GetMaxY DIV 8 );
    OutText( 'Color Numbers' );
    SetTextStyle( SansSerifFont, HorizDir, 4 ); { Labels }

  { Initialize a few global variables: }
    barWidth  := ( ( GetMaxX + 1 ) DIV ( GetMaxColor + 1 ) ) DIV 2;
    labelHeight := 2 * TextHeight( '0' );
    lhd2 := labelHeight DIV 2;  { labelHeight DIV 2 }
    bwd2 := barWidth DIV 2;     { barWidth DIV 2 }
    bwt2 := barWidth * 2        { barWidth times 2 }
END; { Initialize }

{ Draw a single bar filled with a color }
PROCEDURE DrawOneBar( x1, y1, x2, y2 : Integer; color : Word );

BEGIN
    SetFillStyle( SlashFill, color );   { Select pattern, color }
    Bar( x1, y1, x2, y2 );              { Draw filled bar }
    SetColor( White );                  { Outline bars in white }
    Rectangle( x1, y1, x2, y2 )         { Draw outline }
END; { DrawOneBar }


{ Display color number centered at (x,y) under bar }
PROCEDURE LabelBar( x, y : Integer; color : Word );

VAR     s : String[2];          { Color number as a string }

BEGIN
    Str( color, s );            { Convert color to string s }
    OutTextXY( x, y, s )        { Display text at (x,y) }
END; { LabelBar }


PROCEDURE DrawBars;  { Draw color bars and wait for key press }

VAR     x1, y1, y2 : Integer;   { Coordinate values }
        color : Word;           { FOR-loop variable }
        ch : Char;              { Keyboard character }

BEGIN
    x1 := bwd2;                 { Initialize starting x1 value }
    y1 := GetMaxY DIV 3;        { Initialize y1 to top of bar }
    y2 := GetMaxY - labelHeight; { Initialize y2 to bottom of bar }
    FOR color := 0 TO GetMaxColor DO

    BEGIN
        DrawOneBar( x1, y1, x1 + barWidth, y2, color );
        LabelBar( x1 + bwd2, y2 + lhd2, color );
        x1 := x1 + bwt2         { Move x1 right for the next bar }
    END; { while }
    ch := ReadKey  { Wait for keypress }
END; { DrawBars }
```

MEET THE BGI

continued from page 23

Switching from text to graphics. Occasionally, you'll want to switch from a graphics screen to a text screen and then back again. You could call **CloseGraph** to shut down the BGI kernel, and then call **InitGraph** to reinitialize it again, but there's an easier way. Call **RestoreCrtMode** to temporarily return to the display mode that was active prior to calling **InitGraph**. When you're finished with the previous display mode, return to your graphics mode by calling **SetGraphMode** and passing it the **grMode** variable returned by **InitGraph**.

CLASSY CHARACTERS

Fonts. Earlier, I promised to explain more about fonts. The BGI supports two kinds of fonts: bit-mapped and stroked. Characters in *bit-mapped* fonts are composed of rectangular pixel arrays. Characters in *stroked* fonts are composed of vectors—line segments whose sizes and directions are defined as relative to some starting point.

The difference between the two kinds of fonts is important when changing the default size of character images. Blowing up bit-mapped fonts is done by simply enlarging the pixels. This makes the characters look blocky, with huge stair steps (this effect is called *aliasing*) on their sloping parts (see Figure 7). Stroked fonts are enlarged by making their component line segments longer, so aliasing does not occur. Because stroked fonts are defined as collections of line segments rather than pixel blocks, characters look good (in many cases, *better*) when enlarged (see Figure 8). Bit-mapped fonts do offer one advantage—speed. Stroked-font characters, with their many line segments, take longer to draw than do bit-mapped characters.

The BGI kernel currently supports one bit-mapped font and four stroked fonts (see Table 2). For comparison, Listing 2 displays short text lines in all possible fonts. To run the program, you need to place the .CHR files listed in Table 2 into the same directory that you specified with the string parameter to **InitGraph**.

Making your selection. Procedure **TestFont** in Listing 2 shows the correct way to select a font and size. After the **CASE** statement sets string variable **name** to the font's name, a **FOR** loop cycles integer variable **size** from 1 to 8. Most fonts look best with small sizes in this range, but you have to experiment. Different fonts with the same size values appear larger or smaller than others.

Use **SetTextStyle** to select a font, direction, and size. The first parameter in **SetTextStyle** is the font number, which is one of the five constants **DefaultFont**, **TriplexFont**, **SmallFont**, **SansSerifFont**, and **GothicFont**. **DefaultFont** (0) is a bit-mapped font. The other four fonts are stroked.

The second **SetTextStyle** parameter can be either **HorizDir** or **VertDir**. Use **HorizDir** to display text horizontally from left to right, and use **VertDir** to display text vertically from top to bottom. The final **SetTextStyle** parameter specifies the font size.

SetTextStyle loads the appropriate .CHR font file from disk, searching the pathname passed to **InitGraph**. If the kernel can't find this font file, **SetTextStyle** returns an error code through **GraphResult**. (Note that for brevity's sake, Listing 2 does not check for this error.)

Displaying text. OutText and **OutTextXY** display text in the current font, direction, and size, using the color most recently passed to **SetColor**. **OutText** takes a single string parameter and dis-

continued on page 26

```
BEGIN
    grDriver := Detect;
    InitGraph( grDriver, grMode, '' );
    grError := GraphResult;
    IF grError <> GrOk
     THEN
        Writeln( 'Graphics error : ', GraphErrorMsg( grError ) )
     ELSE
        BEGIN
            Initialize;          { Perform various initializations }
            DrawBars;            { Display color bars }
            CloseGraph           { Restore former text mode }
        END
END.
```

LISTING 4: KALEIDO.PAS

```
PROGRAM Kaleidoscope;

(* Displays Kaleidoscope Patterns--Turbo Pascal 4.0--by Tom Swan *)

USES    Crt, Graph;


VAR     grDriver : Integer;      { Graphics driver number }
        grMode   : Integer;      { Graphics driver mode }
        grError  : Integer;      { Graphics error code }


PROCEDURE DoGraphics;  { Display graphics until a key is pressed }

VAR     xmax, ymax : Integer;           { Maximum x, y values }
        x1, y1, x2, y2 : Integer;       { Line endings }
        dx1, dy1, dx2, dy2 : Integer;   { Change in x1,y1,x2,y2 }
        displayPeriod : Word;           { Time between clearing }
        linePeriod : Word;              { Time each pattern lives }
        ch : Char;                      { For clearing keypress }


    PROCEDURE Initialize;  { Perform various initializations }

    BEGIN
        Randomize;               { "Seed" new random sequence }
        displayPeriod := 0;      { Force call to NewDisplayPeriod }
        xmax := GetMaxX DIV 2;   { Set x and y maximums to middle }
        ymax := GetMaxY DIV 2;   {  of display resolution }

      { Restrict viewport to 1/4 entire display.  With clipping
        off, this centers the origin (0,0) and makes mirror
        images in the four quadrants easy to draw. }

        SetViewPort( xmax, ymax, GetMaxX, GetMaxY, ClipOff )
    END; { Initialize }


    { Clear screen and initialize displayPeriod,
      controlling length of time between screen clears: }
    PROCEDURE NewDisplayPeriod;

    BEGIN
        ClearDevice;     { Clear entire display }
        displayPeriod := 6 + Random( 24 )  { 6..29 }
    END; { NewDisplayPeriod }


    { Select coordinates, movements, linePeriod,
      and line color at random: }
    PROCEDURE NewValues;
```

```
BEGIN
    x1  := Random( xMax + 1 );       { x1 <- 0..xmax }
    y1  := Random( yMax + 1 );       { y1 <- 0..ymax }
    x2  := Random( xMax + 1 );       { x2 <- 0..xmax }
    y2  := Random( yMax + 1 );       { y2 <- 0..ymax }
    dx1 := Random( 16 ) - 8;         { dx1 <- -8..+7 }
    dy1 := Random( 16 ) - 8;         { dy1 <- -8..+7 }
    dx2 := Random( 16 ) - 8;         { dx2 <- -8..+7 }
    dy2 := Random( 16 ) - 8;         { dy3 <- -8..+7 }
    linePeriod := 5 + Random(120);  { linePeriod <- 5..124 }
    SetColor( 1 + Random( GetMaxColor ) )
END; { NewValues }

{ Adjust line coordinates, making lines appear to move: }
PROCEDURE MoveCoordinates;

BEGIN
    x1 := x1 + dx1;          { Add appropriate "delta," }

        y1 := y1 + dy1;          {  meaning "change in," value }
        x2 := x2 + dx2;          {  to line end coordinates. }
        y2 := y2 + dy2
    END; { MoveCoordinates }


    PROCEDURE DrawLines;  { Draw lines mirrored in four quadrants }

    BEGIN
        Line( -x1, -y1, -x2, -y2 );     { upper left quadrant }
        Line( -x1,  y1, -x2,  y2 );     { lower left quadrant }
        Line(  x1, -y1,  x2, -y2 );     { upper right quadrant }
        Line(  x1,  y1,  x2,  y2 )      { lower right quadrant }
    END; { DrawLines }


BEGIN
    Initialize;
    REPEAT
        IF displayPeriod <= 0
            THEN NewDisplayPeriod;      { Clear screen }
        NewValues;
        WHILE ( linePeriod > 0 ) AND ( NOT Keypressed ) DO
        BEGIN
            Delay( 5 );                 { Set the speed limit }
            MoveCoordinates;            { Animate display }
            DrawLines;                  { Draw mirror images }
            linePeriod := linePeriod - 1
        END; { while }
        displayPeriod := displayPeriod - 1
    UNTIL Keypressed;
    ch := Readkey
END; { DoGraphics }


BEGIN
    grDriver := Detect;
    InitGraph( grDriver, grMode, '' );
    grError := GraphResult;
    IF grError <> GrOk
     THEN
        Writeln( 'Graphics error : ', GraphErrorMsg( grError ) )
     ELSE
        BEGIN
            DoGraphics;
            CloseGraph
        END
END.
```

MEET THE BGI

continued from page 25

plays its characters beginning at CP. After **OutText**, CP moves to the position just after the last display character. **OutTextXY** ignores CP and takes **x** and **y** integer parameters, plus the string to be displayed.

Listing 2 calls **MoveTo**, initializing CP to a starting position before calling **OutText**. Each new text line must be positioned a little further down the screen, because the characters are larger on each successive line. Therefore, to calculate the position of each new text line, the function **TextHeight** is called from **MoveTo**'s parameter line. **TextHeight** returns the pixel height of a string; in this case, the string consists of the letter M. (M was chosen here because it is about as tall as any character gets.) The sum of the current **y** coordinate value (**GetY**) added to the value returned by **TextHeight**, plus two additional pixels for spacing, yields the new **y** coordinate for the next text line.

Notice that when it calls **OutText**, string variable **name** is concatenated to the quoted string literal "**Font test pattern ABCDEFG 123456 !@#$%^'**" using the string concatenation operator +. Remember, **OutText** and **OutTextXY** take a *single* string parameter. Unlike **Write** and **Writeln**, you can't separate multiple parameters to **OutText** and **OutTextXY** with commas. Also, in order to display integer and real number values, you must first use Turbo Pascal's **Str** procedure to convert such values to strings, and then pass the equivalent string values to **OutText** or **OutTextXY**.

Using Write and Writeln in graphics mode. You can also mix text and graphics with **Write** and **Writeln**, but you can display BGI fonts *only* with **OutText** and **OutTextXY**. In graphics mode,

Write and **Writeln** always use the same system-resident font that they use when called from text mode. If your program uses the **Crt** unit, which links fast, direct video routines to **Write** and **Writeln**, you must set the Boolean variable **DirectVideo** to **False**. If you don't do this, **Write** and **Writeln** won't work on graphics screens. If your program does not use **Crt**, then **Write** and **Writeln** work normally.

BGI ROUNDUP

I'll leave you with two programs that use many of the routines covered in this article. The programs, Listings 3 and 4, are heavily commented and you should be able to explore them without any further help.

Listing 3, which displays a color chart, adjusts for all possible display resolutions and color sets. As I suggested earlier, strive for this same flexibility in your own programs, so that your software runs correctly in a wide variety of PC graphics modes. Why limit your audience?

Listing 4 displays an animated color kaleidoscope that, again, works in all modes supported by BGI graphics. The program uses the technique (mentioned earlier) of setting a viewport without enabling viewport clipping in order to center the origin. This technique divides the display into four quadrants, making the mirror images easier to draw than they would be if the origin was in the top left corner. The program is great for parties and such—you can drag it out when somebody asks, "So what does this computer of yours do?"

By now, you should have a feeling for the depth of BGI graphics capabilities, even though we've only scratched the surface. Of all the graphics systems I've used in various languages, there's no question that the BGI kernel is a top-notch performer. If you're interested in graphics programming on the PC, you no longer have to eye those pretty Amigas and Ataris down at the budget computer store. You may discover that the new BGI frontier, and a copy of Turbo Pascal or Turbo C, are all you need. ■

Tom Swan is the author of Mastering Turbo Pascal 4.0, 2nd Edition, *Howard W. Sams, 1988;* Mastering Turbo Pascal Files, *Howard W. Sams, 1987; and* Programming With Macintosh Turbo Pascal, *John Wiley & Sons, 1987.*

Listings may be downloaded from CompuServe as PASBGI.ARC.

PLOTTING THE MANDELBROT SET WITH THE BGI

Meet the most complex object in mathematics—the Mandelbrot Set—brought to you in living color by the Borland Graphics Interface.

Fred Robinson


PROGRAMMER

Back in the fall of 1985, I sat poring over the latest *Scientific American*, dumbfounded. Professor Dewdney's Computer Recreations article—the cover story, no less—unveiled the Mandelbrot Set, one of the most beautiful and probably the most complex creature ever to emerge from the realm of mathematics.

Well, if you have seen the October, 1985, *Scientific American*, you've probably also been bitten by the Mandelbrot bug. My own experience may be typical. I've implemented software to generate the Set on the Apple II+, and later in several different languages on the PC, culminating with Turbo Pascal 3.0 and the Turbo Pascal Graphix Toolbox. Now the next step has arrived—Turbo Pascal 4.0 and the Borland Graphics Interface team up to provide the Mandelbrot Set generator presented here.

A COMPLEX NOTION

First, let's take a brief look at some concepts that are key to describing the Mandelbrot set: complex numbers, imaginary numbers, and the complex number plane.

Complex and imaginary numbers. A *complex number* contains some form of the square root of -1, which by convention is called *i*. Unlike the square roots of positive numbers, *i* and multiples of *i* cannot be found on the real number line. For this reason, multiples of *i* are called *imaginary numbers* (hence the use of the *i*). *i* isn't normally used in everyday calculations.

A complex number has two parts: a real number part, which is some value along the real number line; and an imaginary number part. For example, the sum of $9.7 + 3.5i$ is a complex number. Complex numbers can be added to, subtracted from, multiplied with, and divided by other complex numbers. (These operations are not as easy to perform on complex numbers as they are on ordinary numbers, but they can be done.)

Complex numbers also have size. The size of a complex number is its distance from the origin, (0,0); this distance is calculated by using the Pythagorean Theorem. For example, the size of $9.7 + 3.5i$ is Sqrt $(9.7^2 + 3.5^2)$, or Sqrt (31.65), or about 10.31213.

The complex number plane. The *complex number plane* is represented by a Cartesian coordinate system whose X axis is labeled the real axis, and whose Y axis is labeled the imaginary axis. In this system, all ordinary real numbers fall on the X axis. Any given point on the plane corresponds to a complex number. For example, the point (9.7, 3.5) corresponds to the complex number $9.7 + 3.5i$.

WHAT IS THE MANDELBROT SET?

The *Mandelbrot Set* is a region of the complex number plane, situated between $-2 + -2i$ and $2 + 2i$, that contains a set of complex numbers. When these numbers are repeatedly subjected to a certain formula, they never achieve a size greater than 2. Naturally, a plane contains just as many points that will exceed 2 as it does points that will not exceed 2. The interesting part comes near the border between the two regions.

The formula that is used to determine which numbers belong to the Mandelbrot Set is deceptively simple: Initialize *z* to the complex point *c* in question, and simply repeat the following equation until either the size of *z* exceeds 2, or else the process has been repeated an arbitrary number of times:

$$z = z^2 + c$$

Once the size of *z* exceeds 2, it never returns to less than 2. Some numbers take quite a while to reach this size, other numbers never do, and still others pass 2 almost immediately. For our purposes (given somewhat slow general-purpose computers), the maximum number of iterations can be cut to about 250.

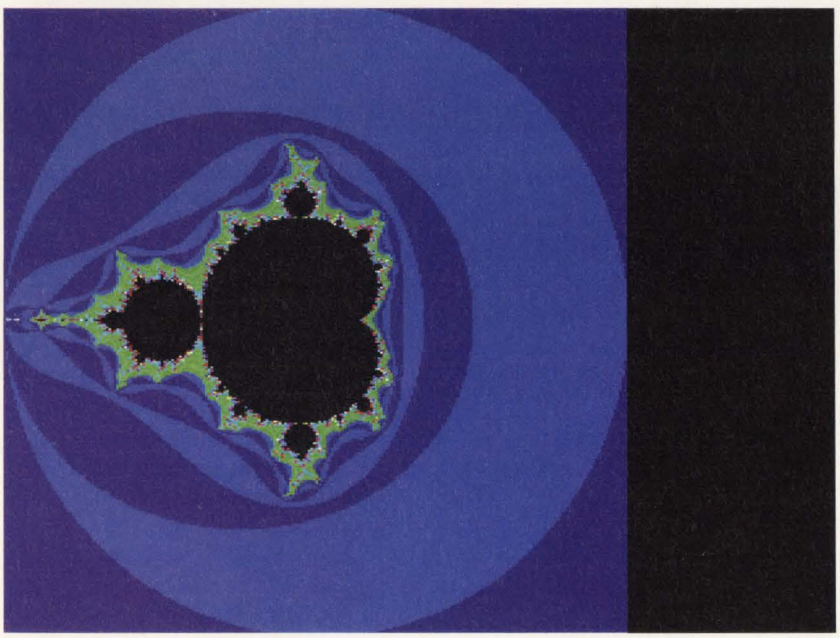


Figure 1. The full Mandelbrot Set, plotted from -2.0 to 2.0 and -2.0i to 2.0i. Other views are obtained by choosing a subset of the image and recalculating it so that the new image occupies the entire image area.

Figure 2. Deep in the heart of the Set, highlighting the fractal nature of the edge of the Set. The perimeter of the Set is infinitely long, even though the Set's area is finite.

THE FRACTAL EDGE

What is the point of all this? Well, the Mandelbrot Set has an edge that is fractal in nature. By specifying a small portion of the complex number plane, mapping it onto a PC display screen, and suitably increasing the iteration limit, you can see details of the fractal edge in ever-decreasing size. Zooming in even closer, the tiny details in a larger picture can be seen to contain even smaller details.

I could go on like this for quite a while, but the Set speaks for itself. Figures 1 and 2 give you some idea of what to expect from a plot of the Set on an EGA in 640 × 350 graphics mode. Figure 1 is a view of the entire Set from a height. The ranges used to generate Figure 1 are –2.0 to 2.0 and –2.0*i* to 2.0*i*. Zooming down several

continued on page 30

```
input real and imaginary ranges
input iteration maximum

for c.i = low_imaginary to high_imaginary
  for c.r = low_real to high_real
    z = c
    repeat
      z = z * z + c
    until size (z)>2 or too many iterations
    if size (z)>2 then
      plot (c.r, c.i)
```

Figure 3. This algorithm determines if a point on the complex number plane belongs to the Mandelbrot Set.

continued from page 29

levels into the Set revealed the image shown in Figure 2.

The first step in generating the Mandelbrot Set is to master complex arithmetic. Complex addition and subtraction are trivial, but complex multiplication is not so easy. Remember how to multiply polynomial expressions? FOIL: First + Outside + Inside + Last. There is enough similarity between a complex number and a polynomial for this technique to work. Multiplying two numbers, A + Bi and C + Di, yields:

AC + ADi + BCi + BDii

This equation is simplified to:

(AC + -BD) + (AD + BC)i

Remember, since i is the square root of -1, $i \times i$ yields -1, producing a negative BD in the result shown above. Multiplying the A's, Bs, Cs, and Ds and then summing the results generates a new complex number.

Once that process is understood, generation of the Set is straightforward. The basic algorithm, which steps across a region one pixel at a time, is shown in Figure 3.

PLOT POINTERS

For the best results, assign a color to each point plotted based upon the number of iterations necessary to determine whether the point belongs to the Set. Points belonging to the Set are usually left black. Points outside of the Set can be plotted many ways; the easiest way is to illuminate those points that require an even number of iterations. This approach is adequate for a two-color display, such as the display supported by the Turbo Pascal Graphix Toolbox.

It's also possible to write a Set generator using the Borland Graphics Interface and Turbo Pascal 4.0. This method results in smaller code and faster compilation, as well as portability to different display devices, including color displays. The use of color offers more possibilities. For

continued on page 32

LISTING 1: MANDEL4.PAS

```
PROGRAM Mandel4;

{This program generates a section of the Mandelbrot Set, can save it
on disk, and use existing Mandelbrot pictures to zoom further into
the Set.}

USES
   Crt, Graph, Cmplx; { CMPLX.TPU is created from CMPLX.PAS }

CONST
   Scan_Width = 359; { 719 (max Hercules) DIV 2 }
   Max_Scan_Lines = 349; { PC3270 maximum }
   Aspect = 0.75; { Typical screen aspect ratio }
   Real_Length = 30;
   Yes_N_No:  SET OF char = ['Y', 'N', 'y', 'n'];
   Yes:  SET OF char = ['Y', 'y'];
   No:  SET OF char = ['N', 'n'];
   TP_Path = 'T:';

TYPE
   Scan_Line = ARRAY [0..Scan_Width] OF byte;
   Scan_Line_Ptr = ^Scan_Line;
   Real_String = STRING[Real_Length];
   Color_Array = ARRAY [0..55] OF integer;

CONST
   Colors_2:  Color_Array = (0, 0, 0, 0, 1, 1, 1, 1,
                             0, 0, 0, 0, 1, 1, 1, 1,
   { Color arrangement for } 0, 0, 0, 0, 1, 1, 1, 1,
   { 2-color screens       } 0, 0, 0, 0, 1, 1, 1, 1,
                             0, 0, 0, 0, 1, 1, 1, 1,
                             0, 0, 0, 0, 1, 1, 1, 1,
                             0, 0, 0, 0, 1, 1, 1, 1);

   Colors_4:  Color_Array = (1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2,
                             3, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1,
   { Color arrangement for } 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3,
   { 4-color screens       } 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2,
                             3, 3, 3, 3, 3, 3, 0, 0);

   Colors_16:  Color_Array = (1,  9, 1,  9, 1,  9, 1,  9,
                              2, 10, 2, 10, 2, 10, 2, 10,
   { Color arrangement for }  3, 11, 3, 11, 3, 11, 3, 11,
   { 16-color screens      }  4, 12, 4, 12, 4, 12, 4, 12,
                              5, 13, 5, 13, 5, 13, 5, 13,
                              6, 14, 6, 14, 6, 14, 6, 14,
                              7, 15, 7, 15, 7, 15, 7, 15);

VAR
   Ch:  Char;
   Low, High, Delta:  Complex;
   Dots_Horizontal, Dots_Vertical, Start_Y, Max_Count, Color_Count,
      Device, Graph_Mode, Max_Colors, Max_X:  integer;
   Use_Color:  Color_Array;
   Picture_Loaded:  boolean;
   File_Name:  STRING[80];

   Data_Line:  Scan_Line;
   Screen_File:  FILE OF Scan_Line;
   Screen:  ARRAY [0..Max_Scan_Lines] OF Scan_Line_Ptr;
   Screen_Data:  RECORD
                    Dots_H, Dots_V, Count, Start:  integer;
                    Low_Real, Low_Imag,
                    High_Real, High_Imag:  Real_String;
                    Note:  String[200]
                 END ABSOLUTE Data_Line;

{*********************************************************************}

PROCEDURE Initialize;

{  This procedure checks for the graphics screen and selects a mode
based on a compromise between resolution and the number of colors.  }

VAR
   X:  integer;

BEGIN
TextMode (LastMode);
TextColor (LightBlue);
TextBackground (Black);
DirectVideo := False;
File_Name := '';
Picture_Loaded := False;
DetectGraph (Device, Graph_Mode);
X := GraphResult;

IF X<>grOk THEN
   BEGIN
   Writeln ('Sorry, I can''t cope with this:  ', GraphErrorMsg (X));
   Halt
   END {* THEN *};

CASE Device OF
   EGA:  Graph_Mode := EGAHi;
   VGA:  Graph_Mode := VGAMed;
   MCGA:  Graph_Mode := MCGACO;
   EGA64:  Graph_Mode := EGA64Lo;
   ATT400:  Graph_Mode := ATT400CO;
   PC3270:  Graph_Mode := PC3270Hi;
   HercMono:  Graph_Mode := HercMonoHi;
   CGA, RESERVED:  Graph_Mode := CGACO

END {* CASE *};

InitGraph (Device, Graph_Mode, TP_Path);

CASE Device OF
   CGA, MCGA, RESERVED,
   ATT400:  BEGIN
            Color_Count := 54;
            Use_Color := Colors_4;
            Max_Colors := 3;
            Max_X := GetMaxX
            END {* CASE CGACO, MCGACO, ATT400CO *};

   EGA, VGA,
   EGA64:  BEGIN
           Color_Count := 56;
           Use_Color := Colors_16;
           Max_Colors := 15;
           Max_X := GetMaxX DIV 2
           END {* CASE EGAHi, VGAHi, EGA64Lo *};

   ELSE BEGIN
        Color_Count := 56;
        Use_Color := Colors_2;
        Max_Colors := 1;
        Max_X := GetMaxX DIV 2
        END {* CASE ELSE *}
END {* CASE *};

FOR X := 0 TO Max_Scan_Lines DO
   New (Screen[X]);

RestoreCrtMode
END {* Initialize *};

{*********************************************************************}

PROCEDURE Plot (X, Y:  integer;
                Color:  word);

{  This procedure plots points on the screen.  For high-resolution-
width screens, two adjacent pixels are set.  }

BEGIN
CASE Device OF
   CGA, MCGA, RESERVED,
   ATT400:  PutPixel (X, Y, Color);

   ELSE BEGIN
        PutPixel (X*2, Y, Color);
        PutPixel (X*2+1, Y, Color)
        END {* CASE ELSE *}
END {* CASE *}
END {* Plot *};

{*********************************************************************}

PROCEDURE Define_Screen;

{This procedure defines the area of the Mandelbrot Set to be viewed.
It can either be typed in at the keyboard, loaded as a partially
completed screen, or as a smaller sector of a completed picture.  }

VAR
   X, Y:  integer;
   Temp, Ratio:  double;

   {*****************************************************************}

   PROCEDURE No_Blank (VAR RS:  Real_String);

   {  This procedure removes leading blanks from the string RS.  }

   BEGIN
   WHILE RS[1]=' ' DO
      RS := Copy (RS, 2, Length (RS)-1)
   END {* No_Blank *};

   {*****************************************************************}

   PROCEDURE Sub_Picture;

   {  This procedure allows the user to select a sub-section of a
   completed screen to be blown up, effectively zooming in on a
   smaller area.

    Pressing keys 2 thru 5 changes the grid on the screen.  A sub-
   section may be chosen by pressing a letter, starting with A in the
   upper left corner and moving across:

      2:  +---+---+
          | A | B |
          +---+---+
          | C | D |
          +---+---+

      3:  +---+---+---+
          | A | B | C |
          +---+---+---+
          | D | E | F |
          +---+---+---+
          | G | H | I |
          +---+---+---+
```

listing continued on page 33

continued from page 30

instance, the CGA medium-resolution mode gives you three colors to use if you reserve black for the Set. More advanced display boards, of course, provide more colors. My own system, which now uses a Genoa Spectrum, generates 16 colors at 320 × 200. This is a good compromise between resolution, image quality, and speed of generation.

Another point: Since every pixel on the screen requires a certain amount of real-number calculation to determine the pixel's color, a math coprocessor is essential. Performing the calculations without a coprocessor takes about four times longer than performing them with a coprocessor. Also, keep in mind that all black points that represent the Set itself require the full number of iterations in order to calculate. Thus, images that contain a great deal of black take longer to calculate than those with less black.

Finally, when you look at very small regions near the edge of the Set, it's necessary to increase the number of iterations in order to reveal details that would otherwise be set to the black color of the Set. Near the edge of the Set, the number of iterations that is required to attain a size greater than 2 increases dramatically. This increase results in longer image generation times.

MANDEL4

Mandel4 (Listing 1) takes advantage of many display types, using the new standard BGI functions to detect and initialize the displays. It accepts starting regions from the keyboard, or else takes a smaller region from a completed screen. **Mandel4** requires the code in CMPLX.PAS (Listing 2), which is a unit that contains several complex-number math routines.

In terms of resolution and color, I've made some compromises when dealing with certain screens. The horizontal resolution of displays that contain a great number of pixels in the horizontal dimension has been halved. This allows faster picture creation with the loss of only a small amount of visual information. In screens with few colors, the colors are grouped together to make the smaller details of the Set more visible on the screen (the alternative is a polychromatic hash where the iteration count changes drastically from one pixel to the next).

Another compromise is the method used to store the pictures. Rather than encoding several pixels into a byte, each pixel is stored in a full byte. This increases speed, and allows up to 256 possible colors for a pixel (this color range is supported by MCGA and VGA displays). This storage method, however, creates rather large picture files, so be careful that you have enough room available on your disk when saving an image as a file.

Mandel4 gives you the opportunity to either load a previously created image (by typing the name of a picture file), or to generate a totally new image (by typing in the real and imaginary ranges, along with the maximum iteration count). The program also allows entry of a note for later reference. Once this information is entered, **Mandel4** plots the area encompassed by the ranges specified.

If you load a partially finished picture, **Mandel4** continues to work on it. If the picture you load is complete, however, it's thrown onto the screen and a grid appears that divides the screen into quarters. By pressing keys 2 through 5, you can change the grid to another grid in the range of 2 × 2 through 5 × 5. Each smaller area has the same proportions as the picture on the screen. An area can be chosen by pressing a letter key, starting with A in the upper left corner and moving across to the right, as shown in Figure 4.

While **Mandel4** is working on a picture, a dot moves across the screen and marks the pixel that is currently being calculated. When the picture is done, or if you press a key during the drawing, **Mandel4** gives you an opportunity to save the image in a file, and then asks if you want to generate another image. You can either start over from scratch, or else use the picture you've just completed as a starting point for the next image.

A	B
C	D

After pressing 2:

A	B	C
D	E	F
G	H	I

After pressing 3:

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

After pressing 4:

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y

After pressing 5:

Figure 4. How the screen is divided after image plotting. To choose a subsection of the image to "zoom," simply press the letter that corresponds to the desired subsection.

Mandel4 plots to many different displays, but picture files plotted on one display can't be moved to another (such as from CGA to EGA).

GET SET

I guess it's possible to go overboard with the Mandelbrot Set, but I think that that's the most fun of all, don't you? Try it and you'll see. ■

Fred Robinson is a computer programmer in the research department of Ross Roy, Inc., an advertising firm in Michigan.

Listings can be downloaded from CompuServe as MANDEL.ARC.

listing continued from page 31

4:

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

5:

A	B	C	D	E
F	G	H	I	J
K	L	M	N	O
P	Q	R	S	T
U	V	W	X	Y

Once a section has been chosen, the program proceeds to calculate and display the smaller section, as large as the screen may allow.}

```
CONST
   Max_Letter:  ARRAY [2..5] OF char = ('D', 'I', 'P', 'Y');

VAR
   Ch:  char;
   New_Size, Size, X, Y, Z, Sector, Sector_X, Sector_Y:  integer;

BEGIN
Size := 1;
File_Name := '';
Ch := '2';

REPEAT
   IF Ch IN ['2'..'5'] THEN
      BEGIN (* Change grid *)
      New_Size := Ord (Ch) - Ord ('0');

      IF Size<>New_Size THEN
         BEGIN
         { Undo existing grid }
         FOR X := 0 TO Dots_Horizontal DO
            FOR Z := 1 TO Size-1 DO
               BEGIN
               Y := Z * Dots_Vertical DIV Size;
               Plot (X, Y, Screen[Y]^[X])
               END (* FOR, FOR *);

         FOR Y := 0 TO Dots_Vertical DO
            FOR Z := 1 TO Size-1 DO
               BEGIN
               X := Z * Dots_Horizontal DIV Size;
               Plot (X, Y, Screen[Y]^[X])
               END (* FOR, FOR *);

         Size := New_Size;

         { Make new grid }
         FOR X := 0 TO Dots_Horizontal DO
            FOR Z := 1 TO Size-1 DO
               BEGIN
               Y := Z * Dots_Vertical DIV Size;
               Plot (X, Y, Max_Colors-Screen[Y]^[X])
               END (* FOR, FOR *);

         FOR Y := 0 TO Dots_Vertical DO
            FOR Z := 1 TO Size-1 DO
               BEGIN
               X := Z * Dots_Horizontal DIV Size;
               Plot (X, Y, Max_Colors-Screen[Y]^[X])
               END (* FOR, FOR *)
         END (* THEN *)
      END (* THEN *);

   Ch := UpCase (ReadKey)
UNTIL (Size IN [2..5]) AND (Ch IN ['A'..Max_Letter[Size]]);

{ Calculate new limits }
Sector := Ord (Ch) - Ord ('A');
Sector_X := Sector MOD Size;
Sector_Y := Size - 1 - Sector DIV Size;
Sub_Comp (High, Low, Delta);
Div_C_By_R (Delta, Size, Delta);
Low.R := Low.R + Delta.R * Sector_X;
High.R := Low.R + Delta.R;
Low.I := Low.I + Delta.I * Sector_Y;
High.I := Low.I + Delta.I;

   WITH Screen_Data DO
      BEGIN
      Start_Y := 0;
      Dots_H := Dots_Horizontal;
      Dots_V := Dots_Vertical;
      Count := Max_Count;
      Str (Low.R, Low_Real);
      Str (Low.I, Low_Imag);
      Str (High.R, High_Real);
      Str (High.I, High_Imag);
      No_Blank (Low_Imag);
      No_Blank (Low_Real);
      No_Blank (High_Imag);
      No_Blank (High_Real)
      END (* WITH *);

   RestoreCrtMode;
   Write
   ('Maximum iteration count = ', Max_Count, '.  Change it? (Y/N) ');

   REPEAT
      Ch := ReadKey
   UNTIL Ch IN Yes_N_No;

   Writeln (Ch);

   IF Ch IN Yes THEN
      BEGIN
      REPEAT
         Write ('Enter maximum iteration count:  ');
         {$I-} Readln (Max_Count) {$I+}
      UNTIL IOResult=0;

      IF Max_Count<10 THEN
         Max_Count := 10;

      Screen_Data.Count := Max_Count
      END (* THEN *);

   Write ('Enter note:  ');
   Readln (Screen_Data.Note);
   SetGraphMode (Graph_Mode)
   END (* Sub_Picture *);

   {*************************************************************}

BEGIN (* Define_Screen *)
Ch := 'N';

IF Picture_Loaded THEN
   BEGIN
   Write ('Use picture in memory? (Y/N) ');

   REPEAT
      Ch := ReadKey
   UNTIL Ch IN Yes_N_No;

   Writeln (Ch)
   END (* THEN *);

IF Ch IN No THEN
   BEGIN
   Write ('Load a picture file? (Y/N) ');

   REPEAT
      Ch := ReadKey
   UNTIL Ch IN Yes_N_No;

   Writeln (Ch);

IF Ch IN Yes THEN
   BEGIN { Load picture file }
   REPEAT
      Write ('Enter name of file:  ');
      Readln (File_Name);
      Assign (Screen_File, File_Name);
      {$I-} Reset (Screen_File) {$I+};
   UNTIL IOResult=0;

   Read (Screen_File, Data_Line);

   FOR X := 0 TO Screen_Data.Start-1 DO
      Read (Screen_File, Screen[X]^);

   Close (Screen_File);
   Picture_Loaded := True
   END (* THEN *)

ELSE
   BEGIN { Get info from keyboard }
   REPEAT
      Write ('Enter range for the real (horiz.) axis:  ');
      {$I-} Readln (Low.R, High.R) {$I+}
   UNTIL (IOResult=0) AND (Low.R<>High.R);

   IF Low.R>High.R THEN
      BEGIN
      Temp := Low.R;
      Low.R := High.R;
      High.R := Temp
      END (* THEN *);

REPEAT
   Write ('Enter range for the imaginary (vert.) axis:  ');
   {$I-} Readln (Low.I, High.I) {$I+}
UNTIL (IOResult=0) AND (Low.I<>High.I);

IF Low.I>High.I THEN
   BEGIN
   Temp := Low.I;
   Low.I := High.I;
   High.I := Temp
   END (* THEN *);

REPEAT
   Write ('Enter maximum iteration count:  ');
   {$I-} Readln (Max_Count) {$I+}
UNTIL IOResult=0;
```

```
  IF Max_Count<10 THEN
     Max_Count := 10;

  Write ('Enter note:  ');
  Readln (Screen_Data.Note);
  Start_Y := 0;
  Sub_Comp (High, Low, Delta);
  Ratio := Delta.I / Delta.R;
  SetGraphMode (Graph_Mode);

  IF Ratio>=Aspect THEN
     BEGIN
     Dots_Horizontal := Round ((Max_X + 1) * Aspect / Ratio) - 1;
     Dots_Vertical := GetMaxY
     END {* THEN *}

  ELSE
     BEGIN
     Dots_Vertical := Round ((GetMaxY + 1) * Ratio / Aspect) - 1;
     Dots_Horizontal := Max_X
     END {* ELSE *};
       WITH Screen_Data DO
          BEGIN
          Dots_H := Dots_Horizontal;
          Dots_V := Dots_Vertical;
          Count := Max_Count;
          Str (Low.I, Low_Imag);
          Str (Low.R, Low_Real);
          Str (High.I, High_Imag);
          Str (High.R, High_Real);
          No_Blank (Low_Imag);
          No_Blank (Low_Real);
          No_Blank (High_Imag);
          No_Blank (High_Real)
          END {* WITH *};

       Picture_Loaded := False;
       File_Name := ''
       END {* ELSE *}
    END {* THEN *};

IF Picture_Loaded THEN
    BEGIN { Dump picture onto the screen }
    SetGraphMode (Graph_Mode);

    WITH Screen_Data DO
       BEGIN
       Start_Y := Start;
       Max_Count := Count;
       Dots_Horizontal := Dots_H;
       Dots_Vertical := Dots_V;
       Val (Low_Real, Low.R, X);
       Val (Low_Imag, Low.I, X);
       Val (High_Real, High.R, X);
       Val (High_Imag, High.I, X)
       END {* WITH *};

    FOR Y := 0 TO Start_Y-1 DO
       FOR X := 0 TO Dots_Horizontal DO
          Plot (X, Y, Screen[Y]^[X]);

    IF Start_Y>GetMaxY THEN
       Sub_Picture { Get a subregion of the completed picture }
    ELSE
       Sub_Comp (High, Low, Delta) { Continue drawing the picture }
    END {* THEN *};

Delta.R := Delta.R / (Dots_Horizontal + 1);
Delta.I := Delta.I / (Dots_Vertical + 1)
END {* Define_Screen *};

{*****************************************************************}

PROCEDURE Generate;

{  This is where most of the program's time is spent, generating the
screen.  The section marked 1* is where code has been optimized by
putting the complex-number math instructions in this procedure rather
than calling the actual procedures.  }

VAR
   X, Y, Count:  integer;
   Z_Point, C_Point:  Complex;
   Temp:  double;

BEGIN {* Generate *}
Plot (Dots_Horizontal, Dots_Vertical, Max_Colors);
C_Point.I := High.I - Start_Y * Delta.I;
Y := Start_Y;

WHILE (Y<=Dots_Vertical) AND NOT KeyPressed DO
   BEGIN
   FillChar (Screen[Y]^, Scan_Width+1, 0);
   C_Point.R := Low.R - Delta.R;

   FOR X := 0 TO Dots_Horizontal DO
      BEGIN
      Plot (X, Y, Max_Colors);
      C_Point.R := C_Point.R + Delta.R;
      Z_Point := C_Point;
      Count := 0;

      WHILE (Count<=Max_Count) AND (Square_Size_Of_C (Z_Point)<4.0) DO
         BEGIN
{ 1*     Mult_Comp (Z_Point, Z_Point, Z_Point);  }
{ 2*     Add_Comp (Z_Point, C_Point, Z_Point);  }

         Temp := Sqr (Z_Point.R) - Sqr (Z_Point.I) + C_Point.R;
         Z_Point.I := 2.0 * Z_Point.I * Z_Point.R + C_Point.I;
         Z_Point.R := Temp;
         Count := Succ (Count)
         END {* WHILE *};

      IF Count<Max_Count THEN
         Screen[Y]^[X] := Use_Color[Count MOD Color_Count];

      Plot (X, Y, Screen[Y]^[X])
      END {* FOR *};

   C_Point.I := C_Point.I - Delta.I;
   Y := Y + 1
   END {* WHILE *};

Screen_Data.Start := Y
END {* Generate *};

{*****************************************************************}

PROCEDURE Wrap_Up;

{  This procedure deals with the shutting down of a picture.  }

VAR
   X:  integer;

BEGIN
Picture_Loaded := True;

IF KeyPressed THEN
   Sound (440)

ELSE
   BEGIN
   Sound (660);
   Delay (20);
   Sound (1000)
   END {* ELSE *};

Delay (50);
NoSound;

Ch := ReadKey;

RestoreCrtMode;
Write ('Save picture? (Y/N) ');

REPEAT
   Ch := ReadKey
UNTIL Ch IN Yes_N_No;

Writeln (Ch);

IF Ch IN Yes THEN
   BEGIN
   IF File_Name<>'' THEN
      BEGIN
      Write ('Save as ', File_Name, '? (Y/N) ');

      REPEAT
         Ch := ReadKey
      UNTIL Ch IN Yes_N_No;

      Writeln (Ch)
      END {* THEN *}

   ELSE
      Ch := 'N';

   IF Ch IN No THEN
      BEGIN
      Write ('Enter filename to save it in:  ');
      Readln (File_Name)
      END {* THEN *};

   Assign (Screen_File, File_Name);
   Rewrite (Screen_File);
   Write (Screen_File, Data_Line);

   FOR X := 0 TO Screen_Data.Start-1 DO
      Write (Screen_File, Screen[X]^);

   Close (Screen_File)
   END {* THEN *};

Write ('Do another? (Y/N) ');

REPEAT
   Ch := ReadKey
UNTIL Ch IN Yes_N_No;

Writeln (Ch)
END {* Wrap_Up *};
```

```
{*****************************************************************}

BEGIN (* main *)
Initialize;

REPEAT
   Define_Screen;
   Generate;
   Wrap_Up
UNTIL Ch IN No
END.
```

LISTING 2: CMPLX.PAS

```
UNIT Cmplx;

{  In the following descriptions,
   Capital letters (A, B) are real numbers or real parts of complex
   numbers.  Lowercase letters (a, b) are real factors of complex
   parts.  i is the square root of -1 (√-1).

CONTENTS:  Add_Comp          (C1, C2, C_Out);
           Sub_Comp          (C1, C2, C_Out);
           Mult_Comp         (C1, C2, C_Out);
           Mult_RC           (C, R, C_Out);
           Sub_C_From_R      (R, C, C_Out);
           Div_C_By_R        (C, R, C_Out);
           Size_Of_C         (C);
           Square_Size_Of_C (C);
}

INTERFACE

TYPE
   Complex = RECORD
               R, I:  double
             END;

PROCEDURE Add_Comp (A, B:  Complex;
                    VAR C:  Complex);

PROCEDURE Sub_Comp (A, B:  Complex;
                    VAR C:  Complex);

PROCEDURE Mult_Comp (A, B:  Complex;
                     VAR C:  Complex);

PROCEDURE Div_Comp (A, B:  Complex;
                    VAR C:  Complex);

PROCEDURE Div_R_By_C (R:  double;
                      C:  Complex;
                      VAR C_Out:  Complex);

PROCEDURE Mult_RC (C:  Complex;
                   R:  double;
                   VAR C_Out:  Complex);

PROCEDURE Sub_C_From_R (R:  double;
                        C:  Complex;
                        VAR C_Out:  Complex);

PROCEDURE Div_C_By_R (C:  Complex;
                      R:  double;
                      VAR C_Out:  Complex);

FUNCTION Size_Of_C (C:  Complex):  double;
FUNCTION Square_Size_Of_C (C:  Complex):  double;

{*****************************************************************}

IMPLEMENTATION

PROCEDURE Add_Comp (A, B:  Complex;
                    VAR C:  Complex);

{  RESULT == (A+ai)+(B+bi) == A+ai+B+bi == (A+B)+(a+b)i  }

BEGIN
C.R := A.R + B.R;
C.I := A.I + B.I
END (* Add_Comp *);

{*****************************************************************}

PROCEDURE Sub_Comp (A, B:  Complex;
                    VAR C:  Complex);

{  RESULT == (A+ai)-(B+bi) == A+ai-B-bi == (A-B)+(a-b)i  }

BEGIN
C.R := A.R - B.R;
C.I := A.I - B.I
END (* Sub_Comp *);

{*****************************************************************}

PROCEDURE Mult_Comp (A, B:  Complex;
                     VAR C:  Complex);

{  RESULT == (A+ai)(B+bi) == AB + Abi + Bai + aibi ==
             (AB-ab)+(Ab+aB)i  }

BEGIN
C.R := A.R * B.R - A.I * B.I;
C.I := A.R * B.I + A.I * B.R
END (* Mult_Comp *);

{*****************************************************************}

PROCEDURE Div_Comp (A, B:  Complex;
                    VAR C:  Complex);

{  RESULT == (A+ai)/(B+bi) == (AB+ab)/(B²+b²)+((aB-Ab)/(B²+b²))i  }

VAR
   D:  double;

BEGIN
D := Sqr (B.R) + Sqr (B.I);
C.R := (A.R * B.R + A.I * B.I) / D;
C.I := (A.I * B.R - A.R * B.I) / D
END (* Div_Comp *);

{*****************************************************************}

PROCEDURE Div_R_By_C (R:  double;
                      C:  Complex;
                      VAR C_Out:  Complex);
VAR
   A:  Complex;

BEGIN
A.R := R;
A.I := 0;
Div_Comp (A, C, C_Out)
END (* Div_R_By_C *);

{*****************************************************************}

PROCEDURE Mult_RC (C:  Complex;
                   R:  double;
                   VAR C_Out:  Complex);

{  RESULT == (C+ci)R == CR+cRi  }

BEGIN
C_Out.R := C.R * R;
C_Out.I := C.I * R
END (* Mult_RC *);

{*****************************************************************}

PROCEDURE Sub_C_From_R (R:  double;
                        C:  Complex;
                        VAR C_Out:  Complex);

{  RESULT == R-(C+ci) == R-C-ci == (R-C)-ci  }

BEGIN
C_Out.R := R - C.R;
C_Out.I := -C.I
END (* Sub_C_From_R *);

{*****************************************************************}

PROCEDURE Div_C_By_R (C:  Complex;
                      R:  double;
                      VAR C_Out:  Complex);

{  RESULT == (C+ci)/R == C/B+ci/R == (C/R)+(c/R)i  }

BEGIN
C_Out.R := C.R / R;
C_Out.I := C.I / R
END (* Div_C_By_R *);

{*****************************************************************}

FUNCTION Size_Of_C (C:  Complex):  double;

{  RESULT == √(C²+c²)  }

BEGIN
Size_Of_C := Sqrt (Sqr (C.R) + Sqr (C.I))
END (* Size_Of_C *);

{*****************************************************************}

FUNCTION Square_Size_Of_C (C:  Complex):  double;

{  RESULT == C² + c²  }

BEGIN
Square_Size_Of_C := Sqr (C.R) + Sqr (C.I)
END (* Square_Size_Of_C *);

END.
```

USING UNITS TO HIDE DATA STRUCTURE DETAILS

Hiding data structure details is more than just keeping secrets—it can make everyone's work easier.

Marshall Brain

SQUARE ONE

There is more to Turbo Pascal's new units feature than separate compilation. In "Getting to Know Units" (*TURBO TECHNIX*, November/December, 1987), Tom Swan explained how units may be used to divide a large program into smaller, more maintainable modules. This is certainly units' major role in Turbo Pascal programming, but like *packages* in ADA and *modules* in Modula-2, units also play a role in facilitating portable coding, and in limiting the use of intermodule "sneak paths" and other undesirable programming habits.

UNIT STRUCTURE

Let's recap unit structure briefly. Every unit has two parts: an interface section and an implementation section.

The *interface* section of the unit contains declarations of constants, types, and variables, as well as the definitions of "public" subprograms and their parameters. These subprograms and declarations are globally "visible" to any program using the unit. For example:

```
UNIT MyUnit;

INTERFACE

VAR
  A,B,C:INTEGER;

PROCEDURE P(VAR I:INTEGER);
```

In any program that uses unit **MyUnit**, the variables **A**, **B**, and **C** will be global variables, and procedure **P** will be a public procedure.

The *implementation* section of a unit comes after the interface section, and is "invisible" to programs that use the unit. The implementation section may contain its own "private" constants, variables and types, but a program that uses the unit cannot access them. This also applies to the actual bodies of the procedures declared in the interface section. These bodies are part of the implementation section, and are also invisible to programs that use the unit.

The interface section of a unit gives a program access to the objects and capabilities inside of the unit. The implementation section allows the programmer to hide data structures and all code used to manipulate those structures.

JUST ENOUGH KNOWLEDGE

When a team of several programmers develops large programs, this dual structure of units forces a kind of modularity that is very different from that obtained by cutting monolithic programs up into libraries of procedures. The structure of units allows each programmer to be assigned a complete, standalone portion of the larger program that can be compiled and tested on its own. The programming team is forced to define and resolve the interfaces among all modules before programming can begin. The interface sections of the units can actually be written as part of the program spec, long before programming begins in earnest. When it comes time to integrate the units together into the complete program, this upfront work on the module interfaces pays off by making the integration and subsequent program verification happen much more quickly and smoothly.

The invisibility of each unit's implementation section gives units another important advantage. The implementation section can be used to hide details that cloud a program's structure, or to hide code that may need to be changed in the future. Programmers who are not responsible for writing the unit itself are given just enough knowledge to use the unit's facilities in their own work, but not enough knowledge to allow them to make unwarranted assumptions about the internals of program modules written by others. "The **IntervalTimer** procedure keeps its seconds count in a **word** variable internally for speed," a programmer might think, "so I can use a **word** variable to hold intervals and not worry about overflow, even though the interface is a **LongInt**." Units would keep that programmer from making that mistake by hiding the way **IntervalTimer** keeps house internally.

Units also allow complex, confusing code to be packaged in a form that is much easier to use. The confusing details are hidden within the unit, while any program using the unit accesses the code through a relatively small number of easy-to-understand procedure and function calls.

MAKING USE OF INVISIBILITY

The true function of any high-level language is the artful hiding of program details. Consider one simple Turbo Pascal statement:

```
New(Node);
```

Here, the program creates a variable on the heap, which is pointed to by a pointer named **Node**. The process behind this statement involves identifying the type to which **Node** points, determining the size of **Node**'s referent, traversing the free list to find a block of free memory large enough to contain the referent, modifying the free list to indicate that another chunk of heap memory has been allocated and used, and finally loading the address of the allocated space into the pointer variable **Node**. Since all of that complication is hidden behind one small statement, the programmer can concentrate instead on what to do with **Node** and its dynamic referent on the heap.

Similarly, hiding certain things in the implementation section of one or more units makes construction of the main program much easier, and also makes code changes completely transparent to all programs that use the unit.

This technique is especially powerful when applied to data structures. Consider this scenario: You have written a large and complex data management program for your company that is based on a very large linked list of data. Many programs written for in-house use revolve completely around a single large data structure in this manner.

As people in your company become dependent on this pro-

continued on page 38

LISTING 1: LIST.PAS

```
unit addr_list;

  {Unit to hide a linked list data structure from the main
   program.}

  {Marshall Brain   Box 37224 Raleigh, NC 27597  ver 1.0  9/13/87}

  INTERFACE
    {This portion of the unit is used to describe the type of
     objects used by the unit and the operations available to
     manipulate those objects. This section is visible to any
     program using this unit.}

    TYPE
      name_string=STRING[10];
      Addr=RECORD
        last_name,first_name:name_string;
        street:STRING[40];
        city:STRING[10];
        state:STRING[2];
        zip:STRING[10];
        phone:STRING[15];
        comment:STRING[40];
      END;

    PROCEDURE load_file(filename:STRING; VAR error:Boolean);
      {LOAD_FILE attempts to load the data structure from
       the filename specified. If unsuccessful, ERROR will
       be true. File is assumed to be in sorted order.}

    PROCEDURE create_file(filename:STRING);
      {creates a new file of name FILENAME.}

    PROCEDURE save_file;
      {SAVE_FILE saves the data structure back to the file it
       was loaded from.}

    PROCEDURE find_first(lname,fname:name_string; VAR rec:Addr;
                         no_match:Boolean);
      {FIND_FIRST will find the first record with a name that
       matches LNAME,FNAME. If a match is found, REC will contain
       the record found. Otherwise, NO_MATCH will be true and REC
       will contain garbage.}

    PROCEDURE find_next(lname,fname:name_string; VAR rec:Addr;
                        no_match:Boolean);
      {FIND_NEXT will find the next record matching LNAME,FNAME.
       It is assumed that FIND_FIRST was used first. NO_MATCH
       is set if there are no matches.}

    PROCEDURE add_rec(rec:Addr; VAR error:Boolean);
      {ADD_REC will add REC to the data structure, maintaining
       that data structure in sorted order by name. If the data
       structure is full, ERROR will be set true.}

    PROCEDURE delete_rec;
      {deletes the last record found using one of the FIND rtns.}

    PROCEDURE change_rec(rec:Addr);
      {replaces the last record found using one of the find rtns
       with rec. First and last name should not be changed, as this
       will destroy the linked list order. If the name needs
       to change, use DELETE_REC and ADD_REC instead.}

    FUNCTION size:word;
      {SIZE will contain the number of records in the data
       structure.}

    FUNCTION full:Boolean;
      {FULL will be false if space remains in the data structure.}
```

```
IMPLEMENTATION
    {This portion of the unit is invisible to the program, and
     can be used to hide that data structure.}

    {The data structure is currently implemented as a linked list.}
    TYPE
      pntr=^ll_rec;
      ll_rec=RECORD
        a:Addr;
        next,prev:pntr;
      END;
    VAR
      first,last,curr:pntr;
      f:FILE of Addr;
      found:Boolean;

    PROCEDURE init;
    {A hidden routine used to init variables.}
    BEGIN
      first:=NIL;
      last:=NIL;
      curr:=NIL;
      found:=False;
    END;

    PROCEDURE load_file{filename:STRING; VAR error:Boolean};
      {LOAD_FILE attempts to load the data structure from
       the filename specified. If unsuccessful, ERROR will
       be true. File is assumed to be in sorted order.}
    VAR temp:Addr; p:pntr; err:Boolean;
    BEGIN
      init;
      {make sure that file exists.}
      Assign(f,filename);
      {$i-} Reset(f); {$i+}
      IF IOResult=0 THEN
      BEGIN
        WHILE NOT EOF(f) DO
        BEGIN
          {append new records to the end of the linked list.}
          Read(f,temp);
          New(p);
          {init p}
          p^.a:=temp;
          p^.next:=NIL;
          p^.prev:=last;
          {create the links.}
          IF (first=NIL) THEN
            first:=p
          ELSE
            last^.next:=p;
          last:=p;
        END;
        Close(f);
      END
      ELSE
        error:=True;
    END;

    PROCEDURE create_file{filename:STRING};
      {creates a new file of name FILENAME.}
    BEGIN
      Assign(f,filename);
      init;
    END;

    PROCEDURE save_file;
      {SAVE_FILE saves the data structure back to the file it
       was loaded from.}
```


continued from page 37

gram, they begin to demand more speed. (And as more data is continually added to the list, the speed, of course, slows down.) After examining your program, you decide that the best way to speed it up is to abandon the linked list structure and switch to an efficient B-tree data structure. Unfortunately, literally thousands of references to the linked list are scattered throughout your program. It would take a great deal of time to find and change all those references, and the process would probably spawn any number of bugs in the code, all of which would have to be eliminated.

Units can prevent situations like this by completely hiding the nature and details of the data structure from the main program. To use units in this way, design a procedural interface to your data structure (in other words, make *no* direct references to the structure itself anywhere in the program) and define this set of routines in the interface section of a unit. For a data management unit, you might decide that the following routines are needed: **create_file**, **load_file**, **save_file**, **add_rec**, **delete_rec**, **change_rec**, **find_first**, **find_next**, **size**, and **full**. With these routines and a data record format in mind, you can write the interface section of your unit.

Once the interface to the unit is designed, you can completely hide your implementation of the data structure in the implementation section of the unit. An example unit, using a linked list data structure implementation, is shown in Listing 1, LIST.PAS. This unit implements the data management routines listed at the end of the previous paragraph. In the **List** unit, the design of the data structure is completely invisible to the main program. Only the data record type, **Addr**, and the headers of the routines needed to manipulate the data structure are visible to users of the unit. Note the critical fact in reading the interface section of **List**: It specifies *what* each routine in the unit does without indicating *how*.

THE BENEFITS OF STRUCTURE SECRECY

When you separate a program from the data structure in this way, you gain four advantages:

1. The program that uses the unit is conceptually easier to understand, because you now access the data structure solely through a set of high-level commands. The absence of more complex low-level data structure manipulation code makes your program much cleaner.
2. The data structure can be completely rebuilt at any time. You might start with an array because it's simple and fast for small quantities of data, then move to a linked list for its larger data space, and switch later to a B-tree for its faster access time with large databases. The main program never changes *at all*, even though the structure containing its data changes dramatically through several revisions.
3. If you ever need these same capabilities and data structure in another program (which is likely), you can easily reuse your unit with only minor modifications to the data record declaration.
4. The use of specific interface routines makes the data structure much more reliable. Once the interface routines have been thoroughly debugged, it's impossible to corrupt the data structure through the main program. This debugged code can then be used over and over again.

Units can hide the details of any kind of data structure you wish to use. For example, a stack unit can be created using **PUSH**, **POP**, and **CLEAR** as the stack interface routines. The stack itself can be implemented using an array, a linked list, a file, or whatever is appropriate, and the main program remains oblivious to implementation details. The same technique can be applied to queues, ring buffers, and so forth.

Porting programs between hardware environments is another instance where hiding

continued on page 40

```
VAR p:pntr;
BEGIN
  Rewrite(f);
  p:=first;
  {loop to end of linked list.}
  WHILE (p<>NIL) DO
  BEGIN
    {some I/O checking could be added.}
    Write(f,p^.a);
    {dispose of LL as it is saved.}
    first:=first^.next;
    dispose(p);
    p:=first;
  END;
  init;
  Close(f);
END;

PROCEDURE find(lname,fname:name_string; VAR rec:Addr;
               VAR no_match:Boolean);
{This hidden routine loops through the LL looking for the
 name passed.}
VAR stop:Boolean;
BEGIN
  stop:=False;
  no_match:=True;
  {loop until end of list, match found, or past where name
   should be.}
  WHILE (curr<>NIL) AND no_match AND NOT stop DO
  BEGIN
    IF (lname>curr^.a.last_name) THEN {check next rec.}
      curr:=curr^.next
    ELSE IF (lname=curr^.a.last_name) THEN
    {check for first name match.}
    BEGIN
      IF (fname>curr^.a.first_name) THEN {check next rec.}
        curr:=curr^.next
      ELSE IF (fname=curr^.a.first_name) THEN {match found.}
      BEGIN
        rec:=curr^.a;
        no_match:=False;
      END
      ELSE {beyond where name can be.}
        stop:=True;
    END
    ELSE {beyond where name can be.}
      stop:=True;
  END;
END;

PROCEDURE find_first{lname,fname:name_string; VAR rec:Addr;
                    no_match:Boolean};
  {FIND_FIRST will find the first record with a name that
   matches LNAME,FNAME. If a match is found, REC will contain
   the record found. Otherwise, NO_MATCH will be true and REC
   will contain garbage.}
BEGIN
  curr:=first;
  find(lname,fname,rec,no_match);
  found:=NOT no_match;
END;

PROCEDURE find_next{lname,fname:name_string; VAR rec:Addr;
                    no_match:Boolean};
  {FIND_NEXT will find the next record matching LNAME,FNAME.
   It is assumed that FIND_FIRST was used first. NO_MATCH
   is set if there are no matches.}
BEGIN
  curr:=curr^.next;
  find(lname,fname,rec,no_match);
  found:=NOT no_match;
END;
```

```
PROCEDURE add_rec{rec:Addr; VAR error:Boolean};
  {ADD_REC will add REC to the data structure, maintaining
   that data structure in sorted order by name. If the data
   structure is full, ERROR will be set true.}
VAR temp:Addr; no_match:Boolean; p:pntr;
BEGIN
  {check for heap overflow.}
  IF (MemAvail>SizeOf(Addr)) THEN
  BEGIN
    {find where new rec should go.}
    curr:=first;
    find(rec.last_name,rec.first_name,temp,no_match);
    {create new rec and link it in.}
    New(p);
    p^.a:=rec;
    p^.next:=curr;
    IF curr=NIL THEN p^.prev:=last ELSE p^.prev:=curr^.prev;
    IF curr=first THEN first:=p
      ELSE IF (curr=NIL) THEN last^.next:=p
        ELSE curr^.prev^.next:=p;
    IF curr=NIL THEN last:=p ELSE curr^.prev:=p;
    error:=False;
  END
  ELSE
    error:=True;
END;

PROCEDURE delete_rec;
  {deletes the last record found using one of the FIND rtns.}
VAR p:pntr;
BEGIN
  IF found AND (curr<>NIL) THEN
  BEGIN
    WITH curr^ DO
    BEGIN
      {unlink rec and dispose of it.}
      IF curr=first THEN first:=next ELSE prev^.next:=next;
      IF curr=last THEN last:=prev ELSE next^.prev:=prev;
      dispose(curr);
    END;
  END;
END;

PROCEDURE change_rec{rec:Addr};
  {replaces the last record found using one of the find rtns
   with rec.}
BEGIN
  IF found AND (curr<>NIL) THEN
    curr^.a:=rec;
END;

FUNCTION size{:word};
  {SIZE will contain the number of records in the data
   structure.}
VAR cnt:word; p:pntr;
BEGIN
  p:=first;
  cnt:=0;
  WHILE (p<>NIL) DO
  BEGIN
    p:=p^.next;
    cnt:=cnt+1;
  END;
  size:=cnt;
END;

FUNCTION full{:Boolean};
      {FULL will be false if space remains in the data structure.}
    BEGIN
      IF MemAvail<SizeOf(Addr) THEN full:=True ELSE full:=False;
    END;

{initialization code for the unit.}
BEGIN
  init;
END.
```

UNITS

continued from page 39

data structure implementation details can be critical. For example, many mainframe programs use large arrays (such as 500 × 500 element two-dimensional integer arrays) that are literally too large to fit into one of the PC's 64K memory segments. Such a program can usually be moved to a PC—memory for the code is available in most cases. But to allow the program to operate, the array has to be broken down into some other form, such as a linked list of smaller one-dimensional arrays on the heap, or even a large disk file. In a large program, the number of direct array references would be huge, and the process of inserting and debugging all of the changes needed to convert the program for execution on a PC could take months. A unit consisting of a pair of array reference routines called **GET** and **SET** can hide the array's implementation, and make changing the data structure much easier when the program is moved to different systems with different restrictions on data size and representation. Turbo Pascal is not available for mainframes, of course, but even in the absence of units, data structure references can be kept out of the main program. The larger principles are valuable with any language in any environment.

When properly used, units offer a number of advantages to the programmer. They allow programs to be broken into reusable modules, and also allow certain details of the program's design to be hidden from the main program. Hiding data structure details in this way makes it easier to design and implement programs cleanly, to modify programs as they evolve, and to port programs among various machine environments. ■

Marshall Brain is a Pascal instructor at North Carolina State University. He can be reached at Box 37224, Raleigh, North Carolina 27627.

Listings may be downloaded from CompuServe as HIDE.ARC.

INTERFACING THE DOS PRINT SPOOLER

Use the DOS Multiplex Interrupt to spool files for printing from within your Turbo Pascal programs.

Duane L. Geiger

WIZARD

Version 3.0 of DOS provides application programs with an interface to some of the utility programs included with DOS. This interface—the *DOS Multiplex Interrupt* ($2F)—allows your Turbo Pascal programs to inspect the installed state of a selected few of these utilities.

PRINT.COM

One of the utilities included with DOS that makes use of the Multiplex Interrupt is PRINT.COM, a resident program that manages print queues. This program allows normal foreground processing to continue while files are printed as a background task. PRINT was the first documented utility to use the Multiplex Interrupt. Using the Multiplex Interrupt in conjunction with PRINT allows you to submit a file to PRINT for printing, remove a file or files from the print queue, or inspect the current status of the print queue.

When installed with its default settings, PRINT uses a little more than 5K of memory for the resident portion of the program and the print queue. PRINT has several command line installation options that change the amount of memory reserved for PRINT's operation. Refer to your DOS reference manual for a complete discussion and examples.

THE MULTIPLEX INTERRUPT

The Multiplex Interrupt ($2F) was introduced into the DOS documentation in version 3.0 as a standard way to inspect the installed status of a few of the utilities bundled with DOS. The use of the Multiplex Interrupt may be compared to the use of the DOS services interrupt ($21), except that instead of providing the familiar DOS services, the Multiplex Interrupt furnishes the services of DOS utility programs.

As of this writing, four DOS utility programs are documented as supporting the Multiplex Interrupt: the Print Queue Manager (PRINT), the Route Disk I/O utility (ASSIGN), the File Sharing utility (SHARE), and the Disk Spanning utility (APPEND). This interrupt provides an excellent method for these utilities to handle their respective functions and provide "hooks" for application programs. In order to avoid conflict with other utilities, each supported utility is assigned a unique identification code, which is reserved by standardizing the interrupt function call in the documentation.

MULTIPLEX CALLING CONVENTION

The Multiplex Interrupt uses a standard calling convention—registers are loaded in a consistent manner and the interrupt is performed. In order to call the Multiplex Interrupt, load the AH register with the appropriate identification code for the utility program you want to access (see Table 1). Then load the AL register with the selected function code (see Table 2), and finally, make the interrupt call:

```
Intr($2F,Regs);
```

When the call returns to your program, check the carry flag. If the carry flag is set, an error has occurred. The appropriate error code appears in register AX (see Table 3).

All of the utility programs listed in Table 1 support the Get Installed State function code (AL=0). Only PRINT supports the additional (AL) functions listed in Table 2. As an example of Get Installed State, see the **ShareInstalled** function in the unit SPOOL.PAS (Listing 1). **ShareInstalled** loads the registers with appropriate values and issues the Multiplex Interrupt. This function tests the version of DOS; if the appropriate version of DOS is running, **ShareInstalled** issues the Get Installed State command for SHARE (AH=$10, AL=$00). If SHARE is resident, the function returns a value of **True**.

PRINT QUEUE MANAGEMENT

The print queue manager (PRINT.COM) supports the functions that allow your programs to take direct

continued on page 42

LISTING 1: PR.PAS

```
PROGRAM Pr;              { Multiplex Interrupt Test                    }
{$R-,S-,I-,D-,T-,F-,V-,B-,N-}
{$M 16384,0,655360 }
{                                                                      }
{           PR.PAS - Interrupt $2F (Multiplex) Demonstration           }
{                              by                                      }
{                         Duane L. Geiger                              }
{                                                                      }
  USES DOS,CRT,Spool;

  VAR
    Installed : Boolean;          { Print Spooler Installed Switch     }
    Name      : PathName;         { Full Path and Filename             }
    Ch        : Char;             { Used to Pause the Program          }

  BEGIN
    {  The first demonstration is to see if SHARE is installed.  The   }
    {  same technique used to detect SHARE can also be used to test    }
    {  any utility which supports Get Installed State.                 }

    Installed:=ShareInstalled;         { If SHARE is installed        }
    WriteLn('Share Installed: ',Installed,' Error:',Error);

    {  The next part of the program simply checks to see if you have   }
    {  PRINT.COM installed, and if you have DOS 3.0 or greater.        }
    {  If either condition is not met, an error is returned.           }

    Installed:=SpoolerInstalled;       { If Print Spooler installed   }
    WriteLn('PRINT.COM Installed: ',Installed,' Error:',Error);
    IF NOT Installed THEN Halt(Error); { Can't continue this program  }

    {  Now queue up a name to the print program.                       }

    Name:='PR.PAS';                    { Name of actual file to queue}
    Write('Submitting: ',Name,' - ');  { Print a test message         }
    SpoolerSubmit(Name);               { Submit name to spooler       }
    IF OK THEN WriteLn('Successful')   { Everything went OK           }
    ELSE WriteLn('Error encountered:',Error); { Display error message}

    Name:='SPOOL.PAS';                 { Submit another file to print}
    Write('Submitting: ',Name,' - ');  { Display a message            }
    SpoolerSubmit(Name);               { Submit name(s) to spooler    }
    IF OK THEN WriteLn('Successful')   { Did it work correctly?       }
    ELSE WriteLn('Error encountered:',Error); { or produce an error   }

    WriteLn('Press Any Key to Continue');   { Now printing PR.PAS     }
    REPEAT UNTIL KeyPressed;           { Wait for a keystroke         }
    REPEAT Ch:=Readkey; UNTIL NOT KeyPressed; { Flush keyboard        }

    SpoolerStatusRead;                 { Pause and display queue      }
    IF OK THEN WriteLn('Successful')   { OK on status read            }
    ELSE WriteLn('Error encountered:',Error); { An Error of some type}

    WriteLn('Press Any Key to Continue');   { Another pause           }
    REPEAT UNTIL KeyPressed;           { Get a keystroke              }
    REPEAT Ch:=Readkey; UNTIL NOT KeyPressed; { Flush keyboard again  }

    Write('Spooler Release - ');       { Print test message           }
    SpoolerStatusEnd;                  { Start up the print job again}
    IF OK THEN WriteLn('Successful')   { It went OK                   }
    ELSE WriteLn('Error encountered:',Error); { or it failed somehow  }

    Write('Canceling: ALL - ');        { Message declaring intent     }
    SpoolerCancelAll;                  { Cancel entire print queue    }
    IF OK THEN WriteLn('Successful')   { It was successful or         }
    ELSE WriteLn('Error encountered:',Error); { it somehow failed     }

    Name:='Pr.Pas';                    { Cancel individual filename   }
    Write('Canceling: ',Name,' - ');   { Display test message         }
    SpoolerCancel(Name);               { Cancel by name               }
    IF OK THEN WriteLn('Successful')   { It will work                 }
    ELSE WriteLn('Error encountered:',Error); { or fail               }

  END.
```

THE PRINT SPOOLER

continued from page 41

TO ACCESS THE RESIDENT PORTION OF:	USE AH VALUE:
PRINT	$01
ASSIGN	$02
SHARE	$10
APPEND	$B7
Reserved by DOS	$00-$BF
Available values	$C0-$FF

Table 1. Multiplex Interrupt ID codes.

The multiplex interrupt was introduced into the DOS documentation in version 3.0 as a standard way to inspect the installed status of a few DOS utilities.

control of the print queue. You may submit files to the print queue, delete files from the print queue, and hold and release the print queue. However, while PRINT.COM is printing from the queue, you cannot *directly* print files to the printer because this would cause quite a mess at the printer itself. If a program attempts to print to the printer while PRINT.COM is printing, a critical error occurs.

The first step in providing queue management support is to report the status of the queue manager. The sample unit provides this facility by returning a Boolean value from function **SpoolerInstalled**. This function tests that the appropriate version of DOS is running, then issues the Get Installed State command for PRINT (AH=$01, AL=$00). If the queue manager is installed, the function returns **True**.

The next step in queue management support is submission of a file to the queue manager. The procedure **SpoolerSubmit** in the sample unit provides this facility. You must call this procedure with the full file- and pathname of the print image file to be placed in the print queue. Note: You may *not* use the DOS wildcard characters (*,?) in the submit filename string.

While PRINT.COM is printing from the queue, you cannot print directly to the printer, because this would cause quite a mess at the printer itself.

REGISTER AL FUNCTION CODE	FUNCTION DESCRIPTION
$00	Get Installed State
$01	Submit File
$02	Cancel File
$03	Cancel All Files
$04	Status Read
$05	Status End
$F8-$FF	Reserved by DOS

Table 2. Multiplex Interrupt function codes.

REGISTER AX ERROR CODES	ERROR DESCRIPTION
1	Invalid Function
2	File Not Found
3	Path Not Found
4	Too Many Open Files
5	Access Denied
8	Queue Full
9	Busy
12	Name Too Long
15	Invalid Drive

NOTE: On an error, the Carry Flag is set.

Table 3. Multiplex Interrupt error codes.

continued on page 44

LISTING 2: SPOOL.PAS

```
UNIT Spool;             {  PRINT.COM Spool Utility                    }
{                                                                     }
{              SPOOL.PAS - Queue Management Routines                  }
{                                 by                                  }
{                            Duane L. Geiger                          }
{                                                                     }
INTERFACE               { Globally Known Types and Variables          }

Uses
  DOS;                  { Standard TP4 DOS Unit                       }

TYPE
  PathName  = STRING[64];         { Path and Filename String Area     }

VAR
  OK        : Boolean;            { Global Flag for Success Tests     }
  Error     : Byte;               { Error Flag if Not OK on Interrupt}

{  Possible error conditions encountered in Functions 1 through 5     }
{  Carry Flag is Set and Register AX contains:                        }
{             1 = Function Code Invalid                               }
{             2 = File Not Found                                      }
{             3 = Path Not Found                                      }
{             4 = Too Many Open Files                                 }
{             5 = Access Denied                                       }
{             8 = Queue Full                                          }
{             9 = Spooler Busy                                        }
{           $0C = Name Too Long                                       }
{           $0F = Drive Invalid                                       }

FUNCTION  SpoolerInstalled  : Boolean;
PROCEDURE SpoolerSubmit( Name : PathName );
PROCEDURE SpoolerCancel( Name : PathName );
PROCEDURE SpoolerCancelAll;
PROCEDURE SpoolerStatusRead;
PROCEDURE SpoolerStatusEnd;

FUNCTION  ShareInstalled    : Boolean;

IMPLEMENTATION

TYPE
  ZName     = ARRAY[1..64] OF Byte;    { ASCIIZ FileName Area         }

VAR
  Regs      : Registers;          { Registers for DOS Unit Interface }
  Major,                          { Major version of DOS installed   }
  Minor     : Word;               { Minor version of DOS             }

PROCEDURE ASCIIZ( Name : PathName; VAR PassName : ZName );
{  Create an ASCIIZ Name from Pascal String }
  TYPE
    ASCIIZName = RECORD               { Use the structure of string }
      LnByte : Byte;                  { Length of the string        }
      NStr   : ZName;                 { Actual characters in string }
    END;
```

```
  VAR
    Nme      : ASCIIZName ABSOLUTE Name;          { Point the string   }

  BEGIN
    Nme.NStr[Nme.LnByte+1] := $00;      { Null terminate the string   }
    PassName := Nme.Nstr;               { Return the ASCIIZ portion   }
  END;

FUNCTION SpoolerInstalled : Boolean;
{  TRUE if Spooler Installed, Otherwise FALSE and an Error            }
{       AL Contains installed Status:                                 }
{          $00 = Not installed, OK to install                         }
{          $01 = Not installed, Not OK to install                     }
{          $FF = Installed                                            }
  BEGIN
    {  First test to see if DOS 3.00 or Greater is Running            }
    IF (Major >= 3) AND             { It's version 3.0 or greater     }
       (Minor >= 0) THEN            { Null then  (It's OK for $2F)    }
    ELSE BEGIN                      { This is not a good version of DOS}
      Error           := 1;         { Not Install and Can't Run       }
      SpoolerInstalled := False;    { Function Return                 }
      Exit;                         { Immeditate Exit                 }
    END;                            { of Error Condition              }

    {  Now test to see if resident portion of PRINT.COM is installed.}
    Error   := 0;                   { Reset System Wide Error         }
    Regs.AH := $01;                 { Select Resident Portion of Print }
    Regs.AL := $00;                 { Request the Installed Status    }
    Intr($2F,Regs);                 { Perform Status Interrupt        }
    IF ((Regs.Flags AND FCarry) <> 0) THEN BEGIN  { Flags Set/Error   }
      SpoolerInstalled := False;    { Set 'Not Installed' Switch      }
      Error           := Regs.AX;{ Load the Error Encountered         }
    END                             { of Error Condition              }
    ELSE                            { The Interrupt Succeeded         }
      SpoolerInstalled := (Regs.AL=$FF); { $FF Indicates Installed    }
  END;

PROCEDURE SpoolerSubmit( Name : PathName );
  TYPE
    Packet = RECORD                 { Submit Packet for PRINT.COM     }
      Level : Byte;                 { Printer Level                   }
      NPtr  : Pointer;              { Pointer to ASCIIZ name          }
    END;

  VAR
    SubPack  : Packet;              { Submit filename packet          }
    PassName : Zname;               { ASCIIZ filename to pass         }

  BEGIN
    Error := 0;                     { Clear the Global Error Flag     }
    OK    := True;                  { Assume it will work             }
    ASCIIZ( Name , PassName );      { Build an ASCIIZ name to Pass    }
    SubPack.Level := 0;             { Print 'Level'                   }
    SubPack.NPtr  := @PassName;     { Pointer to Filename             }
    Regs.AH       := $01;           { Resident portion of PRINT       }
    Regs.AL       := $01;           { Submit a file to PRINT          }
    Regs.DS       := Seg(SubPack);  { Segment pointer to packet       }
    Regs.DX       := Ofs(SubPack);  { Offset to submission            }
    Intr($2F,Regs);                 { MultiPlex interrupt             }
    IF ((Regs.Flags AND FCarry) <> 0) THEN BEGIN   { Flags Set/Error  }
      OK    := False;               { Error in file submission        }
      Error := Regs.AX;             { Load the Error Variable         }
    END;                            { of Error Handling               }
  END;
```

THE PRINT SPOOLER

continued from page 43

SpoolerSubmit prepares a data *packet* and passes the data packet's address to PRINT. This packet contains an ASCIIZ string for the full filename and a *level code*, which must be set equal to zero. (Currently, zero is the only level code accepted by the PRINT program. I suspect that this level code was originally intended to prioritize the print queue, but prioritization was never actually implemented in the PRINT program.) To complete the submission process, the registers are set (AH=$01, AL=$01, and DS:DX is loaded with the address of the submission packet), and the Multiplex Interrupt is issued. If successful, Boolean variable **OK** is set to **True**; otherwise, **OK** is set to **False** and the global variable **Error** is set to the value returned in register AX.

I suspect that the level code was originally intended to prioritize the print queue, but prioritization was never actually implemented in the PRINT program.

The procedures **SpoolerCancel** and **SpoolerCancelAll** handle another important aspect of queue management—the removal of files from the print queue. Unlike the submit function, the cancel function accepts wildcard characters as part of the filename string. To cancel a single filename from the print queue, call **SpoolerCancel** with the name of the file to be removed. To cancel all files in the print queue, use **SpoolerCancelAll**. If successful,

OK becomes **True**; otherwise, **OK** is set to **False** and register AX is copied to the **Error** global variable. When files being printed are canceled, either the message "File [filename.ext] canceled by operator" or "All files canceled by operator" is printed on the printer and a page eject occurs.

The first entry in the queue is the name of the file currently printing, and the end of the queue is marked by an entry whose first character is a null character.

The remaining procedures you'll need to control the print queue are **SpoolerStatusRead** and **SpoolerStatusEnd**, which let you examine the current contents of the queue and resume printing.

SpoolerStatusRead pauses the print queue printing activity and displays the filenames currently pending in the queue. As part of the status read function (AL = $04), a pointer to the first filename in the print queue is returned in registers DS:SI. The print queue consists of a series of 64-byte entries, with each entry terminated by a null character ($00). The first entry in the queue is the name of the file currently printing, and the end of the queue is marked by an entry whose first character is a null character. You may modify **SpoolerStatusRead** to return only

continued on page 46

```
PROCEDURE SpoolerCancel( Name : PathName );
{ Remove a Filename, Wildcards are allowed }
  VAR
    PassName : Zname;                    { ASCIIZ FileName to Pass     }

  BEGIN
    Error := 0;                          { Clear the Global Error Flag }
    OK    := True;                       { Assume it will work         }
    ASCIIZ(Name,PassName);               { Build an ASCIIZ name to pass}
    Regs.AH := $01;                      { Resident Portion of PRINT   }
    Regs.AL := $02;                      { Cancel a File in print queue}
    Regs.DS := Seg(PassName);            { Pointer to the ASCIIZ Name  }
    Regs.DX := Ofs(PassName);            { Offset to ASCIIZ Name       }
    Intr($2F,Regs);                      { Multiplex Interrupt         }
    IF ((Regs.Flags AND FCarry) <> 0) THEN BEGIN   { Flags Set/Error }
      OK    := False;                    { Error in File SubMission    }
      Error := Regs.AX;                  { Load the Error Variable     }
    END;                                 { of Error Handling           }
  END;

PROCEDURE SpoolerCancelAll;
{  Remove all names from print queue }
  BEGIN
    Error := 0;                          { Clear the Global Error Flag }
    OK    := True;                       { Assume it will work         }
    Regs.AH := $01;                      { Resident Portion of Print   }
    Regs.AL := $03;                      { Cancel All Files In Queue   }
    Intr($2F,Regs);                      { MultiPlex Interrupt         }
    IF ((Regs.Flags AND FCarry) <> 0) THEN BEGIN { Flags Set if Error}
      OK    := False;                    { Error in File Submission    }
      Error := Regs.AX;                  { Load the Error Variable     }
    END;                                 { of Error Handling           }
  END;

PROCEDURE SpoolerStatusRead;
{  Pause the Spooler and Read the names  }
  VAR
    QPtr : ^ZName;                       { Pointer to ASCIIZ String    }
    Idx  : LongInt ABSOLUTE QPtr;        { Index Pointer               }
    I    : Byte;                         { String Index                }
    Name : PathName;                     { Constructed File Name       }

  BEGIN
    Error := 0;                          { Clear the Global Error Flag }
    OK    := True;                       { Assume if will work         }
    Regs.AH := $01;                      { Resident Portion of Print   }
    Regs.AL := $04;                      { Hold for Status Read        }
    Intr($2F,Regs);                      { MultiPlex Interrupt         }
    IF ((Regs.Flags AND FCarry) <> 0) THEN BEGIN { Flags Set if Error}
      OK    := False;                    { Error in File SubMission    }
      Error := Regs.AX;                  { Load the Error Variable     }
    END                                  { of Error Handling           }
    ELSE BEGIN                           { Display the Queue           }

      { This section displays the names of all of the files that are }
      { currently in the print Queue.  If you want control, modify   }
      { this PROCEDURE to return the pointer to the names and bypass }
      { this display routine.                                        }
```

```
    QPtr := Ptr( Regs.DS , Regs.SI );{ Point to First Name in Queue}
    WHILE( QPtr^[1] <> $00 ) DO BEGIN  { Start a Display Loop      }
      I := 1;   Name := '';            { Index Pointer             }
      WHILE( QPtr^[I] <> $00 ) DO BEGIN   { Start displaying Names }
        Name := Name + Chr( QPtr^[I] );{ Build the Name String     }
        I := Succ(I);                  { Point to Next Character   }
      END;                             { of a Name Character       }
      WriteLn(Name);                   { Display the Name just built }
      Idx := Idx + 64;                 { Point to Next 64 Bytes    }
    END;                               { of Display Loop           }

  END;                                 { of Queue Display          }
END;                                   { of SpoolerStatusRead      }

PROCEDURE SpoolerStatusEnd;
{  Clear the Paused Condition }
  BEGIN
    Error := 0;                        { Clear the Global Error Flag }
    OK    := True;                     { Assume if will work       }
    Regs.AH := $01;                    { Resident Portion of Print  }
    Regs.AL := $05;                    { Clear Status Read          }
    Intr($2F,Regs);                    { MultiPlex Interrupt        }
    IF ((Regs.Flags AND FCarry) <> 0) THEN BEGIN { Flags Set if Error}
      OK    := False;                  { Error in File SubMission   }
      Error := Regs.AX;                { Load the Error Variable    }
    END;                               { of Error Handling          }
  END;

FUNCTION ShareInstalled : Boolean;
  BEGIN
    {  First test to see if DOS 3.00 or greater is running         }
    IF (Major >= 3) AND                { It's version 3.0 or greater }
       (Minor >= 0) THEN               { Null then  (It's OK for $2F) }
    ELSE BEGIN                         { This is not a good version of DOS}
      Error          := 1;             { Not Install and Can't Run  }
      ShareInstalled := False;         { Function Return            }
      Exit;                            { Immeditate Exit            }
    END;                               { of Error Condition         }

    {  Now test to see if resident portion of SHARE is installed.  }
    Error   := 0;                      { Reset System Wide Error    }
    Regs.AH := $10;                    { Select resident portion of SHARE }
    Regs.AL := $00;                    { Request the installed status }
    Intr($2F,Regs);                    { Perform Status Interrupt   }
    IF ((Regs.Flags AND FCarry) <> 0) THEN BEGIN { Flags Set if Error}
      ShareInstalled := False;         { Set 'Not Installed' switch }
      Error          := Regs.AX;       { Load the Error Encountered }
    END                                { of Error Condition         }
    ELSE                               { The interrupt succeeded    }
      ShareInstalled := (Regs.AL = $FF); { $FF indicates installed  }
  END;

BEGIN
  {  First test to see if DOS 3.00 or Greater is Running           }
  Regs.AH := $30;                      { Request Version Number     }
  MsDos(Regs);                         { Perform Interrupt $21      }
  Major   := Regs.AL;                  { Extract major version of DOS }
  Minor   := Regs.AH;                  { Extract minor version of DOS }
END.
```


continued from page 45

the pointer to the print queue so that you can perform your own decoding, rather than allow the procedure to blindly display the names in the print queue.

Finally, **SpoolerStatusEnd** releases the print queue manager from its paused state after a status read function call. **SpoolerStatusEnd**, or any other function call to PRINT through the Multiplex Interrupt, releases the queue and resumes printing. **SpoolerStatusEnd** is also used to display the contents of the print queue when no other action is to be taken.

You may modify SpoolerStatusRead to return only the pointer to the print queue so that you can perform your own decoding, rather than allow the procedure to blindly display the names in the print queue.

With this set of functions and procedures, you can implement print spool management in your everyday applications. After all, unless you need a long coffee break, why wait for reports to finish printing before you regain control of your computer? ■

Duane L. Geiger, author of Tele-Mark, has been an independent developer and consultant for the last nine years. He lives and works in Newport, Oregon.

Listings may be downloaded from CompuServe as SPOOL.ARC.

EXPLORING THE INTERRUPT VECTOR TABLE

The first 1K of PC RAM is the gatekeeper to your system's DOS and BIOS resources. Feel free to look ... and touch carefully.

Jeff Duntemann

PROGRAMMER

Most people think that DOS is the controlling hand within the IBM PC. While largely true, this idea doesn't credit the assistance of the IBM ROM BIOS, which does much of the work for DOS. Both DOS and the ROM BIOS are called through software interrupts.

YO!

An *interrupt* is a tap on the CPU's shoulder, telling the CPU that it must pay attention to something else *now*. Interrupt mechanisms have always been part of microprocessor systems. Until the development of the 8086 and 8088, all interrupts were hardware interrupts.

Hardware interrupts work this way: An electrical signal on one pin of the CPU chip causes the CPU logic to save the program counter, code segment register (CS), and flags register. The CPU is then free to service the request from outside the chip to execute some bit of code unrelated to its ongoing task. Once the request for service is satisfied, the CPU restores the registers it had saved and picks up its ongoing task as though nothing had happened.

With the 8086 family architecture, Intel presented the concept of the software interrupt. *Software* interrupts work exactly the same way as hardware interrupts, except that the triggering request is a machine instruction (software) rather than an electrical signal on a CPU pin (hardware).

Software interrupts provide standard entry points to system software. When any kind of interrupt happens, the CPU first saves essential registers, and then performs a "long jump" to the location of the interrupt service routine somewhere in memory.

The way that the CPU locates this interrupt service routine is critical. The 8088 recognizes 256 interrupts, numbered from 0 to 255. When an interrupt happens, the CPU must receive the number of the requested interrupt. For hardware interrupts, this number comes from the interrupt priority controller chip outside the CPU. For software interrupts, the interrupt number is built into the interrupt instruction. For example, the 8088 instruction **INT 21** triggers software interrupt 21H.

The first 1024 bytes of the PC memory map are reserved for interrupt vectors. "Vector" can be taken to mean "pointer," and pointers are exactly what occupy those 1024 bytes of memory. Each of the 256 different interrupts has its own 4-byte region of this 1024-byte memory block ($256 \times 4 = 1024$). This 4-byte region contains a 32-bit pointer to the first instruction of the interrupt's service routine.

The first four bytes of 8088 memory contain the vector for interrupt 0. The next four bytes of memory contain the vector for interrupt 1, and so on up to 255 (see Figure 1). Obviously, if the CPU knows the interrupt number, it can multiply that number by four and go immediately to the interrupt vector for any given interrupt. The first two bytes of the interrupt vector are the program counter value for the start of the service routine, and the second two bytes are the code segment value where that service routine exists. The CPU need only load the code segment value into the CS register and then load the

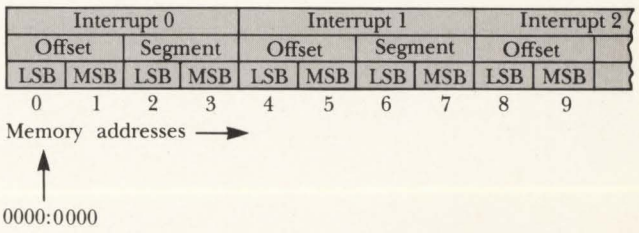


Figure 1. The structure of the 8088 interrupt vector table. There are 256 vectors in the table, each consisting of 4 bytes that represent the service routine segment and offset in the order shown.

continued on page 48

Editor's Note: This article is excerpted from the forthcoming book, *Complete Turbo Pascal, Third Edition*, by Jeff Duntemann. *Complete Turbo Pascal, Third Edition* is scheduled for July, 1988, publication by Scott, Foresman & Company.

LISTING 1: VECTORS.PAS

```
{-------------------------------------------------------------}
{                          VECTORS                            }
{                                                             }
{                   Interrupt vector utility                  }
{                                                             }
{                              by Jeff Duntemann              }
{                              Turbo Pascal V4.0              }
{                              Last update 3/15/88            }
{                                                             }
{ This program allows you to inspect and change 8086 interrupt }
{ vectors, and look at the first 256 bytes pointed to by any  }
{ vector.  This allows the spotting of interrupt service      }
{ routine "signatures" (typically the vendor's copyright      }
{ notice) and also indicates when a vector points to an IRET. }
{                                                             }
{      From: COMPLETE TURBO PASCAL, 3E  by Jeff Duntemann     }
{    Scott, Foresman & Co., Inc. 1988   ISBN 0-673-38355-5    }
{-------------------------------------------------------------}

PROGRAM Vectors;

USES DOS;      { For GetIntVec and SetIntVec }

{$V-}          { Relaxes type checking on string lengths }

CONST
  Up = True;

TYPE
  String80     = String[80];
  Block        = ARRAY[0..255] OF Byte;
  PtrPieces    = ARRAY[0..3] OF Byte;

VAR
  I              : Integer;
  VectorNumber   : Integer;
  Vector         : Pointer;
  VSeg,VOfs      : Integer;
  NewVector      : Integer;
  MemBlock       : Block;
  ErrorPosition  : Integer;
  Quit           : Boolean;
  Command        : String80;
  CommandChar    : Char;


PROCEDURE StripWhite(VAR Target : String);

CONST
  Whitespace  : SET OF Char = [#8,#10,#12,#13,' '];

BEGIN
  WHILE (Length(Target) > 0) AND (Target[1] IN Whitespace) DO
    Delete(Target,1,1)
END;


PROCEDURE WriteHex(BT : Byte);

CONST
  HexDigits : ARRAY[0..15] OF Char = '0123456789ABCDEF';

VAR
  BZ : Byte;

BEGIN
  BZ := BT AND $0F;
  BT := BT SHR 4;
  Write(HexDigits[BT],HexDigits[BZ])
END;
```

INTERRUPT VECTORS

continued from page 47

program counter value into the program counter register—and it's off and running the interrupt service routine.

The important fact here is that the code that wishes to use a software interrupt service routine doesn't need to know where that routine is located in memory—the code only needs to know the interrupt number. Indeed, the actual location of the service routine can change over time as the routine is altered or expanded. As long as the computer's boot or startup code stores the correct interrupt vectors into the lowermost 1024 bytes of memory, software interrupt service routines may be located anywhere and still be used quickly and easily by application programs.

This is the spirit of the IBM PC's ROM BIOS. The BIOS is a collection of software interrupt service routines stored in ROM at the very top of the 8088's memory address space. The interrupt numbers are assigned according to the general function performed by the interrupt service routine. For example, interrupt 16 (10H) controls video services for the PC. Interrupt 22 (16H) controls access to the keyboard.

Not all software interrupts are reserved for the use of the ROM BIOS; most of them are not used at all. DOS uses a few software interrupts, and the Microsoft and Logitech mouse drivers use one. Quite a few peripheral driver programs, in fact, make use of software interrupts. The PC as we know it would have been impossible without them.

GetIntVec AND SetIntVec

The interrupt vector table can be explored with DEBUG, but a fairly simple program makes the vectors easier to read and change. VECTORS.PAS (Listing 1) provides a utility called **Vectors** that displays and changes interrupt vectors, and also displays a hex dump of the first 256 bytes of memory pointed to by a vector. Changing an interrupt vector can be strong medicine, and shouldn't be done unless you know *exactly* what you're doing. Altering the timer tick interrupt carelessly freezes your machine solid in no more than 55 milliseconds' time. But if you intend to write programs that intercept interrupt vectors, **Vectors** can save a lot of aggravation during development.

One problem in reading simple hex dumps of the vector table is that the vectors' components are stored backward in memory from the way we human beings generally write them (see Figure 1). The offsets are stored before the segments, and the least significant bytes of both segments and offsets are stored before the most significant bytes.

Vectors reformats the interrupt vector table for your eyes, and lets you perform certain useful operations on the table. For example, it allows you to zero

out all 32 bits of a vector, and also lets you separately change the offset or segment portion of any vector to the value of your choice.

Vectors centers on two routines from Turbo Pascal's **DOS** unit, **GetIntVec** and **SetIntVec:**

```
PROCEDURE GetIntVec
          (IntNumber : Byte;
           VAR Vector : Pointer);

PROCEDURE SetINtVec
          (IntNumber : Byte;
           Vector : Pointer);
```

In both cases, **IntNumber** contains the number of the interrupt whose vector you wish to read or change, and **Vector** is a generic pointer containing the address read from or written to that vector. **GetIntVec** returns vector **IntNumber** in **Vector**; and **SetIntVec** places the address in pointer **Vector** in the vector table for interrupt **IntNumber**. These are "well-behaved" routines for reading and setting vectors from the interrupt vector table. We say "well-behaved" because there are "ill-behaved" ways to alter the vector table—by use of the **MEM**, **MEMW**, and **MEML** statements.

Why are **MEM**, **MEMW**, and **MEML** ill-behaved?

Well, think about this: Suppose you're in the midst of altering a vector in the table and you have half of a new value written; then something somewhere in the system calls that interrupt. At the moment when the CPU recognizes the interrupt, you may already have a new segment in place, but you may not yet have overwritten the old offset. The CPU sends execution charging off on this half-baked vector (which points into the middle of a data buffer), starts executing data as code, and freezes the machine solid.

DOS has a pair of functions for reading and setting interrupt vectors correctly that first *disable* interrupts before reading or altering a vector. Only when the vector is completely read, or completely changed, will DOS re-enable interrupts. That way, your programs will not end up reading a half-correct vector, or (much worse) allowing the CPU to transfer control to a half-correct vector.

PEEKING AT ISRs

The **Vectors** utility knows another trick—it can provide a look at what any vector is pointing to. Any initialized interrupt vector points to an interrupt service routine (ISR) of some sort. On command, **Vectors** displays a hex dump of the first 256 bytes of memory pointed to by any given interrupt vector. Those wild-eyed folks who read 8086 binary machine code in their heads can track the logic of simple service routines. The rest of us can look for interrupt service routine "signatures," typically in the form of copyright notices embedded in the binary machine code. For example, if you have the Logitech Mouse driver loaded, **Vectors** shows you the Logitech signature at an offset of 16 bytes into the driver, pointed to by interrupt 51 (33H).

Vectors tests every vector that it displays, and indicates whether the vector points to a byte containing

continued on page 51

```
FUNCTION ForceCase(Up : BOOLEAN; Target : String) : String;

CONST
  Uppercase : SET OF Char = ['A'..'Z'];
  Lowercase : SET OF Char = ['a'..'z'];

VAR
  I : INTEGER;

BEGIN
  IF Up THEN FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Lowercase THEN
      Target[I] := UpCase(Target[I])
    ELSE { NULL }
  ELSE FOR I := 1 TO Length(Target) DO
    IF Target[I] IN Uppercase THEN
      Target[I] := Chr(Ord(Target[I])+32);
  ForceCase := Target
END;


Procedure ValHex(HexString : String;
                 VAR Value : LongInt;
                 VAR ErrCode : Integer);

VAR
  HexDigits  : String;
  Position   : Integer;
  PlaceValue : LongInt;
  TempValue  : LongInt;
  I          : Integer;

BEGIN
  ErrCode := 0; TempValue := 0; PlaceValue := 1;
  HexDigits := '0123456789ABCDEF';
  StripWhite(HexString);   { Get rid of leading whitespace }
  IF Pos('$',HexString) = 1 THEN Delete(Hexstring,1,1);
  HexString := ForceCase(Up,HexString);
  IF (Length(HexString) > 8) THEN ErrCode := 9
    ELSE IF (Length(HexString) < 1) THEN ErrCode := 1
  ELSE
    BEGIN
      FOR I := Length(HexString) DOWNTO 1 DO  { For each character }
        BEGIN
          { The position of the character in the string is its value:}
          Position := Pos(Copy(HexString,I,1),HexDigits) ;
          IF Position = 0 THEN   { If we find an invalid character...}
            BEGIN
              ErrCode := I;      { ...set the error code... }
              Exit               { ...and exit the procedure }
            END;
          { The next line calculates the value of the given digit }
          { and adds it to the cumulative value of the string: }
          TempValue := TempValue + ((Position-1) * PlaceValue);
          PlaceValue := PlaceValue * 16;  { Move to next place }
        END;
      Value := TempValue
    END
END;


PROCEDURE DumpBlock(XBlock : Block);

VAR
  I,J,K : Integer;
  Ch    : Char;

BEGIN
```

```
  FOR I:=0 TO 15 DO        { Do a hexdump of 16 lines of 16 chars }
    BEGIN
      FOR J:=0 TO 15 DO    { Show hex values }
        BEGIN
          WriteHex(Ord(XBlock[(I*16)+J]));
          Write(' ')
        END;
      Write('   |');          { Bar to separate hex & ASCII }
      FOR J:=0 TO 15 DO       { Show printable chars or '.' }
        BEGIN
          Ch:=Chr(XBlock[(I*16)+J]);
          IF ((Ord(Ch)<127) AND (Ord(Ch)>31))
          THEN Write(Ch) ELSE Write('.')
        END;
      Writeln('|')
    END;
  FOR I:=0 TO 1 DO Writeln('')
END;  { DumpBlock }


PROCEDURE ShowHelp;

BEGIN
  Writeln;
  Writeln('Press RETURN to advance to the next vector.');
  Writeln;
  Writeln
  ('To display a specific vector, enter the vector number (0-255)');
  Writeln
  ('in decimal or preceded by a "$" for hex, followed by RETURN.');
  Writeln;
  Writeln('Valid commands are:');
  Writeln;
  Writeln
  ('D : Dump the first 256 bytes pointed to by the current vector');
  Writeln
  ('E : Enter a new value (decimal or hex) for the current vector');
  Writeln('H : Display this help message');
  Writeln('Q : Exit VECTORS ');
  Writeln('X : Exit VECTORS ');
  Writeln('Z : Zero segment and offset of the current vector');
  Writeln('? : Display this help message');
  Writeln;
  Write('The indicator ">>IRET" means the vector');
  Writeln(' points to an IRET instruction');
  Writeln;
END;


PROCEDURE DisplayVector(VectorNumber : Integer);

VAR
  Bump : Integer;
  Chunks : PtrPieces;
  Vector : Pointer;
  Tester : ^Byte;

BEGIN
  GetIntVec(VectorNumber,Vector);{ Get the vector }
  Tester := Vector;              { Can't dereference untyped pointer }
  Chunks := PtrPieces(Vector);   { Cast Vector onto Chunks }
  Write(VectorNumber : 3,'  $');
  WriteHex(VectorNumber);
  Write('  [');
  WriteHex(Chunks[3]);        { Write out the chunks as hex digits }
  WriteHex(Chunks[2]);
  Write(':');
  WriteHex(Chunks[1]);
  WriteHex(Chunks[0]);
  Write(']');
  IF Tester^ = $CF           { If vector points to an IRET, say so }
    THEN Write(' >>IRET ')
    ELSE Write('        ');
END;


PROCEDURE DumpTargetData(VectorNumber : Integer);

VAR
  Vector : Pointer;
  Tester : ^Block;

BEGIN
  GetIntVec(VectorNumber,Vector);  { Get the vector }
  Tester := Vector;     { Cast the vector onto a pointer to a block }
  MemBlock := Tester^;      { Copy the target block into MemBlock }
  IF MemBlock[0] = $CF THEN { See if the first byte is an IRET }
    Writeln('Vector points to an IRET.');
  DumpBlock(MemBlock)       { and finally, hexdump the block. }
END;


PROCEDURE ChangeVector(VectorNumber: Integer);

VAR
  Vector : Pointer;
  LongTemp,TempValue : LongInt;
  SegPart,OfsPart : Word;

BEGIN
  GetIntVec(VectorNumber,Vector); { Get current value of vector }
  LongTemp := LongInt(Vector);    { Cast Pointer onto LongInt }
  SegPart := LongTemp SHR 16;     { Separate pointer seg. from off. }
  OfsPart := LongTemp AND $0000FFFF;  { And keep until changed }
  Write('Enter segment ');
  Write('(RETURN retains current value): ');
  Readln(Command);
  StripWhite(Command);
  { If something other than RETURN was entered: }
  IF Length(Command) > 0 THEN
    BEGIN
      Val(Command,TempValue,ErrorPosition);  { Evaluate as decimal }
      IF ErrorPosition = 0 THEN SegPart := TempValue
        ELSE { If it's not a valid decimal value, evaluate as hex: }
          BEGIN
            ValHex(Command,TempValue,ErrorPosition);
            IF ErrorPosition = 0 THEN SegPart := TempValue
          END;
      { Reset the vector with any changes: }
      Vector := Ptr(SegPart,OfsPart);
      SetIntVec(VectorNumber,Vector);
    END;
  DisplayVector(VectorNumber); { Show it to reflect any changes }
  Writeln;
  Write('Enter offset  ');     { Now get an offset }
  Write('(RETURN retains current value): ');
  Readln(Command);
  StripWhite(Command);
  { If something other than RETURN was entered: }
  IF Length(Command) > 0 THEN
    BEGIN
      Val(Command,TempValue,ErrorPosition);  { Evaluate as decimal }
      IF ErrorPosition = 0 THEN OfsPart := TempValue
        ELSE { If it's not a valid decimal value, evaluate as hex: }
          BEGIN
            ValHex(Command,TempValue,ErrorPosition);
            IF ErrorPosition = 0 THEN OfsPart := TempValue
          END
    END;
  { Finally, reset vector with any changes: }
```

INTERRUPT VECTORS

continued from page 49

CFH. This is the machine-code equivalent of the IRET (Interrupt Return) mnemonic. Pointing an interrupt to an IRET instruction is a safety measure that prevents havoc in case an interrupt occurs for which the vector is uninitialized. If an unused vector is made to point to an IRET, the worst that can happen if that interrupt is triggered is nothing at all—the IRET sends execution back to the caller without taking any action.

In the best of all worlds, all unused interrupt vectors are initialized to point to an IRET. But as you'll see once you run **Vectors**, only a few vectors are so disarmed. Most vectors point to segment zero, offset zero (which is in fact an interrupt vector itself—the vector for interrupt 0, the first entry in the vector table). If such an interrupt occurs, the CPU attempts to execute the interrupt jump table as though it were code—which will almost certainly crash the machine hard.

Vectors is simple in operation. It cycles through the 256 interrupt vectors one at a time, displaying the current value of the current interrupt vector, and then pauses for a command. "Jumping" to another interrupt vector is done by entering that vector's value as either a decimal number or a hexadecimal value preceded by a "$."

The E command is used to change a vector value. E prompts individually for the segment and offset portion of the vector. If you don't wish to change one or both of these, simply press Enter and nothing is altered. As with jumping to a new value, vector values can be entered in either decimal or hex.

The command D dumps the 256 bytes pointed to by the current vector. If the first byte of the block is an IRET instruction, **Vectors** displays the string ">>IRET."

The command Z changes both the offset and segment portion of a vector to zero. This is useful in cases where you're testing software that modifies interrupt vectors—and you may be modifying the wrong vectors. Zeroing a vector allows you to come back after your test software has run and to tell at a glance if the zeroed vector or vectors have stayed zeroed.

Either of the commands Q or X exits **Vectors** to DOS.

The hex format display procedure **WriteHex** figures prominently in **Vectors** as the mechanism by which the interrupt vectors are displayed, and also as the core of a hexdump routine, **DumpBlock**, that dumps 256 bytes of memory at the location pointed to by the current vector. ■

Listings may be downloaded from CompuServe as VECTOR.ARC.

```
  Vector := Ptr(SegPart,OfsPart);
  SetIntVec(VectorNumber,Vector);
END;


BEGIN
  Quit := False;
  VectorNumber := 0;
  Writeln('>>VECTORS<<');
  Writeln('By Jeff Duntemann');
  Writeln('From the book: COMPLETE TURBO PASCAL, 3E');
  Writeln('ISBN 0-673-38355-5');
  ShowHelp;

  REPEAT
    DisplayVector(VectorNumber);   { Show the vector # & address }
    Readln(Command);               { Get a command from the user }
    IF Length(Command) > 0 THEN    { If something was typed:     }
      BEGIN
        { See if a number was typed; if one was, it becomes the  }
        { current vector number.  If an error in converting the  }
        { string to a number occurs, Vectors then parses the     }
        { string as a command.   }
        Val(Command,NewVector,ErrorPosition);
        IF ErrorPosition = 0 THEN VectorNumber := NewVector
          ELSE
            BEGIN
              StripWhite(Command);       { Remove leading whitespace }
              Command := ForceCase(Up,Command); { Force to upper case}
              CommandChar := Command[1]; { Isolate first character   }
              CASE CommandChar OF
                'Q','X' : Quit := True;  { Exit VECTORS }
                'D'     : DumpTargetData(VectorNumber); { Dump data  }
                'E'     : ChangeVector(VectorNumber); { Enter vector }
                'H'     : ShowHelp;
                'Z'     : BEGIN            { Zero the vector }
                            Vector := NIL;   { NIL is 32 zero bits }
                            SetIntVec(VectorNumber,Vector);
                            DisplayVector(VectorNumber);
                            Writeln('zeroed.');
                            VectorNumber := (VectorNumber + 1) MOD 256
                          END;
                '?'     : ShowHelp;
              END {CASE}
            END
      END
    { The following line increments the vector number, rolling over }
    { to 0 if the number would have exceeded 255: }
    ELSE VectorNumber := (VectorNumber + 1) MOD 256
  UNTIL Quit;
END.
```

MOUSE MYSTERIES, PART 1: TEXT

Unraveling the mysteries of mouse programming is easy with Turbo C and Turbo Pascal.

Kent Porter

PROGRAMMER

Mice have moved into the PC mainstream. Because of the convenience they offer the user, mice have become synonymous with friendly software and intuitive interaction. From the user's perspective, a mouse can seem like a simple-to-use, yet mysterious device that is probably horrendously complicated to program. From the programmer's viewpoint, however, writing a mouse program is not particularly difficult. In this first part of "Mouse Mysteries," we'll use Turbo C and Turbo Pascal to explore the text mode region of the software world of the mouse. The next article of this two-part series will examine the techniques of mouse programming in graphics mode.

There are relatively few differences between mouse functions in Turbo C and Turbo Pascal. In the program examples, I've purposely called corresponding mouse functions in each language by the same names. Also, I've used the same variables and algorithms in the demonstration programs wherever possible.

WHAT YOU WILL NEED

To incorporate a mouse into the user interface of your program, you need three things:

- The hardware mouse itself
- The mouse device driver
- Software mechanisms to communicate with the mouse

The first two items come from the mouse vendor. The last item consists of the Turbo C and Turbo Pascal programs in Listings 1 and 2, along with the techniques discussed in this article.

Most mouse vendors adhere to the de facto Microsoft standard that governs a two-button mouse, or else they furnish a Microsoft-based superset. Of the latter, the best known vendor is Logitech, who sells a three-button mouse that is compatible with the Microsoft standard (except for operations involving the third button).

This series will explore programming the Microsoft and Logitech device drivers (and by extension, also the great majority of mice by other vendors that emulate these mice) with Turbo C. Throughout these two articles, we'll also point out the differences in mouse programming between Turbo C and Turbo Pascal.

COMMUNICATING WITH THE MOUSE

The only practical channel of communication between the mouse and your software program is through the mouse device driver. The driver is accessible through software interrupt 33H (51 decimal), which is not used by DOS. This interrupt is claimed by the mouse device driver during load-time initialization and thereafter belongs to the driver. Parameters are passed to the driver and back to the caller through the 8086 registers.

To call the mouse, you must place a function code into register AX and execute interrupt 33H. All the mouse calls use full-word registers, and some calls require additional parameters in the registers BX through DX. A few mouse calls also expect segment addresses in the ES register. Mouse inquiry functions return values in the registers BX through DX, wrapping back to AX if returning four values. Your software can then extract these values as integers, assigning and acting on them as appropriate.

THE MOUSE FUNCTIONS

Microsoft furnishes 16 mouse functions, numbered 0 through 15. Logitech's mouse functions are the same, although a few differ slightly to accommodate the third mouse button; there are also two *superset* functions, numbered 16 and 19. The functions from both manufacturers are summarized in Table 1.

continued on page 54

Bradley Ream Rodent courtesy of Round-Up Pet Center, Scotts Valley, CA

Microsoft
LOGITECH Series 7

FUNCTION	PURPOSE	NATURE
0	Initialize mouse	Control
1	Show mouse	Control
2	Hide mouse cursor	Control
3	Get position and button status	Inquiry
4	Set mouse cursor position	Control
5	Get button press information	Inquiry
6	Get button release information	Inquiry
7	Set minimum/maximum columns (x)	Control
8	Set minimum/maximum rows (y)	Control
9	Define graphics pointer shape (1)	Control
10	Define text pointer shape	Control
11	Read motion counters	Inquiry
12	Define mouse event handler	Control
13	Turn light pen emulation on	Control
14	Turn light pen emulation off	Control
15	Set motion-to-pixel ratio (1)	Control
16	Conditional hide mouse cursor (2)	Control
19	Set speed threshold (2)	Control

NOTES:
(1) Covered in Part II.
(2) Logitech only. Not covered here.

Table 1. The 16 Microsoft Mouse function calls, plus those provided by the Logitech mouse.

MOUSE MYSTERIES

continued from page 52

Two sample programs illustrate the most important calling/returning conventions for each of the functions listed in Table 1. MOUSE.INC (Listing 1) is the Turbo C include file that implements the mouse function calls. MOUSE.PAS (Listing 2) is the complementary unit in Turbo Pascal 4.0.

When looking through these listings, keep in mind that the primary differences between the Turbo C and Turbo Pascal libraries relate to the mouse inquiry functions, which are **mReset**, **mPos**, **mPressed**, **mReleased**, and **mMotion**. In Turbo Pascal, structures to be initialized are owned by the calling program and passed as variable **(VAR)** parameters. The Turbo Pascal mouse functions alter the owner's copy of the structure; none of these functions returns anything.

On the other hand, Turbo C mouse functions own the structures as statics, and return pointers to those structures. This follows the spirit of the C language, which is much more pointer-oriented than Pascal. Also, Pascal has no static storage class like that in C.

Therefore, when writing Turbo C programs that incorporate the mouse, be sure to declare pointer variables for the structures that you intend to use, and to initialize those variables with the inquiry functions' returned values.

Function 0: Initializing the mouse (mReset). Any program that uses the mouse must initialize it during the setup phase by calling function 0. Failure to do so means that your program inherits a garbage mouse status from either powerup or the previous program, whichever is most recent.

The reset function clears the previous mouse status, places the mouse cursor in the center of the screen (though the cursor is invisible—see the discussion of functions 1 and 2 below), and sets the scope of operation to the full display. Upon returning, AX contains the mouse status (0 if the mouse device driver is not installed, nonzero if it is installed), and BX contains the number of mouse buttons (2 for Microsoft and clones, 3 for Logitech).

In Turbo Pascal, it's convenient to stuff the returned values into a record, which is why the unit in Listing 2 defines the **resetRec** type. Because this definition is in the interface part of the mouse unit, a program that **USES** this unit can declare **resetRec** variables as though the type were intrinsic.

Note that the **mReset** procedure receives the **resetRec** variable as a **VAR** parameter. In other words, it jointly owns the record with the caller, thus enabling the procedure to pass back the values returned by the mouse device driver. The Pascal equivalents to inquiry functions 3, 5, 6, and 11 operate similarly on records passed as **VAR** parameters in to return results. Let's declare the **resetRec** variable as follows:

```
Var
  theMouse : resetRec;
```

We then initialize the mouse with this call:

```
mReset (theMouse);
```

Afterwards, our program can check to see if a mouse is present by using a test such as:

```
if theMouse.exists then
  {do mouse stuff}
else
  {mouse isn't active}
```

Things are a little different in Turbo C, where we declare pointers to the **resetRec** structure. In Pascal, the call to initialize the mouse is:

```
mReset (theMouse);
```

However, in Turbo C, the mouse initialization call is:

```
theMouse = mReset ();
```

Both calls do the same thing, but the calling sequence is different because of the use of pointers in C.

It's advisable to call **mReset** again at the end of the program. This step restores the device driver to its default state and deactivates the mouse, so that subsequent programs don't inherit an unwanted mouse status.

Function 1: Show mouse cursor (mShow). mReset leaves the mouse cursor off. Therefore, the very next step is usually to turn the mouse cursor on. The only way to do so is via function 1, **mShow**. This control function doesn't return a value, nor does its counterpart, **mReset**.

Function 2: Hide mouse cursor (mHide). This function turns off the mouse cursor without otherwise changing its status. Even though the cursor is "hidden" after a call to function 2 (via **mHide**), it still moves in response to physical travel of the mouse; of course, you're not aware of the cursor's movement until you call **mShow** again and discover it in a different place.

A strange tension exists among functions 0, 1, and 2—they all

control an internal value called the cursor flag, which is a signed integer. **mReset** (function 0) sets the flag to **-1**, making the cursor invisible. **mShow** (function 1) increments the flag to **0**, which tells the device driver that the cursor should be visible. **mHide** (function 2) decrements the flag back to **-1**. The net result is that if you reset the mouse and then call **mHide**, it takes two calls to **mShow** to make the cursor visible, since the internal flag has dropped to **-2**.

Oddly, the mouse device driver doesn't furnish an inquiry to determine the value of the internal cursor flag, so the program cannot ask if the cursor is on or off. Therefore, the programmer is responsible for tracking the mouse cursor's status.

Function 3: Get position and button status (mPos). This inquiry function returns information about the status of the mouse; specifically, it tells the location of the cursor and whether any button is pressed down. All information is reported in *real time*—in other words, if the mouse is in motion, you get a current position report (which is probably not the mouse's destination). Likewise, making infrequent calls to this function can potentially cause a button click to be missed. To capture a click as well as the mouse's position at the ultimate destination, use function 12 (see functions 5 and 6 for more button information). Function 3 is chiefly useful in graphics—within a tight loop, this function can write a pixel to draw a track based on mouse movement, transforming the mouse into a pencil of sorts. Another common use for function 3 is to determine the location of the cursor. Determining the cursor's position can come in handy if you intend to jump it to another place and you need an address to which you'll return the cursor later.

Function 3 returns the button status in BX, the column in CX, and the row in DX. With a two-button mouse, bits 0 and 1 in BX are set if the left and right buttons, respectively, are down. Bit 2 represents the center button in

continued on page 56

LISTING 1: MOUSE.INC

```
/* MOUSE.INC: Turbo C Source code for mouse interface functions.   */
/*    Must #include dos.h before this #include to define the       */
/*    register set used to pass args to device driver              */
/* ------------------------------------------------------------- */
/* Define int for mouse device driver */
#define  callMDD int86(0x33, &inreg, &outreg)

/* Define sorting macros used locally */
#define  lower (x, y)  (x < y) ? x : y
#define  upper (x, y)  (x > y) ? x : y

/* STATIC REGISTERS USED THROUGHOUT */
union REGS  inreg, outreg;

/* STRUCTURES USED BY THESE FUNCTIONS */
typedef struct {
  int exists,                          /* TRUE if mouse is present */
      nButtons;                         /* # of buttons on mouse */
} resetRec;                              /* returned by mReset */

typedef struct {
  int buttonStatus,       /* bits 0-2 on if corresp button is down */
      opCount,                 /* # times button has been clicked */
      column, row;                                    /* position */
} locRec;                 /* returned by mPos, mPressed, mReleased */

typedef struct {
  int hCount,                          /* net horizontal movement */
  vCount;                               /* net vertical movement */
} moveRec;                              /* returned by mMotion */
/* ------------------------------------------------------------- */
/* Following are implementations of the Microsoft mouse functions  */

resetRec *mReset ()
/* Resets mouse to default state. Returns pointer to a structure   */
/* indicating whether or not mouse is installed and, if so, how    */
/* many buttons it has.                                            */
/* Always call this function during program initialization.        */
{
static resetRec  m;

  inreg.x.ax = 0;                                       /* function 0 */
  callMDD;
  m.exists = outreg.x.ax;
  m.nButtons = outreg.x.bx;
  return ( &m );
} /* ----------------------- */
void mShow (void)
/* Makes the mouse cursor visible. Don't call if cursor is already */
/* visible, and alternate with calls to mHide.                     */
{
  inreg.x.ax = 1;                                       /* function 1 */
  callMDD;
} /* ----------------------- */
void mHide (void)
/* Makes mouse cursor invisible. Movement and button activity are  */
/* still tracked. Do not call if cursor is already hidden, and     */
/* alternate with calls to mShow                                   */
{
  inreg.x.ax = 2;                                       /* function 2 */
  callMDD;
} /* ----------------------- */
locRec *mPos (void)
/* Gets mouse cursor position and button status, returns pointer   */
/* to structure containing this info                               */
{
static locRec  m;

  inreg.x.ax = 3;                                       /* function 3 */
  callMDD;
```

```
  m.buttonStatus = outreg.x.bx;                    /* button status */
  m.column = outreg.x.cx;                          /* horiz position */
  m.row = outreg.x.dx;                             /* vert position */
  return (&m);
} /* ------------------------ */
void  mMoveto (int newCol, int newRow)
/* Move mouse cursor to new position */
{
  inreg.x.ax = 4;                                  /* function 4 */
  inreg.x.cx = newCol;
  inreg.x.dx = newRow;
  callMDD;
} /* ------------------------ */
locRec  *mPressed (int button)
/* Gets pressed info about named button: current status (up/down), */
/* times pressed since last call, position at most recent press.   */
/* Resets count and position info. Button 0 is left, 1 is right on */
/* Microsoft mouse.                                                */
/* Returns pointer to locRec structure containing info.            */
{
static locRec  m;

  inreg.x.ax = 5;                                  /* function 5 */
  inreg.x.bx = button;              /* request for specific button */
  callMDD;
  m.buttonStatus = outreg.x.ax;
  m.opCount = outreg.x.bx;
  m.column  = outreg.x.cx;
  m.row     = outreg.x.dx;
  return (&m);
} /* ------------------------ */
locRec  *mReleased (int button)
/* Same as mPressed, except gets released info about button */
{
static locRec  m;

  inreg.x.ax = 6;                                  /* function 6 */
  inreg.x.bx = button;              /* request for specific button */
  callMDD;
  m.buttonStatus = outreg.x.ax;
  m.opCount = outreg.x.bx;
  m.column  = outreg.x.cx;
  m.row     = outreg.x.dx;
  return (&m);
} /* ------------------------ */
void  mColRange (int hmin, int hmax)
/* Sets min and max horizontal range for mouse cursor. Moves   */
/* cursor inside range if outside when called. Swaps values if */
/* hmin and hmax are reversed.                                 */
{
  inreg.x.ax = 7;                                  /* function 7 */
  inreg.x.cx = hmin;
  inreg.x.dx = hmax;
  callMDD;
} /* ------------------------ */
void  mRowRange (int vmin, int vmax)
/* Same as mHminmax, except sets vertical boundaries. */
{
  inreg.x.ax = 8;                                  /* function 8 */
  inreg.x.cx = vmin;
  inreg.x.dx = vmax;
  callMDD;
} /* ------------------------ */
void  mGraphCursor (int hHot, int vHot, unsigned maskSeg,
                    unsigned maskOfs)
/* Sets graphic cursor shape */
{
struct SREGS  seg;

  inreg.x.ax = 9;                                  /* function 9 */
```

MOUSE MYSTERIES

continued from page 55

the Logitech mouse and other three-button mice. A zero in any of these bit positions means the corresponding button is not currently being pressed.

Function 4: Set mouse cursor position (mMoveto). This function moves the cursor directly to the specified column and row. The mouse device driver is sensitive to the display adapter in use, but handles positioning coordinates differently than you might expect. Even in text mode, the device driver regards the screen as a 640 × 200-pixel array (which is the same as the CGA high-resolution graphics mode). Under this scheme, a character cell is 8 × 8 pixels for the normal 80 × 25 text display.

The mouse cursor position in text mode is always the pixel address of the upper left corner of the current character cell. Thus, home position is 0,0; the position of the next cell to the right is 8,0; the cell directly below that is located at 8,8; and so forth. To calculate the mouse cursor position, multiply the text column and row coordinates by eight. For instance, the approximate center of the text screen is the pixel address of 39,12; this maps to 312,96 in mouse cursor coordinates. To translate in the reverse direction, divide the mouse cursor coordinates by eight.

In graphics mode, the cursor moves smoothly, pixel by pixel. In text mode, however, it jumps from cell to cell in order to avoid occupying two or more cells at the same time. If you pass coordinates to **mMoveto** that are not multiples of eight (e.g., 313,97), the device driver ignores the remainders and positions the cursor at the next lower integral text cell (312,96).

Function 5: Button press information (mPressed). The mouse device driver records how many times each button is pressed. **mPressed** calls function 5 to get this information for any specific button. The button numbers are: 0 = Left; 1 = Right; for three-button mice only, 2 = Middle.

A request for press information

about a specific button clears its *history*. Thus, a call might reveal that button 1 has been pressed twice. The next time you ask **mPressed** about button 1, it returns zero if the button was not pressed since that earlier call.

This function also returns the button status, which is real-time information about all buttons (exactly as in **mPos**). Additionally, function 5 provides the column and row of the cursor's position during the last time the button of interest was pressed. If the button is currently down (the status bit is ON), the position report is the current location; otherwise, the position report is historical information.

mPressed is of limited usefulness—if it's called from within a loop, the loop might iterate many times while the user has the button pressed down. Each time, **mPressed** reports the same button operation as though it were a new event. **mReleased** (described below) is a better choice, since it waits to update the internal counter until the user has released the button.

Function 6: Get button released information (mReleased). This function is similar to **mPressed**, except that it reports how many times the button of interest has been released after being pressed. It's a more reliable indicator of a given button's activity than is **mPressed**, simply because a release is a singular event that isn't tricked by a button's current status.

Functions 7 and 8: Set Min/Max columns and rows (mColRange and mRowRange). These two functions limit the operational area of the mouse cursor, much like fencing the backyard to confine the dog. Function 7 (**mColRange**) governs the range of columns, and function 8 (**mRowRange**) controls the rows in which the cursor can appear. When these two functions are invoked, the cursor—if previously located outside the defined area—moves inside of it. Thereafter, the cursor simply refuses to cooperate if you try moving it beyond any boundary.

The default condition gives the

continued on page 58

```
  inreg.x.bx = hHot;                /* cursor hot spot: horizontal */
  inreg.x.cx = vHot;                  /* cursor hot spot: vertical */
  inreg.x.dx = maskOfs;
  seg.es = maskSeg;
  int86x (0x33, &inreg, &outreg, &seg);
} /* ----------------------- */
void  mTextCursor (int curstype, unsigned arg1, unsigned arg2)
/* Sets text cursor type, where 0 = software and 1 = hardware)     */
/* For software cursor, arg1 and arg2 are the screen and cursor    */
/*   masks.                                                        */
/* For hardware cursor, arg1 and arg2 specify scan line start/stop */
/*   i.e. cursor shape.                                            */
{
  inreg.x.ax = 10;                                    /* function 10 */
  inreg.x.bx = curstype;
  inreg.x.cx = arg1;
  inreg.x.dx = arg2;
  callMDD;
} /* ----------------------- */
moveRec  *mMotion (void)
/* Reports net motion of cursor since last call to this function   */
{
static moveRec  m;

  inreg.x.ax = 11;                                    /* function 11 */
  callMDD;
  m.hCount = _CX;                                  /* net horizontal */
  m.vCount = _DX;                                    /* net vertical */
  return (&m);
} /* ----------------------- */
void mInstTask (unsigned mask, unsigned taskSeg, unsigned taskOfs)
/* Installs a user-defined task to be executed upon one or more    */
/*   mouse events specified by mask.                               */
{
struct SREGS  seg;

  inreg.x.ax = 12;                                    /* function 12 */
  inreg.x.cx = mask;
  inreg.x.dx = taskOfs;
  seg.es = taskSeg;
  int86x (0x33, &inreg, &outreg, &seg);
} /* ----------------------- */
void mLpenOn (void)
/* Turns on light pen emulation. This is the default condition.    */
{
  inreg.x.ax = 13;                                    /* function 13 */
  callMDD;
} /* ----------------------- */
void mLpenOff (void)
/* Turns off light pen emulation. */
{
  inreg.x.ax = 14;                                    /* function 14 */
  callMDD;
} /* ----------------------- */
void mRatio (int horiz, int vert)
/* Sets mickey-to-pixel ratio, where ratio is R/8. Default is 16   */
/*   for vertical, 8 for horizontal */
{
  inreg.x.ax = 15;                                    /* function 15 */
  inreg.x.cx = horiz;
  inreg.x.dx = vert;
  callMDD;
} /* ----------------------- */
```

LISTING 2: MOUSE.PAS

```
Unit mouse;

{-------------------------------------------------------}
{ For Turbo Pascal Release 4.n. Won't work with older levels }
{ Implements calls to mouse device driver. Works with the    }
{ Logitech and Microsoft mice, and anything else compatible. }
{                                                            }
{                          NOTE!!!                           }
{      COMPILER OPTIONS MUST BE SET TO FORCE FAR CALLS!      }
{      If not, user defined task installed by mInstTask      }
{                  will crash the system.                    }
{-------------------------------------------------------}

Interface

{$U \tp}
Uses DOS;                       { For interrupts and registers }

Const
  MDD = $33;                { Interrupt for mouse device driver }

Type
  resetRec = record          { Initialized by mouse function 0 }
    exists   : Boolean;             { TRUE if mouse is present }
    nButtons : integer;                   { # buttons on mouse }
  End;

  locRec = record     { Initialized by mouse fcns 3, 5, and 6 }
    buttonStatus,     { bits 0-2 on if corresp button is down }
    opCount,                { # times button has been clicked }
                              { opCount not returned by fcn 3 }
    column,                                          { position }
    row      : integer;
  End;

  moveRec = record               { Initialized by mouse fcn 11 }
    hCount,                          { net horizontal movement }
    vCount : integer;                  { net vertical movement }
  End;

Var  Reg : Registers;

{ These are the Microsoft mouse functions }
Procedure mReset (var mouse : resetRec);        { function  0 }
Procedure mShow;                                { function  1 }
Procedure mHide;                                { function  2 }
Procedure mPos (var mouse : locRec);            { function  3 }
Procedure mMoveto (col, row : integer);         { function  4 }
Procedure mPressed (button : integer;
                    var mouse : locRec);        { function  5 }
Procedure mReleased (button : integer;
                    var mouse : locRec);        { function  6 }
Procedure mColRange (min, max : integer);       { function  7 }
Procedure mRowRange (min, max : integer);       { function  8 }
Procedure mGraphCursor (hHot, vHot : integer;
                   maskSeg, maskOfs : word);    { function  9 }
Procedure mTextCursor (ctype, p1, p2 : word);   { function 10 }
Procedure mMotion (var moved : moveRec);        { function 11 }
Procedure mInstTask (mask,
                     taskSeg, taskOfs : word);  { function 12 }
Procedure mLpenOn;                              { function 13 }
Procedure mLpenOff;                             { function 14 }
Procedure mRatio (horiz, vert : integer);       { function 15 }
{ ------------------------------------------------------- }
```

MOUSE MYSTERIES

continued from page 57

cursor access to the entire display area; these procedures override the default. A typical application is to limit the cursor to a pop-up dialog box. To do so, follow these steps:

1. Get the current cursor position with **mPos**.
2. Create the pop-up.
3. Set the boundaries with **mColRange** and **mRowRange**.
4. Perform the operations necessary to make the pop-up go away.
5. Restore full-screen mouse operation with another set of calls to these procedures, this time passing the absolute boundaries of the display as parameters.
6. Use **mMoveto** with the cursor position saved in step 1 to put the cursor back where it was during step 1. Note that both procedures sort the parameters to make sure that they're in correct order.

Function 10: Set text cursor (mTextCursor). You have your choice of two text cursors when using a mouse; function 10 lets you select a cursor and specify how it will appear. The hardware cursor selection (option 1) puts the video adapter's text cursor under control of the mouse, providing a single cursor on the display (although the text I/O position does not move with the mouse; see the discussion of MOUSDEMO later in this article). The software cursor (**curstype = 0** in Listing 1, or **ctype = 0** in Listing 2) lets you have two cursors on the display at the same time. One cursor operates normally in response to I/O, while the other cursor is controlled by the mouse. To reduce user confusion, the software mouse cursor can have a unique appearance provided by any of the 256 text characters. My favorite is character 18H, which is an upward-pointing arrow.

The software cursor is the default. Thus, if you call **mReset** and then **mshow** without an intervening call to function 10, you get a mouse cursor independent of the normal hardware cursor. Furthermore, this default software

cursor is a simple full-cell block that inverts the attributes of any character cell it occupies. For example, if you have gray letters on a black background, the cell occupied by the software mouse cursor contains a black letter on a gray background. Unlike the hardware cursor, the software cursor does not blink.

To change the software cursor's attributes, pass two masks in the CX and DX registers (in Listing 1, use arguments **arg1** and **arg2**, respectively; in Listing 2, use parameters **p1** and **p2**, respectively). **arg1** is the screen mask; it should have the value 77FFH if the cursor is a see-through rectangle, and 0000H if the cursor uses one of the 256 characters as its shape. **arg2** defines the cursor itself. Place a foreground/background attribute byte in the upper eight bits, and place the ASCII value of the cursor shape (character) in the lower eight; use ASCII character 0 if the cursor is to be a simple rectangle. For example, the patterns for a see-through block cursor are 77FFH and 7700H; the patterns for a gray up-arrow cursor against a black background are 0000H and 0718H (where 18H is the arrow symbol). Complete the parameter setup by passing 0 as the **curstype** argument.

A hardware cursor is simpler to define. Set up **arg1** and **arg2** with the starting and ending scan lines of the cursor. The top scan line is always 0. On a monochrome adapter, the bottom scan line is 12. For all graphics boards operating in text mode, the bottom scan line is 7 (just as it is for ROM BIOS interrupt 10H, function 1). To tell the mouse device driver to take control of the hardware cursor, pass the value **1** as the **curstype** parameter.

MICE AT WORK

The MOUSDEMO program (MOUSDEMO.C in Listing 3 and MOUSDEMO.PAS in Listing 4) has two phases. The first phase demonstrates the software cursor, which is an up-arrow that travels one cell at a time around the text display in response to mouse movement. When the software cursor moves into a cell already occupied by a character, the cur-

continued on page 60

```
IMPLEMENTATION

Function lower (n1, n2 : integer) : integer;  { Local to unit}
Begin
  If n1 < n2 then lower := n1
  Else lower := n2;
End;

Function upper (n1, n2 : integer) : integer; { Local to unit }
Begin
  If n1 > n2 then upper := n1
  Else upper := n2;
End;
{ ------------------------- }
Procedure mReset;         { Resets mouse to default condition }
Begin
  reg.ax := 0;
  intr (MDD, reg);
  if reg.ax <> 0 then
    mouse.exists := TRUE
  else
    mouse.exists := FALSE;
  mouse.nButtons := reg.bx;
End;
{ ------------------------- }
Procedure mShow;                  { Make mouse cursor visible }
Begin
  reg.ax := 1;
  intr (MDD, reg);
End;
{ ------------------------- }
Procedure mHide;                { Make mouse cursor invisible }
Begin
  reg.ax := 2;
  intr (MDD, reg);
End;
{ ------------------------- }
Procedure mPos;               { Get mouse status and position }
Begin
  reg.ax := 3;
  intr (MDD, reg);
  mouse.buttonStatus := reg.bx;
  mouse.column := reg.cx;
  mouse.row := reg.dx;
End;
{ ------------------------- }
Procedure mMoveto;      { Move mouse cursor to new location }
Begin
  reg.ax := 4;
  reg.cx := col;
  reg.dx := row;
  intr (MDD, reg);
End;
```

```
{ -------------------------- }
Procedure mPressed;  { Get pressed info about a given button }
Begin
  reg.ax := 5;
  reg.bx := button;
  intr (MDD, reg);
  mouse.buttonStatus := reg.ax;
  mouse.opCount := reg.bx;
  mouse.column := reg.cx;
  mouse.row := reg.dx;
End;
{ -------------------------- }
Procedure mReleased;      { Get released into about a button }
Begin
  reg.ax := 6;
  reg.bx := button;
  intr (MDD, reg);
  mouse.buttonStatus := reg.ax;
  mouse.opCount := reg.bx;
  mouse.column := reg.cx;
  mouse.row := reg.dx;
End;
{ -------------------------- }
Procedure mColRange;            { Set column range for mouse }
Begin
  reg.ax := 7;
  reg.cx := lower (min, max);
  reg.dx := upper (min, max);
  intr (MDD, reg);
End;
{ -------------------------- }
Procedure mRowRange;              { Set row range for mouse }
Begin
  reg.ax := 8;
  reg.cx := lower (min, max);
  reg.dx := upper (min, max);
  intr (MDD, reg);
End;
{ -------------------------- }
Procedure mGraphCursor;          { Set mouse graphics cursor }
Begin
  reg.ax := 9;
  reg.bx := hHot;
  reg.cx := vHot;
  reg.dx := maskOfs;
  reg.es := maskSeg;
  intr (MDD, reg);
End;
```


continued from page 59

sor temporarily replaces the text. When the cursor moves on, the text reappears.

The software cursor phase works like this: As you move the cursor around with the mouse, notice that a mouse position report appears on the display each time that you click the left button. There are two cursors on the screen: the normal flashing underscore (which is the hardware cursor) and the up-arrow cursor controlled by the mouse.

This condition continues under control of the first Turbo C **DO** loop (**REPEAT..UNTIL** in the Turbo Pascal program) until you click the right mouse button. At that point, the demo moves into the second phase to illustrate the hardware cursor.

The hardware cursor comes under control of the mouse as a result of the second call to **mTextCursor**, which sets the cursor shape to a small block the size of a character.

In this part of the demo, move the mouse cursor and click the right button to signal that you want to enter text, then begin typing after the program prints a question mark. End the input by pressing Enter, then move the cursor elsewhere and repeat. You can end the program by clicking the right button.

Even though the mouse takes over the hardware cursor's shape and position, it does not affect the location of normal I/O operations. Thus, regardless of the physical cursor's location on the display, an input or output proceeds from wherever the last I/O occurred. This is admittedly strange, but true.

Therefore, if you want the I/O location to correspond to the position of the mouse-controlled cursor, you have to track the cursor. In Turbo Pascal, use **GotoXY** as shown in the second **REPEAT..UNTIL** loop of MOUSDEMO.PAS. This process involves converting the mouse position into normal text coordinates through integer division by eight.

Note that MOUSDEMO.C has a couple of local functions—**clrScr**

and **gotoxy**—to emulate intrinsic Pascal procedures. Both of these functions call ROM BIOS interrupt 10H, which furnishes video services. The **clrScr** function merely resets the display to the current video mode, causing the adapter to clear the screen. The **gotoxy** function calls ROM BIOS function 2, which repositions the cursor to the specified row and column. While these functions have nothing to do with the mouse, they're useful additions to your Turbo C bag of tricks.

Similarly, the **mMoveto** after the **gets** in MOUSDEMO.C (**Readln** in MOUSDEMO.PAS) is necessary to relocate the mouse cursor back to the beginning of the output string. Otherwise, the cursor remains stationary at the end of the output string until you move the mouse again, at which point the cursor jumps back to the start of the string and commences motion from that position.

Note that this clean-up process resets the mouse cursor. This is in deference to subsequent programs, including DOS, which might otherwise be peculiarly affected if the mouse is still "alive."

The MOUSDEMO program is highly *mouse-intensive.* It expends a great deal of programmatic energy to get information about the mouse, even though a mouse is an accessory and not the principal input device for most software. For example, a menu-driven application might accept several alternative forms of input for selecting a menu choice:

- Cursor keys followed by Enter
- An alphabetic key identifying a choice
- Mouse movement and a click

The introduction of a mouse, with all of the attendant inquiry and control calls, greatly complicates this situation. Life would be much easier if we could simply check periodically to see if the user has done something with the mouse, and if so, act upon it. Fortunately, mouse function 12 lets us do precisely that.

continued on page 62

```
{ -------------------------- }
Procedure mTextCursor;              { Set mouse text cursor }

   { NOTES:                                                 }
   { Type 0 is the software cursor. When specified, p1 }
   {   and p2 are the screen and cursor masks.          }
   { Type 1 is the hardware cursor. When specified, p1 }
   {   and p2 are the start and stop scan lines, i.e.   }
   {   the cursor shape.                                    }

Begin
  reg.ax := 10;
  reg.bx := ctype;
  reg.cx := p1;
  reg.dx := p2;
  intr (MDD, reg);
End;
{ -------------------------- }
Procedure mMotion;   { Net movement of mouse since last call }
                            { Expressed in mickeys (1/100") }
Begin
  reg.ax := 11;
  intr (MDD, reg);
  moved.hCount := reg.cx;
  moved.vCount := reg.dx;
End;
{ -------------------------- }
Procedure mInstTask;             { Install user-defined task }
Begin
  reg.ax := 12;
  reg.cx := mask;
  reg.dx := taskOfs;
  reg.es := taskSeg;
  intr (MDD, reg);
End;
{ -------------------------- }
Procedure mLpenOn;   { Turn on light pen emulation (default) }
Begin
  reg.ax := 14;
  intr (MDD, reg);
End;
{ -------------------------- }
Procedure mLpenOff;           { Turn off light pen emulation }
Begin
  reg.ax := 15;
  intr (MDD, reg);
End;
{ -------------------------- }
Procedure mRatio;                { Set mickey to pixel ratio }

   { NOTES:                                          }
   { Ratios are R/8.                                 }
   { Default is 16 for vertical, 8 for horizontal }

Begin
  reg.ax := 15;
  reg.cx := horiz;
  reg.dx := vert;
End;
{ -------------------------- }

End.
```

LISTING 3: MOUSDEMO.C

```
/* MOUSDEMO.C:  Demo of basic mouse operations */

/* INCLUDES */
#include <stdio.h>
#include <dos.h>
#include <mouse.i>

/* CONSTANTS */
#define HARDWARE  1                                  /* cursor types */
#define SOFTWARE  0
#define LEFT      0                                 /* mouse buttons */
#define RITE      1
#define ROMBIOS   int86 (0x10, &inreg, &outreg)      /* BIOS calls */

/* LOCAL FUNCTIONS */
void  clrScr (void);
void  gotoxy (int col, int row);

/* ------------------------------------------------------------ */
main ()
{
resetRec   *theMouse;                        /* from reset function */
locRec     *its;                            /* from mouse inquiries */
int        col, row;
char       input [80];

  clrScr ();                                          /* clear screen */
  theMouse = mReset ();                               /* reset mouse */
  if (theMouse->exists) {                 /* do following if it exists */

/* Software mouse cursor */
    puts ("Software cursor:");
    printf ("Demo of a mouse with %d buttons\n", theMouse->nButtons);
    puts ("Move the mouse around and click the left button");
    puts ("Click the right button for hardware demo\n");
    mTextCursor (SOFTWARE, 0x0000, 0x0718);       /* set s/w cursor */
    mShow ();                                        /* turn it on */
    do {
      its = mReleased (LEFT);                    /* check left button */
      if (its->opCount > 0) {
        mHide ();                        /* cursor off in case of scroll */
        printf ("\nMouse is at col %d, row %d", its->column,
                its->row);                          /* position report */
        mShow ();                                    /* cursor back on */
      }
      its = mReleased (RITE);                   /* check right button */
    } while (its->opCount == 0);             /* repeat until operated */

/* Now do hardware mouse cursor demo */
    clrScr ();                                          /* clear screen */
    puts ("Hardware cursor:");
    puts ("Move the mouse, click left button");
    puts ("Type something and press Enter");
    puts ("Click right button to end demo");
    theMouse = mReset ();                               /* reset mouse */
    mTextCursor (HARDWARE, 2, 5);                    /* set h/w cursor */
    mShow ();                                          /* cursor on */
    do {
      its = mReleased (LEFT);                    /* check left button */
      if (its->opCount > 0) {                    /* if operated . . . */
        col = its->column / 8;                /* compute text position */
        row = its->row / 8;
        gotoxy (col, row);                                /* go there */
        putchar ('?');                                     /* prompt */
        gets (input);
        mMoveto (its->column, its->row);          /* restore position */
      }
      its = mReleased (RITE);                   /* check right button */
    } while (its->opCount == 0);             /* repeat until operated */
```

MOUSE MYSTERIES

continued from page 61

Function 12: Define mouse event handler (mInstTask). This rather esoteric function keeps the code free of frequent mouse checks. In fact, function 12 is so powerful that you can incorporate a mouse into your program with very few function calls.

mInstTask (see Listings 1 and 2) attaches a user-defined routine (a "task") to the mouse device driver. When calling **mInstTask**, you effectively tell the device driver, "Watch for certain events. When they happen, call my routine." Thus, a portion of your program becomes an extension of the device driver.

THE EVENT HANDLER

An *event* is any of the mouse actions listed below. You pass a 16-bit mask in the CX register to tell the device driver which events to watch for. The bit numbers and their corresponding events are given in Table 2. For example, if you want the device driver to call your routine when either the left or right button is released, pass the mask 0014H. Note that bits 7 through 15 are unused.

BIT	EVENT
0	Mouse cursor moved
1	Left button pressed
2	Left button released
3	Right button pressed
4	Right button released
5	Middle button pressed (Logitech only)
6	Middle button released (Logitech only)

Table 2. Mouse events and their corresponding mask bits.

You must also give the address of your event handler routine to the task installer. Pass the address' segment as the second parameter and its offset as the third parameter, as in the example below:

```
mInstTask(0X14,seg(handler),
  ofs(handler));
```

Thereafter, whenever the left or right button is released, the device driver automatically hands control to your routine via a far call.

When calling your handler, the device driver passes the information to the registers summarized

in Table 3. You can examine these registers to find out which event occurred and where it occurred. I'll cover this process presently, but first let's look at how to write the event handler.

REGISTER	DATA
AX	Event that occured (a bit set as above)
BX	Status of the buttons (1 bit if down)
CX	Horizontal cursor postion
DX	Vertical cursor position

Table 3. Returning data from mouse events.

Everything about your mouse event handler is the same as an interrupt service routine except that the event handler must exit via a far return. In Turbo Pascal, you must exit via a **RETF** instruction, rather than with the **IRET** instruction that Turbo Pascal automatically inserts into the machine code before the **END** of a procedure identified as **INTERRUPT**. As a result, you have to code the entire 12-byte exit processing sequence with inline code, as shown in MOUSEVNT.PAS (Listing 5). This code, by the way, is invariant for any mouse event handler, so you can safely insert it "as is" into your own routines.

The most important restriction on an interrupt service routine is that it cannot perform any I/O, DOS, or ROM BIOS calls. Since it's seemingly cut off from the world, what can the event handler possibly do? The answer is, "Not much"—nor should it. The object of the game is to handle the event as quickly as possible and return, so that the device driver can release control and let the running program resume. Consequently, the event handler should confine itself to saving a record of the event that the program can process at its convenience.

DOING IT IN TURBO PASCAL 4.0

Turbo Pascal 4.0 interrupt procedures have access to global variables due to the compiler-inserted entry processing (which

continued on page 64

```
/* Clean up and end of job */
    mTextCursor (HARDWARE, 6, 7);      /* use 11, 12 if mono board */
    mReset ();                                        /* reset cursor */
    clrScr ();                                        /* clear screen */
  } else
    puts ("Mouse not present in system. Demo can't run.");
} /* ----------------------- */
void  clrScr (void)                              /* clears the screen */
          /* Uses ROM BIOS int 10h to reset video mode to itself */
{
struct REGS  inreg, outreg;

  inreg.h.ah = 0x0F;                      /* first get current mode */
  ROMBIOS;
  inreg.h.al = outreg.h.al;                                /* copy mode */
  inreg.h.ah = 0;                         /* now reset to same mode */
  ROMBIOS;
} /* ----------------------- */
void  gotoxy (int col, int row)            /* position text cursor */
                                          /* Uses ROM BIOS int 10h */
{
struct REGS  inreg, outreg;

  inreg.h.ah = 2;                     /* function 2 sets cursor pos */
  inreg.h.bh = 0;                        /* video page 0 is active */
  inreg.h.dh = row;
  inreg.h.dl = col;
  ROMBIOS;
} /* ----------------------- */
```

LISTING 4: MOUSDEMO.PAS

```
PROGRAM mousdemo;     { Demo of the mouse unit }

USES dos, crt, mouse;

CONST hardware = 1;                                  { cursor types }
      software = 0;
      left     = 0;                                  { mouse buttons }
      right    = 1;

VAR  theMouse : resetRec;                        { for mouse fcn 0 }
     its      : locRec;                          { for mouse inquiries }
     col, row : integer;
     input    : string [80];


BEGIN
  CLRSCR;                                          { clear the screen }
  mReset (theMouse);                              { initialize the mouse }
  IF theMouse.exists THEN                    { and make sure we have one }

{ Do the software mouse demo first }
    BEGIN
      WRITELN ('Software cursor:');
      WRITELN ('Demo of a mouse with ',
               theMouse.nButtons, ' buttons');
      WRITELN ('Move the mouse around and click the left button');
      WRITELN ('Click the right button for hardware mouse demo');
      WRITELN;
      mTextCursor (software, $0000, $0718);          { set s/w cursor }
      mShow;                                         { turn cursor on }
      REPEAT
        mReleased (left, its);                     { check left button }
        IF its.opCount > 0 THEN BEGIN
          mHide;                                { in case screen scrolls }
          WRITELN ('Mouse is at column ', its.column,
                   ', row ', its.row);
          mShow;                                     { cursor back on }
        END;
```

```
      mReleased (right, its);                { check right button }
    UNTIL its.opCount > 0;                  { loop if not released }

{ Now do the hardware mouse demo }
    CLRSCR;
    WRITELN ('Hardware cursor:');
    WRITELN
      ('Move the mouse, click left button');
    WRITELN ('Type something and press Enter');
    WRITELN ('Click right button to end demo');
    mReset (theMouse);                      { clear old mouse status }
    mTextCursor (hardware, 2, 5); { set h/w cursor as small block }
    mShow;                                    { and turn it on }
    REPEAT
      mReleased (left, its);                { check left button }
      IF its.opCount > 0 THEN BEGIN
        col := its.column DIV 8;          { compute text position }
        row := its.row DIV 8;
        GOTOXY (col, row);                  { move text pointer }
        WRITE ('?');                                { prompt }
        READLN (input);                       { get the input }
        mMoveto (its.column, its.row);        { restore position }
      END;
      mReleased (right, its);               { check right button }
    UNTIL its.opCount > 0;                  { loop if not released }

{ Clean up afterwards }
    mTextCursor (hardware, 6, 7); { if graphics board, else 11,12 }
    mReset (theMouse);                      { reinitialize mouse }
    CLRSCR;
  END
ELSE
    WRITELN ('Mouse not present in system');
END.
```

LISTING 5: MOUSEVNT.PAS

```
PROGRAM mousevnt;                       { Illustrates mouse function 12 }

USES dos, crt, mouse;

TYPE mEvent = record                        { for recording mouse event }
       event,
       btnStatus,
       horiz,
       vert   : WORD;
     END;

VAR  mous : mEvent;
     m    : resetRec;

{ ------------------------------------------------------------------ }
PROCEDURE handler     { Mouse event handler called by device driver }
   (Flags, CS, IP, AX, BX, CX, DX, SI, DI, DS, ES, BP : WORD);

INTERRUPT;
```

MOUSE MYSTERIES

continued from page 63

occurs at the **BEGIN** keyword). This code saves all the CPU registers and then sets DS to point to the data segment, enabling the routine to read or write any program global. The procedure also has direct access to registers, since they're named as parameters. Consequently, you can simply copy from registers into the fields of the record defined as the **mEvent** type (or into any other variable of type **WORD**). Afterward, control passes through the inline sequence and returns to the device driver.

In this way, the handler quickly saves a record of the event and returns. A more sophisticated program than the demo in Listing 5 might instead stick the record into a linked list serving as a queue, so that several events can be stored up.

In either case, the program must look periodically to see if a mouse event has occurred (by checking if the event field is nonzero or if the queue pointer is not nil). If a mouse event has occurred, the program must then take action. A necessary part of the action is resetting the indicator so that the same event isn't processed more than once. The statement **mous.event := 0;** in the demo loop performs this step.

DOING IT IN TURBO C

The MOUSEVNT.C program (Listing 6) is functionally similar to its Turbo Pascal counterpart, with one notable exception—the location of the buffer that saves the mouse event status. The Pascal version treats this buffer as a global, which is accessible from the interrupt-class handler. The same thing could be done in Turbo C by making the handler function an interrupt-class process. However, I treated it differently due to a fundamental difference between the Turbo Pascal and Turbo C compilers.

Turbo Pascal makes it easy to insert the necessary inline code to

continued on page 66

Basically speaking, there's one choice . . . Turbo Basic!

Compare the BASIC differences!

	Turbo Basic 1.1	QuickBASIC 4.0 Compiler	QuickBASIC 4.0 Interpreter
Compile & Link to stand-alone EXE	*3 sec.*	7 sec.	——
Size of .EXE	*28387*	25980	——
Execution time w/80287	*0.16 sec.*	16.5 sec.	21.5 sec.
Execution time w/o 80287	*0.16 sec.*	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 at 10 MHz. The benchmark tests compiler's ability to optimize loop-invariant code, unused code, expression and conditional evaluation.

Turbo Basic® is the BASIC that lets even beginners write polished, professional-looking 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—not just a cramped 64K.

The critics agree. The choice is basic. Turbo Basic from Borland.

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.
- ***The Editor Toolbox*** is all you need to build your own text editor or word processor, including source code for two sample editors.

New!

*60-Day Money-back Guarantee**

"With a total programming environment like Borland International's Turbo Basic at the ready, even novice programmers can soon write programs that look as if they've been polished by a professional . . . 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] is easy enough to use for the beginning programmer and has enough power and sophistication for the professional. It is an unbelievable achievement for the price.

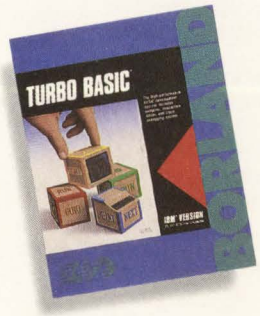
Giovanni Perrone, PC Week

[Turbo Basic] simply blew away my optimization test . . . these test results were truly wonderful surprises.

T.A. Elkins, Computer Language

Turbo Basic, simply put, is an incredibly good product.

William Zachman, Computerworld"



For the dealer nearest you call (800) 543-7543

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 1.0. 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.

BORLAND INTERNATIONAL

```
BEGIN
  mous.event     := AX;
  mous.btnStatus := BX;
  mous.horiz     := CX;
  mous.vert      := 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 }
       $58/                { POP  AX }
       $CB );              { RETF    }
END;
{ -------------------------- }

BEGIN
{ Set up screen }
  CLRSCR;
  GOTOXY (17, 25);
  WRITE ('Press left button for position, right to quit');
  GOTOXY (27, 1);
  WRITELN ('MOUSE EVENT-HANDLING DEMO');

{ Set up mouse }
  mReset (m);
  IF m.exists THEN BEGIN
    mInstTask ($14, seg (handler), ofs (handler));
    mous.event := 0;
    mShow;

{ Loop to perform demo }
    REPEAT
      IF mous.event = 4 THEN BEGIN
        mHide;
        WRITELN ('X = ', mous.horiz : 5, ', Y = ', mous.vert : 5);
        mShow;
        mous.event := 0;
      END;
    UNTIL mous.event = $10;

{ Clean up and quit }
    mHide;
    mReset (m);
  END;
  CLRSCR;
  END.
```

LISTING 6: MOUSEVNT.C

```
/* MOUSEVNT.C: Illustration of mouse function 12 */

/* INCLUDES */
#include <stdio.h>
#include <dos.h>
#include <mouse.i>

/* DEFINES */
#define ROMBIOS  int86 (0x10, &inreg, &outreg)

/* TYPE DEFINITION */
typedef struct {            /* structure for recording mouse event */
  unsigned  event,                          /* event that occurred */
            btnStatus,                    /* current button status */
            horiz, vert;          /* position where event occurred */
} mEvent;
```

MOUSE MYSTERIES

continued from page 64

exit properly from the handler. Turbo C also supports inline code, but you need to also have an assembler that the compiler can run as a child process in order to translate the inline assembly language. If you don't have an assembler, you can't use inline code. Thus, I chose a different way to store the mouse event status.

When DOS starts a program, it attaches a prefix structure called the Program Segment Prefix (PSP) to the memory image. The lower half of this 256-byte structure is rigidly formatted and should never be modified by the program. The upper 128 bytes, however, are available for the program to use. Turbo C programs don't ordinarily use this space, so we can safely grab some of it.

You can get the segment address of the PSP in a couple of ways. The simplest way is to read the system global **_psp**, which is furnished by DOS.H and is automatically initialized when the program begins execution. The **_psp** global contains the PSP segment. Note that the mouse setup in MOUSEVNT.C uses **_psp** to initialize the pointer to the event buffer.

Because it's called by the mouse device driver (and therefore inherits the device driver's data segment), the handler routine has no access to globals within MOUSEVNT.C (physically, the handler is a part of MOUSEVNT.C; functionally, it's a subroutine of the device driver). For that reason, we have to use another way to calculate the PSP segment. The PSP has a value 10H less than the code segment in the Turbo C *near* memory models (Tiny, Small, and Compact). Thus, we can easily find the PSP segment by subtracting 10H from the CS register (see the second statement in the handler function).

While MOUSEVNT.C is a simple demonstration of mouse event handling, it contains all of the elements necessary for more complex applications. The main part of the program constantly loops. With each loop, the program checks the event field to see if an event has occurred; if one has occurred, the program takes

action. The left button

mous->event = = 4

produces a position report on the screen. The right button

mous->event = = 0x10

terminates the loop, ending the program.

Function 13 and 14: Light Pen Emulation (mLpenOn/Off). The reset function sets light pen emulation ON by default. Software that expects light pen input then interprets the mouse cursor as the light pen's position.

Not much software uses a light pen, so it's seldom necessary to call these functions. You may need them when you're writing an application that uses a light pen that's independent of the mouse. In this case, you should turn emulation OFF immediately after reset, so that your program isn't confused by dual signals.

By now, you're probably eager to start creating your own spectacular user interface around a mouse. You should have enough information and the right tools to begin. I leave you with three pieces of advice about mouse programming:

1. Use a handler installed via function 12 as much as possible, rather than other status-checking calls.
2. Check the mouse event status often, preferably in a loop. Check elsewhere as well if you leave the loop for lengthy periods.
3. When you don't want the mouse to be available to the user, save the cursor position and hide it. Later, when reopening the mouse for business, reset the cursor position and restore the cursor to its former position.

See you in Part 2, where we'll hunt down the mouse in graphics territory. ■

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

Listings may be downloaded from CompuServe as CMOUSE.ARC.

```
/* LOCAL PROTOTYPES */
void clrScr (void);
void gotoxy (int, int);
/* ------------------------------------------------------------ */
void far handler (void)   /* event handler called by device driver */
{
mEvent  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->event = a;                      /* stuff registers into it */
  save->btnStatus = b;
  save->horiz = c;
  save->vert = d;
} /* ----------------------- */
main ()
{
mEvent  far *mous;
resetRec    *m;

/* Set up screen */
  clrScr ();                                          /* clear screen */
  gotoxy (17, 24);
  printf ("Press left button for position, right to quit");
  gotoxy (27, 0);
  puts ("MOUSE EVENT-HANDLING DEMO");

/* Set up mouse */
  m = mReset ();                                  /* initialize mouse */
  if (m->exists) {                           /* if mouse exists . . . */
    mInstTask (0x14, FP_SEG (handler), FP_OFF (handler));
    mous = MK_FP (_psp, 0x00C0);          /* point to event buffer */
    mous->event = 0;                        /* reset event signal */
    mShow ();                                     /* show cursor */

/* Loop to perform demo */
    do {
      if (mous->event == 4) {     /* if left button operated . . . */
        mHide ();                 /* hide cursor in case of scroll */
        printf ("\nX = %3d, Y = %3d", mous->horiz, mous->vert);
        mShow ();                              /* show cursor again */
        mous->event = 0;                      /* reset event signal */
      }
    } while (mous->event != 0x10);        /* loop til right button */

/* Clean up and quit */
    mHide ();                                        /* cursor off */
    mReset ();                                 /* reinitialize mouse */
  }
  clrScr ();                                          /* clear screen */
} /* ----------------------- */
void clrScr (void)                                    /* clear screen */
{          /* Uses ROM BIOS int 10h to reset video mode to itself */
struct REGS  inreg, outreg;

  inreg.h.ah = 0x0F;                            /* get current mode */
  ROMBIOS;
  inreg.h.al = outreg.h.al;                              /* copy mode */
  inreg.h.ah = 0;                             /* reset to same mode */
  ROMBIOS;
} /* ----------------------- */
void gotoxy (int col, int row)             /* position text cursor */
{                                         /* uses ROM BIOS int 10h */
struct REGS  inreg, outreg;

  inreg.h.ah = 2;                      /* fcn 2 sets cursor position */
  inreg.h.bh = 0;                          /* video page 0 is active */
  inreg.h.dh = row;
  inreg.h.dl = col;
  ROMBIOS;
} /* ----------------------- */
```

++, --

There's more to adding or subtracting by one than meets the eye.

Bruce F. Webster

SQUARE ONE

Part of the strength and power of Turbo C comes from your freedom to build tight, precise expressions that do exactly what you want without taking up a lot of source code. Much of this freedom comes from C's large list of operators and the flexibility you have in using them.

New users of C sometimes have problems with four of C's operators: preincrement, predecrement, postincrement, and postdecrement. The names seem formidable, but you can easily break them down: "pre" means "before," "post" means "after," "increment" means "add 1," and "decrement" means "subtract 1." So, assuming that **x** is a variable in some expression, Table 1 shows you how the operators look and what they do.

OPERATOR NAME	EXAMPLE	MEANING
Preincrement	**++x**	add 1 to **x** before evaluating the expression
Predecrement	**--x**	subtract 1 from **x** before evaluating the expression
Postincrement	**x++**	add 1 to **x** after evaluating the expression
Postdecrement	**x--**	subtract 1 from **x** after evaluating the expression

Table 1. Turbo C's increment and decrement operators.

Consider the postincrement operator. The Turbo C statement

```
x++;
```

is roughly equivalent to the Turbo Pascal statement:

```
x := x + 1;
```

I say "roughly" because you can put **x++** into expressions, something the Pascal language won't let you do with **x := x + 1.**

For example, the following statement means "assign the value of **x** to **y**, then add 1 to **x**":

```
y = x++;
```

If **x** had a value of 42 prior to executing that statement, then **x** would have a value of 43 and **y** would have a value of 42 *after* the statement executed.

Notice that the statement below has a completely different result:

```
y = ++x;
```

This statement adds 1 to **x** *before* assigning **x** to **y**, so that if **x** starts out with a value of 42, both **x** and **y** end up with a value of 43 after the statement executes.

Perhaps the most common use of these operators is within loops, especially **for** loops. The typical format is:

```
for (i= start; i <= finish; i++) {
  <statements>;
}
```

This **for** loop executes <**statements**> with the variable **i** going from **start** to **finish**, being incremented by 1 each time. You can build the same structure with a **while** loop:

```
i = start;
while (i <= finish) {
  <statements>;
  i++;
}
```

These examples show how the increment and decrement operators work, but they don't really demonstrate these operators' usefulness. Consider the short program in Figure 1.

The third executable statement—the **while** loop—does all the work of counting how many characters are in the string variable **line**. Note that the **while** structure itself doesn't have any statements to execute; the counting work is done within the expression **line[count ++]** inside the parentheses.

How does this work? First, remember that the **while** loop will repeat its evaluation of the expression within the parentheses until the expression takes on a value of 0 (or **FALSE**). Strings in C (such as **line**) are terminated with a NUL (ASCII 0) character. So, this loop will sit there, with **count** incrementing itself, until a NUL character is encountered at **line[count]**.

The variable **count** starts off at 0, so the **while** loop tests **line[0]**, then increments **count**. The loop continues this process, incrementing **count** by 1 each

continued on page 70

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.

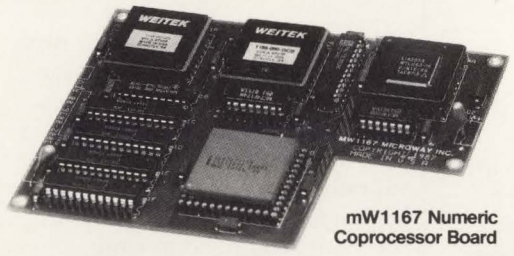
The benefits of our new technologies include:

- 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



mW1167 Numeric Coprocessor Board

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 $1495
mW1167 20 MHz $1995

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 meg[1] ...$1495
Monoputer T800-20 with 2 meg[1] ...$1995

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 meg[1] $3495
Quadputer 4 T414-20 with 4 meg[1] ...$6000

[1] Includes 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 $795
80387-16 16 MHz $495
80287-10 10 MHz $349
80287-8 8 MHz $259
80287-6 6 MHz $179
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

```
main()
  {
   char   line[128];
   int    count;

     count = 0;                /* initialize the counter to 0 */
     scanf("%s",line);         /* get a string from the user  */
     while (line[count++]);    /* count chars in the string   */
     printf("There are %d characters in the line\n",--count);
   }
```

*Figure 1. Doing real work with an empty **while** statement.*

```
{
float *myptr,mylist[20];

myptr = &mylist[0]; /* point myptr at start of mylist */
while (myptr < &mylist[20]) {
  *myptr = 1.0;
  myptr++;
}
```

Figure 2. Incrementing printer referents.

++, - -

continued from page 68

time, until **line[count]** is NUL, and the loop ends. Note, however, that **count** will be incremented one last time *after* the NUL is detected, and so **count** will be too large by 1. That's why we put **--count** in the **printf()** call: we first decrement **count** by 1, *then* print out its value.

POINTERS

When a pointer-type variable is being incremented or decremented, the ++ and -- operators work a little differently. They still add or subtract a fixed value from their operands, but that value now depends upon the pointer type—you can no longer assume the value to be 1.

Suppose you have some data type called **mytype**, and that **myptr** is declared to be a pointer to **mytype** as shown below:

```
mytype    *myptr;
```

Given the above declaration, the expression

```
myptr++;
```

no longer means "add 1 to **myptr.**" Instead, it means "add **sizeof(mytype)** to **myptr**." In other words, the address stored in **myptr** is incremented by the number of bytes that a variable of type **mytype** uses.

How would this be helpful? Consider the code in Figure 2. Here, **mylist** is a list of 20 elements of type **float**, and **myptr** is a pointer to type **float**. The **while** loop starts with **myptr** pointing to the first element in **mylist**. Each pass through the loop points **myptr** to the next element in the list. The expression **myptr ++**, adds the size (in bytes) of type **float** to **myptr**, "bumping" **myptr** to the next element in **mylist**.

If the size of the **float** type is not taken into account, incrementing **myptr** only points **myptr** to the second byte in the same floating point number that **myptr** pointed to originally—perhaps a useful thing to do from time to time, but *not* what we want to do here.

Pointers, then, are a special case for incrementing and decrementing. The assumption in incrementing or decrementing a pointer is that the pointer is pointing to a list of items that have the same data type. The idea is to point to the next (or previous) item in the list, *not* simply to add or subtract one byte to or from the address of the pointer's referent.

PITFALLS

You need to be aware of potential problems using these operators. For example, consider the following statements:

```
x = 10;
y = 3 * x++ + 5 * --x;
```

After these statements execute, what values do **x** and **y** have? The answers are 10 and 72, respectively. Why? Because the **--x** takes place before the expression is evaluated at all, reducing **x** to 9. 3 * 9 + 5 * 9 is 72. **x** is then incremented back up to 10 with **x++**.

A second problem, alluded to above, deals with pointers. In the **test()** function in the following program, note that we're passing a pointer to a **long** variable, so that we can change it within **test()** and have that change reflected elsewhere:

```
void test(val)
long *val;
{
   *val++;
}
main()
{
   long   i;
   i = 0;
   test(&i);
   printf("i = %d \n",i);
}
```

However, when the program runs, you'll find that **i** is not (necessarily) 1. That's because the expression ***val** ++ within **Test()** does a postincrement on the *address* in **val** rather than on the long value *pointed to* by ***val**. To increment the value, you need to write **(*val)** ++ instead. This forces the ++ operator to act upon ***val** rather than just **val**.

CONTROL, CONTROL

The increment and decrement operators in C are popular because they're concise, useful, and allow tremendous control over values in iterative operations. Two cautions when you use them:

- Comment heavily; the terseness of C in general, and the terseness of these operators in particular, makes extra commenting necessary.
- Be sure that in any given expression, increment and decrement do what you want them to do. Look carefully at which side of the operand the operator is placed, and make sure you understand the difference between incrementing or decrementing a pointer and incrementing or decrementing the pointer's referent.

Look before you code. That's one of several important axioms in working with C, and never more necessary than when working with ++ and --. ■

Bruce Webster is a computer mercenary living in the Santa Cruz mountains. He can be reached via MCI MAIL (as Bruce Webster) or on BIX (as bwebster).

A QUATTRO SAVE TRANSLATOR

Quattro's open architecture lets you create your own translators for writing spreadsheets to disk in any format you choose.

Bruce F. Webster

PROGRAMMER

With release of the Quattro Developer's Toolkit, you can join the Quattro add-in development team and begin writing your own extensions and additions to Borland's Professional Spreadsheet. Starting in this issue, *TURBO TECHNIX* will present a series of articles on creating custom Quattro add-in programs with Turbo C and Turbo Pascal. Because special libraries and start-up code are needed to create Quattro add-ins, these programs assume that you have the Quattro Developer's Toolkit.

In this issue, we'll look at file translator drivers, or "translators." A translator performs one of two tasks, depending upon whether information is moving into or out of Quattro. A *retrieve translator* converts a file into a series of records that Quattro understands. A *save translator* saves a Quattro spreadsheet as a disk file using a defined format.

As a simple example, consider a translator that saves a spreadsheet to disk. Specifically, the translator saves the current Quattro spreadsheet as a plain ASCII text file, with each row of cells taking up one line and all cells within a row separated by commas. Cells that contain text (such as *label cells*) are saved as text surrounded by quotation marks; thus, any commas embedded in the text won't be confused with commas that separate two adjacent cells. Empty rows are inserted as blank lines; empty cells in a non-empty row are inserted as a pair of double quotes ("").

HOW A SAVE TRANSLATOR WORKS

When Quattro saves a file to disk, it checks the extension of the output filename provided by the user, then loads the corresponding file translator. Quattro looks in its current directory for a translator file named FSxxx.TRN, where xxx is the output file extension. For example, if Quattro is told to save the current spreadsheet to BUDGET.ASC, it looks for a file named FSASC.TRN, loads that file, and then uses it to save the spreadsheet.

Quattro opens the output file itself and starts passing *spreadsheet records* to the save translator. Quattro contains two types of spreadsheet records: cell and noncell. A *cell* record holds the contents of a given cell (i.e., the spreadsheet's data), and can be one of several types (most notably, **INTEGER**, **NUMERIC**, **LABEL**, and **FORMULA**). *Noncell* records hold spreadsheet settings and other descriptive information.

Quattro passes these spreadsheet records one at a time to the save translator. The translator then decides what to do with each record. Typically, the translator converts each record to a format appropriate to the type of file it's creating. Next, the translator writes the translated record to that new file. While the translator can ignore or combine records, it always receives each record one at a time, in this order:

- WKQBOF (WKQ Beginning-Of-File) record
- all noncell records
- all cell records
- WKQEOF (WKQ End-Of-File) record

The translator can request that Quattro skip all the noncell records; likewise, after receiving all the noncell records, the translator can request that Quattro skip all the cell records.

STRUCTURE OF A TRANSLATOR

A save translator must contain at least these four functions: **init_save()**, **cvt_rec()**, **end_nonc()**, and **end_save()**.

init_save(). This function handles initialization of the save translator. (**init_save()** does *not* open the output file—that step is performed by Quattro.) **init_save()** takes the following form:

```
int  init_save(unsigned h,
               char *filename,
               unsigned password);
```

continued on page 74

Borland's New Professional Spreadsheet

Quattro: Twice the speed. Twice the power. Half the price.

Quattro™, our new professional spreadsheet, proves there are better and faster ways to do everything. To do graphics. To recalculate. To do macros. To save and retrieve. To search, sort, load. To do anything and everything that state-of-the-art spreadsheets should do.

Quattro gives you presentation-quality graphics

Quattro brings new highs in quality graphics to your spreadsheet. It also brings Postscript™ support and new variety and diversity to the kinds of graphs and graphics you can produce from your spreadsheet, and you can produce hard copy of your graphics—with either printer or plotter—without leaving the spreadsheet. All you do is hit "Print." Quattro makes it easy to get hard copy—and you don't have to buy a separate graphics program.

*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.

Quattro recalculates much faster than you-know-who

The smartest and fastest way to recalculate a spreadsheet is to do what Quattro does, something called "intelligent recalc," which in English means you only re-count the numbers that count.

In a spreadsheet, not all numbers are born equal and changing one number doesn't always change *everything*, so Quattro recalculates just the formulas that matter, not all the formulas it knows. (You wouldn't reshoot a whole movie just because you changed one scene, but unfortunately, that's the way 1-2-3 does it—and that's why it takes so long.)

Quattro demystifies Macros and makes your work go faster

Using macros—electronic shortcuts—is easy with Quattro. Quattro offers a complete macro debugging environment as you "single-step" through your macros and record them as you work.

BI 1206A

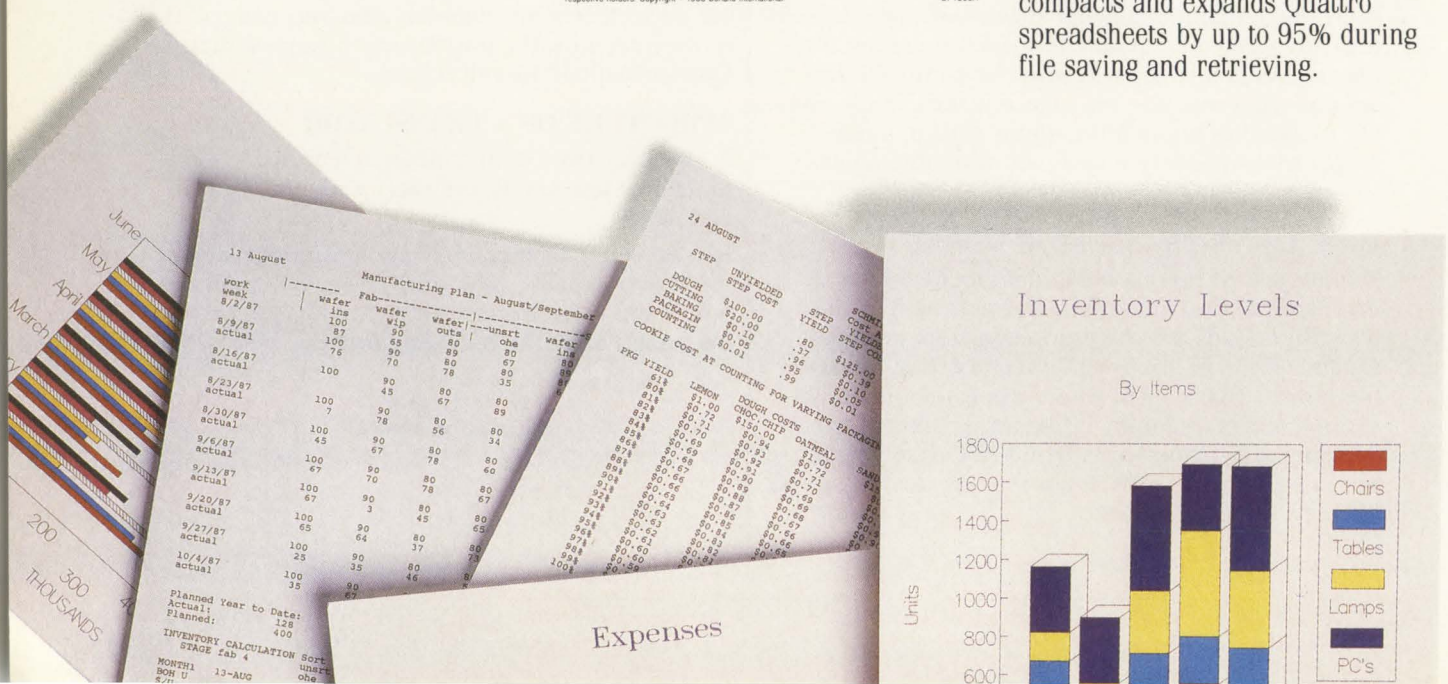
If you know how to use 1-2-3, you know how to use Quattro

You don't have to learn a whole new program. Quattro works directly with all 1-2-3 file formats. No importing/exporting or macro translation is required. Quattro can also load and save ASCII, Paradox®, and dBASE® files.
Compatible with 1-2-3? Yes.
Faster than 1-2-3? Yes.
Technically superior to 1-2-3? Yes.
Half the price of 1-2-3? Yes!

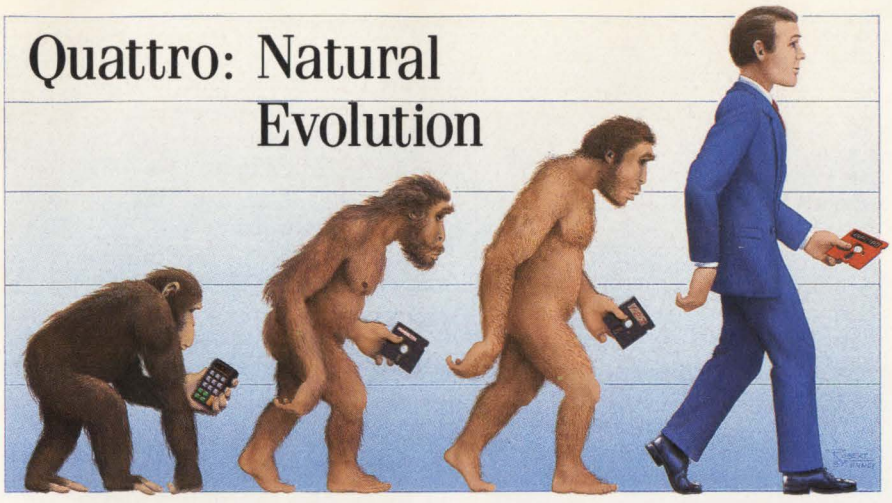
> "... It's a perfect choice for either the novice spreadsheet user or for someone who has mastered every arcane twist of 1-2-3."
>
> *Ezra Shapiro, Byte*

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.



Quattro: Natural Evolution



1970
HAND-HELD CALCULATOR

1979
VISICALC®

1982
LOTUS 1-2-3®

1987
QUATTRO™

Quattro: The Professional Spreadsheet

	FEATURE	QUATTRO	LOTUS 2.01
SPEED	ReCalc Cash Flow Model (5K cells)	.27 sec.	2.90 sec.
	Delete Row 15K cells (Recalc Time)	.76 sec.	2.38 sec.
	Load File (15K cells)	15.9 sec.	19.8 sec.
	Page Down (A1 to A1000)	12.2 sec.	17.4 sec.
GRAPHICS	Presentation-quality Graphics	YES	NO
	Graph Types	10	6
	Integrated Graph Printing	YES	NO
	Full Graph Customization	YES	NO
	On-Screen Font Styles	11	1
	PostScript Support	YES	NO
VERSATILITY	User-modifiable Menus	YES	NO
	Menu Shortcuts	YES	NO
	Pull-down menus	YES	NO
	Point and Press Editing	YES	NO
	Automatic Installation	YES	NO
POWER	Macro Learn Mode	YES	NO
	Maximum Number of Macros	Unlimited	27
	Single Step Macro Debugging Environment	YES	NO
	Price	$247.50	$495

*60-Day Money-Back Guarantee**

Benchmark details available upon request.

Get Quattro, the professional spreadsheet for only $247.50

Quattro is so advanced it's easy to use and it's half the price of 1-2-3. It's fully compatible with all your existing 1-2-3 files and macros—but it makes everything in them look better, print better and makes your work go faster.

For the dealer nearest you or a brochure,

Call (800) 543-7543

BORLAND INTERNATIONAL

For the IBM PS/2 and the IBM family of personal computers and all 100% compatibles.

QUATTRO

```
typedef struct {
   unsigned rectype;  /* record type */
   unsigned reclen;   /* record body length */
   char     recbody;  /* start of record body */
} REC_HDR;

typedef struct {
   int          doserr;  /* error code (0 if none) */
   unsigned int rectype; /* record type */
   int          reclen;  /* record length */
   char        *recdata; /* pointer to record body */
} RECDESC;
```

*Figure 1. The **REC_HDR** and **RECDESC** types used by a Quattro save translator.*

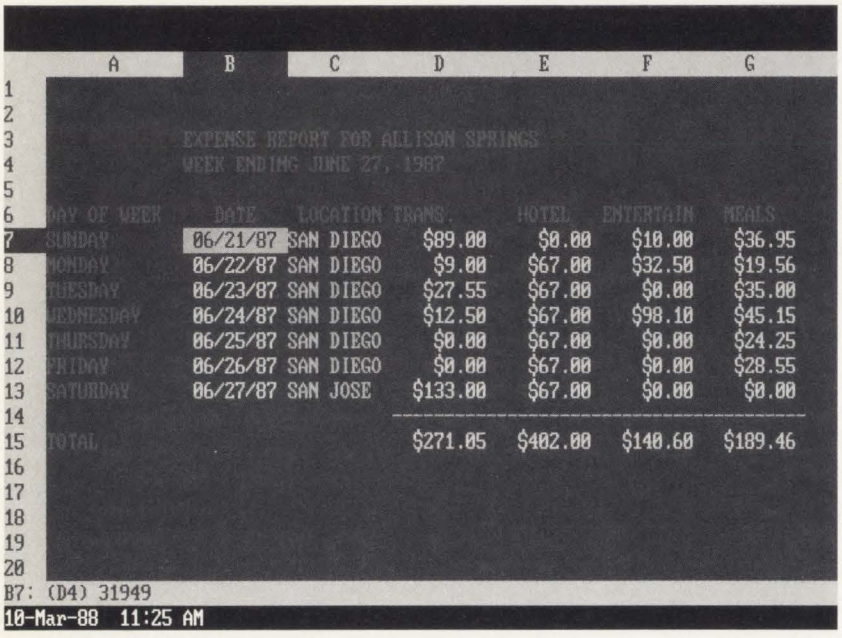


Figure 2. The spreadsheet SAMPLE.WKQ on the Quattro screen.

```
"","EXPENSE REPORT FOR ALLISON SPRINGS"
"","WEEK ENDING JUNE 27, 1987"

"DAY OF WEEK","DATE","LOCATION","TRANS.","HOTEL","ENTERTAIN","MEALS","TOTAL","","DAY OF WEEK","DATE","MEALS"
"SUNDAY",31949,"SAN DIEGO",89,0,10,36.95,135.95,"","TUESDAY",31951,35
"MONDAY",31950,"SAN DIEGO",9,67,32.5,19.56,128.06,"","SUNDAY",31949,36.95
"TUESDAY",31951,"SAN DIEGO",27.55,67,0,35,129.55,"","WEDNESDAY",31952,45.15
"WEDNESDAY",31952,"SAN DIEGO",12.5,67,98.1,45.15,222.75
"THURSDAY",31953,"SAN DIEGO",0,67,0,24.25,91.25
"FRIDAY",31954,"SAN DIEGO",0,67,0,28.55,95.55
"SATURDAY",31955,"SAN JOSE",133,67,0,0,200
"","","","-","-","-","-","-"
"TOTAL","","",271.05,402,140.6,189.46,1003.11
```

Figure 3. The spreadsheet SAMPLE.WKQ , translated to comma-delimited ASCII format.

THE SAVE TRANSLATOR

continued from page 71

h is the DOS file handle for the file that Quattro has opened. **filename** is the complete DOS pathname, and **password** is a flag that indicates if the user has entered a password.

init_save() returns one of two values: **0** or **SKIP_NON_CELLS**. If **init_save()** returns **0**, then Quattro passes the noncell records, starting with BOF; if **init_save()** returns **SKIP_NON_CELLS**, Quattro passes only the cell records.

cvt_rec(). This function retrieves the Quattro spreadsheet records one at a time. It converts each record to the required output format, and writes that converted record to the output file. **cvt_rec()** typically appears as shown below:

```
RECDESC *cvt_rec(REC_HDR r);
```

Figure 1 shows the types **REC_HDR** and **RECDESC**, which are defined in the header files included with the Quattro Developer's Toolkit. The record type and length fields together are called a *record header*. The field **recbody** is the beginning (the first byte) of the *record body*.

The basic structure of **cvt_rec()** is usually a **switch** statement, based on **r.rectype**, which handles each record according to its type.

cvt_rec() is expected to return a pointer to a **RECDESC** structure. This structure must be declared as static so that it continues to exist between calls to **cvt_rec()**. The entire **RECDESC** structure should be initialized to zero. However, the **doserr** field can be used to prematurely abort the entire save process, in case of file corruption or other errors. To abort, set

doserr to one of the following values, which are predefined in the header files:

DE_CORRUPT—Save the file to disk with this record as the last record.

DE_ABORT—Don't save the file; delete the specified file from disk.

end_nonc(). Quattro calls **end_nonc()** to signal that all of the noncell (spreadsheet information) records have been passed to the translator. **end_nonc()** is shown below:

```
int end_nonc(void);
```

All records passed by Quattro *after* it calls **end_nonc()** are cell records. At this point, you can tell Quattro to skip all the cell records, according to which of the following two values are returned by **end_nonc()** to Quattro:

0—Pass all cell records.

SKIP_CELLS—Skip all cell records.

If **end_nonc()** returns **SKIP_CELLS**, then Quattro completes the save process without passing any additional spreadsheet records.

end_save(). **end_save()** is provided for the convenience of the translator, and typically is used to release memory heaps or to close any auxiliary files opened by the translator. (Quattro opens and closes the data file selected by the user.) A return value is not required. **end_save()** is shown below:

```
void end_save(void);
```

More about translator functions. You can, of course, have many more functions within your save translator. However, you must include the four functions discussed above, and they must contain the declarations shown above; otherwise, you won't be able to link those functions with the necessary routines to produce an executable file translator. Also, you *cannot* put a **main()** function into your translator!

A SIMPLE TRANSLATOR

FSASC.C in Listing 1 is a simple save translator that writes a Quattro spreadsheet as an ASCII

continued on page 76

LISTING 1: FSASC.C

```
/* FSASC.C -- Comma-delimited save driver

   author:       David Golden (minor changes by Bruce F. Webster)
   last update:  08 March 1988
   compiler:     Turbo C 1.5
   link files:   COSAVEC.OBJ, QC.LIB, CC.LIB

   This example uses 'cvt_rec' to convert records.  Formula records
   are converted to either a NUMBER or LABEL record.
*/

#include <quattro.h>     /* general Quattro Toolbox header file */
#include <qdriver.h>     /* additional header file for translators */

char out[300];           /* output buffer must be external to any
                            procedure to ensure it stays in existence
                            upon return to Quattro */

unsigned handle;         /* file handle stored here */


/* * * Procedure specific to saving comma-delimited * * */

/*
   Procedure 'fillin' that 'fills in' by generating:
   1) 'null' fields that serve as placeholders for blank cells ( "" )
   2) 'null' lines that serve as placeholders for blank rows
       ( just CR/LF )

Receives: pointer to cell record that contains header information
            needed (the cell col and row)

Returns:  nothing
*/

int lastcol=0;  /* column for last cell written */
int lastrow=0;  /* row for last cell written */

int foundinrow=0;  /* 'true' if cell already written out
                      for this row */

/* all record types have header info in common */
void fillin(NUMBER_REC *r)
{
   int    i,num_lines,num_flds,str_len;
   int    col = r->col;
   int    row = r->row;

   if(row > lastrow)
   {  /* generate end of line sequence followed by as many blank lines
         as needed */
      out[0] = 0x00;  /* ensure we concatenate at start of buffer */
      for( num_lines = row-lastrow, i=0 ; i < num_lines ; i++ ) {
         strcat(out,"\r\n");
         /* output the buffer every 125 iterations to guarantee we
            don't overflow:  125*2=250 ( <300 ) */
         if( (i%125)==124 ) {
            _write(handle, out, strlen(out) );
            out[0] = 0x00;
```

```
        }
      }
      if( strlen(out)>0 ) /* flush buffer */
         _write(handle, out, strlen(out) );

      lastrow=row;
      lastcol=0;
      foundinrow=0;
   }

   /* generate field separators to go between this field
      and previous one, and if necessary create null fields
      for the empty cells */
   out[0] = 0x00;
   if (foundinrow)
      strcat(out,",");

   /* generate null fields (including preceding comma) if an empty
      cell preceded this one (in the same row) */

   /* see if we need to create any null fields to precede this one */
   if(!foundinrow)
      num_flds=col;              /* row starts with empty cells */
   else
      num_flds=col-lastcol-1;

   for( i=0 ; i<num_flds ; i++ ) {
      strcat(out,"\"\",");
      if( (i%90) == 89 ) {
         _write(handle, out, strlen(out) );
         out[0] = 0x00;
      }
   }
   if( (str_len=strlen(out)) > 0 ) /* flush buffer */
      _write(handle, out, str_len );

   lastcol = col;
   foundinrow = 1;
   return;
}

/* init_save() function

   Initialize save driver.  This is the first procedure called.
   Should save as an extern the file handle passed to it since
   it is not passed on subsequent calls.  Quattro opens and
   closes the output file.

   Return value is either 0, meaning to pass non-cell records then
   to pass cell records, or SKIP_NON_CELLS to skip non-cell records.

   NOTE: this version will cause two warnings, since we use neither
         'filename' nor 'password'. You can safely ignore these
         warnings.
*/
```

THE SAVE TRANSLATOR

continued from page 75

(plain text) file, with one row per line. Cells in a given row are separated by commas; empty cells are inserted as a pair of double quotes (""). Labels are enclosed by double quotes to identify them as labels and to prevent any confusion that could be caused by the (entirely legal) presence of a comma within a label.

In addition to the four required translator functions, FSASC.C contains the function **fillin()**, which checks for empty cells and rows located between the cell just passed to FSASC.C and the last cell that FSASC.C handled. **fillin()** inserts a pair of double quotes ("") to indicate empty cells, and inserts blank lines for empty rows. All cells in a given row are separated by commas.

The function **init_save()** just saves a copy of the file handle and returns a value of **SKIP_NON_CELLS**, telling Quattro that the translator only wants cell records.

Since the noncell records are being skipped, the function **end_nonc()** won't be called by Quattro and doesn't have to do anything. It still must be declared, however, in order for the translator to link properly.

cvt_rec() is called once by Quattro for each nonempty cell. Quattro starts with the first (topmost) nonempty row and sends all nonempty cells (from left to right) in that row before moving to the next nonempty row. **cvt_rec()** calls **fillin()** to insert any empty rows or cells, then writes out the numeric value or label string.

In this case, **end_save()** doesn't do anything, but it needs to be declared so that the translator can be linked.

COMPILING AND LINKING

A file translator's internal structure is different from the internal structure of a regular .EXE file.

Specifically, a file translator has different startup code and uses a special library (which replaces the regular Turbo C Runtime Library). To create the translator, you must have the Quattro Developer's Toolkit, which contains all the files you need.

A file translator must be compiled with Turbo C's Compact memory model. The startup code file is C0SAVEC.OBJ (instead of the regular C0C.OBJ). Since the special library is QUATTROC.LIB, which contains all the I/O routines used in FSASC.C, you don't have to link in CC.LIB. If you use any math routines, then you must link in a special library, QMATHC.LIB, which replaces MATHC.LIB.

If you're using the Turbo C Integrated Environment (TC.EXE), you'll need to use a project file. For FSASC.C, create the file FSASC.PRJ, which contains the following:

```
c:\tc\lib\c0savec.obj
fsasc.c
c:\tc\lib\qc.lib
```

This file presumes that your library directory is C:\TC\LIB; you may need to change this path to reflect your own hard disk setup.

Note that when you list C0SAVEC.OBJ as the first file, Turbo C's linker uses it (instead of C0C.OBJ) as the startup code. Be sure to set the compiler model to Compact. After you compile and link, you'll have a file named FSASC.EXE. Rename it to FSASC.TRN, and copy it to your Quattro disk or directory for testing and use.

If you're using the Turbo C command-line compiler (TCC.EXE), compile your program to .OBJ code only. Use the Compact model and tell TCC where the include directory is located:

```
C>tcc -c -mc -Ic:\tc\include fsasc
```

Invoke TLINK.EXE and tell it which startup code and libraries to use:

```
C>tlink c:\tc\lib\c0save fsasc,
  fsasc.trn,, c:\tc\lib\qc
```

continued on page 78

```
init_save(h,filename,password)
unsigned h;          /* file handle */
char *filename;      /* filename -- has Pascal style length byte as
                        initial byte, null-terminated */
unsigned password;   /* password flag : 1 if one entered, 0 if not */
{
   handle = h;
   return SKIP_NON_CELLS;
}

/* end_nonc() function

   Called after all the non-cell records have been passed.
   Return SKIP_CELLS to skip the cell records or 0 to not skip.
*/

int end_nonc(void)
{
   return 0;
}

/* cvt_rec() function

   Convert given record.  Called to present a worksheet record to
   the driver.  The translator can either write the record information
   to disk or ignore the record.

   This procedure is passed a pointer to the worksheet record header.
   It should return a pointer to a record descriptor structure.
   On a 'save' operation the 'doserr' and 'reclen' fields are used
   to return status information to Quattro.

   The doserr field must be given one of the following status codes:
      DE_CORRUPT  save the file to disk with this as the last record
      DE_ABORT    do not save the file; delete it from disk
      ACCEPTED    all is okay; send the next record
*/

RECDESC rec;         /* record descriptor to return.  This must
                        be external to ensure it remains in existence
                        after a pointer to this structure is returned
                        to Quattro.  */

RECDESC * cvt_rec(REC_HDR *r)
{

   unsigned rectype;    /* record type */
   void     *body;      /* pointer to record body */

/*   qTempHeap(4*1024); /* create heap for internal translator use */

   rectype = r->rectype;   /* fetch record type */

   rec.doserr = ACCEPTED;  /* Primary return flag says all is OK */
   rec.reclen = ACCEPTED;  /* Secondary return flag says all is OK. */

   body = (void *) &r->recbody;    /* pointer to body */

   switch (rectype) {
```

```
   case INTEGER: {
      INTEGER_REC *c = (INTEGER_REC *) body;

      fillin( (NUMBER_REC *) c);  /* typecast done
                                     to prevent warning */
      sprintf(out,"%d",c->value);
      _write(handle, out, strlen(out));
      break;
   }

   case NUMBER: {       /* includes numeric-valued formulas */
      NUMBER_REC *c = (NUMBER_REC *) body;

      fillin(c);
      sprintf(out,"%g",c->value);
      _write(handle, out, strlen(out));
      break;
   }

   case LABEL: {       /* includes string-valued formulas */
      LABEL_REC *c = (LABEL_REC *) body;

      fillin( (NUMBER_REC *) c); /* typecast done
                                    to prevent warning */
      /* skip leading format byte: */
      sprintf(out,"\"%s\"",c->text+1);
     _write(handle, out, strlen(out));
     break;
   }

   default:             /* ignore WKQEOF end of file record */
      break;
}

return &rec;            /* return pointer to record descriptor */

}

/* end_save() function

   Called after all of the records have been passed to the driver.
   Gives the translator the chance to close ancillary files,
   deallocate heaps, etc.

*/

void end_save(void)
{
/*   qTempDeall();   /* release temporary memory heap */
   return;          /* There is no return value */
}
```

THE SAVE TRANSLATOR

continued from page 77

The result is FSASC.TRN; copy this file to your Quattro disk or directory.

USING A TRANSLATOR

Using a translator is easy: just save a spreadsheet with the appropriate file extension. For example, load in the file SAMPLE.WKQ, which comes on your Quattro disks. (Figure 2 shows a portion of SAMPLE.WKQ as it appears on your screen from within Quattro.) Now, save SAMPLE.WKQ to disk as SAMPLE.ASC, using the **S**ave option on Quattro's **F**ile menu. Behind the scenes, Quattro automatically looks for the save translator file FSASC.TRN, loads it from disk into memory, and uses it to write the spreadsheet to disk in ASCII format. The result is an ASCII file that looks like the file in Figure 3.

What if the appropriate file translator can't be found on disk? Quattro then saves the file to disk in the standard .WKQ format.

ADD-IN VERSATILITY

This should give you a feeling for what it's like to use the Quattro Developer's Toolkit. Most of the add-in programs that you can create follow the same general format of a set of functions called by Quattro at specific times or in response to specific events. In future issues, we'll talk about how to write custom @ functions, as well as general add-in programs in Turbo C and Turbo Pascal. ■

Bruce Webster is a computer mercenary living in California. He can be reached via MCI MAIL (as Bruce Webster) or on BIX (as bwebster.)

Listings may be downloaded from CompuServe as OUTRAN.ARC.

A MEMORY-RESIDENT CLOCK UTILITY

No matter what else your PC is doing, it can always give you the time of day.

Ron Sires

WIZARD

Memory-resident utilities have taken the PC world by storm. Once installed, these programs remain in your computer's memory and wait for an event that tells them to take action. The signal event may be user-generated, such as striking a mouse button or a particular key combination; or it may be computer-generated, like reading a disk or writing to the screen. After the utility has performed its task, it returns to its inactive state in the computer's memory, waiting for the signal event to occur again.

The signal event is termed an *interrupt*, and the procedure that is called when the interrupt occurs is an *interrupt service routine*, or ISR. The interrupts are numbered from 0 to 255, and many pass control to ISRs in the PC ROM BIOS when they occur. A memory-resident program may replace a ROM BIOS routine with its own ISR, but a well-behaved resident program will save the address of the interrupt's original ISR. When the interrupt is generated, the program will call the original ISR, after taking its own action. This allows more than one memory-resident utility to service the same interrupt.

CLOCK.COM, a memory-resident clock utility application, incorporates an ISR and demonstrates how Turbo C makes the development of well-behaved memory-resident programs very easy. CLOCK takes over the timer control interrupt (interrupt 1CH), which the PC generates about 18.2 times a second. Every time this interrupt occurs, CLOCK inspects the system time and displays it in the upper right corner of the screen if the time isn't already displayed there. Every time the minutes roll over to a new hour, the speaker beeps twice.

INSTALLING THE ISR

The source code for CLOCK.COM is given in Listing 1. The following global declaration in CLOCK.C defines **oldtick** as a pointer to an interrupt function that doesn't return a value:

```
void interrupt (*oldtick)();
```

Pointer **oldtick** saves the address of the timer interrupt's original service routine. The Turbo C keyword **interrupt** causes the compiler to add ISR housekeeping code to the beginning and end of the function. When the function terminates, the computer returns to the state it was in before the function took control.

The original ISR will be replaced by **tickintr()**, which is also declared as an interrupt function. This process of saving and replacing ISRs is handled by four lines of code from **main()**, shown in Figure 1.

The number of an interrupt, in this case interrupt 1CH, is passed to Turbo C's **getvect()** function. **getvct()** returns the address of that interrupt's current ISR, which is also known as the *interrupt vector*. This address is saved in **oldtick**. The **disable()** function turns off interrupts, so that the computer doesn't attempt to call the ISR before it's completely installed. The number of the interrupt being replaced, and the new ISR's address, are passed to the **setvect()** function. Turbo C interprets the identifier **tickintr** (without the parentheses) as the address of the function, rather than as a call to the function. Finally, **enable()** turns the interrupts back on once the new ISR is completely and safely installed.

WHAT TIME IS IT?

The ISR **tickintr()** uses the Turbo C **biostime()** function to determine the time. The invocation **biostime(0,0L)** returns the number of timer ticks (in the form of a long integer value) that has occurred since system midnight. These timer ticks occur at a rate of 18.20648193 per second, according to IBM's ROM BIOS listing. For CLOCK's purposes, rounding this rate to 18.2065 ticks per second yields sufficient accuracy.

continued on page 80

continued from page 79

First, we want to convert this number of ticks into the number of minutes that have passed since system midnight. The formula to use is

$$\text{minutes since midnight} = \frac{\text{ticks since midnight}}{18.2065 \text{ ticks/second} \times 60 \text{ seconds/minute}}$$

or:

$$= \frac{\text{ticks since midnight}}{1{,}092.39 \text{ ticks/minute}}$$

In a structured language such as C, designing a good data structure may be even more important than using a correct program control structure.

One major consideration in any memory-resident program (especially one that executes 18.2 times per second!) is to use as little time and memory as possible. Since floating point arithmetic is very costly in terms of both time and memory, it's best to use integer arithmetic. To perform this division using integers, we apply the mathematical fact that multiplying the top and bottom of a fraction by the same number doesn't change the value of the fraction. The first instruction in **tickintr()** multiplies the top and bottom of our fraction by 100, converting ticks into minutes without resorting to floating point arithmetic:

```
mins_aft_mid =
(biostime(0,0L) * 100L) / 109239L;
```

From this number, **tickintr()** derives **rawhour**, which is the hour on a 24-hour clock; **hour**, which is the hour on a 12-hour clock; and **minute**, the number of minutes after the hour.

```
oldtick = getvect(0x1C); /* Save original ISR in oldtick */
disable();               /* Disable interrupts.          */
setvect(0x1C, tickintr); /* Replace ISR with tickintr()  */
enable();                /* Allow interrupts again.      */
```

*Figure 1. The process of saving the old ISR interrupt vector and replacing it with **tickintr**.*

```
typedef struct SCR_LOC {
          char s_char, s_attr;
} SCR_LOC;                      /* One screen location.       */
typedef SCR_LOC SCRLINE[80];    /* One screen line.           */
SCRLINE   far *scr;             /* Entire screen.             */
```

Figure 2. Declaration of the screen access data structures.

```
MyLine[5][10].s_char = 'Z';  /* Put a 'Z' at row 5, col. 10 */

HomeAttr = MyLine[0][0].s_attr; /* Store the attribute of */
                                /* the upper left corner. */
```

Figure 3. Direct assignments to the video refresh buffer.

```
Start  Stop   Length Name              Class

00000H 00F9DH 00F9EH _TEXT             CODE
00FA0H 01247H 002A8H _DATA             DATA
01248H 0124BH 00004H _EMUSEG           DATA
0124CH 0124DH 00002H _CRTSEG           DATA
0124EH 0124EH 00000H _CVTSEG           DATA
0124EH 0124EH 00000H _SCNSEG           DATA
0124EH 01297H 0004AH _BSS              BSS
01298H 01298H 00000H _BSSEND           BSSEND
```

Figure 4. CLOCK's memory usage from the map file.

STRUCTURES AND SCREENS

In a structured language such as C, designing a good data structure may be even more important than using a correct program control structure. A data structure that corresponds closely with the object it models makes a program clearer and more efficient. CLOCK's method for accessing the display screen illustrates such a data structure.

Each position on the PC's screen corresponds to a pair of bytes in memory—the first byte is the ASCII code of the character, while the second byte is the attribute or color of the character and its background. Two thousand of these pairs exist; one pair corresponds to each location on the 25-row, 80-column display. These character-attribute pairs—called the *video refresh buffer*—begin at hex address B000:0000 for display adapters connected to a monochrome screen, or at B800:0000 for adapters connected to a color screen.

Once given the row and column of a character or attribute, CLOCK's video access data structures can read or write that character or attribute at any screen location. For example, the structures allow the program to easily put an 'S' at row 5, column 20; or to find the attribute of row 3, column 79. The video access data structures are set up in the declarations (excerpted from CLOCK.C) in Figure 2.

In Figure 2, the **typedef** keyword defines a complex variable in a series of logical steps. Unlike the declaration of a variable, a **typedef** declaration does not allocate memory for a variable. Instead, **typedef** creates a new data type that is equivalent to some usual C type. The first **typedef** declares the **SCR_LOC** type to be a structure of two bytes, where

one byte is called **s_char** and the other is called **s_attr**. Thus, if **MySpot** is a variable of type **SCR_LOC**, and **MySpot** occupies the same place in video memory as the screen's upper left corner, the character at that screen location is **MySpot.s_char**, and the attribute is **MySpot.s_attr**. The assignment below puts an 'A' at that screen location:

```
MySpot.s_char = 'A';
```

The second declaration in Figure 2 defines the **SCRLINE** type as an array of 80 **SCR_LOC** structures. This array is analogous to a single line on the screen; however, it comes close to meeting the goal of direct screen access by row and column because of the way C handles array indexing. If **MyLine** is a variable of type **SCRLINE** and is located at the beginning of video memory, then **MyLine[0]** is the first screen line, **MyLine[1]** is the second, and **MyLine[20]** is the 21st. This works because incrementing the index of an array by one causes the memory address that is accessed to be incremented by the *size* of the array's type. **MyLine**'s type is **SCRLINE**, and its size is equal to that of one screen line. This means that **MyLine[1]** is one screen line past **MyLine[0]** and **MyLine[20]** is 20 screen lines past **MyLine[0]**. Although this process accesses the screen by row, what about providing access to individual locations on that row? **MyLine[n]**, where **n** is any integer, is of type **SCRLINE**, which means that it is an array of 80 **SCR_LOC** structures. A second array index, specifying an element number from 0 to 79, allows access to individual elements of the array. Since each array element is a structure, the structure member must also be named for direct access to the screen. Assignments such as those in Figure 3 then become possible.

One further puzzle, glossed over in the above paragraphs, remains to be solved. When a variable of type **SCRLINE** is declared, Turbo C allocates space in CLOCK's data segment for that variable. In order for the screen access scheme to work, the variable must be located at the beginning of video memory. The only way to access a specific address in memory, such as B000:0000, is to use a pointer. In fact, for the single-user, single-task PC, a pointer and a memory address are virtually the same thing. The final trick to accessing the video memory as though it were a 25 × 80 array of character-attribute pairs is to declare a pointer to such an array, assign the desired memory location to that pointer, and then treat the pointer as the name of an array.

Since most memory-resident programs must be executed as .COM rather than .EXE files, those programs must be compiled to the Tiny memory model.

The pointer **scr** is defined as a far pointer to **SCRLINE** so that both a segment and an offset can be specified. **scr** is given its value through a call in **main()** to Turbo C's **MK_FP** library function (see "Building Far Pointers with **MK_FP**," *TURBO TECHNIX*, March/April, 1988, p. 61.) The pointer takes on a value of either B800:0000 or B000:0000, depending on whether or not a color display is in use. This allows assignments to and from **scr[row][column].s_char** and **scr[row][column].s_attr** to be used for reading from and writing to the screen, as shown in the **disptime()** function.

MAKING CLOCK MEMORY-RESIDENT

When a normal program terminates, it gives all of its allocated space back to DOS. For a program to remain resident, it must retain its memory allocation after control returns to DOS. Turbo C provides this ability with the **keep()** function, which takes the two integer parameters **status** and **size**. **status** is the exit status to be returned to DOS, and **size** is the amount of memory measured in paragraphs (one paragraph equals 16 bytes) that the program retains. It seems strange that this function requires that the program's size be known while the program is being developed, but there is a way around this requirement.

Since most memory-resident programs, including CLOCK, must be executed as .COM files rather than as .EXE files, those programs must be compiled to the Tiny memory model, which puts all of the programs' code and data into 64K of memory. This means that CLOCK can safely be compiled with **size** = 4096, because the program will retain as much memory as it could possibly need (64K). However, it's desirable for resident programs to use as little memory as possible, so once the program has been tested and debugged to its final form, the link map is used to find the actual number of paragraphs that the program requires. The following command line compiles and links CLOCK in the Tiny memory model, and generates a link map file called CLOCK.MAP:

```
tcc -mt -M clock.c
```

The map file is an ASCII text file that contains information about how CLOCK's memory will be

continued on page 82

LISTING 1: CLOCK.C

```
/* Clock.c -- Memory-resident program to display a clock in the    */
/*            upper right corner of the screen                    */
/* (c) Copyright 1988, Sires Software, All Rights Reserved.       */
/*                                                                */
/* CAUTION: Do not run CLOCK from within the Turbo C integrated   */
/*      environment.  The computer may hang if a memory-resident  */
/*      program is installed from within another program.         */
/*                                                                */
/* To compile from command line:    tcc -mt -M clock.c            */
/* To convert .EXE to .COM file:    exe2bin clock.exe clock.com   */
/*                                                                */

#define COPYRIGHT "CLOCK 1.0 (c) Copyright 1988, Sires Software, "\
                  "All Rights Reserved."
#include <dos.h>
#include <bios.h>

#define TRUE   -1
#define FALSE   0

typedef struct SCR_LOC {
    char    s_char, s_attr;
}  SCR_LOC;                             /* One screen location      */
typedef SCR_LOC     SCRLINE[80];        /* One screen line          */
SCRLINE             far *scr;           /* Pointer to entire screen */

char    attr;
void    interrupt (*oldtick)();

void disptime(int hour, int minute, int rawhour)
{
    int     i;

    scr[0][70].s_char = (hour > 9) ? '1' : ' ';
    scr[0][71].s_char = (hour % 10) + '0';
    scr[0][72].s_char = ':';
    scr[0][73].s_char = (minute / 10) + '0';
    scr[0][74].s_char = (minute % 10) + '0';
    scr[0][75].s_char = ' ';
    scr[0][76].s_char = (rawhour < 12) ? 'A' : 'P';
    scr[0][77].s_char = 'M';
    for (i=70; i<78; i++)
        scr[0][i].s_attr = attr;
    return;
}

void beep(void)
/*  Routine will beep the speaker twice briefly. */
{
    unsigned int     i, j;

    for (i=0; i < 2; i++)
    {
            sound(512);     /* Beep at approx. middle C     */
            for (j=0; j < 40000; j++) ; /*  for 1/2 second. */
            nosound();      /* Turn off speaker             */
            for (j=0; j < 20000; j++) ; /*  for 1/4 second. */
```

CLOCK

continued from page 81

allocated when CLOCK is run. The portion of CLOCK that pertains to **size** is given in Figure 4. The last value in the **Stop** column is the number of bytes (in hex) used by the program. To convert this value to the number of paragraphs, just shift it one hexadecimal digit to the right and round the last digit up. In the example, the number of bytes is 01298H, which converts to 012AH paragraphs (remember that in hexadecimal arithmetic, 9 + 1 = A). Thus, the **keep()** function for this example is **keep(0,0x12A)**. As was done in CLOCK, this number may be rounded up to err on the side of safety.

In order for CLOCK to work properly, it must be converted from the .EXE file produced by Turbo C to a .COM file. This conversion is done by the EXE2BIN program, which comes with DOS:

```
exe2bin clock.exe clock.com
```

CLOCK.C's internal arrangement, with **main()** last and all functions defined before they are used, is due to a peculiar limitation of EXE2BIN, which requires that the source code be in that format in order for the .EXE file to be converted.

LIMITATIONS AND POSSIBLE ENHANCEMENTS

CLOCK operates in text mode only. If you shift into graphics mode with CLOCK in operation, CLOCK's output will not be visible, because the text video buffer is at a different location from the graphics video buffer.

Text editors that work directly to screen memory may inadvertently copy CLOCK's display to other parts of the screen when they scroll the top line downward. A great many programs reserve the top line for status information, and CLOCK's display may obscure important information. You can reposition CLOCK's output simply by specifying a different row and column position.

Because CLOCK performs direct screen accesses in the interest of speed, it will create video "snow" on an old-style IBM CGA display. A strategy for eliminating this interference can be found on pages 79-80 of Ray Duncan's excellent book, *Advanced MS-DOS* (reviewed in *TURBO TECHNIX*, March/April, 1988). Also, CLOCK does not check to see if a copy of itself is already resident and operating. Such a check can be performed in several ways. Most involve inspection of the code at some offset from the referent of interrupt 1CH's vector for a *signature* consisting of the program's name or some other unique string of bytes like SIRESCLOCK.

A program can determine if it is already installed by inspecting memory at some offset from the referent of interrupt 1CH's vector.

This simple version of CLOCK works well; take the time to improve and enhance it to your own satisfaction. My own commercial product, the Sires Alarm Clock, is a greatly enhanced version of CLOCK. Turbo C handles all of the difficult work—the rest is up to your imagination. ■

Ron Sires is president of Sires Software, a database consulting and software development firm in Berkeley, California. He may be reached at Sires Software, 2925 M.L. King, Jr. Way, Berkeley, California 94703.

Listings may be downloaded from CompuServe as CLOCK.ARC.

```
    }
    return;
}

void interrupt tickintr(void)
{
    int     rawhour, hour, minute;
    int     mins_aft_mid;
    static  newhour=TRUE;   /* Should beep() be called
                               when minute == 0?  */

    /* Use 18.2065 ticks/sec                              */
    /* 18.2065 ticks/sec * 60 secs/min * 100 == 109239L */
    mins_aft_mid = (biostime(0, 0L) * 100L) / 109239L;
    rawhour = mins_aft_mid / 60;
    minute = mins_aft_mid % 60;
    if (minute % 10 + '0' != scr[0][74].s_char  /* Does the time  */
     || minute / 10 + '0' != scr[0][73].s_char) /* need to be put */
                                               /* on the screen? */
    {
        hour = (rawhour > 12 ? (rawhour - 12) :
               (rawhour == 0 ? 12 : rawhour));
        disptime(hour, minute, rawhour);
        if (minute == 0)
        {
            if (newhour)
            {
                beep();
                newhour = FALSE;
            }
        }
        else
            newhour = TRUE;
    }
    (*oldtick)();
    return;
}

int color_adpt(void)
/* Return 0 if monochrome adapter, 1 if color adapter */
{
    return ((biosequip() & 0x0030) != 0x0030);
}

main()
{
    puts(COPYRIGHT);

    scr = MK_FP((color_adpt() ? 0xB800 : 0xB000), 0x0000);
    attr = ((scr[0][0].s_attr >> 4) + (scr[0][0].s_attr << 4)) & 0x77;

    /* Clock Installation  */
    oldtick = getvect(0x1C);    /* Save original ISR in oldtick.    */
    disable();                  /* Disable interrupts.              */
    setvect(0x1C, tickintr);    /* Replace ISR with tickintr().     */
    enable();                   /* Allow interrupts again.          */

    keep(0, 0x0130);       /* Terminate but stay resident with exit  */
}                          /* status 0, reserving 130H paragraphs.   */
```

TURBO PROLOG 2.0: INTELLIGENT EVOLUTION

Turbo Prolog spreads its wings as the second generation is born.

Michael Floyd

One of the most exciting aspects of working with the *Turbo* languages is watching their evolutionary process. Like the caterpiller's transformation into a butterfly, the Turbo languages periodically undergo a metamorphic process to emerge as a new creature.

Turbo Prolog has undergone its first transformation, and has emerged wearing the colors of 2.0. Some of the things you'll see in the new version of Turbo Prolog include the ability to create and manipulate multiple databases, color graphics through the Borland Graphics Interface (BGI), new debugging capabilities, conditional compilation, extensions to the language, a full-featured development environment, and a new and improved manual.

If you're a Turbo Pascal, Turbo C, or Turbo Basic programmer, you may be quite surprised at some of Turbo Prolog 2.0's new features. If you're already a Turbo Prolog programmer, you'll want to watch closely as this tale of evolution unfolds.

THE DATABASE CONCEPT

By far, the most significant change in Turbo Prolog 2.0 is its handling of the database—or should I now say, databases. You'll notice first that there are two types of databases—internal and external. An *internal* database corresponds to the dynamic database in earlier versions of Turbo Prolog. However, you can now have more than one internal database. The following statements, for example, declare two internal databases:

```
DATABASE - db1
   pred1
   pred2
   ...
   predN

DATABASE -db2
   predA
   predB
   ...
   predZ
```

These statements declare the **db1** database with the database predicates **pred1**, **pred2**, and so on; and the **db2** database with the database predicates **predA**, **predB**, and so on to **predZ**. (Database predicates behave just as they do in earlier versions.)

In Turbo Prolog 2.0, built-in predicates such as **asserta**, **assertz**, **consult**, and **save** have been extended to address specific databases. For example, to assert **pred2("fact")** into the the **db1** database, make the following statement:

```
asserta(pred2("fact"),db1)
```

Another important feature to note is that Turbo Prolog 2.0 supports local and global databases. Our discussion up until now has referred to *local* databases, which are databases declared locally in a program or module. In addition to local databases, you can declare *global* databases that are shared between modules. In a global database declaration, the keyword **global** must preface the **database** keyword.

EXTERNAL DATABASES

External databases extend the capabilities of the internal database in a number of ways. First, external databases can be placed in conventional RAM (memory below 640K), in extended memory (EMS), or in a disk file. In addition, external databases can load and store data in binary form. Finally, the use of B+ trees in external databases allows more efficient handling of data than is possible in internal databases (because of an internal database's sequential nature).

An external database has two parts: data items, which are Turbo Prolog terms stored in chains; and an index value that corresponds to a B+ tree. *Chains* are a way of grouping like data (terms) into a structure that can be referenced. There is no

continued on page 86

Bradley Ream


INIT+MYPROLOG+MYASM
SYS,MIXPROG,,PROLOG
⟨name(*)list⟩
⟨flow(*)spec⟩
(._or....if_.)
expression
NEC MULTISYNC
CONT
BRIGHT
IBM
datadesk

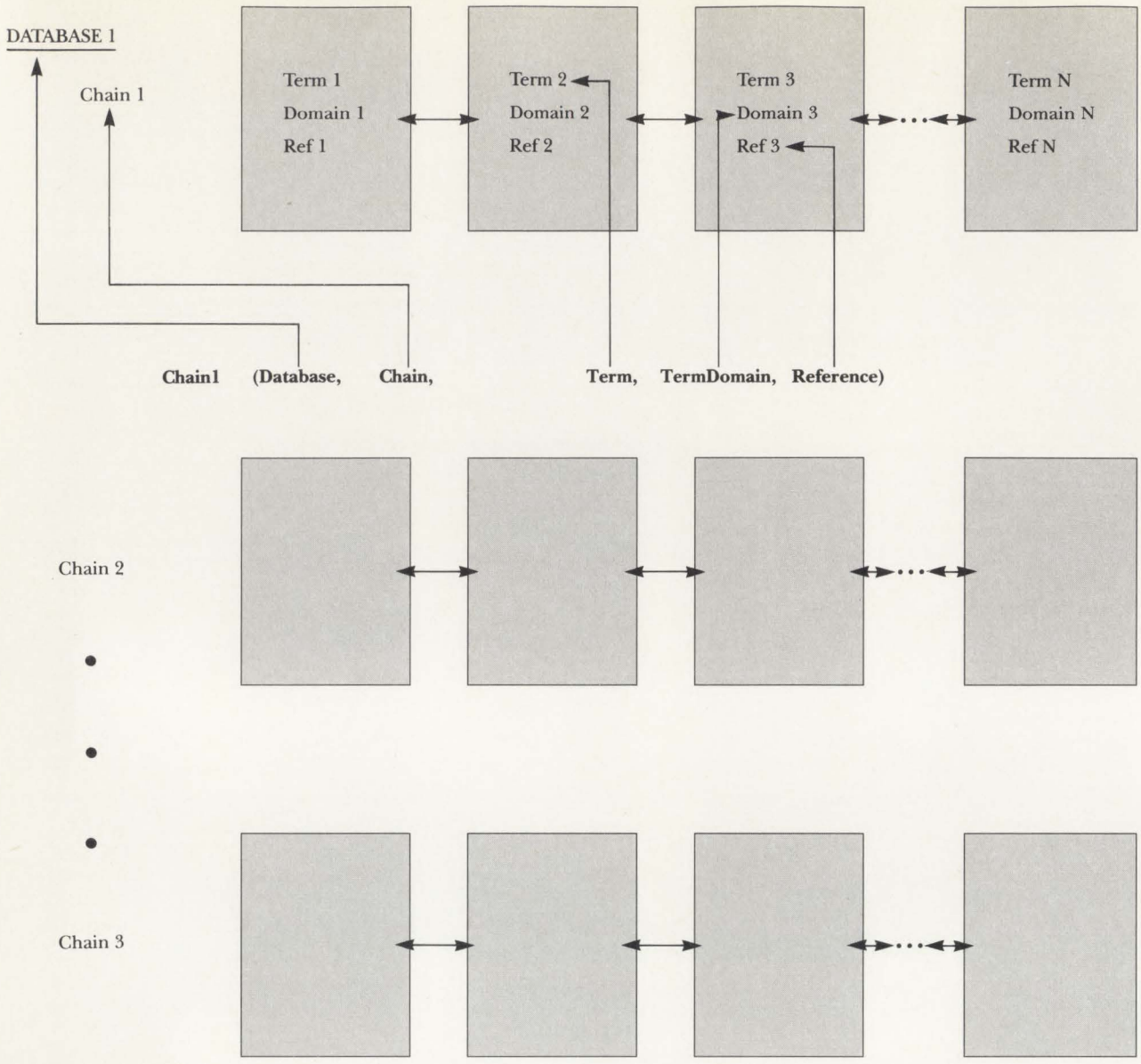


Figure 1. Structure of chains in the external database.

INTELLIGENT EVOLUTION

continued from page 84

practical limit on the number of terms that a chain can contain. In addition, there is no limit to the number of chains that can be stored in an external database. Figure 1 shows the structure of a chain.

Chains *are a way of grouping like data (terms) into a structure that can be referenced.*

A chained item consists of the name of the database that contains the item, the name of the chain, the domain that the chain belongs to, the actual term in the chain, and a special reference value that allows quick lookups within the chain.

As an example, imagine a case where you have a database that tracks the PCs in a company. One chain could correspond to the

individuals in the company, and another chain could contain specific information about individual PCs. By linking these chains together through their reference values, you can easily add a relational capability to the databases.

The second part of an external database is a reference to a B+ tree. In Turbo Prolog, a B+ tree is a data structure that is contained in an external database. Each entry in the B+ tree consists of a key string and a database reference number. When creating a new database entry, the programmer defines a *key string* that can be referenced during lookups. The *database reference* number is created by Turbo Prolog as each entry is created, and can also be referenced directly. When searching for a record, the programmer references the key string to be searched on, and Turbo Prolog returns the associated reference number. This reference number is then used to retrieve the actual record from the database.

BGI

If you haven't seen the Borland Graphics Interface (BGI) in Turbo Pascal 4.0 or Turbo C 1.5, you're in for a real treat. The BGI is a library of graphics routines that does everything from drawing specialized character fonts to detecting the type of graphics card installed in your PC. The library includes high-level routines to draw lines, circles, ellipses, arcs, rectangles, and polygons. The library also contains a number of routines for drawing two- and three-dimensional bars, and pie slices for creating charts. In addition, the BGI provides a number of patterns that can be used to fill any object. (For a complete discussion of the Borland Graphics Interface, see "Meet the BGI," elsewhere in this issue.)

The BGI's sixty or so graphics routines are accessed through built-in predicates. For instance, drawing a circle is a simple matter of specifying the **X** and **Y** coordinates and the radius of the circle in the following call:

```
circle(X,Y,R)
```

The BGI's coordinate system is similar to the system used in the turtle graphics of Turbo Prolog 1.x, although the scaling has changed. The upper left corner of the screen is designated as (0,0). The X (row) and Y (column) values increment from that point according to the screen mode the system is in. For instance, on a CGA system in low-resolution mode (320 × 200, four colors), the bottom right corner is designated as (319,199). Figure 2 shows the coordinate system used for a CGA in low-resolution mode.

One interesting new feature is that predicates can now have multiple arities. The programmer simply makes a declaration for each arity that will be supported by a program.

The BGI also supports the notion of viewports. As the name implies, a *viewport* is a window to a (possibly) larger graphics image. Viewports use a clipping system so that portions of the image that are not in the current viewport (or window) are not visible (i.e., clipped).

The BGI routines allow you to draw either bit-mapped or stroked fonts. *Bit-mapped* fonts are generated as an 8 × 8 matrix of pixels. Bit-mapped fonts are quicker to draw, since they are drawn pixel by pixel. To create larger bit-mapped fonts, however, the matrix must be multiplied by a scaling factor—this results in poorer resolution since the font is, in effect, magnified.

A *stroked* font, on the other hand, is defined by a set of vectors that describes each character. Since these vectors must be interpreted, they are slower to draw. On the positive side, stroked fonts retain their resolution for larger characters.

The best news for Turbo Prolog programmers is the BGI's drivers. These drivers support the Hercules, CGA, MCGA, EGA, and VGA cards, as well as the IBM 3270 and AT&T 400-line graphics cards. No longer does the programmer have to resort to C or assembly language for such support.

LANGUAGE EXTENSIONS

Borland has the reputation of listening to its customers and implementing their suggestions in future versions of products. Turbo Prolog 2.0 is no exception. Many enhancements to the language resulted directly from user suggestions. Some of these enhancements are changes to existing built-in predicates, while others are new predicates and compiler directives. The list of enhancements is quite long, so I'll just highlight some of the major points.

One interesting new feature is that predicates can now have multiple arities. The programmer simply makes a declaration for each arity that will be supported by the program. For example, consider the following program fragment:

continued on page 88

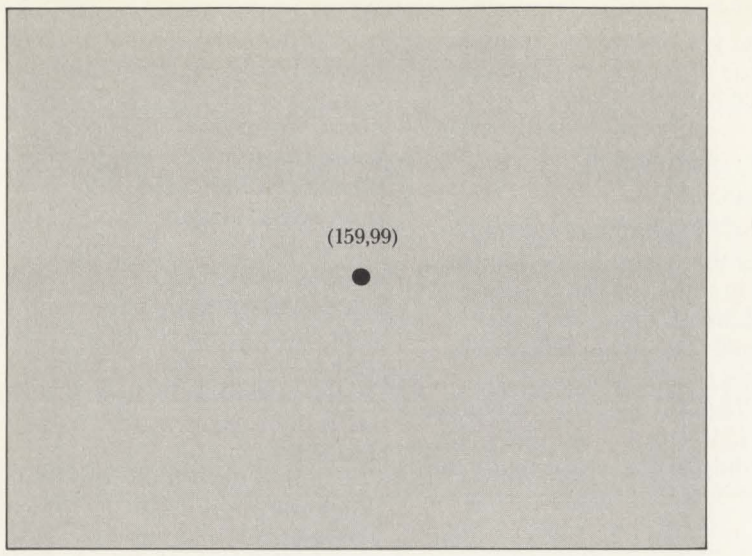


Figure 2. Coordinate system for a CGA system.

INTELLIGENT EVOLUTION

continued from page 87

```
predicates
   run
   run(integer)

clauses
   run:-
   makewindow(1,2,3,"",0,0,25,80),
      run(0).
   run(X):-
      X <= 100,
      Y = X+1,
      write(Y),
      run(Y).
```

Notice that the first **run** clause has an arity of 0 (referred to as run/0) while the second **run** clause has an arity of 1 (run/1). Also notice that it is possible to call one clause from the other.

Another enhancement is conditional compilation, which compiles a given section of a program only if a condition is satisfied. Conditional compilation is particularly useful for generating different versions of the same program. The syntax takes the form:

```
ifdef ConstantID
   ...
   elsedef
   ...
   endef
```

For instance, we can write a routine to set the graphics mode, based on the type of graphics card installed in the PC:

```
predicates
...

constants
   egaCard = 1

ifdef egaCard
   goal
      graphics(5,1,1),
      write("System in EGA Mode),
elsedef
   goal
      graphics(1,1,1),
      write("System in CGA Mode"),
enddef
```

This program fragment demonstrates another new feature—**constants**. As in other languages, once the value of the constant has been set, it remains the same throughout the program.

Turbo Prolog 2.0 has a number of new window-handling features, including the ability to change window colors during program execution. The user of a program can now modify window color and size at runtime.

Other features include built-in error handling and reporting, extended text mode handling, character manipulation similar to already existing built-in string manipulation, and more.

THE DEVELOPMENT ENVIRONMENT

Turbo Prolog's windowed development environment, in a sense, has served as the prototype for Turbo Basic, Turbo C, and Turbo Pascal 4.0, and is now a Borland standard. Many of the changes in the Turbo Prolog 2.0 development environment were made to bring it more into line with the other Borland compiler environments. Now, if you're familiar with one Borland development environment, you are familiar with them all.

Many of the changes in the Turbo Prolog 2.0 development environment were made to bring it more into line with the other Borland compiler environments.

Notice, for example, that most of the hot keys that perform actions such as running a program from the editor (Alt-R), or exiting from the development environment (Alt-X), have been standardized. I typically shift from Turbo C to Turbo Prolog (especially in a joint development) and couldn't survive without this kind of standardization. If you prefer other assignments, you can redefine the hot keys easily from within the environment.

When starting up Turbo Prolog 2.0, you're immediately placed in the editor, ready to program. "But," you say, "I wanted to load in a program first." No problem—Turbo Prolog (along with all of the compilers) now takes command-line arguments. For instance, to load MYPROG.PRO into the environment upon start-up, simply issue the following command at the DOS prompt:

```
PROLOG -E MYPROG
```

As you go through the pulldown menus, notice that the **F**iles pulldown menu has been moved to the far left. If you've worked with Turbo Pascal 4.0, Turbo C, or Turbo Basic, this standard menu will be familiar to you. For instance, when browsing for files using the **L**oad option, you can now easily move from one directory to another by highlighting and selecting the appropriate directory name.

Next in the pulldown menu system are the **E**dit and **R**un buttons. As always, the **R**un option compiles your program to memory and then executes that program. This provides the power of a compiler, combined with the instant feedback normally seen only in interpreters.

New to Turbo Prolog is the ability to set up linker options in the environment. Particularly useful is the ability to include libraries in the link command. Thus, modules written in other languages that require their own runtime library support can now be linked in directly from the development environment.

In addition, you have complete control over other options, including compiler directives from within the development environment. The programmer can set up options without hard coding them into the program. Some of the new compiler directives allow better control over heap, stack, code, and trail sizes, overflow checking, and the generation of error and warning messages.

DOCUMENTATION

Unlike many compiler manuals, the *Turbo Prolog Owner's Handbook* includes a complete language tutorial. This tutorial is a must for a language like Turbo Prolog, which is a new area for even seasoned programmers. While the original *Turbo Prolog Owner's Handbook* was just under 250 pages, the tutorial section alone in 2.0 is now around 300 pages. The tutorial takes the reader through the Prolog language, step by step, and includes programming exercises to test the reader's understanding. Solutions to the exercises are provided.

Modules written in other languages that require their own runtime library support can now be linked in directly from the development environment.

The manual also discusses other topics such as the BGI, database programming techniques, system-level programming, and so forth. In general, the reader will find excellent coverage of each topic, plus many more example programs.

The reference section of the manual provides a quick lookup to all of Turbo Prolog's built-in predicates. This section has been expanded to include full descriptions of the predicates and includes example programs for each. These example programs show how a particular predicate works, and give the reader a feel for the predicate's application.

THE SECOND GENERATION IS HERE

Prolog has long been known as a prototyping language because applications can be quickly modeled in the language. Since Prolog programs provide a *logical* description of a problem, the prototype served as a program specification in the development process. After setting up working models in Turbo Prolog, programmers usually developed their projects in a language such as C. But with the extensions added by the Turbo flavor, many developers find that once the prototype is done, so is the project.

Turbo Prolog 2.0 adds many features (such as conditional compilation) that previously were seen only in languages such as Pascal and C. The development environment provides total control over compiler directives so that various versions of a program can be customized. Turbo Prolog 2.0 extends the Prolog database concept significantly with the implementation of chained terms to add true relational capability, and the addition of B+ trees to allow sorting and retrieval of data at lightning speeds. In addition, with tools such as the BGI at your command, adding sophisticated graphics is a snap.

The evolution of Turbo Prolog puts the power of Artificial Intelligence in your hands. The extensions to Turbo Prolog add power to traditional applications as well. Because of its declarative nature, Turbo Prolog allows you to write many applications in only one-tenth the lines of code usually required with traditional languages. This difference translates into savings in development time, and ultimately reduces cost. Now, the only question is whether you want to become a conscious part of Turbo Prolog's evolution ... and the answer should be easy. ■

WHAT'S IN A LIST?

In list processing, a little bit of recursion goes a long way.

Keith Weiskamp

SQUARE ONE

One of the powerful features of Turbo Prolog—a feature that gives Turbo Prolog the edge over its more procedural cousins, such as Turbo C or Turbo Pascal—is its ability to easily process lists. When you're programming in Turbo Prolog, you don't have to worry about all of the traditional programming headaches like pointers, memory allocation, and the linked-list data structures from computer science 101. Of course, the trade-off in Turbo Prolog is that you have to master recursion.

In this article, we'll explore the fundamentals of list processing and recursion in Turbo Prolog. We'll also build some useful tools to show you how to work with lists in Turbo Prolog.

STARTING WITH THE BASICS

Most programming languages provide some type of data structure, such as an array, for list representation. The limitation of such built-in data structures is that they don't allow you to construct *dynamic* lists—lists that can grow and shrink during the execution of a program. Fortunately, Turbo Prolog provides real dynamic list processing capabilities.

A list in Turbo Prolog is simply a sequence of zero or more elements. Because a list is a dynamic structure, you don't have to specify its size. Lists are easy to represent in Turbo Prolog. (They are so easy, in fact, that you might feel a little guilty about using them!) A list is constructed by enclosing elements between the brackets [ ] as shown:

```
[1,2,3,4]
[one,two,three,four]
["one","two","three","four"]
[]
```

A comma separates each element in the list. If you forget a comma, the compiler gives you a friendly reminder. Note that we represent an empty list by using just the brackets [].

Each element or *member* in a list can be defined as either a standard domain type, such as an integer or a character, or a user-defined domain type. Keep in mind, however, that Turbo Prolog places one important restriction on list elements: all of the elements in a list must be of the same domain type. Therefore, the following lists aren't valid:

```
["one","two",3,4]
[5.7,10.9,20,'c']
```

We can handle lists containing elements of different types through the use of complex objects. That subject, however, is beyond the scope of this article.

Now that you know what a list looks like, you're probably wondering how to declare one. Like all user-defined domain types, lists must be declared in the **domains** section of a program. To declare a list domain, use the * symbol to indicate that the specified domain name represents a list. For example, the following declaration creates a domain called **strlist**, which is defined as a list of strings:

```
domains
   strlist = string*
```

This domain can then be used in a predicate declaration such as:

```
predicates
   search(strlist,string,integer)
```

Given this declaration, the predicate search accepts arguments like:

```
search(["door","window","wall"],
       "door",Pos)
search([],"wall",Pos)
```

PROCESSING A LIST

When processing a list, one of the first things to do is access a single element in the list. In order to do this, Turbo Prolog divides a list into two components, a head and a tail. The *head* is the first element in the

list; the *tail* is the list that is left when the first element (the head) is removed. The symbol | is used to separate the head and the tail. As an example, let's attempt to match the list **[1,2,3,4]** with the terms:

```
[H|T]
```

We get the following result:

```
H = 1
T= [2,3,4]
```

The task of dividing a list into a head and a tail is handled by Turbo Prolog's built-in unification algorithm. For example, let's define a predicate and clause **head_tail** as:

```
domains
   strlist = string*

predicates
   head_tail(strlst)

clauses
   head_tail([Head|Tail).
```

Now, we call **head_tail** with the goal:

```
head_tail(["red","green","blue"]).
```

Upon execution, the list **["red","green","blue"]** is passed to the **head_tail** clause. Because the variables **Head** and **Tail** are *uninstantiated* (i.e, have no value), Turbo Prolog *binds* (assigns) the first element of the list to **Head**, and assigns the rest of the list to **Tail**. In this case, the string **red** is bound to the term **Head**, and the list **["green", "blue"]** is bound to the term **Tail**. Here's a quick program that applies the head/tail relationship:

```
domains
  slist = symbol*

predicates
  div_list(slist)

clauses
 div_list([Head|Tail]):-
  write("\nThe first element is: ",
        Head),
  write("The rest of the list is: ",
        Tail).
```

div_list simply displays the two components of a list, the head and the tail. Let's give the goal:

```
div_list([this,is,a,list]).
```

In this case, Turbo Prolog responds with:

```
The first element is: this
The rest of the list is:
        ["is","a","list"]
```

Lists behave differently depending upon whether the head and tail are bound or free, and how the list is passed. For instance, our last example removes the head from a list. We add an element to a list by instantiating the head of the list. Consider the following clause, which writes a list to the screen:

```
  domains
     intlist = integer*

predicates
  add_elem(integer,intlist,intlist)

clauses
    add_elem(Element,List1,List2):-
       Lists2 = [Element|List1].
```

Now, let's give the goal:

```
add_elem(1,[2,3,4],X).
```

Turbo Prolog responds with:

```
X = [1,2,3,4]
```

When specifying a head/tail relationship, the rule is: If the head is bound to a value, that value is added to the front of the list; if the head of the list is a free variable, that variable is bound to the first element in the list.

A LITTLE BIT OF RECURSION

In Turbo Prolog, recursion is used to perform most of the useful list processing tasks. After all, most list operations—such as finding a member in a list, or counting the number of elements in a list—require some method of stepping through a list. The typical method in a procedural language like C is a loop. In Turbo Prolog, we rely upon recursion or backtracking to handle looping.

To show how recursion is used to access a list, let's modify our previous clause, **div_list**, so that we can step through a complete list. Here is the new version:

```
domains
  ilist = integer*

predicates
  div_list(ilist)

clauses
 div_list([]).
 div_list([Head|Tail]):-
  write("\nThe first element is: "
        ,Head),nl,
  write("The rest of the list is: ",
       Tail),
  div_list(Tail).
```

Note that **div_list** now has two clauses. The first clause, which is known as an *anchor* clause, simply tests for the empty list. This clause terminates the recursion when we get to the end of the list. The second clause, which performs the actual processing, separates the head from the tail of the list and then displays both the head and the tail on the screen. Finally, **div_list** calls itself (the recursive call) with the tail of the list. Therefore, each time **div_list** is called, the list is reduced by one element—the current head. Each time the recursive call is made, the first **div_list** clause tests to see if the list is empty ([]). If the list is empty, this clause succeeds and we are done. For example, let's call **div_list** with:

```
div_list([1,2,3]).
```

The goal produces the following output:

```
The first element is: 1
The rest of the list is: [2,3]
The first element is: 2
The rest of the list is: [3]
The first element is: 3
The rest of the list is: []
```

BUILDING LIST TOOLS

The basic recursive technique illustrated in the previous example can easily be applied to construct the fundamental list processing operations such as appending elements, searching for elements, and deleting elements. To show you the power of Turbo Prolog's

continued on page 92

LISTING 1: EXAMPLES.PRO

```
/* get an element at specified position */

domains
   ilist = integer*

predicates

   index(ilist, integer, integer)

clauses

   index([Head |_], 1, Head).

   index([_|Tail], Pos, Elem) :-
      Pos > 1,
      New_pos = Pos - 1,
      index(Tail, New_pos, Elem).


/* The Append Tool */

domains

    clist = char*
    ilist = integer*
    rlist = real*
    stlist = string*
    slist = symbol*

predicates

    append( clist, clist, clist ) /* Append 2 character lists. */
    append( ilist, ilist, ilist )  /* Append 2 integer lists. */
    append( rlist, rlist, rlist )  /* Append 2 real lists. */
    append( stlist, stlist, stlist ) /* Append 2 string lists. */
    append( slist, slist, slist )    /* Append 2 symbol lists. */

clauses

    append([ ], List, List ).

    append([ Head | List1 ], List2, [ Head | Rest ] ) :-
        append( List1, List2, Rest ).



/* The Reverse Tool */

include "append.pro"

predicates

    reverse( clist, clist )   /* Reverse the character list. */
    reverse( ilist, ilist )   /* Reverse the integer list. */
    reverse( rlist, rlist )   /* Reverse the real list.*/
    reverse( stlist, stlist )   /* Reverse the string list. */
    reverse( slist, slist )   /* Reverse the symbol list. */

clauses

    reverse( [ ], [ ] ).

    reverse( [ Head | Tail ], Result ) :-
        reverse( Tail, Temp ),
        append( Temp, [ Head ], Result ).
```

WHAT'S IN A LIST?

continued from page 91

list processing capabilities, let's construct some list processing tools.

The first tool, called **index**, is shown in Listing 1. This predicate determines which element is at a specified position in a list, and in one respect, illustrates how a list in Turbo Prolog can be accessed like an array. **index** takes three arguments:

```
index(List,Pos,Member)
```

The first argument, **List**, supplies the list to be searched. In our example, this argument must be a list of integers; you can also modify **List** to support lists of other domain types, such as strings and characters. The argument **Pos** specifies the position of the element to be accessed, and **Member** is used to return the element. For example, the following goal binds the variable **X** to the list element **76**:

```
index([20,45,76,89],3,X).
```

This produces the same effect as does the following statement, which is written in Turbo Pascal:

```
X := List[3];
```

Of course, in order to access the list in Turbo Prolog, we must use recursion. Note that the predicate **index** contains two clauses. The first clause terminates the recursion, and the second clause decrements the list index and calls itself recursively until the desired index position is reached.

The next tool that we'll construct is **append**, which is also shown in Listing 1. This predicate joins two lists together and produces a new list. The general format for **append** is:

```
append(List1,List2,List3)
```

Here, the arguments **List1** and **List2** provide two lists of any standard domain type, which can be joined to create a new list that is bound to the variable argument

List3. For example, to combine the list **[a,b,c]** with **[d,e,f]**, we give the goal:

```
append([a,b,c], [d,e,f], L)
```

In this case, Turbo Prolog responds with:

```
L = [a,b,c,d,e,f]
```

The **append** predicate first copies one element at a time from **List1** to **List3**. This step is performed by the clause:

```
append([Head|List1],List2,
                [Head|Rest]):-
   append(List1,List2,Rest).
```

Each time **append** calls itself, the current first element (**Head**) of **List1** becomes the first element of **List3**. This process continues until **List1** is empty. The following anchor clause terminates the recursion:

```
append([],List,List ).
```

This anchor clause also makes sure that **List2** is joined with **List3**. How is this done? Well, remember that **List3** is represented as the two components:

```
[Head|Rest]
```

Therefore, when the recursion terminates, the term **Rest** is bound to the second list.

The last tool provided in Listing 1 illustrates how the **append** predicate can be used to alter the order of a list. In this case, the predicate **reverse** is used to reverse a list. For example, let's give the goal:

```
reverse([1,2,3,4],L).
```

This goal produces the result:

```
L = [4,2,3,1]
```

It's easy to reverse a list by using recursion. The technique involves stepping through the list by allowing the second **reverse** clause to call itself, as shown below:

```
reverse([Head|Tail],Result):-
   reverse(Tail,Temp),
   append(Temp,[Head],Result).
```

When the list becomes empty, the recursion is terminated by the clause:

```
reverse([],[]).
```

Once the list becomes empty, the new list is created by appending elements in the reverse order from which they were removed. To help you see how this is done, let's trace through the **reverse** predicate using the list **[1,2,3,4]** as the first argument. Each time **reverse** calls itself, the list is separated as shown:

```
Call 1:
        Head = 1
        Tail = [2,3,4]
Call 2:
        Head = 2
        Tail = [3,4]
Call 3:
        Head = 3
        Tail = [4]
Call 4:
        Head = 4
        Tail = []
```

At this point, the list is empty and the first clause terminates the recursion. Therefore, the next step consists of the following calls to **append** to construct a new list:

```
append([],4,Result)
append([4],3,Result)
append([4,3],2,Result)
append([4,3,2],1,Result)
```

Although the **append** predicates are listed together, they are actually called each time the recursion unwinds one level. Remember that **reverse** called itself recursively four times in order to get to the end of the list. Therefore, when the recursion stops, it must unwind one level at a time.

END OF THE TOUR

This concludes our quick tour of Turbo Prolog's list processing capabilities. We started with the fundamentals of list construction techniques, and then wrote a few predicates to illustrate how lists can be processed with Turbo Prolog. Along the way, we've investigated most of the basic techniques for writing recursive clauses. The key to writing useful list processing predicates in Turbo Prolog lies in using recursive programming techniques. ■

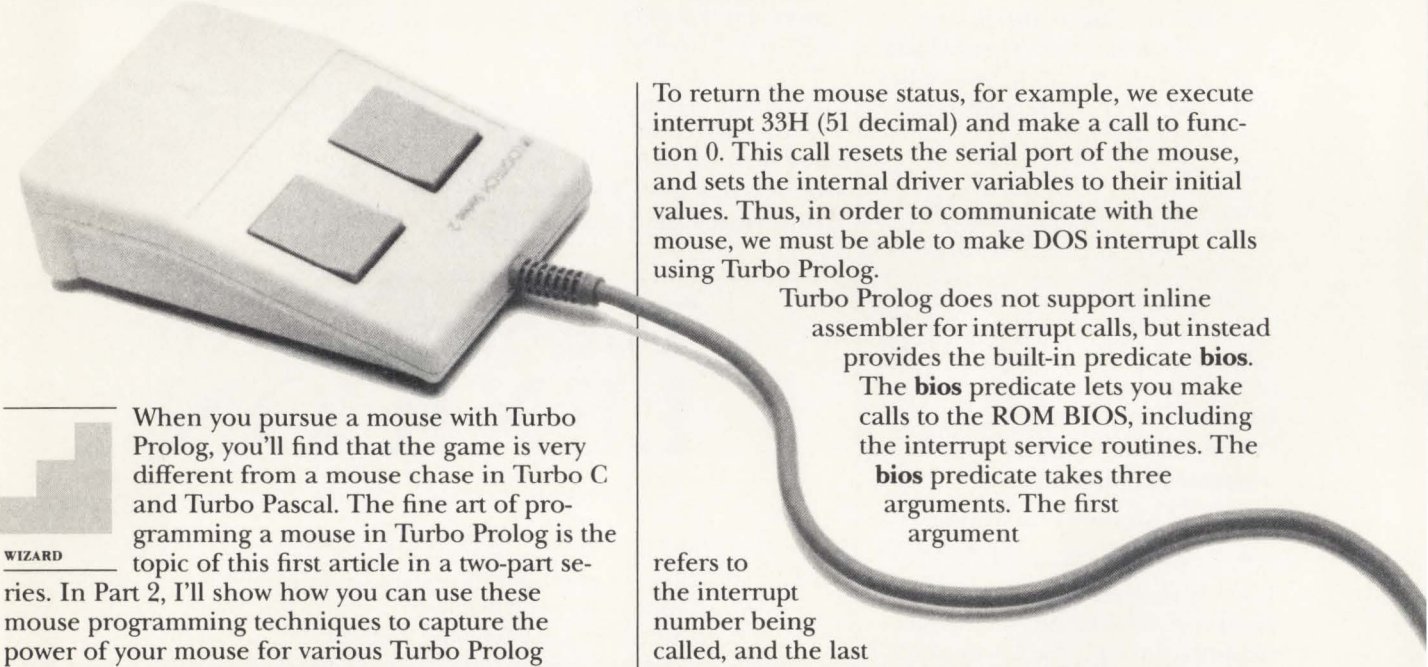
Keith Weiskamp is a cofounder of PC AI *magazine, and is coauthor of* Artificial Intelligence Programming with Turbo Prolog.

Listings may be downloaded from CompuServe as LISTP.ARC.

PLAYING CAT AND MOUSE IN TURBO PROLOG

Don't grab your mouse by the recursive tail just yet! You may need to backtrack.

Safaa H. Hashim



WIZARD

When you pursue a mouse with Turbo Prolog, you'll find that the game is very different from a mouse chase in Turbo C and Turbo Pascal. The fine art of programming a mouse in Turbo Prolog is the topic of this first article in a two-part series. In Part 2, I'll show how you can use these mouse programming techniques to capture the power of your mouse for various Turbo Prolog applications.

We won't explore the basics of mouse programming in this discussion, because they are thoroughly covered in Kent Porter's article "Mouse Mysteries," elsewhere in this issue. I highly recommend that you read "Mouse Mysteries" before you jump into Turbo Prolog's cat-and-mouse techniques. Particularly important are the sections on mouse functions and how to communicate with the mouse.

This article deals specifically with the Microsoft Mouse. If you use a different mouse, don't worry—most mouse packages contain a driver that emulates the Microsoft Mouse driver.

BUILDING THE BETTER MOUSE TRAP

As discussed in "Mouse Mysteries," the driver for the Microsoft Mouse uses a software interrupt for communicating between the mouse and the computer. To return the mouse status, for example, we execute interrupt 33H (51 decimal) and make a call to function 0. This call resets the serial port of the mouse, and sets the internal driver variables to their initial values. Thus, in order to communicate with the mouse, we must be able to make DOS interrupt calls using Turbo Prolog.

Turbo Prolog does not support inline assembler for interrupt calls, but instead provides the built-in predicate **bios**. The **bios** predicate lets you make calls to the ROM BIOS, including the interrupt service routines. The **bios** predicate takes three arguments. The first argument refers to the interrupt number being called, and the last two arguments refer to compound objects corresponding to the registers before and after the **bios** call. The general form of the **bios** predicate is:

```
bios(InterruptNo,RegistersIn,
     RegistersOut)
```

Internally, Turbo Prolog defines a special domain, called the **reg** domain, for **RegistersIn** and **RegistersOut**. The **reg** domain takes the form:

```
reg(AX,BX,CX,DX,SI,DI,DS,ES)
```

The arguments of the **reg** domain represent eight 16-bit 8088 registers. We use these registers to pass

parameters to an interrupt service (**RegistersIn**), as well as to return the result of the execution of that service (**RegistersOut**).

To see how the **bios** predicate works in relation to the mouse, let's consider an example. Listing 1 defines two clauses: One clause checks to see if the mouse is installed; the other clause initializes the mouse. To see how Listing 1 reacts to different situations, enter the following goal before and after installing your mouse driver:

```
Goal: msm_chk_init(STATUS)
```

Note in Listing 1 that the first call in **msm_chk_init** is a **bios** call to interrupt 33H (the **$** indicates a hex value). We must specify the function call (in this case, 0) in the AX register, so the first argument of **reg** (for the input registers) is set to **0**. This call does not use the other registers. However, since the domain has an input flow pattern, Turbo Prolog requires that the entire domain be instantiated. Therefore, the other seven register slots are padded with **0**s.

The status value is returned back in the AX register, so we specify the variable **AX** in the first slot of the output registers. Again, we're not concerned with the other registers, so we've used the anonymous variable (_).

After making the **bios** call, **msm_chk_init** calls **report_init_status**, which reports whether or not the mouse is installed.

Using the same principles described in Listing 1 for initializing the mouse, we can write other clauses to call all of the driver functions.

Listing 2 documents

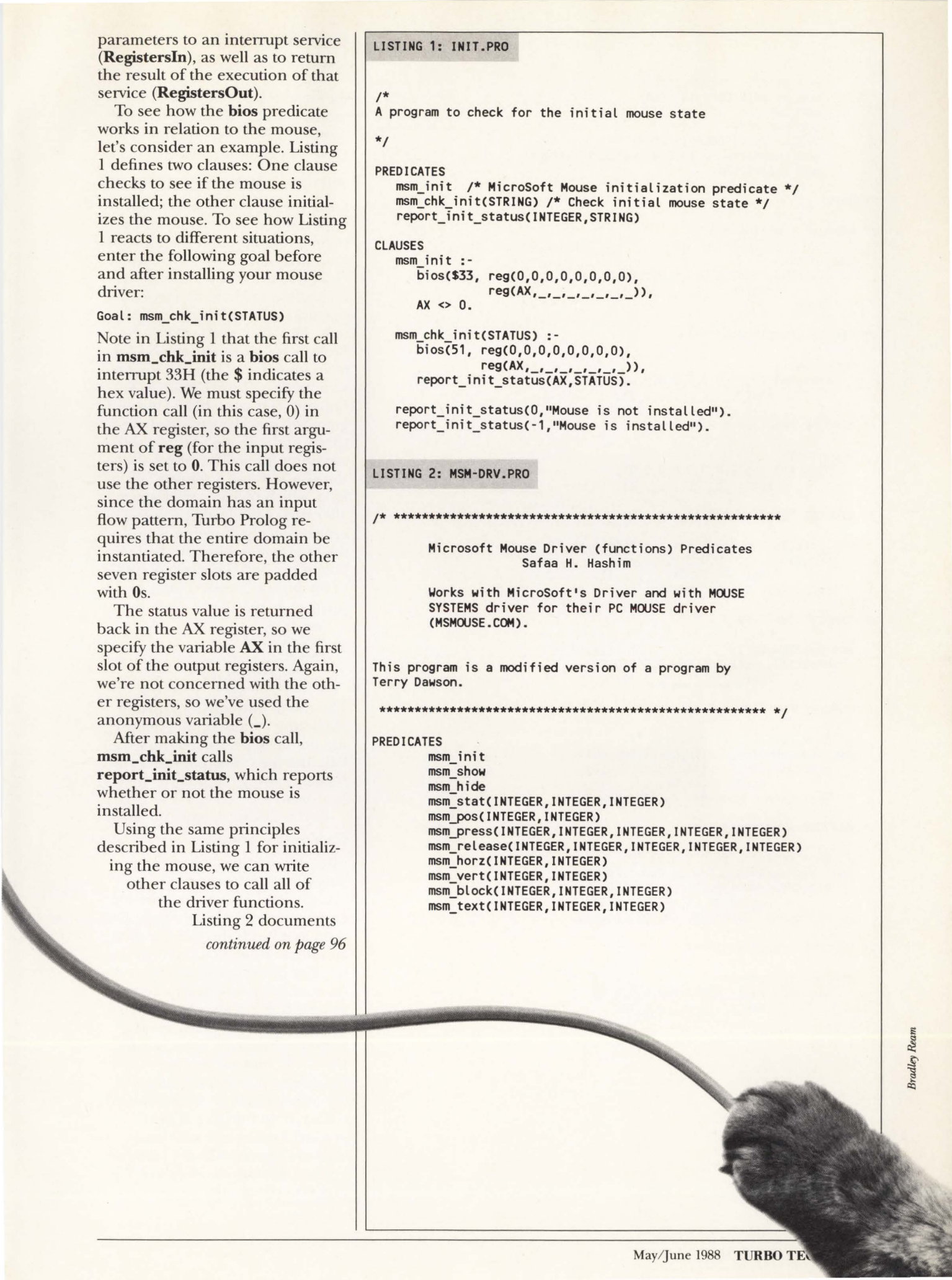
continued on page 96

LISTING 1: INIT.PRO

```
/*
A program to check for the initial mouse state

*/

PREDICATES
   msm_init  /* MicroSoft Mouse initialization predicate */
   msm_chk_init(STRING) /* Check initial mouse state */
   report_init_status(INTEGER,STRING)

CLAUSES
   msm_init :-
      bios($33, reg(0,0,0,0,0,0,0,0),
                reg(AX,_,_,_,_,_,_,_)),
      AX <> 0.

   msm_chk_init(STATUS) :-
      bios(51, reg(0,0,0,0,0,0,0,0),
               reg(AX,_,_,_,_,_,_,_)),
      report_init_status(AX,STATUS).

   report_init_status(0,"Mouse is not installed").
   report_init_status(-1,"Mouse is installed").
```

LISTING 2: MSM-DRV.PRO

```
/* *******************************************************

        Microsoft Mouse Driver (functions) Predicates
                    Safaa H. Hashim

        Works with MicroSoft's Driver and with MOUSE
        SYSTEMS driver for their PC MOUSE driver
        (MSMOUSE.COM).


This program is a modified version of a program by
Terry Dawson.

 ******************************************************* */

PREDICATES
        msm_init
        msm_show
        msm_hide
        msm_stat(INTEGER,INTEGER,INTEGER)
        msm_pos(INTEGER,INTEGER)
        msm_press(INTEGER,INTEGER,INTEGER,INTEGER,INTEGER)
        msm_release(INTEGER,INTEGER,INTEGER,INTEGER,INTEGER)
        msm_horz(INTEGER,INTEGER)
        msm_vert(INTEGER,INTEGER)
        msm_block(INTEGER,INTEGER,INTEGER)
        msm_text(INTEGER,INTEGER,INTEGER)
```



Bradley Ream

```
        msm_mickey(INTEGER,INTEGER)
        msm_user(INTEGER,INTEGER)
        msm_pen_on
        msm_pen_off
        msm_ratio(INTEGER,INTEGER)
        msm_cond(INTEGER,INTEGER,INTEGER,INTEGER)
        msm_ds(INTEGER)

CLAUSES

/* MSMOUSE FUNCTION # 0 */

   msm_init:-
      bios($33,reg(0,0,0,0,0,0,0,0),
               reg(AX,_,_,_,_,_,_,_)),
      AX<>0.

/* MSMOUSE FUNCTION # 1 */

   msm_show:-
      bios($33,reg(1,0,0,0,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 2 */

   msm_hide:-
      bios($33,reg(2,0,0,0,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 3 */

   msm_stat(Button,Row,Col):-   /* (o,o,o) */
      bios($33,reg(3,0,0,0,0,0,0,0),
               reg(_,BX,CX,DX,_,_,_,_)),
      Button=BX, Col=CX, Row=DX.

/* MSMOUSE FUNCTION # 4 */

   msm_pos(Row,Col):-           /* (i,i,i) */
      bios($33,reg(4,Col,Row,0,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 5 */

   /* Flow pattern: (i,o,o,o,o) */
   msm_press(Button,Status,Count,Row,Col):-
      bios($33,reg(5,Button,0,0,0,0,0,0),
               reg(AX,BX,CX,DX,_,_,_,_)),
      Status=AX, Count=BX, Col=CX, Row=DX.

/* MSMOUSE FUNCTION # 6 */

   /* Flow pattern: (i,o,o,o,o) */
   msm_release(Button,Status,Count,Row,Col):-
      bios($33,reg(6,Button,0,0,0,0,0,0),
               reg(AX,BX,CX,DX,_,_,_,_)),
      Status=AX, Count=BX, Col=CX, Row=DX.

/* MSMOUSE FUNCTION # 7 */

   msm_horz(Min,Max):-          /* (i,i) */
      bios($33,reg(7,0,Min,Max,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).
```

CAT AND MOUSE

continued from page 95

the various function calls to the mouse. Again, "Mouse Mysteries" provides a detailed explanation of each function call.

PROGRAMMING TECHNIQUES

Once the basic calls have been implemented, we're ready to use them in our applications. First, we must determine how to continuously poll the mouse. Our initial inclination might be to embed the function calls within a recursive loop. However, even with Turbo Prolog's *tail recursion optimization* techniques (see "The Tail Recursion Tiger," *TURBO TECHNIX*, January/February, 1988) your program will quickly run out of stack space, and will end abruptly with a runtime error. Therefore, the solution is to use a **repeat/fail** loop, which reclaims stack space after each pass. Listing 3 uses a **repeat/fail** combination to poll the mouse and to report the last button pressed. The loop occurs in the **button_status** clause:

```
button_status :-
   msm_init,
   msm_show,
   repeat,
      msm_stat(X,_,_),
      button(X,_).
```

The call to **button(X,_)** provides the failing condition. If a button has not been pressed, **button(X,_)** fails and the clause backtracks to **repeat**, starting the polling process over.

To run Listing 3, issue the goal:

```
Goal: button_status.
```

After the mouse is initialized (**msm_init**) and its cursor is displayed (**msm_show**), the program enters the **repeat** loop. The call to **msm_stat** binds the variable **X** to an integer value that refers to the number of the pressed button. If no button is pressed, the returned value is **0**. **X** is then passed on to **button**, which checks to see if a button has actually been pressed (**X <> 0**). **button** then uses **Bmeaning** to report the name of the pressed button. As mentioned earlier, if no button has been pressed, **button** fails and backtracks to **repeat**. When a button is pressed, **button** succeeds and the program terminates.

We can expand the techniques in Listing 3 to allow the continuous use of the mouse, even after we have pressed a button. This approach is handy in an application that uses the mouse for continuous interaction with the user. We do this by appending a **fail** to the end of the **button** clause:

```
button(X,Y) :-
   X <> 0,  /* When X=0 then no
               button is pressed */
   Bmeaning(X,Y),
   write("\n",Y,"\n"),
   fail.
```

This guarantees that **button** will always fail and that the program will backtrack to **repeat** (in **button_status**), starting the whole process over again. In this case, the only way to stop the program is to use the Ctrl-Break sequence.

Another useful Turbo Prolog predicate reports the cursor position at the click of a particular mouse button. POSITION.PRO (Listing 4) is an example of this "reporting" technique. To invoke POSITION.PRO, issue the goal:

```
Goal: report_pos
```

The system then enters text mode, which is the default mode; now you can move the cursor with the mouse. When you press the left button, the program writes the row and column of the cursor position. When you move the cursor to a new position and press the left button again, the program writes the new row and column values. This continues each time you press the left button. When you press the right button, the screen changes into graphics mode and displays the graphics cursor (an arrow). If you press the left button now, the X and Y coordinates of the cursor are represented as an integer between 0 and 31999 (the range of 0–31999 is the coordinate system used by Turbo Prolog in graphics mode).

Notice two predicates in Listing 4—**txt_pos** and **txt_pos1**. The clauses for these two predicates are:

```
txt_pos :-
   makewindow(1,7,0,"Text Mode",
              0,0,25,80),
   write("\nPress left button to
         locate cursor position"),
   repeat,
      msm_stat(Button,Row,Col),
      txt_pos1(Button,Row,Col).

txt_pos1(1,Row,Col):-
```

continued on page 98

```
/* MSMOUSE FUNCTION # 8 */

   msm_vert(Min,Max):-          /* (i,i) */
      bios($33,reg(8,0,Min,Max,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 9 */

   msm_block(Row,Col,Mask):-    /* (i,i,i) */
      bios($33,reg(9,Col,Row,Mask,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 10 */

   msm_text(Select,Screen,Cursor):-     /* (i,i,i) */
      bios($33,reg(10,Select,Screen,Cursor,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 11 */

   msm_mickey(Row,Col):-                /* (o,o) */
      bios($33,reg(11,0,0,0,0,0,0,0),
               reg(_,_,CX,DX,_,_,_,_)),
      Col=CX, Row=DX.

/* MSMOUSE FUNCTION # 12 */

   msm_user(Mask,Address):-             /* (i,i) */
      bios($33,reg(12,0,Mask,Address,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 13 */

   msm_pen_on:-
      bios($33,reg(13,0,0,0,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 14 */

   msm_pen_off:-
      bios($33,reg(14,0,0,0,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 15 */

   msm_ratio(Vert,Horz):-               /* (i,i) */
      bios($33,reg(15,0,Horz,Vert,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 16 */

   msm_cond(UY,UX,LY,LX):-              /* (i,i,i,i) */
      bios($33,reg(16,0,UX,UY,LX,LY,0,0),
               reg(_,_,_,_,_,_,_,_)).

/* MSMOUSE FUNCTION # 19 */

   msm_ds(MPS):-                        /* (i) */
      bios($33,reg(19,0,0,MPS,0,0,0,0),
               reg(_,_,_,_,_,_,_,_)).


/* *************** End of MSM-DRV.PRO ****************** */
```

LISTING 3: BUTTONS.PRO

```
/* ******************************************************* */
/* This is a testing program to report the pressed button. */
/* ******************************************************* */

include "msm-drv.pro"

PREDICATES
   button_status
      repeat
      button(INTEGER,STRING)
         Bmeaning(INTEGER,STRING)

CLAUSES
   button_status :-
      msm_init,
      msm_show,
      repeat,
         msm_stat(X,_,_),
         button(X,_).

   repeat.
   repeat :- repeat.

/* Table of Mouse Buttons combination for PC MOUSE (3 button mouse).

   For users of Microsoft Mouse only 0,1,2,3 combinations are
   applicable.

   Button Status is an integer referring to the pressed
   button combinations. The following combinations are
   recognized:

   0 initial status (no button is pressed)
   1 left ;              ....... (left)
     left + right (6)
   2 right;              ....... (right)
     right + middle (6)
   3 left + right        ....... (left + right)
   4 middle;             ....... (middle)
     middle + left;  (6)
     middle + right; (3)
   5 left + middle + right; .... (left + middle)
     middle + left (6)
   6 right + middle      ....... (right + middle)
   7 left + middle + right ..... (left + middle + right)
*/

   button(X,Y) :-
      X <> 0,  /* When X=0 then no button is pressed */
      Bmeaning(X,Y),
      write("\n",Y,"\n").

   Bmeaning(1,"left button is pressed").
   Bmeaning(2,"right button is pressed").
   Bmeaning(4,"middle button is pressed").

/* ******************** END OF BUTTONS.PRO **********************/
```

LISTING 4: POSITION.PRO

```
/* ******************************************************

This program shows a technique to return cursor postion
in both text mode (row and column of cursor), and
graphics mode (X and Y coordinates of cursor).

 ****************************************************** */
```

CAT AND MOUSE

continued from page 97

```
    NewRow = Row/8,
    NewCol = Col/8,
    cursor(NewRow,NewCol),
    write(NewRow,",",NewCol),
    fail.
txt_pos1(2,_,_).
```

After the **repeat** subgoal, we get the status of the mouse (**Button**, **Row**, and **Col**), which we pass to **txt_pos1**. There are two **txt_pos1** clauses, which are selected by pressing the appropriate button on the mouse. Pressing the left button instantiates **Button** to the value **1**, which matches with the first **txt_pos1** clause. Notice that the **fail** in this clause causes backtracking (in **txt_pos**) back to **repeat**. This backtracking allows the program to mark many positions on the screen, as long as you only press the left button. When you press the right button, the program transfers control to the second clause for the **txt_pos1** predicate, the subgoal **txt_pos(2,_,_)** succeeds, and control returns to **report_pos**.

END OF A TAIL

This brings us to the end of this Turbo Prolog cat-and-mouse game—at least for the moment. During the chase, we've explored a number of mouse programming techniques that are unique to Turbo Prolog. First, we've examined how basic mouse functions are called with the **bios** predicate. Second, we've used backtracking instead of recursion to continuously poll the mouse for activity. Third, we've explored how to associate the mouse with specific actions in our program.

The mouse can be used in Turbo Prolog applications in a number of ways. For instance, you can use the mouse to dynamically move Turbo Prolog windows on the screen by pointing to a window and dragging it to a new position. This same technique can also be used to resize a Turbo Prolog window.

Another possible Turbo Prolog mouse application is to select strings of text displayed on the screen, and then move or copy those strings to a new screen location. In graphics mode, the mouse can be used to point or to draw

lines, rectangles, polygons, and so forth; and to indicate the direction for rotation, reflection, perspective, and other kinds of graphical transformations of those figures.

In the second article of this two-part series, we'll chase our mouse into two Turbo Prolog applications. The first application will use the mouse with a pop-up menu; in fact, we'll modify the Turbo Prolog Toolbox's menu tools to work with the mouse. The second application will allow us to scroll text in a window by using horizontal and/or vertical scroll bars similar to those on the Macintosh. Tune in next time as the Turbo Prolog mouse chase continues. ■

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*, Alameda, California: SYBEX Inc., 1986.

Nath, Sanjiva. *Turbo Prolog: Features for Programmers*, Portland, Oregon: MIS Press, 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.

Safaa H. Hashim is a graduate student at the Computer Science Division, University of California, Berkeley.

Listings may be downloaded from CompuServe as MOUSE1.ARC.

```
include "msm-drv.pro"        /* include mouse driver file */

PREDICATES

   report_pos
   txt_pos
      txt_pos1(INTEGER,INTEGER,INTEGER)
   gra_pos
      gra_pos1(INTEGER,INTEGER,INTEGER)
   repeat

CLAUSES

/* ******************************************************* */
/* REPORT CURSOR POSITION BOTH IN TEXT, AND GRAPHIC MODE  */
/* ******************************************************* */

   report_pos :-
      msm_init,
      msm_show,
      repeat,
         txt_pos,
         gra_pos.

/* ********************** */
/* TEXT MODE              */
/* ********************** */

   txt_pos :-
      makewindow(1,7,0,"Text Mode",0,0,25,80),
      write("\n Press left button to indicate cursor position"),
      repeat,
         msm_stat(Button,Row,Col),
         txt_pos1(Button,Row,Col).

/* If Button = 1, you pressed left button to report position */

   txt_pos1(1,Row,Col):-
      NewRow = Row/8, /* scale cursor row position to text mode */
      NewCol = Col/8, /* scale cursor col position to text mode */
      cursor(NewRow,NewCol),
      write(NewRow,",",NewCol),
      fail.

/* If Button = 2, you pressed the right button to end text mode */

   txt_pos1(2,_,_).

/* ********************** */
/* GRAPHIC MODE           */
/* ********************** */

   gra_pos :-
      graphics(2,1,4),
      write("\n Press left button to indicate cursor position"),
      msm_show,
      repeat,
         msm_stat(Button,X,Y),
         gra_pos1(Button,X,Y).

   gra_pos1(1,X,Y) :-  /* left button is pressed */
      NewX = (X/200)*31999, /* scale X coord. to graphic mode */
      NewY = (Y/640)*31999, /* scale Y coord. to graphic mode */
      Xpos=X/8, Ypos=Y/8, cursor(Xpos,Ypos),
      write(NewX , "," , NewY),
      fail.

   gra_pos1(2,_,_).

   repeat.
   repeat :- repeat.

/* ******************** END OF POSITION.PRO *********** */
```

VARIABLE VARIATIONS

The area in which a variable is known can be as important as the data it contains.

David A. Williams

SQUARE ONE

Modern BASIC compilers such as Turbo Basic not only enable your programs to run faster, but they also have many features that make your programs easier to write, debug, and maintain. Examples include procedures (subprograms), multiline defined functions, block-structured program statements, and the "scoping" of variables. Of these extensions, variable scoping is by far the subtlest and most alien to the original spirit of Dartmouth BASIC. The scoping of Turbo Basic's variables is well worth a close and thorough look.

GLOBAL HEGEMONY

Interpreted BASIC treats all variables as global. Once you declare a variable, it is available to be read or modified by any statement in the program. This makes each variable identifier absolutely unique. You can use a given variable name for only one entity in a single program, whether the variable is in the main program or in a small subroutine.

This is no great hardship if you write small, simple programs; you can easily remember which variable names have been used. The larger the program, however, the more likely you are to make a mistake. Index variables, which act as counters in **FOR..NEXT** loops, are especially difficult to track. Consider the following program fragment:

```
CLS
DIM ...
I=37.4
X=23
.
.
GOSUB CalcIt
.
.
PRINT I*X
.
.
```

During development of this program, you decide to save some effort and "drop in" the small section of code shown below that you've lifted from another program:

```
CalcIt:
FOR I=1 TO 10
  FOR J=1 TO 20
    A(I)=A(I)*B(J)
  NEXT J
NEXT I
RETURN
```

Note that the variable **I** appears in both sections of code, even though there is no logical connection between the two sections. The **I** in subroutine **CalcIt** is used only as a **FOR..NEXT** loop counter and bears no relation to variable **I** in the main program. The two sections do not conflict with one another from a *syntactic* standpoint, but they may logically interfere with one another unless you take specific steps to ensure that they don't.

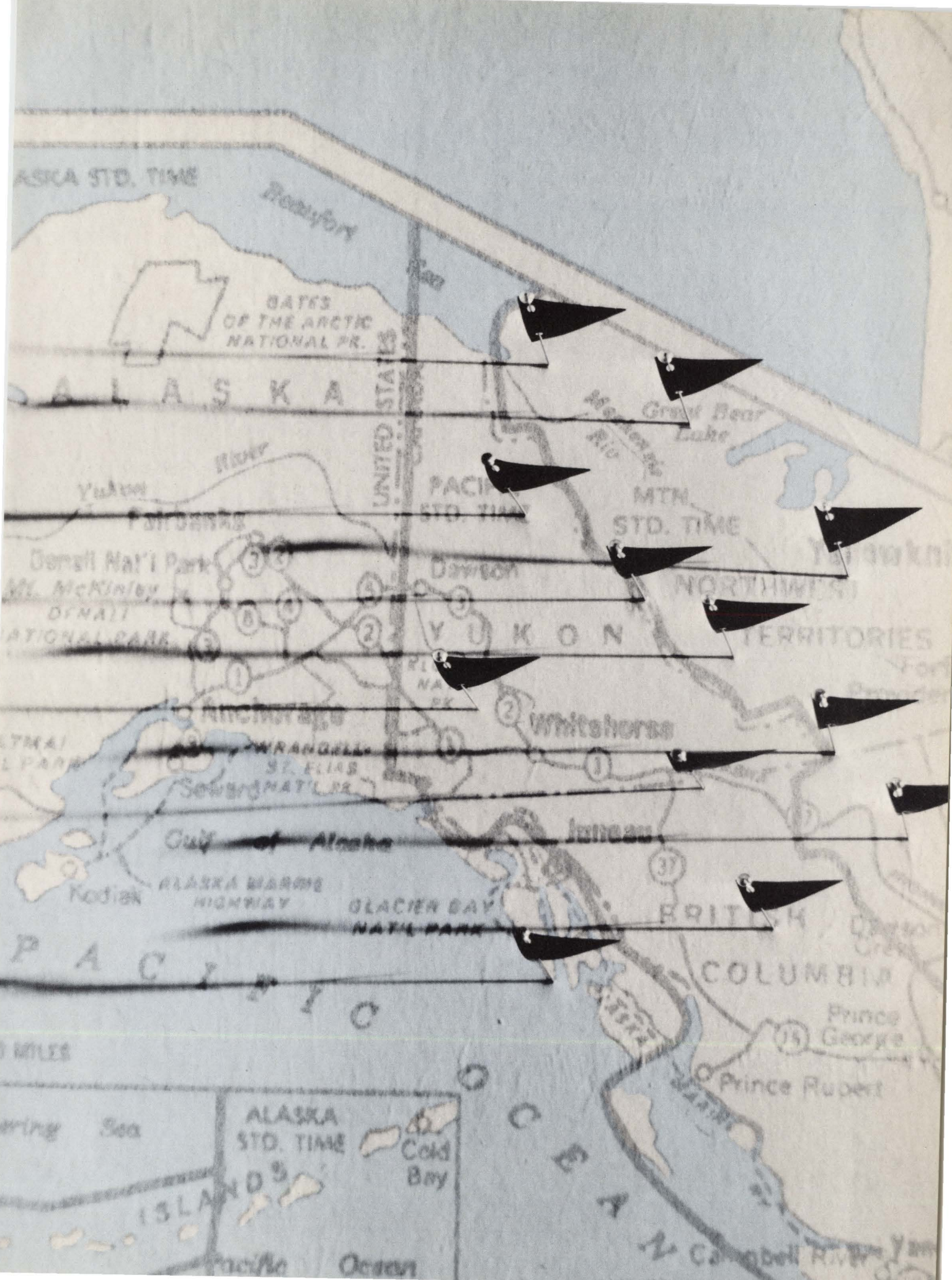
This limitation makes it very difficult to write generic routines that you can combine into a single library file and use in many different programs. A library of such commonly used routines can save you many hours of programming and debugging time and let you concentrate on creating your main program. Libraries are most comprehensible if you establish a system in which a particular variable name always represents the same kind of quantity, or is always used for the same purpose in every routine. Using the variable **I** only within **FOR..NEXT** loops would be a good example of this sort of convention. Before you can follow this system, however, you need a way to keep the multiple instances of identifier **I** in all of your various library routines from conflicting with one another.

Language designers created the concept of scope to meet this need. The language compiler restricts the use of local variables to a certain area of the program. This area, called the local variable's *scope*, is the region of the program in which the variable is "known" (i.e., where the variable is available to be read from or written to).

The compiler builds a conceptual fence that isolates one section of the program from another. Since program code on one side of the fence is independent from that on the other, you can have identical

continued on page 102

Bradley Ream


ASKA STD. TIME
Boundary
GATES OF THE ARCTIC NATIONAL PK.
ALASKA
UNITED STATES
Great Bear Lake
River
Yukon
Fairbanks
PACIFIC STD. TIME
MTN. STD. TIME
Denali Nat'l Park
Mt. McKinley
DENALI
NATIONAL PARK
Dawson
YUKON
TERRITORIES
Anchorage
Whitehorse
WRANGELL-ST. ELIAS
Seward
NAT'L PK.
Gulf of Alaska
Juneau
Kodiak
ALASKA MARINE HIGHWAY
GLACIER BAY NAT'L PARK
PACIFIC
COLUMBIA
Prince George
Prince Rupert
MILES
ALASKA STD. TIME
Cold Bay
Pacific Ocean

continued from page 100

variable names for different variables, as long as each instance of a name resides within its own fenced-off backyard. Turbo Basic constructs these fences around routines written as procedures and as defined functions, and also around chained programs. On the other hand, program code imported with the **$INCLUDE** metastatement, and subroutines called via **GOSUB**, share the scope of the main program and do not have a separate scope.

THE VARIABLE BESTIARY

While the term "local" is often applied to any variable whose access is limited in some way, the true definition of *local variable* under Turbo Basic is more precise. A running program, upon entering a procedure or function, establishes local variables on the stack. These local variables disappear and their values are lost during stack clean-up when the program exits the procedure or function.

A *static variable* is similar to a local variable in that its scope is limited to the routine in which it is declared. However, the compiler assigns each static variable a permanent location in memory, so the static variable retains its existence and its value even after the program exits the routine that owns the variable.

Of course, you usually don't want to completely isolate subprograms from your main program. You have to pass variables and values to the routine and send the results back to the main program. One way of doing this is with an *argument list*, a list of variables or values passed to the procedure or function. Variables in the argument list are available to both the caller and the routine being called. There are limits to the use of argument lists, most notably that array variables cannot be passed to functions as arguments. Arrays may, however, be passed as arguments to procedures.

Shared variables are available to both the subprogram in which they are declared and to the main program, but not to other subprograms. They can be used to augment the argument list of a subprogram, because variables of any type can be shared variables. Arrays declared in a function should be declared as **SHARED** with the main program, because (as mentioned above) array variables cannot be passed to functions through argument lists.

With earlier BASICs, the term global variable is sometimes used interchangeably with shared variable, but a *global variable* is available to *all* parts of a program without restriction. Any program statement in the main program, in any procedure, or in a function can access a global variable. Turbo Basic, however, does not support global variables in this sense. By default the scope of a variable in Turbo Basic is limited to the entity in which it is defined (either the main program or a subprogram), and no single statement declares a variable to be global. To make a Turbo Basic variable effectively global, you must declare it as **SHARED** within every subprogram in the program.

In certain situations, a variable's status defaults to local, static, or shared. But you can also declare the status of a variable with the **LOCAL**, **STATIC**, and **SHARED** statements. You can only use these statements within procedures and functions, and they must appear before the code body of the procedure or function.

PROCEDURES AND THEIR VARIABLES

A *procedure* is a multiline program segment bounded by the **SUB** and **END SUB** statements. The main program calls a procedure with a **CALL** statement, which may have an argument list. Although the *Turbo Basic Owner's Handbook* states otherwise on page 355, variables declared within a procedure are static by default, but you may also declare variables to be local or shared. You cannot declare any variable in a **LOCAL**, **STATIC**, or **SHARED** statement if that same variable also appears in the argument list.

Consider the following example:

```
CLS
A=3
B=56
X=5
Y=2
C=6
CALL TEST(X,Y)
PRINT C,X,A,Y
END
SUB TEST(A,D)
  LOCAL B
  SHARED C
  B=A^2
  C=C+A
  D=4*B
  E=E+1
END SUB
```

The **PRINT** statement displays:

```
11   5   3   100
```

The variable **B** in the procedure is declared as local; it is maintained on the stack and its value will be lost when the program exits the procedure. The variable **B** in the procedure is different from the variable **B** in the main program, because their scopes do not intersect.

The main purpose of the **SHARED** statement in procedure **TEST** is to make the main program variable **C** available to **TEST** without having to put **C** in the argument list. Shared variable **C** starts out with the value 6 assigned in the main program. **C** is then changed by the procedure **TEST**; when printed, **C** has the new value of 11.

X is not changed by procedure **TEST**, but **Y** assumes the value of **D**. This example illustrates how a value may be returned to the main program through the argument list. The main program's variable **A** does not change, since it is independent of the variable **A** in the procedure's argument list.

You must keep default conditions in mind, but it is not necessarily a good idea to depend on them. Being explicit about variable scoping costs nothing in code speed and costs very little in compilation time. **E**, which by default is static, is initialized to 0 when the program begins execution, and is incremented by 1 each time the procedure **TEST** executes. The incremented value remains in memory between executions of **TEST**. There is no reason to declare **B** as local, as opposed to letting it be static by default (like

variable **E**). Good practice, however, suggests that you declare the procedure's own variables with the **STATIC** or **LOCAL** statements as desired so that anyone looking at your program listing (including you) can see the nature of your variables at a glance. Furthermore, Borland does not guarantee that default conditions will not change in future releases of the compiler. To this end, we should add the statement **STATIC E** to the example program.

SHARING STRATEGIES

For any given variable, you have to choose between using the argument list or the **SHARED** statement when you are designing a procedure. If your procedure is for use within a single program, the **SHARED** statement is very convenient. On the other hand, if you are designing a routine for incorporation into a library that is used by many programs, it is better to pass everything possible through argument lists. You will be able to use the routine in any program without worrying about matching shared variable names.

You can pass arrays either through the argument list, or by declaring them as **SHARED** within the procedure. With the former method, use the format **arrayname(n)**, where **n** is the number of dimensions. Keep in mind that passing arrays in this way applies *only* to procedures; arrays cannot be passed to functions as arguments. In the **SHARED** statement, use **arrayname()**.

An additional advantage of using argument lists in passing values to procedures is that the actual parameters passed as arguments can be different variables on each invocation, whereas a shared variable is always the same variable on every invocation. For example, consider the following procedure header:

```
SUB AVERAGE(NumArray(1),Average)
```

Here, the formal parameter **NumArray(1)** specifies that a one-dimensional array may be passed to **AVERAGE**. It doesn't say *which* array. In other words, if you have two arrays, **Array1** and **Array2**, either one may be passed to **AVERAGE** in the **NumArray(1)** formal parameter.

On the other hand, if you don't include an array formal parameter in the argument list, and instead declare **Array1** as **SHARED** within procedure **AVERAGE**, the actual array variable **Array1** is the only array that will be available to **AVERAGE**.

DEFINED FUNCTIONS

Support of multiline defined functions is one of Turbo Basic's more significant enhancements. Older BASIC interpreters limit defined functions to a single line. Using multiple lines in Turbo Basic allows the creation of considerably more complex functions.

While similar to procedures, defined functions have several distinctive properties. You execute a function by placing its name in an expression, as in **A=FNTEST(X,Y)**. The function name takes on a value during the function's execution, and this value is returned to the caller as though the name of the function were the name of a variable.

A function is not required to return a value in the function name; it may simply perform its work by executing some statements. Values may also be passed back to the caller through shared variables. Note that, unlike procedures, functions *cannot* pass values back to the main program through the argument list. Another difference between functions and procedures is that variables appearing in a function's argument list are local to that function, but other variables in the function are by default shared with the main program unless declared otherwise.

The following example illustrates some of the properties of variables in defined functions:

```
CLS
X=5
Y=3
A=17
ANSWER=FNTEST(X,Y)
PRINT ANSWER,C,A
END
DEF FNTEST(A,B)
  FNTEST=A^2+2*A*B+B^2
  C=A+B
END DEF
```

The **PRINT** statement displays the output:

```
64    8    17
```

continued on page 104

continued from page 103

A function is given a value by assigning the value of some expression to the function name, as in the statement:

```
FNTEST=A^2+2*A*B+B^2
```

The main program can access this value by using the function name in an expression. The expression does not have to be an assignment statement. I could also have written:

```
PRINT FNTEST(X,Y)
```

Note that variable **C**, which is a shared variable by default, is available to the main program even though it was never declared there. Shared variables are the best way to operate on arrays, since you can't pass complete arrays to functions through the argument list. Only the individual elements of an array can be passed in this way.

A and **B** are local by virtue of appearing in the argument list. If your function requires the use of temporary variables, you can declare them with the **LOCAL** or **STATIC** statements to avoid interference with main program variables. Again, good practice dictates that you declare all of your variables as having a specific scope, rather than rely upon default conditions. This means that we should add the statement **SHARED C** to the **FNTEST** function.

Defined functions have another property not shared by procedures, as shown below:

```
CLS
X=1
Y=2
Z=3
DUMMY=FNTEST(X,Y,Z)
PRINT X
CALL TEST(X,Y,Z)
PRINT X
END
DEF FNTEST(A,B,C)
  A=A+B*C
END DEF
SUB TEST(A,B,C)
  A=A+B*C
END SUB
```

The function and the procedure seem identical, yet the first **PRINT X** statement will display 1, whereas the second will display 7. In the first case, the function's argument list passes only the value of the variable **X**. The function cannot change **X** since it doesn't know where **X** is stored. This is called *passing by value*. In the second case, the procedure's argument list passes the address of the variable **X**. The procedure reads the original value, performs the operation, then returns the result to the variable **X**. This is called *passing by reference*. If you change the **CALL** statement to **CALL TEST((X),Y,Z)**, the value of **X** will not change because this tells the compiler to pass only the value of **X**.

SINGLE-LINE FUNCTIONS

Single-line functions have properties similar to multiline functions that don't use the **LOCAL**, **STATIC**, or **SHARED** statements. Variables in the argument list are local, and all variables within the function's definition are shared with the caller. The following example illustrates a simple single-line function:

```
CLS
B=10
X=2
PRINT FNTEST(X)
END
DEF FNTEST(Y)=B+Y
```

The **PRINT** statement displays 12.

CHAINED PROGRAMS

Chained programs are relics of the days of small memory systems and the limited power of BASIC interpreters. Programs of a reasonable size often could not fit completely in memory, so they were divided into multiple modules that passed control from one to another, with each module taking its turn in memory while it executed. Turbo Basic's ability to use all available memory will probably allow you to combine previously chained programs into a single module. Turbo Basic does support chaining to a limited extent, and the subject of local variables is not complete without mentioning chaining.

With the statement **CHAIN <filespec>**, a program can pass execution to a second program that has been compiled with the .EXE or .TBC extension. The second program can pass execution back either to the first program, to a third program, or to DOS. The scope of all variables in a series of chained programs is limited to the single program that contains those variables. In other words, if **PROGA** chains to **PROGB**, **PROGB** has no knowledge of, or access to, variables within **PROGA**.

The **COMMON** statement allows you to share variables among programs in a chained series of programs. **COMMON**, accompanied by a variable list, must appear in both the called and the calling programs. The variable list in each program must contain the same number and type of variables, listed in the same order. To share variables **A**, **B**, **MyArray**, and **BUF$**, the following statement must appear in all programs that will share these variables, *in exactly this form*:

```
COMMON A, B, MyArray(1), BUF$
```

The scope of all variables cited in the **COMMON** statement is thus expanded to include all programs that contain the **COMMON** statement. Again, it is crucial that each variable in all **COMMON** statements has the same name, the same type, and is in the same order in all instances of the **COMMON** statement, or else a runtime error will occur.

VARIABLE STARS

As author Tom Swan has said, statements are what a program *does*, and variables are what a program *knows*. When first learning a programming language, users often look at the statements that make up a program without thinking very much about the variables that are acted upon by those statements. Taking the time to understand how scoping affects Turbo Basic's variables will make it much easier to create programs that work correctly and read well, both now and six months from now. ■

David A. Williams is a principal staff engineer for a major aerospace company. He can be reached at 2452 Chase Circle, Clearwater, Florida 34624.

INSTANTANEOUS HELP SCREENS

Tuck some helpful information in the screens hiding behind your screen.

Ralph Roberts

PROGRAMMER

Instantaneous help screens are an easy way to add professional "slickness" to your Turbo Basic software. These screens appear instantly at the touch of a key, then neatly disappear until called again.

You can have up to eight display "pages" in text mode, but only on CGA, EGA, or VGA adapters. The page being processed and/or displayed is set using the **SCREEN** statement. The general format is:

```
SCREEN [mode] [,[colorflag] [,[apage] [,[vpage].
```

The mode must be 0 (i.e., the text default), and **colorflag** is not used (though you can use the **COLOR** statement as much as desired). The trick is the last two parameters, **apage** and **vpage**. **apage**, an integer value from 0 to 7, controls which text page is written to. **vpage** selects the one shown.

In a help screen application, you want to devise a routine to load help screens during program initialization so that the screens do not appear until they're called. Use event trapping to make a screen pop into place when a specific "help" key is pressed. You write to one screen while showing another.

At the start of the program, F1 is defined as invoking the **HELP** subroutine; the F10 **EndIt** subroutine provides an exit from the program. When a subroutine is assigned to a function key, you must turn the key on. Turbo Basic then checks between each statement to see if that key has been pressed. If it has, the associated operation is performed.

Note that this will *not* work on the IBM Monochrome Display Adapter, which contains only 4K of text buffer; this is enough for a single page and no more. ■

Ralph Roberts is a freelance writer and ham radio operator (WA4NUO) who has written books on many topics, including Turbo Basic, Turbo Prolog, Reflex, and autograph collecting.

Listings may be downloaded from CompuServe as HLPSCR.ARC.

LISTING 1: SCREEN.BAS

```
' ::::::::: Instantaneous Help Screens by Ralph Roberts ::::::::::::

  CLS : ON KEY (1) GOSUB ONE : ON KEY (2) GOSUB TWO
  ON KEY (10) GOSUB EndIt : KEY (1) ON : KEY (2) ON : KEY (10) ON
  SCREEN ,,1,0      ' Write to Screen 1 but show Screen 0!

COLOR 14            ' Make it a different color
FOR X% = 1 TO 42    ' Use integers for loops, it's faster
  PRINT "This is an instantaneous HELP Screen. ";
NEXT X% : PRINT     ' This loop invisibly fills the HELP screen
PRINT : PRINT "                  (press any key to return to program)"
COLOR 15             ' color for second HELP screen

SCREEN ,,2,0
FOR X% = 1 TO 42    ' Use integers for loops, it's faster.
  PRINT "This is yet another HELP Screen. ";
NEXT X% : PRINT     ' This loop invisibly fills the 2nd HELP screen
PRINT : PRINT "                  (press any key to return to program)"
COLOR 7             ' Restore color to default
SCREEN ,,0          ' Restore screen writing to Screen 0

MainProgram:
  LOCATE 1,20  : PRINT "I n s t a n t a n e o u s   S c r e e n s"
  LOCATE 25,20 : PRINT "Hit F1 for Help, F2 or Help2, or F10 to end";
  LOCATE 12,30 : COLOR 0,7 : PRINT DATE$,TIME$; : COLOR 7,0
  GOTO MainProgram

  EndIt:
    CLS : COLOR 7,0 : END
  ONE:
    HelpScreen = 1 : GOTO HELP
  TWO:
    HelpScreen = 2 : GOTO HELP
  HELP:
    SCREEN ,,HelpScreen     ' POP up the called for screen
    WaitABit:
      A$ = INKEY$ : IF A$ = "" THEN WaitABit
    SCREEN ,,0      ' restore main program screen & continue operation
  RETURN
```

PICK A FILE, ANY FILE

File selection menus let your user "peruse and choose," with a little help from the Turbo Basic Database Toolbox.

Marty Franz

Sooner or later, your programs will need to get a filename from your user. When that time arrives, you could execute an **INPUT** statement asking your user to type the drive, directory, and filename. In this situation, the task of specifying a legal filename (eight letters, digits, and common symbols; followed by a period; optionally followed by one to three characters) falls completely upon your user—and leaves plenty of room for a naive user to make a mistake.

Isn't there a better way to ensure that you get a correct filename from your user? Wouldn't it be nicer if your user could highlight a file in a list that you display on the screen, and then select that file by pressing Enter?

You can make this scenario a reality by rounding out your Turbo Basic programs with the routines demonstrated in the file selection program PICKER.BAS (Listing 1). PICKER prompts the user for a drive and directory, and then displays a list of the files in that directory. The selected file is highlighted by using the up and down arrow keys, and the Home and End keys. If Enter is pressed, the highlighted file is displayed for confirmation; if Esc is pressed, entry is aborted.

Although PICKER is a simple program, it illustrates the use of the various Turbo Basic routines described below. These routines, which can be modified and incorporated into your own programs, provide good examples of the use of Turbo Basic's powerful **CALL INTERRUPT** statement. They also demonstrate the use of subroutines from the Turbo Basic Database Toolbox for handling the details of getting input from the user, scrolling, and writing to the screen.

INTERRUPT HINTS

Before we examine the implementation of Turbo Basic's **CALL INTERRUPT** statement in PICKER, let's look for a moment at the way that Turbo Basic handles interrupts.

Using interrupts from interpreted BASICA is tedious and requires an assembler. You have to write assembly language routines to pass control and parameters from your BASIC program to the interrupt routine, then generate the interrupt, and finally clean up the stack and come back. Turbo Basic, however, contains a **CALL INTERRUPT** statement that allows you to generate any valid 8088 interrupt by specifying the interrupt number. For example, to call DOS using the **CALL INTERRUPT** statement, you code:

```
CALL INTERRUPT &H21
```

Turbo Basic provides a **REG** buffer to hold 8088 register values before and after an interrupt call. The *Turbo Basic Owner's Handbook* tells exactly which subscripted element in the **REG** buffer stands for which register in the buffer. For example, element 0 holds the flags register, element 1 holds the AX register, and so on. Rather than attempting to remember all these correlations, you should always include the file REGNAMES.INC from the Turbo Basic distribution disk, and define named constants for the frequently used register values. Your code will be much more readable if you refer to registers and DOS function call services by their symbolic names:

```
%FindFirst=&H4E00
REG %AX,%FindFirst
```

This is much easier to read than the literal equivalent:

```
REG 1,&H4E00
```

Following readability conventions makes future maintenance of your program much easier.

To make effective use of **CALL INTERRUPT**, you should have both a DOS technical reference manual and a good reference to PC BIOS routines. Other books on IBM PC assembly language are also helpful.

LOOKING FOR FILES

The routines **MsDos** and **FNDosError** set the registers and check for errors; we can also call them to perform directory searching chores. Searching

a DOS directory involves two calls: Find First Matching File and Find Next Matching File. The logic for using these calls (assuming that you already have a file mask) can be expressed in pseudo-code:

```
Find first file matching the mask
If there was a file found then
  Do
    Find the next file
         that matches the mask;
  Loop until no more files
End If
```

The first of PICKER's two main parts contains two versions of this logic. The version in the subroutine **FNCountFiles** returns a count of the files matching the mask and enables you to dynamically allocate an array to hold the filenames. Another version of this logic is in the subroutine **FNGetFiles**, which fills a string array with all the files that match the mask. The following Turbo Basic code statements let you use these routines:

```
Mask$="*.*"
NumberFiles=FNCountFiles(Mask$)
DIM FileNames$(NumberFiles)
CALL GetFiles(Mask$,FileNames$())
```

If a nonexistent or illegal mask is entered, the routines find no files. In this case, calling the function **FNDosError** returns the reason why no files are found. While officially the code 2 signifies No Files Found, and the code 18 indicates No More Files, any return code other than zero is treated in the same way. Let's amend the logic given above to be the following:

```
Mask$="*.*"
NumberFiles=FNCountFiles(Mask$)
IF NumberFiles>0 THEN
  DIM FileNames$(NumberFiles)
  CALL GetFiles(Mask$,FileNames$())
ELSE
  PRINT "No files found."
END IF
```

As shown here, the first part of PICKER handles the actual directory search; we'll jump farther into PICKER shortly. First, however, you need to know about three important wrinkles with respect to directory searches in BASIC.

Attribute Mask. Searching directories requires you to specify an attribute mask, along with the filename. The *attribute mask* tells DOS which types of directory entries you are seeking. The exact bit meanings (when bits are equal to one) are given below:

continued on page 108

LISTING 1: PICKER.BAS

```
' PICKER.BAS:   Demonstration file picker program
'
' version:      2-10-88
' compiler:     Turbo BASIC 1.0
' uses:         REGNAMES.INC, ENTSUBS.BOX, SCRNASM.BOX, SCRNSUBS.BOX
' module type:  .EXE
'
' (C) Copyright 1987 Marty Franz
'
' This program prompts the user for a list of files and displays
' them, allowing selection using the up and down arrow keys,
' Home, End, and Esc.

DEFINT A-Z

$INCLUDE "REGNAMES.INC"
$INCLUDE "ENTSUBS.BOX"          'From the Turbo BASIC Database Toolbox
$INCLUDE "SCRNASM.BOX"
$INCLUDE "SCRNSUBS.BOX"

%False=0
%True=NOT(%False)

%GetDTA=&H2F00                  'Dos and Bios calls
%FindFirst=&H4E00
%FindNext=&H4F00

%SearchAttribute=&H21           'file attribute: R/O + archive
%FileNameOffs=30                'filename offset within DTA

%NoError=0                      'Dos error code

SUB EntUserHook(Ch$)
  ' This routine is required by the Turbo BASIC Database Toolbox entry
  ' functions.
  Ch$=INKEY$
END SUB

DEF FNLo(X)
  ' Return the low byte of integer X.
  FNLo=X MOD 256
END DEF

DEF FNHi(X)
  ' Return the high byte of integer X.
  FNHi=INT(X/256)
END DEF

SUB StringAddr(Segment,Offset,S$)
  ' Subroutine to get the segment and offset of a string.  Gets
  ' the segment from the first two bytes of the default segment.
  ' Gets the offset by reading the string descriptor and getting
  ' the third and fourth bytes of that.
  LOCAL O
  Segment=PEEK(0)+256*PEEK(1)
  O=VARPTR(S$)
  DEF SEG = VARSEG(S$)
  Offset=PEEK(O+2)+256*PEEK(O+3)
  DEF SEG
END SUB

DEF FNASCIIZ$(Segment,Offset,MaxLen)
  ' Get an ASCIIZ string from memory at the location given by
  ' Segment:Offset.  MaxLen is a check on the length of the
  ' string to get.
  LOCAL P,N,S$
  S$=""
  N=0
  P=Offset
  DEF SEG = Segment
```

```
  WHILE PEEK(P) AND N<=MaxLen
    S$=S$+CHR$(PEEK(P))
    INCR P
    INCR N
  WEND
  DEF SEG
  FNASCIIZ$=S$
END DEF

DEF FNDosError
  ' Return the last DOS error code received.
  IF (REG(0) AND &H01) THEN
    FNDosError=REG(%AX)
  ELSE
    FNDosError=0
  END IF
END DEF

SUB MsDos(N)
  ' Perform MS DOS call N.  Assumes the other registers have been
  ' set up already.  N is used as AX value.
  REG 1,N
  CALL INTERRUPT &H21
END SUB

SUB GetDosDTA(Segment,Offset)
  ' Get the current Dos DTA segment and offset.  If both are zero
  ' then an error occurred.
  Offset=0:Segment=0
  CALL MsDos(%GetDTA)
  IF FNDosError=%NoError THEN
    Offset=REG(%BX)
    Segment=REG(%ES)
  END IF
END SUB

DEF FNCountFiles(FileSpec$)
  ' Return a count of the files matching the filespec.  Use this
  ' subroutine to pre-allocate arrays for sorting and choosing.
  ' If it returns 0 then no files were found; this means checking
  ' FNDosError for the reason.
  LOCAL Segment,Offset,Mask$
  Mask$=FileSpec$+CHR$(0)
  CALL StringAddr(Segment,Offset,Mask$)
  REG %CX,%SearchAttribute
  REG %DX,Offset
  REG %DS,Segment
  CALL MsDos(%FindFirst)
  IF FNDosError=%NoError THEN
    Count=0
    DO
      Count=Count+1
      CALL MsDos(%FindNext)
    LOOP UNTIL FNDosError
    FNCountFiles=Count
  ELSE
    FNCountFiles=0
  END IF
END DEF

SUB GetFiles(FileSpec$,FileArray$(1))
  ' Fill a string array with the files that match the filespec
  ' passed.  Fills up to upper bound of FileArray from lower
  ' bound.
  LOCAL I,Segment,Offset,Mask$,DTASeg,DTAOffs
  CALL GetDosDTA(DTASeg,DTAOffs)
  Mask$=FileSpec$+CHR$(0)
  CALL StringAddr(Segment,Offset,Mask$)
  REG %CX,%SearchAttribute
  REG %DX,Offset
```

PICK A FILE

continued from page 107

&H01: file is read-only
&H02: hidden file
&H04: system file
&H08: volume label
&H10: subdirectory
&H20: archive bit

The archive bit is set whenever the file is written to and closed. It's used by the DOS BACKUP utility to tell if the file has been changed since the last time it was written to. After the backup has been made, this bit is reset to zero.

You can combine these bits into a single binary byte to tell DOS exactly what kinds of files you want to look for. PICKER uses the value &H21 to include read-only and nonarchive files in the search, and to omit everything else. You may also include hidden files, system files, and subdirectories in the search.

Disk Transfer Address. File information is placed in a *Disk Transfer Address*, or DTA, which is an area that DOS uses to pass information about directory entries. The layout of the DTA when you search directories is summarized in Table 1.

21 bytes – reserved by DOS
1 byte – attribute found
2 bytes – file's time
2 bytes – file's date
2 bytes – low word of file's size
2 bytes – high word of file's size
13 bytes – name and extension of file

Table 1. The structure of the DOS Disk Transfer Area.

The file's name and extension are stored as a proper filename ending with a zero byte. (This type of filename is called an ASCIIZ string in the DOS documentation.) The function **FNASCIIZ** retrieves this filename from the DTA, and the subroutine **GetDosDTA** finds the segment and offset of DOS's current DTA. DOS assumes that you already know which drive and path to search, and only gives you the specific filename and file-related information (date, time, and size).

File specification mask. You need to pass an ASCIIZ string to DOS containing the actual file specifi-

continued on page 110

"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"

Actual photograph of Turbo C graphics displayed on IBM 8514 screen.*

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.

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***

For the dealer nearest you,

Call (800) 543-7543

BORLAND INTERNATIONAL

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 metafile 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.

BI 1221

```
  REG %DS,Segment
  CALL MsDos(%FindFirst)
  IF FNDosError=%NoError THEN
    I=1
    DO
      FileArray$(I)=FNASCIIZ$(DTASeg,DTAOffs+%FileNameOffs,12)
      INCR I
      Call MsDos(%FindNext)
    LOOP UNTIL FNDosError OR I>UBOUND(FileArray$(1))
  END IF
END SUB

SUB Choose(Pick,Visible,N,Choices$(1))
  ' Subroutine to choose an item from a string array.  Pick holds
  ' the item number the user chose.  Visible is the number of
  ' items on the screen... it can be less than N, the number of
  ' items in the array.  If so, the routine will scroll the list
  ' up and down.  The array Choices$ contains the items.  When
  ' complete, the variable EscPressed will be %True if the user
  ' aborted, %False otherwise.  Keys handled are Enter, Esc, the
  ' up and down arrows, Home, and End.
  SHARED EscPressed
  SHARED Ent.NotAvailable,Ent.Escape,Ent.CR,Ent.UpLine,Ent.DnLine,_
         Ent.Home,Ent.End
  LOCAL From,I,R,Top,Bottom,Margin,Cmd
  Top=CSRLIN:Margin=POS(0)
  Bottom=Top+Visible-1
  Pick=1
  From=1
  GOSUB DisplayChoices
  Done=%False
  DO
    GOSUB Highlight
    DO
      CALL GetKeyStroke(Cmd)
    LOOP UNTIL Cmd<>Ent.NotAvailable
    GOSUB LowLight
    SELECT CASE Cmd
    CASE Ent.Escape
      Done=%True
      EscPressed=%True
    CASE Ent.CR
      Done=%True
      EscPressed=%False
    CASE Ent.UpLine
      IF Pick>1 THEN
        DECR Pick
        IF R>Top THEN
          DECR R
        ELSE
          CALL Scroll(1,Top,Bottom,Margin,80,&H07)
          LOCATE R,Margin
          PRINT Choices$(Pick);
        END IF
      ELSE
        CALL MinorErrorSound
      END IF
    CASE Ent.DnLine
      IF Pick<N THEN
        INCR Pick
        IF R<Bottom THEN
          INCR R
        ELSE
          CALL Scroll(0,Top,Bottom,Margin,80,&H07)
          LOCATE R,Margin
          PRINT Choices$(Pick);
        END IF
      ELSE
        CALL MinorErrorSound
      END IF
```

PICK A FILE

continued from page 108

cation mask to search for. The string is passed to DOS by loading the DX register with the string's offset, and loading the DS register with the string's segment. You can't just use Turbo Basic's **VARSEG** and **VARPTR** functions to set these registers, because **VARSEG** and **VARPTR** return the segment and offset of the string's descriptor, rather than of the string itself. To find the actual segment and offset of the string's data, you need to obtain the string data segment from the first two bytes of Turbo Basic's default data segment. The subroutine **StringAddr**, which uses the third and fourth bytes of the string's descriptor, finds the offset of the string data within this segment. The code that accomplishes the sequence is near the top of both the **FNCountFiles** function and the **GetFiles** procedure.

CHOOSING A FILE

The second part of PICKER is the subroutine **Choose**, which uses subroutines and shared variables that are found in the Turbo Basic Database Toolbox. **Choose** displays the contents of the array on the screen and allows the user to pick from among the displayed filenames. This straightforward subroutine displays as many names as it can, then lets you pick one by moving a reverse video bar up and down on the screen. The up and down arrows move the highlight bar up or down the list. The Home key goes to the top of the list, and the End key goes to the bottom. Study the code and you'll see that it's easy to support additional keys, such as PgUp or PgDn. The logic for **Choose** can be summarized in the following pseudo-code:

```
Display as many items as you can
Start with the first one
  Do
    Get a keypress
    If Enter or Esc pressed then
      Done
    Else
      Move the highlight bar
              as required
    End if
  Loop until done
```

When **Choose** is called, it assumes that the cursor is at the

top left corner of the area to be used for input. The parameters passed to **Choose** are:

Pick	The number of the item actually picked by the user.
Visible	The number of items in the list that can be seen at one time.
N	The total number of items in the list.
Choices$()	A string array containing the items to be displayed and chosen from.

Notice that nothing in the design of **Choose** limits it to the selection of filenames only. You can modify **Choose** to select anything that can be kept in a list in your programs, such as menu options, names, and so forth.

Also notice that you can choose both the number of items in the list and the number of items that are visible at any one time on the screen. **Choose** will scroll the list up or down as needed when you move the highlight bar.

Choose scrolls the screen with the Turbo Basic Database Toolbox's **Scroll** subroutine, which uses the PC's BIOS VIDEO service (interrupt 10H, subfunctions 6 and 7) to scroll the screen one line up or down. Another subroutine, **ClrArea**, uses the Toolbox function **WriteScreenArea** to clear an area of the screen that is designated by the row and column of its top left and bottom right corners. The subroutine **ClrEol** simulates the Turbo Pascal function of the same name, and also uses **WriteScreenArea** to clear a single line from the cursor to the edge of the screen.

Choose contains a shared variable named **EscPressed**. This variable is set to the constant **%True** when the Esc key, rather than Enter, is used to exit the subroutine. This process allows you to check whether the user has aborted a file selection session.

The function **GetKeyStroke** is worth further explanation. This function is a Turbo Basic Database Toolbox subroutine that retrieves a keypress as an ASCII number, rather than as a string. For extended keys (such as the

continued on page 112

```
    CASE Ent.Home
      From=1
      Pick=1
      GOSUB DisplayChoices
    CASE Ent.End
      From=N
      Pick=N
      GOSUB DisplayChoices
    END SELECT
  LOOP UNTIL Done
  EXIT SUB

DisplayChoices:
  ' Display all the choices starting with From on the screen.
  ' Clears the display area first.
  CALL ClrArea(Top,Bottom,Margin,80)
  I=From
  R=Top
  COLOR 7,0
  DO
    LOCATE R,Margin
    PRINT Choices$(I);
    INCR I
    INCR R
  LOOP UNTIL I>N OR R>Bottom
  R=Top
  RETURN

Highlight:
  ' Highlight the current choice.  Displays this in reverse
  ' video.
  LOCATE R,Margin
  COLOR 0,7
  PRINT Choices$(Pick);
  RETURN

Lowlight:
  ' Lowlight the current choice before moving on.  Displays the
  ' choice in normal video.
  LOCATE R,Margin
  COLOR 7,0
  PRINT Choices$(Pick);
  RETURN
END SUB

SUB ClrArea(TopRow,BottomRow,LeftCol,RightCol)
  ' Clear an area of the screen using WriteScreenArea
  LOCAL NumberOfRows,NumberOfCols,NumberOfChars,ClrText$,ClrAttr$
  NumberOfRows=BottomRow-TopRow+1
  NumberOfCols=RightCol-LeftCol+1
  NumberOfChars=NumberOfRows*NumberOfCols
  ClrText$=STRING$(NumberOfChars," ")
  ClrAttr$=STRING$(NumberOfChars,&H07)
  CALL WriteScreenArea(TopRow,LeftCol,NumberOfRows,NumberOfCols,_
    ClrText$,ClrAttr$)
END SUB

SUB ClrEol
  ' Clear a line from the cursor to the end using ClrArea
  CALL ClrArea(CSRLIN,CSRLIN,POS(0),80)
END SUB

' Main program

CALL ScrnInit                    ' Required by SCRNSUBS and SCRNASM
CALL InitEntry                   ' Required by ENTSUBS
DIM FileNames$(100)
COLOR 7,0:CLS
LOCATE 1,1:PRINT "File Picker Demo";
LOCATE 3,1:PRINT STRING$(80,196);
```

```
Done=%False
EntSpec$=""
Prompt$="Drive and Directory: "
FieldSize=80-LEN(Prompt$)-1
DO
  CALL PromptEntry(FieldSize,FieldSize,"",2,1,&H07,&H70,CHR$(13),3,_
       Prompt$,EntSpec$,Changed,ExitKey)
  IF Changed AND LEN(EntSpec$)>0 THEN
    Spec$=EntSpec$+"\*.*"
    N=FNCountFiles(Spec$)
    IF N>0 THEN
      LOCATE 24,1
      CALL ClrEol
      PRINT "Files listed:";N
      CALL GetFiles(Spec$,FileNames$())
      LOCATE 4,20
      CALL Choose(Pick,19,N,FileNames$())
      IF NOT(EscPressed) THEN
        LOCATE 24,1
        CALL ClrEol
        PRINT "You chose ";FileNames$(Pick);
      ELSE
        CALL MinorErrorSound
      END IF
    ELSE
      LOCATE 24,1
      CALL ClrEol
      PRINT "No files found...";
      CALL MinorErrorSound
    END IF
  ELSE
    Done=%True
  END IF
LOOP UNTIL Done
CLS
END
```

PICK A FILE

continued from page 111

function keys, Alt keys, and cursor pad keys), the second byte of the extended ASCII value is added to 255. For example, the Home key, 75, is translated as 326. When you include the Toolbox subroutines, you can use shared variables such as **Ent.Home** for these commonly used keycodes. If you aren't building strings out of the user's keypresses, **GetKeyStroke** is the preferred way to work with keypresses.

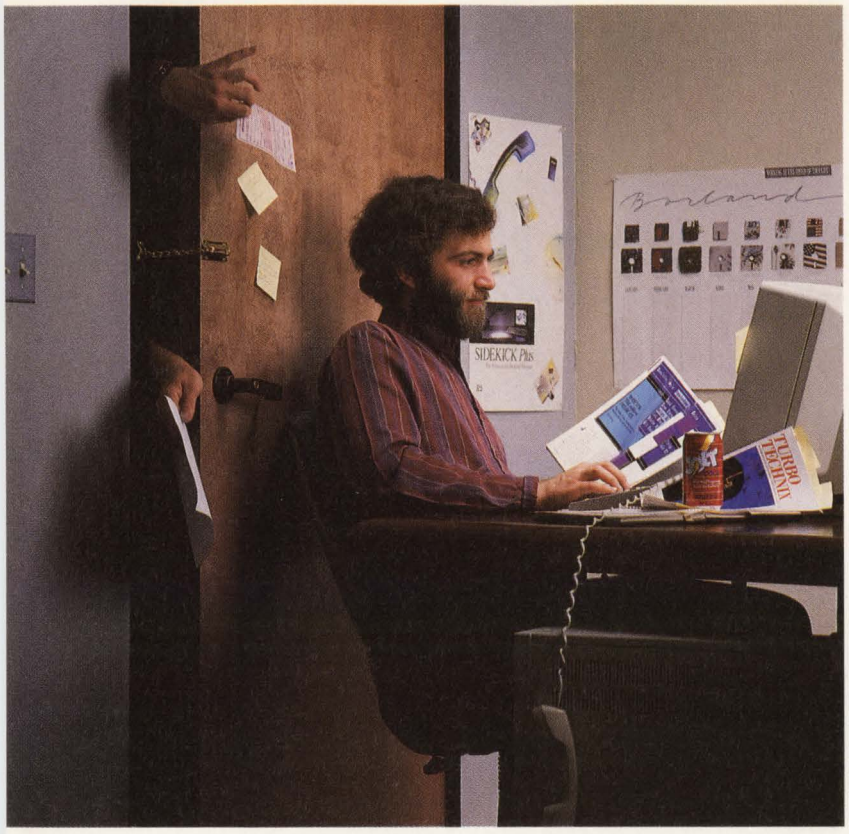
The Toolbox subroutine **PromptEntry**, called from the main program of PICKER, retrieves the drive and directory. Normal video attribute (7H) is used for the prompt area, and reverse video (70H) is used for the input area. This powerful Toolbox subroutine makes it easy to handle the user's input because it supports color changes, field lengths, and even the editing keys. (The Turbo Basic Database Toolbox's screen functions are described in "Turbo Basic Screens At Assembler Speed," *TURBO TECHNIX*, March/April, 1988.)

TAKE YOUR PICK

The routines included in PICKER constitute a simple toolbox for searching for files using the DOS FIND FIRST and FIND NEXT services, and for picking items from lists. You can also break each of the two parts of PICKER out into its own **$INCLUDE** file for use in other programs. There's plenty of room for improvement here, including defining the screen colors through variables, putting multiple columns of choices on the screen, and handling additional movement keys. But the routines as written illustrate the basics well enough for you to move forward on your own. Use them in good health, and from now on don't be shy about asking your users to pick a file. ■

Marty Franz is a programmer who frequently writes on microcomputer topics. He lives in Kalamazoo, Michigan.

Listings may be downloaded from CompuServe as PICKER.ARC.



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.

SEPTEMBER/OCTOBER 1988
ISSUE CLOSING DATE: JUNE 23

Multitask Turbo Pascal applications under DOS ... learn how to use linked lists in Turbo C ... understand PAL procedure memory management ... save and load EGA screens from Turbo Basic ... add a pattern-matcher to the MicroStar editor from the Turbo Pascal Editor Toolbox ... learn about storing data in 286 extended memory ... our columnists, our critiques, and lots more!

NOVEMBER/DECEMBER 1988
ISSUE CLOSING DATE: AUGUST 25

More on Turbo Pascal multitasking ... take the mystery out of structures and unions in Turbo C ... write a code-generating script in PAL ... emulate SQL in Turbo Prolog ... rotate Turbo Basic GET/PUT bitmaps in a hurry ... create custom disk formatter programs in Turbo Pascal ... expert advice from our columnists, and much more!

CALL NOW RESERVE YOUR TURBO TECHNIX SPACE TODAY!

Office of the Publisher
(408) 438-9321

Publisher
Marcia Blake

Advertising Sales Manager
John Hemsath

Western Office
(714) 858-0408
Janet Zamucen

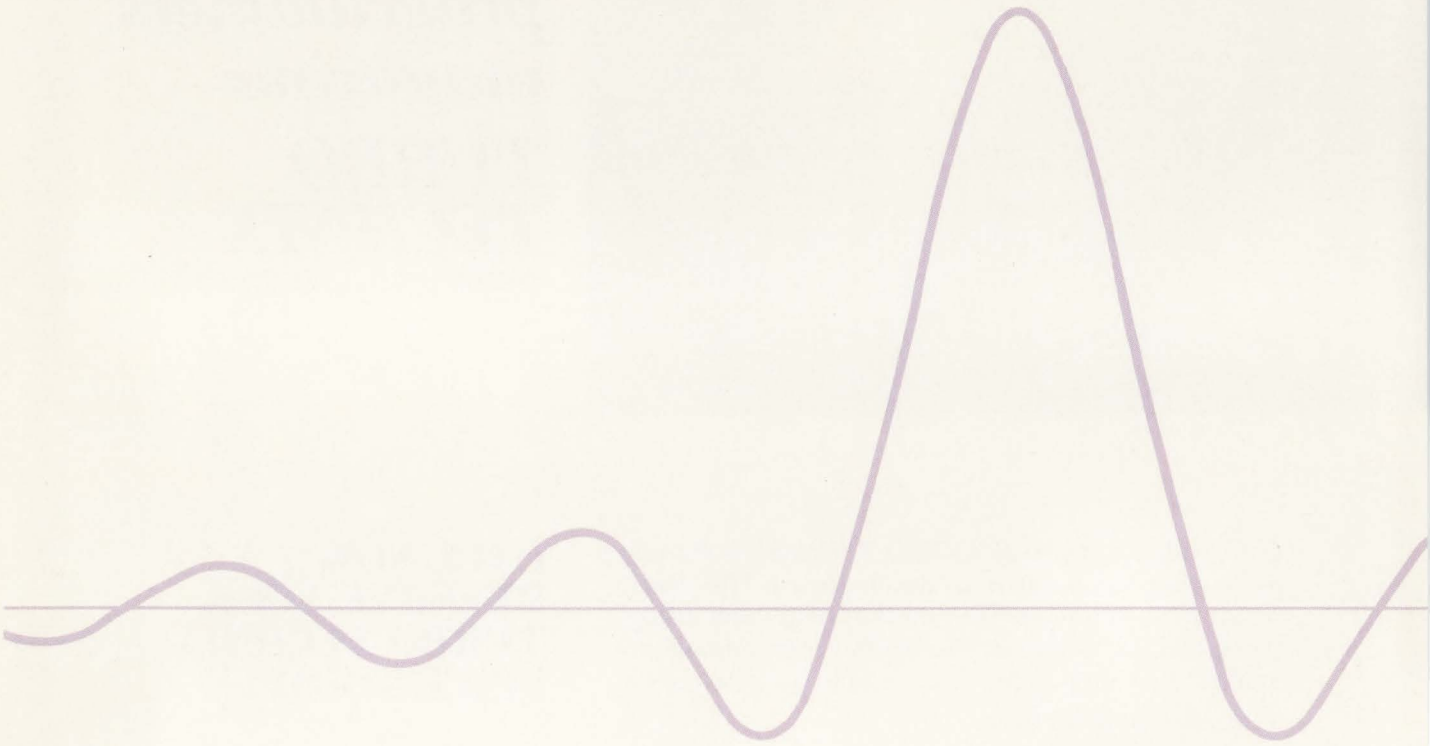
New England/
Mid-Atlantic Office
(617) 848-9306
Merrie Lynch
Nancy Wood

Southern Office
(813) 394-4963
Megan Patti

PLOTTER SUPPORT, TURBO STYLE

Don't rely on canned programs to do your plotting—send your own commands from any Turbo language.

William H. Murray and Chris H. Pappas



PROGRAMMER

Many scientists, engineers,and mathematicians use plotters on a regular basis to output information from commercial programs such as AutoCad, ASYST, smARTWORK, and Energraphics. Pen and ink drawings done on a plotter are crisp and neat. However, it's often difficult to write your own software to take greater advantage of the plotter. Fortunately, many plotters now available respond to the Hewlett Packard Graphics Language (HPGL) command set, which is summarized in Table 1. Describing these commands in detail is beyond the scope of one article, but most plotter manuals discuss the commands in sufficient depth.

What the manuals typically do *not* provide are language-specific programming examples. The best way to learn anything is by example, and here we provide simple examples of how to use the HPGL commands in all four Turbo languages. To take advantage of these four example programs, you need a plotter capable of understanding the HPGL command set. HPGL is as close as we come to a plotter interface standard these days, and most low-end plotters support it. Plotter manufacturers usually provide chapters in their hardware documentation to help you interface their plotters with your programs, and these chapters should say if your plotter understands HPGL. (If you're unsure about your plotter, contact the manufacturer directly.)

GIVE US SOME KIND OF SINE

The example plotter program is a popular graphics routine that plots a curve of **SIN(X)/X**.

COMMAND	EXAMPLE	DESCRIPTION
AA	AA 2016,2016,90,5;	Draws arcs (absolute)
AR	AR 1016,1016,90,5;	Draws arcs
CA	CA 3	Gets alternate character set
CI	CI 250,5	Draws circle
CP	CP 2,-2	Character plot
CS	CS 2	Gets standard character set
DC	DC	Reactivates auto pen lift
DF	DF	Sets plotter's default function
DI	DI 1	Direction of line lettering
DP	DP	Deactivates auto pen lift
DR	DR 1	Direction of line lettering
DT	DT*; bill*	Character (*) is terminator for label
EA	EA 1000,1000;	Rectangle at absolute coordinate
ER	ER 500,500	Rectangle at relative coordinate
EW	EW 500,30,90,5;	Perimeter of a pie segment
FT	FT 3,50,45	Fill instruction
IM	IM 19;	Reports unmasked program errors
IN	IN	Reinitializes plotter
IP	IP 500,250,1500,1250;	Assigns new coordinates
IW	IW 500,0,625,250;	Boundary for plotting
LB	LB THIS IS MY LABEL	Draws a string of characters
LT	LT 2,5;	Line type, for style of line
OA	OA	Outputs pen coordinates & physical status
OC	OC	Outputs pen coordinates & logical status
OD	OD	Actual coordinates & physical status
OE	OE	Outputs first unmasked instructions
OF	OF	Outputs size of plotter unit
OH	OH	Upper and lower plotter area
OI	OI	Outputs plotter identification
OO	OO	Outputs options of plotter
OP	OP	Current coordinates of P1 and P2
OS	OS	Output status byte
OW	OW	Coordinates of window
PA	PA 500,500;	Moves pen to specified position
PD	PD	Pen down
PR	PR 500,500;	Moves pen to relative coordinates
PS	PS 3;	Sets paper size
PT	PT .5	Pen point thickness (mm)
PU	PU	Pen up
RA	RA 600,600;	Shades rectangle
RO	RO 90;	Rotates plotter coordinates
RR	RR 600,600;	Shades relative rectangle
SA	SA	Selects recent character set
SC	SC 0,10365,0,7962;	Scale
SI	SI .6,.5;	Size of character set
SL	SL 1 (45 degrees)	Slants character set
SM	SM +;	Character and vector system
SP	SP 5;	Selects pen
SR	SR 1,1;	Width, height character size
SS	SS	Selects most recent character set
TL	TL 1;	Horizontal and vertical ticks
UC	UC	Designs symbols and characters
VS	VS 15;	Pen velocity
WG	WG 500,30,90,5;	Fills pie segment
XT	XT	Draws vertical tick at position
YT	YT	Draws horizontal tick at position

Table 1. Hewlett Packard Graphics Language commands.

Figure 1. A plot of the curve represented by the function **SIN(X)/X**, *as produced by the listings in this article.*

Most plotters communicate with your PC through a serial port. Before executing any of the example programs, you need to set the baud rate and other communication parameters of one of your serial ports to match the requirements of your plotter. In our case, the plotter requires a baud rate of 9600, no parity check, 8 data bits, and 1 stop bit. We've used serial port COM1. Setting things up is

continued on page 116

LISTING 1: PLOT.BAS

```
REM PROGRAM PLOT.BAS

REM LOAD BLACK PEN AND KEEP IT UP
LPRINT "IN SP 1"

REM PLOT A SIN(X)/X CURVE IN BLACK
FOR X=3 TO 1033
  Z=((X*10)-5182.5)*0.0035
  IF Z=0.0 THEN Z=0.1
  Y=(2000+3000*SIN(Z)/Z)
  LPRINT "PA",(X*10),",",(Y),"PD"
NEXT X
LPRINT "PU"

REM DRAW A BLUE BORDER AROUND WHOLE SIN(X)/X PLOT
LPRINT "SP 6 PA 0,0 PD EA 10365,7962 PU"

REM DRAW A RED LINE THROUGH MID POINT OF PLOT
LPRINT "SP 2 PA 0,2000 PD PA 10365,2000 PU"

REM DUMP PLOTTER BUFFER, ETC.
LPRINT "PA 0,7962 SP"
```

LISTING 2: PLOT.PAS

```
PROGRAM PLOT(INPUT,OUTPUT);

USES Printer;

VAR
  X,Y : Integer;
  Z   : Real;


PROCEDURE DRAWER;

BEGIN
  (* LOAD BLACK PEN AND KEEP IT UP *);
  WRITELN(LST,'IN SP 1;');

  (* PLOT A SIN(X)/X CURVE IN BLACK *);
  FOR X:= 3 TO 1033 DO
  BEGIN
    Z:=((X*10)-5182.5)*0.0035;
    IF Z=0.0 THEN Z:=0.1;
    Y:=ROUND(2000+3000*SIN(Z)/Z);
    WRITELN(LST,'PA ',(X*10),',',(Y),' PD;')
  END;
  WRITELN(LST,'PU;');

  (* DRAW A BLUE BORDER AROUND WHOLE SIN(X)/X PLOT *);
  WRITELN(LST,'SP 6 PA 0,0 PD EA 10365,7962 PU;');

  (* DRAW A RED LINE THROUGH MID POINT OF PLOT *);
  WRITELN(LST,'SP 2 PA 0,2000 PD PA 10365,2000 PU;');

  (* DUMP PLOTTER BUFFER, ETC. *)
  WRITELN(LST,'PA 0,7962 SP;');

END;

BEGIN
  DRAWER;
END.
```

LISTING 3: PLOT.C

```
/* PROGRAM PLOT.C */

#include <stdio.h>
#include <math.h>
```

PLOTTER SUPPORT

continued from page 115

simple via two invocations of the DOS MODE command:

```
C>MODE COM1:9600,N,8,1,P
C>MODE LPT1:=COM1
```

You can replace **COM1** with **COM2** in the two commands if you intend to use serial port COM2. The **P** (short for **P**rinter or **P**lotter) in the first command avoids possible time-out errors; omit it if you intend to communicate with a remote communications service rather than a desktop peripheral.

PLOTTING BY EXAMPLE

Listing 1 is a complete Turbo Basic program that generates the plot of the sine curve described earlier. Turbo Basic **LPRINT** commands direct program output to the printer port. The various HPGL commands are enclosed within the **LPRINT** statements (refer to Table 1 for more information about each command).

HPGL is as close as we come to a plotter interface standard these days, and most low-end plotters support it.

Listing 2 is an equivalent Turbo Pascal program. In this example, output is directed to the printer by using the **LST** device file in each **Writeln** statement. Otherwise, the program is structured very much like the Turbo Basic program.

Listing 3 is a Turbo C program. In C, there is no simple way to send output to the printer when using the **printf** command. It is much easier to write and compile

the program so that the plotter commands are sent to standard output. You can then redirect standard output to the chosen serial port using DOS command line output redirection. This process requires that you compile the program to a .EXE file and then leave the Turbo C programming environment. When you are ready to execute the program, use the DOS redirection command:

```
C>PLOT > PRN
```

Once plotter commands are sent to standard output, you can redirect standard output to the chosen serial port using DOS command line redirection.

Listing 4 completes the set of examples with a Turbo Prolog program that generates the same plot.

Figure 1 is the sample plot generated by each program. The actual plot is in color, with a blue border and a red line through the midpoint of the plot, but it's reproduced here in black and white. It's simple once you see how it's done! ■

Bill Murray and Chris Pappas are professors of computer science at Broome Community College. Together they have written six books for Osborne McGraw-Hill. Their book on the IBM PS/2 Model 80 will be released this summer.

Listings may be downloaded from CompuServe as HPPLOT.ARC.

```
main()
{
int    x,y;
float  z;

  /* LOAD BLACK PEN AND KEEP IT UP */
  printf("in sp 1;");

  /* PLOT A SIN(X)/X CURVE IN BLACK */
  for (x=1; x<1034; x++)
  {
   z=((x*10.0)-5182.5)*0.0035;
   if (z==0.0) z=0.1;
   y=(int)(2000.0+3000.0*sin(z)/z);
   printf("pa");
   printf("%d",x*10);
   printf(",");
   printf("%d",y);
   printf("pd;");
  }
  printf("pu;");

  /* DRAW A BLUE BORDER AROUND WHOLE SIN(X)/X PLOT */
  printf("sp 6 pa 0,0 pd ea 10365,7962 pu;");

  /* DRAW A RED LINE THROUGH MID POINT OF PLOT */
  printf("sp 2 pa 0,2000 pd pa 10365,2000 pu;");

  /* DUMP PLOTTER BUFFER, ETC. */
  printf("pa 0,7962 sp;");
}
```

LISTING 4: PLOT.PRO

```
predicates
   plot(real)
   checkZ(real,real)
   drawer

goal
   writedevice(printer),
   drawer,
   writedevice(screen),
   write("done").

clauses
   drawer:-
      write("in sp 1;"),   /* Load Black Pen */
      plot(3),
      write("pu;"),

      /* Draw a blue border around the plot */
      write("sp 6 pa 0,0 pd ea 10365,7962 pu;"),

      /* Draw a red line through the midpoint of the plot */
      write("sp 2 pa 0,2000 pd pa 10365,2000 pu;"),

      /* Dump Plotter Buffer, etc. */
      write("pa 0,7962 sp;").

   plot(1034):-!.
   plot(X):-
      TempZ = ((X*10) - 5182.5)* 0.0035,
      checkZ(TempZ,Z),
      Y = (2000 + 3000* sin(Z) /Z),
      TenX = X*10,
      write("pa",TenX,",",Y,"pd;"),
      NewX = X+1,
      plot(NewX).

   checkZ(Temp,Val):-
      Temp = 0,!,
      Val = 0.1;
      Temp = Val.
```

BACKGROUND COLOR MAGIC

You can trade those blinking characters for eight new background colors. Here's how.

Mark Novisoff


WIZARD

If you're like most programmers, you probably think that the PC can display only eight background colors in text mode. Not true—you can get 8 *additional* background colors on a color monitor, for a total of 16. One word of caution—these eight new colors come at the cost of blinking foreground characters.

The text video attribute byte uses 4 bits for foreground color (giving us 16 colors), 3 bits for the background color (giving us 8 colors), and 1 bit indicating whether or not the foreground character is blinking. Listing 1 (MAGIC.BAS) reprograms the video controller to interpret the blink bit as background color information. This allows us to use 4 bits for background color, for 16 background colors.

MAGIC also detects the presence of an EGA or VGA adapter, which is necessary because the method for reprogramming the controller differs depending upon which graphics board is installed. When a CGA is present, we simply do an **OUT** to the video controller chip. When an EGA or VGA is installed, we must use a BIOS service since the EGA/VGA controller is not 100 percent compatible with the CGA controller. We detect these adapters by calling two BIOS services—the first detects VGA adapters, and the second detects both VGAs and EGAs. If neither BIOS service recognizes the installed adapter, the adapter is a CGA.

After MAGIC checks whether an EGA or VGA is installed, it displays characters in foreground colors from 0 to 15 against background colors from 0 to 7. At the point where MAGIC reprograms the video controller, the background cells beneath the blinking characters magically change to eight new background colors—and the blinking stops.

It's really a small price to pay. ■

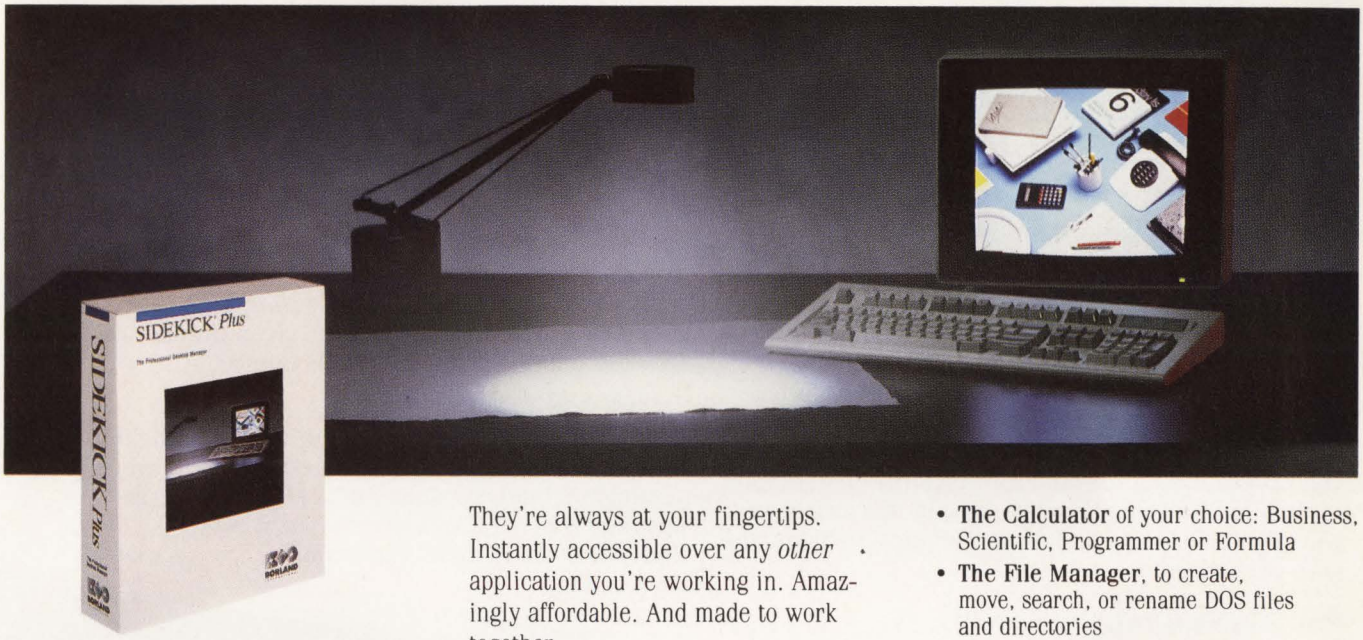
Mark Novisoff is the president of MicroHelp, Inc., and is the author of Mach 2 for Turbo Basic, an assembly language subroutine library.

Listings may be downloaded from CompuServe as COLORS.ARC.

LISTING 1: MAGIC.BAS

```
'  MAGIC.BAS by Mark Novisoff.

'  This program demonstrates how you can display a total
'  of 32 background colors on a color monitor.

'  Make sure that you "zoom" the run window out to the full screen!

  Defint A-Z
  True = -1
  False = 0
  Reg 1, &H1a00              ' This is a VGA specific BIOS call
  Call Interrupt &H10
  Al = Reg(1) Mod 256        ' Get low byte (al register)
  If Al <> &H1A Goto No.Vga  ' If Al<>&H1a then there is no VGA
  Bl = Reg(2) Mod 256        ' Get low byte
  If Bl = 7 Goto Found.Vga   ' If Bl=7 or 8, then a VGA is there
  If Bl <> 8 Goto No.Vga     ' Not there
Found.Vga:
  Vga = True
  Goto Do.display
No.Vga:                      ' No VGA, but maybe an EGA
  Reg 1, &H1200              ' EGA/VGA BIOS service
  Call Interrupt &H10
  Bl = Reg(2) Mod 256        ' Get low byte
  If Bl <> &H10 Then
    Ega = True               ' Then we have EGA
  End if
Do.display:
  Cls
  For Foreground=0 to 31
    For Background=0 to 7
      Color Foreground,Background
      Print "aaa ";
    Next
  Next
  Print
  Print
  Color 7,0
  If EGA + VGA <> 0 Goto EGA ' If either one was detected

CGA:  '--------- For CGA Adapters ----------

  Input "Press <Enter> to disable blink ", A$
  Out &H3D8, 9     ' for mono use Out &H3B8
  Input "Press <Enter> to enable blink ", A$
  Out &H3D8, &H29 ' for mono use Out &H3B8
  End

EGA:  '-------- For EGA/VGA Adapters --------
  Input "Press <Enter> to disable blink ", A$
  Reg 2,0
  Reg 1,&H1003               ' BIOS service to disable/enable blink
  Call Interrupt &H10
  Input "Press <Enter> to enable blink ", A$
  Reg 2,1
  Call Interrupt &H10
```

Lots of software packages help you work; only one helps you work smarter... SideKick Plus!



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.

To add all the productivity applications in SideKick Plus separately, you'd spend hundreds of dollars—and drain your computer's memory dry. SideKick Plus takes as little as 64K of your computer's RAM . . . *you* decide exactly how much. And *you* set up just the productivity applications you need.

They're always at your fingertips. Instantly accessible over any *other* application you're working in. Amazingly affordable. And made to work together.

Here's What You Get!

- **The PhoneBook**, complete voice and data communications that you can set to take place in the background
- **The Time Planner**, to manage your schedule with alarms, repeating appointments and more
- **The Notepad**, nine file editor Notepads, up to 11,000 words each. Use up to nine notepads at once!
- **The Clipboard**, to transfer information between files or applications with a single keystroke
- **Outlook: The Outline Processor**, to organize your thoughts
- **The Calculator** of your choice: Business, Scientific, Programmer or Formula
- **The File Manager**, to create, move, search, or rename DOS files and directories
- **The ASCII Table**, organized to help you find and paste characters fast
- **Supports both EMS and extended memory**: if you have an Intel® Above™ Board or extended memory you can load in the SideKick Plus desk accessories and leave most of your conventional memory for your other applications

Includes 5¼" and 3½" disks.
Hard disk required.

60-day money-back guarantee*

BORLAND INTERNATIONAL

For the dealer nearest you
Call (800) 543-7543

*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.

Minimum System Requirements: For IBM™ PS/2, IBM family of personal computers, and all 100% compatibles. Operating system: PC-DOS (MS-DOS®) 2.0 or later. Minimum system memory: 384K bytes. Minimum resident memory size: 64K. Hard disk required.

SIDEKICK OWNERS: GET A GREAT DEAL FROM YOUR DEALER AND A $25 REBATE FROM BORLAND!

$25 REBATE

Go to your favorite retailer for a great deal on new SideKick Plus. And, because you're a SideKick owner, we'll make it an even better deal—with a $25 rebate from Borland! To receive your rebate, you must return your completed SideKick Plus registration form from your manual, a copy of your dated SideKick Plus sales receipt, and this original completed coupon (including your original SideKick serial number.*) to:

Borland International, 4585 Scotts Valley Drive, P.O. Box 660005, Scotts Valley, CA 95066-0005

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____

Original SideKick Serial No. _____
(must be included to process rebate)*

$25 REBATE

*If your copy of SideKick does not have a serial number, remove and return the front cover of your original SideKick manual. To take advantage of this rebate you must purchase SideKick Plus by May 31, 1988 and return the rebate request to Borland by July 31, 1988. This offer is good for one rebate per registered copy of original SideKick. Not good with any other offer from Borland. Please allow 6 to 8 weeks for delivery of rebate. Offer good in the U.S. and Canada only.

PAL'S TOOLS FOR FINANCIAL INFORMATION

Money is no object when you write your financial analysis programs in PAL.

Todd Freter

PROGRAMMER

No matter how obliquely we try to put it, at some level business programming is the technology of managing money. Apart from a simple program to manage a business contact file, few PAL applications for business will *not* involve financial programming of some sort. PAL is well-equipped to deal with financial matters, and in this article we will take a close look at PAL techniques involving the coin of the realm.

PAL'S FINANCIAL FUNCTIONS

Among the 126 predefined functions in PAL are 4 functions devoted exclusively to financial analysis. Like functions in other programming languages, these financial functions take arguments and return single values. PAL's financial functions are:

- Future value, **FV**: The future value of equal, regular payments on a loan or to an annuity fund for a certain number of time periods.
- Payment, **PMT**: The periodic payment amount to pay off an amortized loan.
- Present value, **PV**: The present value of equal, regular payments on a loan or withdrawals on an investment for a certain number of time periods.
- Net present value, **CNPV**: The value in current money of future cash flows associated with an investment.

These four financial functions fall into two categories: single-record financial functions, and multiple-record financial functions.

SINGLE-RECORD FINANCIAL FUNCTIONS

The *single-record* financial functions **FV**, **PMT**, and **PV** perform operations on one or more data items as arguments that together constitute a single-record occurrence of financial data. These functions do not operate on sets of values in a column.

Future Value, FV. This calculation demonstrates the cumulative result of making payments to an annuity fund by showing what those payments, at the specified interest rate, will yield after a given number of payments. PAL's future value function is:

```
FV(Num1,Num2,Num3)
```

Num1 is a numeric expression representing the periodic payment, **Num2** is a numeric expression representing the interest rate per period for which each payment is made (expressed as a decimal), and **Num3** is a numeric expression representing

the number of periods during which payments are made.

Note that each of these arguments is a numeric expression. As in other programming languages, functions in PAL allow you to supply expressions that evaluate to numbers.

PAL calculates future value with the formula:

```
FV = p*((1+r)n-1)/r
```

p is the payment amount, **r** is the nominal interest rate, and **n** is the number of payment periods.

For instance, to calculate the future value of an annuity with a 9.6 percent annual interest rate to which you make quarterly payments of $325.39 for 15 years, run this PAL script:

```
MESSAGE FV(325.39,0.024,60)
SLEEP 10000
```

The **MESSAGE** command displays the result of the calculation. The **SLEEP** command gives you 10 seconds to see that in 15 years you'll be sitting on the tidy sum of $42,700.87 (which isn't bad, considering that your total outlay is 60 payments of $325.39, or $19,523.40).

You can make this script more universal and friendly by using display statements and variables to contain user input for the function's arguments. The modified script is given in Listing 1, FUTURE.SC.

Payment, PMT. This calculation determines the amount that must be paid every period to pay off an amortized loan for a given number of periods at a specified interest rate. An amortized loan is a mortgage-type loan, in which interest and principal are paid off together, unlike consumer loans, such as credit accounts or automobile loans. You cannot use **PMT** to determine the payment on non-amortized loans. PAL's payment function is:

```
PMT(Num1,Num2,Num3)
```

Num1 is a numeric expression representing the amount of the loan (principal), **Num2** is a numeric expression representing the effective interest rate per period for which each payment is made (expressed as a decimal), and **Num3** is a numeric expression representing the number of periods during which payments are made.

PAL calculates the payment with the formula:

```
PMT = P*(r/(1-(1+r)-n))
```

P is the principal loan amount, **r** is the nominal interest rate per period (*not* per year!), and **n** is the

continued on page 122

continued from page 121

number of payment periods in the term of the loan.

For instance, to determine the monthly payment for a home equity loan of $30,000 at 10.5 percent per year for 15 years, with payments due at the end of each month, run the following script:

```
MESSAGE
  PMT(30000,(.105/12),(15*12))
SLEEP 10000
```

The monthly payment will be $331.62. To calculate the total cash outlay of the loan (principal and interest, exclusive of loan fees), multiply the payment amount by the number of payment periods. In this case, $331.62 × 180 = $59,691.60.

You can easily generate a script that accepts user input for the **PMT** function's arguments and displays the value that **PMT** returns.

Present Value, PV. This function represents the initial or principal value of an amortized loan. **PV** tells you how high a mortgage you should apply for, based upon the payment you can afford to make, the effective interest rate, and the number of payment periods in the loan contract. PAL's present value function is:

```
PV(Num1,Num2,Num3)
```

Num1 is a numeric expression representing the periodic payment amount, **Num2** is a numeric expression representing the effective interest rate per period for which each payment is made (expressed as a decimal), and **Num3** is a numeric expression representing the number of periods in which payments are made.

PAL calculates present value with the formula:

```
PV = p*((1-(1+r)-n)/r)
```

p is the principal loan amount, **r** is the nominal interest rate per period, and **n** is the number of payment periods in the term of the loan.

For instance, if you can afford a monthly mortgage payment of $675.00, and your research shows the two best loans are for 11.5 percent for 15 years or 12.25 percent for 30 years, run the following script to determine the amount of principal for which you can apply under either loan:

```
x = PV(675, (0.115/12), (15*12))
y = PV(675, (0.1225/12), (30*12))
? "Apply for ", x,
  " at 11.5% for 15 years."
? "Apply for ", y,
  " at 12.25% for 30 years."
SLEEP 10000
```

A multiple-record function is the natural companion of a relational database system, which stores data in tables made up of rows (records) and columns (fields).

The result tells you to apply for $57,781.71 on the 15-year loan and $64,414.76 on the 30-year loan. Now use the **FV** function to determine the relative costs of both loans (the quantity of principal plus interest).

MULTIPLE-RECORD FINANCIAL FUNCTION

PAL's *multiple-record* financial function **CNPV** performs an operation on values that together constitute a multiple-record occurrence of financial data. A multiple-record function that operates on a set of like values is the natural companion of a relational database management system, which stores data in tables made up of rows (records) and columns (fields). **CNPV** performs operations such as the calculation of the average value in a set of values. **CNPV**'s name begins with a C to indicate that it operates on a set of values in a column. This distinction reflects the nature of the Paradox database in which the financial information is stored.

Net Present Value, CNPV. This function evaluates investment opportunities based upon the time value of money being invested. *Net present value* analyzes an investment based on a series of real or projected cash flows per time period, as a function of the interest paid to finance the investment. If the net present value calculation returns a positive value, the investment should be profitable. PAL's net present value function is:

```
CNPV(TableName,FieldName,Number)
```

TableName is a string (not an expression) representing the name of the table with a numeric field whose entries represent investment cash flows, **FieldName** is a string (not an expression) representing the name of the numeric field in **TableName** containing cash flows for regular periods, and **Number** is a numeric expression (expressed as a decimal) representing the effective interest rate per period for which each payment is made. Passing the name of a non-numeric field in **FieldName** causes a PAL script error.

PAL calculates net present value with the formula:

```
CNPV = sum(p=1 to n) of Vp/(1+r)p
```

p is the period associated with a cash flow amount, **Vp** is the cash flow in the **p**th period, **r** is the nominal interest rate per period, and **n** is the number of payment periods for which there is cash flow.

Consider an apartment building, with income from its units

PERIOD	Month	NetCash
1	Jan	-200
2	Feb	2000
3	Mar	-150
4	Apr	1000
5	May	600
6	Jun	750
7	Jul	200
8	Aug	-175
9	Sep	450
10	Oct	1200
11	Nov	-75
12	Dec	-150

Figure 1. A table of cash flow values for one year. Each value consists of income plus expenses for a particular month of the year in question.

and costs associated with each apartment or for the building as a whole. Let's assume you can secure a loan at 11.75 percent to finance the apartment building. Let's also assume that you can project the monthly cash flow based on rental income and expenses over the next twelve months, as shown in the Paradox table in Figure 1. To evaluate the net present value of this investment's cash flow at 1.15 percent per month, run the PAL script NETPV.SC in Listing 2. NETPV.SC evaluates the cash flow for the investment and determines its net present value. If the result of the calculation is positive, the script embeds it in an encouraging message; if negative, the script embeds it in a cautionary message. In this case, the net present value is $5138.68, and the program exhorts you to invest. Of course, much more information goes into a decision to invest money, but net present value is a useful tool for evaluating the cash flow projections of the investment.

PAL's net present value calculation presumes two things:

- The initial cash flow before payments start is zero; and
- Payments are made at the end of the payment period.

The apartment building example assumes that you buy the property with no down payment. If the investment involves an initial cash flow **I**, then you simply add it to the calculated net present value:

```
I+CNPV(TableName,FieldName,Number)
```

If you make a down payment, it must be added as an initial (negative) cash flow to the net present value. If the down payment is greater than $5138.68, then your investment in the building may not be profitable for the first year. You can continue projecting future cash flows and include them in a net present value calculation to determine when the investment will become profitable.

Net present value *analyzes an investment based on a series of real or projected cash flows per time period, as a function of the interest paid to finance the investment.*

BUYERS

Name	MaxDownPmt	MaxMonthlyPmt
Feldman	35000.00	1900.00
Ruzicka	19500.00	1750.00
Grijalva	22000.00	1750.00
Chung	44000.00	1750.00
Bedrosian	29000.00	1750.00

Figure 2. A table listing the maximum possible down payment and monthly payment for a list of home buyers.

USING PAL'S FINANCIAL FUNCTIONS

Let's incorporate PAL's financial functions into a simple real estate application. Based on a minimum of information about buyers, agents can identify optimal financing for a home purchase using either or both of two simple PAL scripts: Listing 3, DOWNPMT.SC; and Listing 4, PAYMENT.SC.

These scripts work with two tables, **buyers** and **lenders**. The **buyers** table, as shown in Figure 2, contains three fields:

- **Name** identifies the buyer (or buyers) with a single surname.
- **MaxDownPmt** contains the maximum down payment the buyer(s) can afford.
- **MaxMonthlyPmt** contains the maximum payment the buyer(s) can manage each month.

The **lenders** table, as shown in Figure 3, has six fields:

- **Lender** identifies a lending institution by name.
- **DownPct** contains the percentage of the purchase price required as a down payment for the loan.
- **Rate** contains the fixed rate of interest on the loan expressed as a percentage.
- **Term** contains the term of the loan expressed in years.
- **Points** contains the number of points that the borrower must pay for the loan.
- **Fees** contains the loan fee for the particular loan.

continued on page 124

LENDERS

Lender	DownPct	Rate	Term	Points	Fees
Barney's Mortgage	17.5	12.25	30	1.5	500.00
Barney's Mortgage	20	11.75	15	1.5	500.00
Honest Abe's Loans	15	13	30	2	450.00
Honest Abe's Loans	15	12.5	15	2	450.00
Debtor's Bank	20	11.75	15	1.5	550.00
Debtor's Bank	20	12.5	30	1.5	550.00

Figure 3. A table of loan term information from various lenders.

LISTING 1: FUTURE.SC

```
@0,0 ?? "Enter payment amount:"
@0,36 ACCEPT "n" TO amount
@1,0 ?? "Enter annual rate:"
@1,36 ACCEPT "n" TO rate
@2,0 ?? "Enter term (number of years):"
@2,36 ACCEPT "n" TO term
@3,0 ?? "Enter number of payments per year:"
@3,36 ACCEPT "n" TO payments ; e.g., to convert yearly rates
                             ; for monthly or quarterly payments
answer = FV(amount, rate/100/payments, term*payments)
@5,0 ?? "If you pay $", amount, " ", payments,
    " times per year at ", rate, "% interest for ", term,
    " years,"
@6,0 ?? "the future value will be $", answer, "."
SLEEP 10000
```

LISTING 2: NETPV.SC

```
x = CNPV("building", "NetCash", .0115)
IF x > 0
  THEN MESSAGE "Net present value is ", x, ", so go for it!"
  ELSE
    MESSAGE "Net present value is ", x , ", so don't touch it!"
ENDIF
SLEEP 10000
```

LISTING 3: DOWNPMT.SC

```
password "onlynames" ; permit access to Name field only
view "buyers"        ; put buyers table on workspace
moveto field "Name"  ; move cursor to Name field
wait table           ; display protected table to user so
                     ; that user can select name of buyer
  prompt "Move cursor to name and press F2 for report."
until "F2"           ; select name where cursor is
n = [Name]           ; store current value in n
                     ; withdraw Name-only access to buyers
                     ; table and clear all passwords
unpassword "onlynames"
menu {Tools} {More} {Protect} {ClearPasswords}
password "readfields" ; permit access to all fields
view "buyers"        ; put buyers table on workspace again
moveto field "Name"  ; move cursor to Name field
locate n             ; find the name browsed and chosen by user
                     ; create output file based on name
                     ; of user and assign file name to f
f = "c:" + n + ".dwn"
x = [maxdownpmt]     ; assign buyer's MaxDownPmt to x
y = [maxmonthlypmt]  ; assign buyer's MaxMonthlyPmt to y
                     ; Write report header to file that
                     ; identifies buyer and stated down payment
print file f "Report for buyer: ", n, "\n\n",
  "Based on your stated maximum down payment, $", x, "\n",
  "consider the following financing:\n\n"
view "lenders"       ; put lenders table on workspace
scan "lenders"       ; perform following calculations on all
                     ; records of lenders and write results to
                     ; file, record by record -
                     ; calculate down payment corrected for fees,
                     ; points, and "reserve" fund stored in prin
prin = ((x * 100/[lenders->downpct] * .95) *
       (100 - [lenders->points])/100) - [lenders->fees]
                     ; with correct loan amount (prin), use PMT
                     ; function to calculate the monthly payment
pay = pmt(prin, ([lenders->rate]/100)/12, [lenders->term]*12)
if pay <= y          ; if buyer can afford this monthly payment
  then               ; then write that information to file
  print file f "You can finance $", prin, " at ",
    [lenders->rate], "% for \n", [lenders->term],
    " years with a monthly payment of $", pay,
    " from \n", [lenders->lender], ".\n"
```

PAL FINANCIAL TOOLS

continued from page 123

Because the **buyers** table contains sensitive information about each buyer's finances, passwords are applied so that only the **Name** column can be browsed with the **WAIT TABLE** command. In this way, when either script is played, the Grijalvas will not see how much the Feldmans can afford to put down on a house. Once the buyer's name is selected, a different password allows the script to read the values that the user cannot directly access.

Because the buyers table contains sensitive information, passwords are applied so that only the Name column can be browsed with the WAIT TABLE command.

Two simple PAL scripts match prospective buyers with realistic financing. Listing 3, **Downpmt**, evaluates each of the lenders' plans in terms of the buyer's stated down payment. The program then determines how much money the buyer could borrow on the basis of the down payment under each plan, and tests whether the buyer can afford the monthly payments for the plan, based on the buyer's stated budget.

Listing 4, **Payment**, evaluates each of the lenders' plans in terms of the buyer's stated monthly payment. **Payment** then deter-

mines how much money the buyer could borrow on the basis of the monthly payment under each plan, and tests whether the buyer can qualify for the loan, based on the buyer's down payment.

More elaborate formatting of the output report is always an option, and should be added to any commercial application.

Both scripts assume that the buyers have not included loan points and fees in estimating their down payments. The scripts also reduce the buyer's stated down payment by five percent to accommodate unanticipated expenses that often crop up in the purchase of a home. (This five percent adjustment is arbitrary and can be modified as the PAL programmer sees fit.) **Downpmt** and **Payment** write the results of their evaluations into DOS files with simple formats. The formatting of this information is simple to keep the scripts short and their use of PAL's financial functions clear. More elaborate formatting is always an option and should be added to any commercial application.

These scripts illustrate how PAL's financial functions can provide the heart of a convenient script that supports real financial and commercial situations. ■

Todd Freter is Senior Writer/Editor at Ansa Software.

Listings may be downloaded from CompuServe as PALFIN.ARC.

```
  else                ; if buyer cannot afford this, write also
    print file f "The monthly payment $", pay, " at ",
    [lenders->rate], "% for \n", [lenders->term],
    " years from ", [lenders->lender], " exceeds your stated \n",
    "budget of $", [buyers->maxmonthlypmt], ".\n"
endif                 ; close affordability test
endscan               ; conclude scan operation
                      ; withdraw read-only access to buyers
unpassword "readfields"
clearall              ; clear both tables from workspace
```

LISTING 4: PAYMENT.SC

```
password "onlynames" ; permit access to Name field only
view "buyers"        ; put buyers table on workspace
moveto field "Name"  ; move cursor to Name field
wait table           ; display protected table to user so
                     ; that user can select name of buyer
  prompt "Move cursor to name and press F2 for report."
until "F2"           ; select name where cursor is
n = [Name]           ; store current value in n
                     ; withdraw Name-only access to buyers
                     ; table and clear all passwords
unpassword "onlynames"
menu {Tools} {More} {Protect} {ClearPasswords}
password "readfields" ; permit access to all fields
view "buyers"        ; put buyers table on workspace again
moveto field "Name"  ; move cursor to Name field
locate n             ; find the name browsed and chosen by user
                     ; create output file based on name
                     ; of user and assign file name to f
f = "c:" + n + ".pmt"
x = [maxdownpmt]     ; assign buyer's MaxDownPmt to x
y = [maxmonthlypmt]  ; assign buyer's MaxMonthlyPmt to y
                     ; Write report header to file that
                     ; identifies buyer and stated monthly payment
print file f "Report for buyer: ", n, "\n\n",
  "Based on your stated maximum monthly payment, $", y, "\n",
  "consider the following financing:\n\n"
view "lenders"       ; put lenders table on workspace
scan "lenders"       ; perform following calculations on all
                     ; records of lenders and write results to
                     ; file, record by record -
                     ; calculate down payment corrected for fees,
                     ; points, and "reserve" fund stored in prin
prin = pv(y, ([lenders->rate]/100)/12, [lenders->term]*12)
                     ; calculate the loan amount for which buyer
                     ; could afford the monthly payments
downpay = (.95 * prin * [lenders->downpct]/100 * (100 -
          [lenders->points])/100) - [lenders->fees]
                     ; calculate down payment corrected for fees,
                     ; points and "reserve" based on the loan
                     ; amount that the buyer's monthly payment
                     ; could carry
if downpay <= x      ; if buyer can afford this down payment
  then               ; then write the information to the file
  print file f "You can finance $", prin, " at ",
    [lenders->rate], "% for \n", [lenders->term],
    " years with a down payment of $", downpay, "\n",
    " and a monthly payment of $", y, " from ",
    [lenders->lender], ".\n"
  else               ; if buyer cannot afford this, write also
    print file f "The required down payment of $", downpay,
    " to finance ", prin, " at ", [lenders->rate],
    "% for \n", [lenders->term], " years from ",
    [lenders->lender], " exceeds your stated \n",
    "maximum for a down payment of $", x, ".\n"
endif                ; close affordability test
endscan              ; conclude scan operation
                     ; withdraw read-only access to buyers
unpassword "readfields"
clearall             ; clear both tables from workspace
```

BUILDING AN ADDRESS DATABASE WITH SPRINT

Build a database right into your word processor—in the word processor's own language.

Neil Rubenking

PROGRAMMER

Sprint is subtitled "The Professional Word Processor," but it's much more than that—Sprint's macro language is as powerful as any programming language. If you're familiar with C or Forth, you'll recognize parts of Sprint's macro language. However, other parts of the language are designed especially for word processing, and are quite different. For example, Sprint has 16 *Q-registers* instead of string variables; each of these registers holds any amount of text up to an entire document.

Besides the obvious word processing functions, Sprint's macro language invokes software interrupts, and reads or writes any memory location or port. As an example, you can force Caps Lock on with this Sprint macro:

```
40h->PeekSeg (Peek 17h | (1 << 6)) -> Peek 17h
```

This get-to-the-hardware power lets Sprint take full control of the PC when necessary. In the process, Sprint accomplishes things that one would not expect of a word processor, such as the creation and maintenance of a macro-based address database.

THE DATABASE

The macro file DATABASE.SPM (Listing 1) uses Sprint's menuing and text manipulation abilities to create a handy address database that you can load with names of frequent correspondents. This database macro lets you insert name and address information into your current document by choosing the name you want from a pop-up Sprint menu. Maintenance of the database is handled by loading and running DATABASE.SPM through the **M**acros item on the **U**tilities menu (Alt-U); this displays a menu with the two options, **A**dd New and **D**elete. Since deleting doesn't make sense until the database contains entries, I'll discuss **A**dd New first.

ADDING A NEW ENTRY

When you select the **A**dd New menu entry, the **AddNew** macro asks you for information via a prompt in the highlighted bar at the bottom of Sprint's screen. **AddNew** first requests the name that is to appear on the menu, and then prompts you to specify three lines of text to be associated with that name. This text can be a person's name plus two lines of address information, or any other three lines of text—it's up to you. Now for the magic—the Sprint code modifies itself to include the new name choice as part of the macro source code in the file DATABASE.SPM.

The next time you run the **database** macro, the name you previously entered appears as a new choice on the menu. When you select that name, the macro inserts the name and its three associated text lines into your document at the current cursor position. You can keep adding names to the macro—when the macro contains more names than can fit on the screen, simply page through the additional choices as you would with any other Sprint menu. An onscreen menu produced by the **database** macro is shown in Figure 1. The three-line address at the top of the document was inserted by pressing the Enter key when the corresponding name was highlighted.

HOW IT WORKS

Sprint is a new language to almost everyone, so I'll walk through Listing 1 step-by-step. The file DATABASE actually holds three separate macros. The first macro, **AddNew**, adds names to the database by modifying the last macro, which is **database**. The **database** macro is a simple Sprint menu. The title of the menu follows the word "menu" itself, and parentheses enclose the menu choices. Each menu choice consists of a line that appears *in* the menu, a number of Sprint commands, and a comma to separate the menu choice from the next choice. The remaining macro, **DeleteOne**, removes records from the database. Let's look at the **database** macro after one name has been added to it, as shown in Figure 2.

continued on page 128

Introducing Paradox 386. Optimized for 386 environments and up to 5 times faster than Paradox 2.0

Designed exclusively for 80386-based systems, Paradox® 386 runs up to 5 times faster than Paradox 2.0. It's that much faster because it accesses your full linear address space, which in English means it uses all the memory you or your company paid for when you put together your 386 systems.

With Paradox 386, the old 640K limits are a thing of the past.

It runs in the DOS environment and is a modern must for power users confronted with very large tables (tens of thousands of records or more) and/or large applications. Paradox 386 makes more ambitious mainframe-type applications feasible on the PC because it addresses up to 16 megabytes of memory.

But back to speed. Paradox processes data 32 bits at a time instead of just 16 bits at a time, so you race rather than run.

Paradox 2.0 and 386. The same but different

Paradox 386 gives you all the functionality, power, and flexibility that earned Paradox the Software Digest top-rating for PC relational databases.* Like Paradox 2.0, the all-new Paradox 386 gives you the speed and power to:

- Query multiple tables simultaneously
- Use an unlimited number of selection criteria
- Do pattern matches and relational operations on tables with up to 2 billion records

And Paradox 386 makes fast work of big operations like:

- Retrieving data with large single or multi-table queries
- Paging through screens of information as you view large tables
- Sorting tables with several thousand records
- Developing applications with the Paradox Personal Programmer 386
- Running large memory-intensive applications

Paradox 386 also lets you share data on local area networks with workstations running different network versions of Paradox.

Make the most of your fast new 386 hardware with our fast new 386 software

Get up to speed with Paradox 386. Your 386-based system has the racing engine, Paradox 386 is the racing fuel.

For information on upgrading from Paradox 2.0 to 386, call (800) 543-7543.

Works with the Intel® Inboard™

*60-Day Money-back Guarantee***

For the dealer nearest you or a brochure, call (800) 543-7543

BORLAND INTERNATIONAL

*Software Digest Ratings Report, March 1986, July 1987.

**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.

BI 1203A

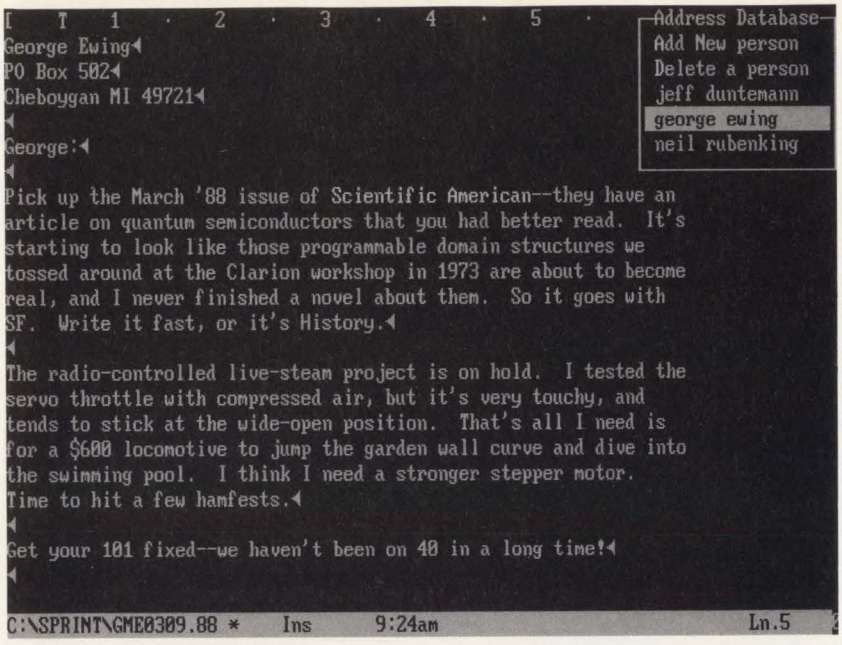


Figure 1. A Sprint screen showing the menu created by ***database****.*

```
database : menu "Address Database" (
    "Add New person " AddNew,
    "Delete a person" DeleteOne,
    "Joe"
    "Joe Jones"
    "\n123 Pleasant St."
    "\nAtown, USA\n")
```

Figure 2. The ***database*** *macro, after one name has been added.*

SPRINT DATABASE

continued from page 126

The first menu choice is still **A**dd New; selecting this choice runs the macro **AddNew**, as described before. **D**elete is still the second menu choice. However, the menu now contains a third choice, **J**oe. If you select **J**oe, Sprint executes the commands following that line up to the comma (or to the final parenthesis). In this case, the commands are text strings that Sprint inserts in the document. Note the **\n** in the last two lines—this command stands for "new line." Without this command, the macro inserts the following text into your document as one line:

```
Joe Jones123 Pleasant St.Atown, USA
```

As you add more people to the database, you'll create similar entries for each new person.

The **AddNew** macro uses one of Sprint's multiple buffers to hold the text of the macro file. With the command **2 open**, we locate DATABASE.SPM and read it into a buffer. At the end of the macro, the command **close** closes the temporary buffer and returns to the previous buffer. The **2** before **open** tells Sprint to search the path for the requested file, if that file is not found in the current directory. Naturally, the macro quits with an error message if it can't find DATABASE.SPM.

After the **open** command is issued, the text of DATABASE.SPM is stored in a buffer. Before adding a new name, we have to make sure there's enough room for it. In theory, names could be added until the menu lines total more than 4K, but Sprint's 42K macro space would probably be overrun first. To avoid running out of macro space, the database is arbitrarily limited to 75 names. We need to count the lines in the **database** macro to be sure it doesn't already contain 75 names.

First, find the beginning of the **database** macro. The string "database:" occurs three times: twice in search macros, and the third time as a macro name itself. To locate the third occurrence, use the command **3 repeat**, which makes Sprint repeat the search three times. The current line number is stored in the variable **holdnum**.

Next, locate the last nonempty line of the macro. **toend** goes to the end of the file and performs a character search in reverse for a right parenthesis, using the command:

```
r (')' csearch)
```

After the search, the line number of the found line is stored in **line**. If this line number is less than or equal to the sum of **holdnum** plus 4 lines for each of 75 entries, the macro has room for more entries. Otherwise, the program sends a message and quits.

If room is available for a new entry, delete the final parenthesis using **del**. Next, insert a comma, a new line, and several spaces in the form of this string:

```
",\n    "
```

In general, you tell a macro to insert text in a document by simply placing the text in quotes. You can also explicitly call the macro **insert**. To insert FOO, for example, you would type:

```
insert "FOO"
```

In addition, you need to use **insert** to insert the contents of a Q-register into a document. Be sure to surround the request with parentheses to make it clear that this is *not* a request to *fill* the Q-register, as in the example below:

```
(insert Q1)
```

You can't insert every character this way, though. For example, the quote character delimits strings. To insert a literal quote character, precede it with a backslash, as in \".

Input from the user is needed for the next steps. In Sprint, it's easy to read text into a Q-register. The macro **set Q0** fills **Q0** with the user's response to the prompt "Enter text:". To specify a different prompt, use the **message** macro to put the prompt on the status line first. Before all but the

second input request, clear **Q0** with the command **set Q0 ""**. Since the actual address name is almost certainly related to the menu name, leave the name in **Q0**.

Notice the string **n** in three places. The doubled slash character is no accident—in Sprint, the slash is the signal for a special character. **n** means new line, **t** means tab, and so on. The idea is to insert the two literal characters \ and **n** into the text in the Q-register, so that they can be added to the source code of the **database** macro. Sprint interprets **n** (with a single slash) as a request to insert a new line; the extra slash in **n** tells Sprint *not* to interpret the character that follows the slash.

After all the lines are written to the buffer, restore the final parenthesis and write the changed text back out to the macro file DATABASE.SPM. This step is handled by the command **Write "%"**, which tells Sprint to write the current buffer to the current filename. The last step is to activate the newly modified macro with the command **mread "database"**.

DELETING AN ENTRY

The **DeleteOne** macro lets you use any part of a name or address to select an entry to delete. Suppose you want to delete Joe Jones, but you can only remember that he lives on Pleasant Street ... or was it Pheasant? Enter "P?eas" at the prompt; when the macro finds a match, it displays the whole line and asks you to confirm. Suppose that it displays "Alice Pleasance Liddell." Since that's not the entry you want, answer the prompt with NO. The next match that you see is "123 Pleasant St.", which is the one you want. If you confirm the deletion, **DeleteOne** removes this entry and then reloads the changed macro file.

DeleteOne locates DATABASE.SPM and finds the start of the database macro within the file, just as **AddNew** does. Then **DeleteOne** prompts you for the name that you wish to delete. The search occurs within one big

continued on page 130

LISTING 1: DATABASE.SPM

```
;DATABASE Macro set
int holdnum

AddNew : ;MACRO to add a new person
    if (2 open "database.spm") {
        ; Get to the start of the database macro
        3 repeat { 1 search "database :" }
        line->holdnum
        toend r (')' csearch)
        ; Be sure there aren't TOO many already.
        ; We set a limit of 75 names
        if line <= 4*75+holdnum {
            del ",\n    "
            set Q0 ""
            message "Name to appear on menu: "
            set Q0 $
            ; to allow first letter conflicts:
            mark (to Q0 while !isend ToLower)
            "\"" (insert Q0) "\"$\n    "
            Message "Name line for address: "
            set Q0 $
            "\"" (insert Q0) "\"$\n    "
            set Q0 ""
            message "Street line: "
            set Q0 $
            "\"\\n" (insert Q0) "\"$\n    "
            set Q0 ""
            message "City,State,Zip line: "
            set Q0 $
            "\"\\n" (insert Q0) "\\n\"$)"
            write "%"
            close
            mread "database"
        }
        else {
            close
            message "\nOnly 75 entries allowed -- sorry.  "
  }
    }
    else {
        bell message "\nFile DATABASE.SPM not found"
    }
;
DeleteOne : ; MACRO to delete a person
    set Q0 ""
    message "Delete who: "
    (set Q0)
    if (2 open "database.spm") {  ; big IF
        ; Get to the start of the database macro
        3 repeat { 1 search "database :" }
        down down
```

```
        DO { ; Offer matching entries for deletion
            if !(3 search Q0) { ; 2 means allow wildcards
                bell message "Sorry, " (message Q0)
                message " NOT FOUND"
                0 wait close break
            }
            set Q2 ""
            ; Put the whole found LINE in Q2
            (tosol copy toeol Q2) tosol
            (message "Delete ")
            (message Q2)
            if ask "  (Y/N)" {  ; if yes delete
                setmark
                ; Get to the start of the matched entry
                r (',' csearch) down tosol ;; added tosol!
         ; Now delete it
                4 repeat { delete (toeol c) }
                ; If we erased the final ")", restore it.
                if !(')' csearch) {

                    toend r (',' csearch) del ")"
                }
                write "%"
                close
                mread "database"
                break
            }
            else {
              ; Not that one? Move down a line so we
              ; don't find the same one again!
                down tosol
            }
        } ; big DO
    } ; big IF
    else {
        bell message "File database.SPM not found"
    }
;

database : menu "Address Database" (
    "Add New person " AddNew,
    "Delete a person" DeleteOne)
```


continued from page 129

DO loop. Notice that **break** occurs in two places, which are the exits from the loop. The loop ends if it fails to find the string or if you confirm that the string it finds is correct. When you confirm an entry for deletion, the macro deletes all four lines. In order to do this, the macro has to locate the beginning of the entry. (Remember, the macro can match any of the entry's four lines.) The following line of commands first searches backward to the comma that ends the previous entry, and then moves down to the start of the next text line:

```
r (',' csearch) down tosol
```

The deletion of the actual text information in the macro is performed by this line of code:

```
4 repeat { delete (toeol c) }
```

Delete toeol simply deletes to the end of the line. To delete the line-end character, add the **c** to direct the macro to delete one more character past the end of visible text.

After making the deletion, **DeleteOne** saves and reloads DATABASE.SPM, just as **AddNew** did. Of course, if no name was selected for deletion, the reloading process is skipped.

GIVE IT A TRY

The **AddNew** macro is ready to roll—you can type it in or download it from CompuServe and run it. Since the names are in a standard text file, they can even be edited with Sprint. The ability to edit the macro file also allows you to export names from a separate database program, such as Paradox, and then insert them into the macro—but don't forget that 75-name limit! ■

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 SPDBAS.ARC.

BINARY ENGINEERING

How loosely are you coupled?

Bruce F. Webster

Last issue, we looked at using pre- and postconditions to implement abstract data types, and created a standalone module that implements a tic-tac-toe board. We provided a listing of the module, and used the module to write a program that plays tic-tac-toe. You can plug the tic-tac-toe module into a program of your own design, without having to copy any other portion of the tic-tac-toe program, by simply declaring in your program that you're using the module.

By contrast, I recently tried to extract the routines that handle the creation, display, and selection of menus out of a text editor's source code. This task initially appeared easy, because the highest level menu routines were set off in a separate module. It turned out, however, that those routines called procedures in several other modules; the routines also used a number of global variables that were used by still other modules. Attempts to extract those procedures and variables showed obscure linkages to yet other procedures and variables. In the end, I gave up, feeling as though I were trying to remove the entire nervous system—intact—from an animal.

The difference between last issue's tic-tac-toe module and the menu module described above lies in their degrees of coupling. *Coupling* refers to the interconnectedness between two pieces of code involving parameters, global variables, etc. Most often, one of the pieces of code is in a program, and the other piece is located in a subroutine or collection of subroutines used by that program. Coupling is usually a function of how communication takes place between the pieces of code, and which procedures are called by both pieces.

Coupling refers to the interconnectedness between two pieces of code involving parameters, global variables, and so on.

Code is often described as being "loosely coupled" or "tightly coupled." The tic-tac-toe module was loosely coupled with the tic-tac-toe program. It's an easy task to extract a *loosely coupled* module from a program for use in another program. It's also easy to write another version of that module and then drop the new version in place of the first one, as long as each version's interface is the same.

By contrast, the menu routines in the second example are *tightly coupled*, both to the editor and to one another. They make extensive use of the editor's global variables, and call many procedures used by other sections of the editor. Hence, it's almost impossible to extract the menu-handling routines in a form that can be used by any other program. Also, it's difficult to write a new implementation of the menu routines that could be used in place of the current routines.

Does this mean that loose coupling is always good, or that tight coupling is always bad? Not necessarily. In some situations, tight coupling is more convenient; in other situations, it's essential. But before we explore tight coupling further, let's look more closely at loose coupling.

LOOSE COUPLING

Imagine for a moment that you want to enhance your stereo system, which currently has several components, including an amplifier, a turntable, and a tape deck. You decide to add a compact disc player, so you connect it to the system with a couple of audio cables. When you get a two-drive cassette deck, you simply unplug the old cassette deck and plug in the new one. Finally, you decide that vinyl is passé and you remove your turntable from the system altogether.

Throughout this process your stereo system continues to work just fine. Adding the CD player gives the system new capabilities without diminishing existing ones. Swapping the tape decks increases the functionality of the tape deck subsystem. Removal of the turntable shrinks the stereo system's size and functionality, but doesn't affect any other functions in the system.

continued on page 132

continued from page 131

This stereo system is loosely coupled. The CD player (a module of the system) connects with the amplifier (and thus with the rest of the system) via two simple cables. For the most part, the CD player's make, model, and features are irrelevant to the amplifier, which only knows about the CD player's two cables that carry the sound for the amplifier to process and send to the speakers. Likewise, the CD player knows nothing about the amplifier other than that the amplifier expects two cables. The CD player has no connection at all with the speakers, even though they ultimately play (make audible) the signals.

What's the benefit of this loose coupling? As shown, you can update, modify, or rearrange your stereo system with a minimum of hassle or problems. As newer and better components come along, you can upgrade your system accordingly. (My modest system, which has been evolving for several years now, has none of its original components left.)

Getting a little closer to home, coupling applies to computer hardware as well. Consider the typical DOS-based system, which has a series of expansion slots based on a standard interface that allows you to upgrade and modify your system. The system usually has a socket for an optional math coprocessor, and you can often upgrade other aspects of the system as well, such as its disk drives.

Loose coupling works well with hardware; does it work equally well with software? Yes—but as with hardware, loose coupling in software requires some forethought and discipline. The result, though, can be well worth it.

Consider, for example, the case of a large software project that involves several programmers. Each programmer has responsibility for a section of the finished program. The software design can be separated into loosely coupled modules, with the interface for each module clearly specified. Each programmer can then work independently to develop modules according to the specifications, much as a host of companies can manufacture CD players based on the specifications for input (the standard compact disc format) and output (two channels of sound). If all the programmers follow the agreed-upon specifications, the resulting modules will—in theory, at least—easily plug together to form the finished program.

Global variables between two pieces of code tend to bond tightly; to make the code more loosely coupled, these global variables typically need to be replaced by parameters.

The same principle applies when you're doing all the programming yourself. By breaking the program into separate, loosely coupled modules, you make your overall design task easier. You also make it easier to recycle your code into other programs, since you can plug in those modules elsewhere with little or no modification.

HOW TO DO IT

We've established that loose coupling is desirable under some circumstances. The next question is: How do you achieve it?

Think again about the stereo system example. The CD player and the amplifier contain very complex electronics, yet the two are not connected with a mass of wires. Two cables suffice, since the amplifier cares only about the left and right channels coming out of the CD player.

You must similarly limit the connections in your code, because the more connections you include, the more tightly your code will be coupled. Possible connections are mutually referenced constants, data types, variables, and subroutines—any of these connections between several pieces of code can cause tight coupling, and decoupling the code can be messy. Global variables between two pieces of code tend to bond more tightly; to make the code loosely coupled, these variables typically need to be replaced by parameters.

Let's start by talking about a single subroutine. The first step in making it loosely coupled with the rest of the code is to pass all information via parameters, rather than through global variables. Consider the sort routine in Listing 1. This routine presumes that the program contains two global variables: **List** (an array of type **Integer**) and **Count** (an integer variable). It sorts the integers in **List** into ascending order, using the selection sort method. This routine is tightly coupled to the rest of the program, and it can only sort the one integer array (**List**). To use Listing 1 in another program, that program must also have an integer array named **List**, as well as an integer variable named **Count**.

Now look at the version of the sort routine in Listing 2. This version expects to be passed two parameters—an array and the number of elements in the array. Some coupling is still going on, since Listing 2 expects the global types **ListType** and **BaseType** to be declared; data types, however, tend to be less binding than variables. In fact, Listing 2 is set up so that you can declare **BaseType** to be any one of a number of types in your program—**Char**, **Byte**, **ShortInt**, **Integer**, **Word**, **LongInt**, **Real**, **Single**, **Double**, **Extended**, **Comp**, any enumerated data type, or any string type. All you must do is declare **BaseType** appropriately, with **ListType** declared as an array of **BaseType** (with some specific limit), and the procedure still works correctly. Note, however, that **ListType** has a fixed length, and that it's the only type of array

you can pass to **Sort** within the program.

Finally, look at the sort routine in Listing 3. This routine is completely decoupled from the program, and requires no global declarations whatsoever. Listing 3 is restricted to arrays of four base types: **ShortInt**, **Integer**, **LongInt**, and **Real**. The type is indicated by the **Size** parameter, which should be 1, 2, 4, or 6, respectively. If **Size** holds any other value, then the nested function **LessThan** always returns a value of **False**, leaving the array untouched. There is also an arbitrary (though very large) limit on the array size, based on constraints imposed by the Turbo Pascal compiler. The rewards, however, are twofold: you can drop this routine into any program without having to modify either the program or this procedure; and you can pass an array of any length (up to the indicated maximum) to this routine.

So much for decoupling a single procedure; what about a whole group of them? This can be either easier or more difficult, depending upon what you're decoupling. In many cases, any shared constants, data types, variables, and subroutines can all be put into a single, separate unit or module, along with the routines to be decoupled.

In the last issue, for example, we looked at **TicTac**, a unit for playing tic-tac-toe. This unit implements a constant (**GLim**), a few data types (**Move**, **Location**, **Game**), and a number of procedures and functions. **TicTac** makes no external references, and so can be dropped into any program.

We also looked at the **Moves** unit, which allows the computer to play tic-tac-toe. This unit is not as loosely coupled—it depends upon the **TicTac** unit for data types and subroutine calls. **Moves** presents two variables and one procedure to the main program; the program uses them to generate the necessary moves. The **Moves** unit, however, doesn't depend upon anything from the main program, so this unit can also be dropped into another program *if* the **TicTac** unit is in that program also.

Finally, the **GameIO** unit depends upon both the **TicTac** unit and the standard **CRT** unit—**GameIO** requires the presence of both of these units in order to function. In turn, **GameIO** presents to the main program a couple of data types and four procedures, which can be used to display the game and to prompt for moves.

Tightly coupled code—both customized and inline—is almost always faster than more general, loosely coupled code.

These units are loosely coupled to each other, and even more loosely coupled to the main program. The main program "knows" almost no details of how the tic-tac-toe game is implemented, how moves are generated, or how the game in progress is displayed. If you modify the **Moves** unit to play a more (or less) intelligent game, the program itself won't be aware of the changes.

By contrast, changing from the tic-tac-toe program's text-based display to a graphics display would require more work, since the **GameIO** unit and the main program are more tightly coupled in their mutual use of the **CRT** unit and in their presumption of a text display. To loosen the bonds, we would need to add a clear-screen procedure and a write-string procedure to **GameIO**. All game I/O would then be directed through the unit, and we could switch to a graphics display without making any changes to the main program.

TIGHT COUPLING

If loose coupling is so great, why don't you do it all the time? There are three basic reasons: speed, memory, and convenience. Let's look at each of them.

Speed. Compare the sort routines in Listings 1 and 3. Which do you think executes more quickly? Casual examination indicates that the version in Listing 1 is faster; this version has less overhead and uses fewer instructions to perform the same operation. Also, the code generated to reference parameters is often more complex than that generated to reference global variables, due to stack manipulation, indirect addressing, and similar issues. Actual tests show that the routine in Listing 1 sorts a list of 1000 random integers more than three times faster than the routine in Listing 3.

Must this always be the case? Essentially, yes. Tightly coupled code—both customized and inline—is almost always faster than more general, loosely coupled code. The issue then becomes one of tradeoffs: Is the increase in speed sufficient to justify the loss of flexibility? That's a decision you must make for yourself, case by case.

Memory. The second reason for considering tightly coupled code is memory. Again, a quick comparison of Listings 1 and 3 reveals which one produces more machine code. In addition to producing more code, general routines often require more data space as well in order to handle a wider variety of situations, especially error conditions.

Even more significantly, passing all information as parameters can create a lot of stack overhead, especially if you have a large amount of global information that is needed by many different routines. At the very least, you need to pass addresses or pointers to those data structures; this process does use less space, but can still significantly affect stack overhead, especially if you have any recursive routines.

Convenience. The last reason to use tightly coupled code is convenience. Looking at Listings 1 and 3, which do you think would be

continued on page 134

LISTING 1: SORT1.PAS

```
procedure Sort;
{
   preconditions:  List is an array[1..Limit] of Integer
                   Count is of type Word
                     and in the range 0..Limit
   postconditions: The elements 1..Count of List are sorted
                     in ascending order
}
var
  Top,Min,K,Temp : Integer;
begin
  for Top := 1 to Count-1 do begin
    Min := Top;
    for K := Top+1 to Count do
      if List[K] < List[Min]
        then Min := K;
    if Top <> Min then begin
      Temp := List[Top];
      List[Top] := List[Min];
      List[Min] := Temp
    end
  end
end; { of proc Sort }
```

LISTING 2: SORT2.PAS

```
procedure Sort(var List : ListType; Count : Word);
{
   preconditions:  BaseType is a type for which the operators
                     :=, <>, and < all are defined
                   ListType = array[1..Limit] of BaseType
                   Count is in the range 0..Limit
   postconditions: The elements 1..Count of List are sorted
                     in ascending order
}
var
  Top,Min,K : Integer;
  Temp      : BaseType;
begin
  for Top := 1 to Count-1 do begin
    Min := Top;
    for K := Top+1 to Count do
      if List[K] < List[Min]
        then Min := K;
    if Top <> Min then begin
      Temp := List[Top];
      List[Top] := List[Min];
      List[Min] := Temp
    end
  end
end; { of proc Sort }
```


continued from page 133

easier to write off the top of your head? Which appears easier to debug and get running? (As a matter of fact, I did have a bug in the routine I wrote for Listing 3: I started the three local arrays with an index of 0 instead of 1. Took me a while to track the problem down.)

Again, if your routines need to access a lot of data structures, it can be painful to have long parameter lists to every procedure and function. Of course, not all code needs to be decoupled; routines that are naturally grouped together can communicate via global (to them, at least) data structures.

Some years back, I coauthored a large, realtime, high-resolution graphics computer adventure game. The final code was around 15,000 lines of Pascal and another 5,000 lines of assembly language. Because of severe constraints on memory and a need for as much speed as possible, my set of loosely coupled units became ever more tightly coupled as time went on. Case in point: the graphics library, which started out as a general graphics library that did lots of error checking and little game-specific graphics. Once speed and memory constraints began to crop up, though, we started pruning—many safeguards were removed; code that presumed a lot about the rest of the game was added; and references were made to global data structures that were shared with the rest of the game. The final graphics library was so tightly coupled to the game itself as to be useless for any other application without very significant rewriting—but it achieved its purpose of making the game both possible and fast enough to be accepted by its users.

Each unit in the game was tightly coupled with the main program. A few very large global data structures were used to contain the complete game state. This approach, of course, made it easy to save and restore the game in progress by simply writing the

data structures to disk to be saved, then reading them back in to be restored. Passing the data structures through all the different layers of procedure calls would have been ridiculous, especially given the tight memory and speed constraints. Instead, the data structures were assumed global and were used freely by all routines.

There was, however, significant decoupling between units due to the simple fact that we couldn't fit more than a few units into memory at any one time. Each unit or set of units handled a distinct, nonoverlapping function. When that function (such as ship repair) took place, the required unit was loaded in from disk; when the function was done and another function (such as navigation) took place, the old unit was flushed out, and the new one was loaded in.

STAY LOOSE

As a general rule, loose coupling is preferable to tight coupling. Loose coupling allows for modular design, structured programming, and recycled code, and generally makes it easier to debug, maintain, and upgrade programs. Loose coupling is based largely on heavy use of parameter lists, the avoidance of communication via global variables, and the formation of separately compiled modules or units.

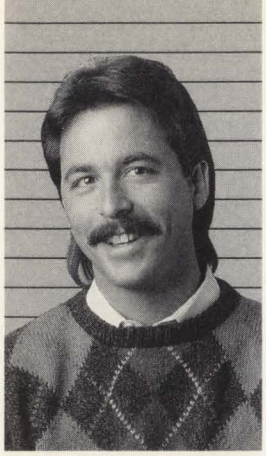
Tight coupling, even though it introduces problems, is sometimes necessary and desirable. You can generally use tight coupling to improve performance, reduce code size, and add convenience. However, these benefits need to be weighed against increased complexity in debugging, maintenance, and upgrading. Whether you decide upon loose coupling or tight coupling, you'll find that the use of good design and coding techniques does wonders. ■

Bruce Webster is a computer mercenary living in California. He can be reached via MCI Mail (as Bruce Webster) or on BIX (as bwebster).

Listings may be downloaded from CompuServe as COUPLE.ARC.

LISTING 3: SORT3.PAS

```
procedure Sort(Size,Count : Word; var List);
{
   preconditions:  Size has a value of 1, 2, 4, or 6
                   Count is in the range 0..Limit
                   List is an array of type ShortInt, Integer,
                     LongInt, or Real
                   the upper bound of Limit and the type of List
                     are determined by Size as follows:
                     Size  max value of Limit   type of List
                     ----  -----------------   ------------
                       1        64,000            ShortInt
                       2        32,000            Integer
                       4        16,000            LongInt
                       6        10,667             Real
    postconditions: The elements 1..Count of List are sorted
                     in ascending order
}
var
  SList        : array[1..64000] of ShortInt absolute List;
  IList        : array[1..32000] of Integer  absolute List;
  LList        : array[1..16000] of LongInt  absolute List;
  RList        : array[1..10667] of Real     absolute List;
  Top,Min,K    : Word;

  function LessThan(I,J : Word) : Boolean;
  begin
    case Size of
      1 : LessThan := SList[I] < SList[J];
      2 : LessThan := IList[I] < IList[J];
      4 : LessThan := LList[I] < LList[J];
      6 : LessThan := RList[I] < RList[J];
      else LessThan := False
    end
  end; { of locproc LessThan }

  procedure Swap(I,J : Word);
  var
    K          : Word;
    Temp       : ShortInt;
    ch         : char;
  begin
    I := 1 + (I-1)*Size;
    J := 1 + (J-1)*Size;
    for K := 0 to Size-1 do begin
      Temp := SList[I+K];
      SList[I+K] := SList[J+K];
      SList[J+K] := Temp
    end
  end; { of locproc Swap }

begin
  for Top := 1 to Count-1 do begin
    Min := Top;
    for K := Top+1 to Count do
      if LessThan(K,Min)
        then Min := K;
    if (Min <> Top)
      then Swap(Min,Top);
  end;
end; { of proc Sort }
```



LANGUAGE CONNECTIONS

Creating a work of art often means a blending of media—and of tools.

Michael Floyd

I like analogies. Analogies help me bridge the conceptual gap between what I know and what I'm struggling to learn. When I encountered a list in Turbo Prolog for the first time, I immediately looked for an analogous concept in a procedural language—and picked the array. At first, my analogy between Turbo Prolog's list and the array worked well. I soon found, however, that drawing parallels between procedural and declarative languages can be dangerous. Armed with my analogy, I expected the list to perform in the same way that an array performs. Instead, I discovered that an array handles programming activities that are difficult to do with a list.

These differences became more apparent when I wanted to search through a large list and needed to improve the search strategy. A binary search would have been ideal, but this search method requires individual elements to be addressed directly. However, list elements by nature are accessed sequentially. (For more information about lists, refer to "What's In a List," elsewhere in this issue).

Fortunately, with the marriage of Turbo Prolog and Turbo C, you can combine the power of list processing with the strength of arrays. In this article, we'll look at lists and arrays from both sides of the fence. We'll examine ways to represent a Turbo Prolog list in Turbo C, and see how to convert the list to an array and back to a list again. Finally, we'll present an example that sorts and searches a list using Turbo C's **qsort** and **bsearch** routines.

Editor's Note: Due to the very recent announcement of Turbo Prolog 2.0, we were unable to reflect 2.0 in this article. Note that some of the techniques described are specific to Turbo Prolog 1.x.

LISTS IN TURBO C

Internally, Turbo Prolog represents a list in memory as a linked list. Each record in the linked list contains three fields. The first field, which is one byte in size, is the list functor. The second field is the actual element in the list; the size of this field is the number of bytes that corresponds to the element type. The third field is a pointer (represented as a far pointer) to the next element in the list; this field's size is four bytes. The three fields are represented below as a data structure in Turbo C:

```
typedef struct ilist {
   char functor;
   int value;
   struct ilist *next;
   } intlist;
```

functor indicates the type of element that is referenced by the list record. If **functor** = **1**, the record is a list element; if **functor** = **2**, the record is a special value that indicates the end of the list. **value** is the actual element being referenced, and **next** is another **ilist** that references the next element in the list.

CONVERTING A LIST TO AN ARRAY

To convert a list to an array, we need to know how many elements are in the list. We must also know the data type of the list in order to allocate space for the array to be created. Once the array is created, convert the list by copying elements from the list to the array. To do this, we'll create the function **ListToArray**, which takes two arguments. The first argument is a pointer to the list to be converted, and the second argument points to the resulting array. The function itself returns an integer value that corresponds to the number of elements in the list. This example uses a **for** loop to tally the number of elements in the list:

```
int ListToArray(intlist *list,
                int **ResultArray)
{
  intlist *savelist = list;
  int *array;
  int      i = 0;
  for(i=0; list->functor == 1;
      list = list ->next)
      i++;
  array=palloc(i*sizeof(int));
  list = savelist;
  for(i=0; list ->functor==listfno;
      list=list->next)
      array[i++]=list->value;
  *ResultArray=array;
  return(i);
  }
```

Next, we must allocate space for the array on the stack. As always, use the routines provided in CPINIT.OBJ to manage the heap and stack in order to guarantee that heap space and stack space are handled using Turbo Prolog's rules. The **palloc** routine allocates space on the global stack.

Once the list has been converted, we can process the array in any manner we choose. In order to get the results back to the Turbo Prolog module, consider **ArrayToList**, which appears below. This function reverses the process and converts the array to a list:

```
void ArrayToList(int array[],int n,
                 intlist **list)
{ int i;

  for (i=0; i<n; i++)
  {
   intlist *p = *list
    = palloc(sizeof(intlist));

   p->functor = listfno;
   p->value = array[i];
   list = &(*list)->next;
  }

  {
   intList *p = *list
     = palloc(sizeof(char));
   p->functor = nilfno;
  }
}
```

ArrayToList converts an array to a Turbo Prolog list, using **palloc** to allocate memory on the global stack for each element of the list, and then builds the list by traversing through the array. Notice that the last element of the list is special—in effect, this element says, "I am the last element, because the value at **p→functor** is **nilfro**." This last element is very important and must be specified.

There is one final point: **ArrayToList** receives its list from the Turbo Prolog calling module via a pointer (to the list), and passes the result **list** back as a pointer to a pointer. Turbo Prolog gets the list from the address specified by the pointer to the list.

SORTING A LIST

As mentioned earlier, we sometimes want to search efficiently through large lists of data. With the techniques implemented so far, we can pass a list to Turbo C and convert it to an array. Now, we'll use Turbo C to search through the array and retrieve a particular item.

A number of search algorithms are available, but I prefer using the capabilities that are already built into a language. Turbo C implements the library routine **bsearch** to perform a binary search. Although a binary search expects the data to be sorted in ascending order, this is not a problem—we can use the library routine **qsort** to perform a *quicker sort* (an optimized quick sort) on the array before it's passed to **bsearch**. Let's look at the sort routine first.

In Turbo C, we'll define a function, callable from Turbo Prolog, to perform the following actions: take a list from Turbo Prolog, convert the list to an array, pass the array to **qsort**, and convert the sorted array back to a list that is then returned to Turbo Prolog. This Turbo C function is shown below:

```
void sortlist_0(IntList *InList,
                IntList **OutList)
{
  int  n, *Array;
  n = ListToArray(InList, &Array);
  qsort(&Array[0],n,sizeof(int),
        compare);
  ArrayToList(Array,n,OutList);
}
```

Since the function is being called from Turbo Prolog, that function is defined as **void**. Note that (as always) an **_0** has been appended to the function name to coincide with Turbo Prolog's naming conventions. After the variable declarations, **ListToArray** is called to perform the conversion. We assign the number of elements in the list (returned by the function) to **n**.

qsort takes four arguments: the address of the first element in the array, the number of elements in the array, the size of each element in the array, and the name of a function that is defined by the programmer. The purpose of this programmer-defined function is to compare two elements and then return a value based on the results of that comparison. **qsort** calls the comparison function successively to compare two individual elements until all of the elements in the array have been sorted. Depending on how the comparison function is written, the array can be sorted in ascending or descending order. This **compare** function sorts in ascending order:

```
int compare(void *p1, void *p2)
{
  return(*(int *)p1-*(int *)p2);
}
```

To see how **compare** is used, consider two elements to be sorted: **32** and **19**. **compare** takes **32** as the first argument and **19** as the second argument, and then takes their difference (**32 - 19 = 13**). The positive value indicates that **p1** is greater than **p2**, and the values are swapped. Table 1 shows the possible results of a comparison.

Once the array is sorted, the final task is to convert the array back to a list (**ArrayToList**) and pass the list back to Turbo Prolog.

SEARCHING A LIST

Now that we've sorted a Turbo Prolog list, let's use Turbo C's **bsearch** to search the list for particular items. Again, create a call to a function. The process is similar to that used on **sortlist**, but add an argument that specifies the element to be searched for in the list. Because there is no need to pass a list back to Turbo Prolog, the **outlist** parameter has been removed:

```
void search_0(IntList *InList,
              int *key)
{
  int  n, *Array, *found;
  n = ListToArray(InList, &Array);
  found = (int *) bsearch(&key,
          &Array[0],n,sizeof(int),
          compare);
  if (found == 0)
  {
     fail_cc();
  }
```

We give the Turbo C function some Turbo Prolog flavor by allowing **search_0** to either succeed or fail, based on the results of the search. By calling Turbo Prolog's **fail_cc** library routine, the call to **search** fails if **bsearch** does not find the element being searched for.

In PSORT.PRO (Listing 2), the **find** clause is called to search for a specified element in the array. **find** calls the Turbo C **search** function. **find** is nondeterministic, so if **search** fails, Turbo Prolog back-

continued on page 138

Items compared	Returns
*p1 < *p2	an integer < 0
*p1 = = *p2	0
*p1 > *p2	an integer > 0

Table 1. A list of possible values returned by ***compare****.*

LISTING 1: CSORT.C

```
void  _bsearch(void *key, void *base, int nelem, int width,
                        int (*fcmp)());

void    _qsort  (void *base, unsigned nelem, unsigned width,
                        int (*fcmp)(const void *, const void *));
#define qsort _qsort
#define bsearch _bsearch
#define listfno 1
#define nilfno  2

void *palloc(unsigned);

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 = palloc(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);
  }

void ArrayToList(int Array[],int n,IntList **List)
{ int i;
              /* Allocate a record for each element. */
  for (i=0; i<n; i++)
  {  IntList *p = *List = palloc(sizeof(IntList));
     p->Functor = listfno;
     p->Value = Array[i];
     List = &(*List)->Next;
  }
             /* Allocate the last record in the list. */
  {
   IntList *p = *List = palloc(sizeof(char));
   p->Functor = nilfno;
  }
}

        /* Compare two items in an array */
int compare(void *p1, void *p2)
{
  return(*(int *)p1-*(int *)p2);
}

       /* Sort a Turbo Prolog List */
void sortlist_0(IntList *InList, IntList **OutList)
{
  int  n, *Array;
  n = ListToArray(InList, &Array);
  qsort(&Array[0],n,sizeof(int),compare);
  ArrayToList(Array,n,OutList);
}
```

LANGUAGE CONNECTIONS

continued from page 137

tracks to the second **find** clause, which informs the user that the element being searched for is not in the list.

ON THE TURBO PROLOG SIDE

The program in Listing 1 (CSORT.C) implements all of the techniques discussed so far. Listing 2 (PSORT.PRO) is the Turbo Prolog module that orchestrates the sorting and searching. As always, Turbo Prolog must be the main module.

Starting with the **goal**, PSORT.PRO first calls **cpinit** to initialize the memory management routines. Then **run** is invoked to get the list items from the user. List items are asserted into the Turbo Prolog database as they are entered. Once the user terminates the list (by entering **-999**), **findall** is used to collect all the database entries into a list. The list is then passed to the **sortlist** function created earlier.

Once the list is sorted, we create a window displaying the sorted list, and pause so that the user can examine the list. The user is then prompted to enter another value, which is passed to the **find** clause. As mentioned earlier, **find** immediately calls **search** (written in Turbo C), passing the list sorted by **sortlist** and the value to search for in the list. If **search** succeeds, a message is displayed indicating that the value was found in the list. If **search** fails (via **fail_cc** in the Turbo C function), Turbo Prolog backtracks to the next **find** clause and displays a message that the element was not found.

PUTTING IT ALL TOGETHER

I won't go into all of the agonizing details of the link process here. If you read "Language Connections" regularly, you probably have this step memorized by now! If you want to know more about the link process, refer to any of the earlier "Language Connections," or consult the *Turbo C User's Guide.*

Remember to use the Large memory model when compiling

the Turbo C module. Also, don't forget to set the **G**enerate underbars compiler option **OFF**, and **U**se register variables **OFF**. Although not required, I like to set **J**ump optimization **ON**.

To link our project together, we must use a command line linker such as TLINK.EXE, which is provided on the Turbo C disk. The files to be linked (in order) are:

- INIT.OBJ—Turbo Prolog's initialization module;
- CPINIT.OBJ—Turbo C's initialization module to handle memory management by Turbo Prolog rules;
- PSORT.OBJ—the compiled Turbo Prolog module;
- CSORT.OBJ—the compiled Turbo C module;
- PSORT.SYM—the symbol table created when compiling the Turbo Prolog module;
- PROLOG.LIB—the Turbo Prolog Runtime Library; and
- CL.LIB—the Turbo C Large memory model library.

Your command line to TLINK should look something like this:

```
TLINK INIT CPINIT PSORT CSORT
      PSORT.SYM,SORT,,PROLOG+CL
```

By the way, here's one final note about linking this type of project—we can't use Turbo Prolog's project facility (in the Turbo Prolog development environment) because we must link in Turbo C's Large memory model library. Turbo Prolog does not yet let us link in other libraries.

THE RIGHT TOOLS FOR THE JOB

No single set of tools is appropriate for every task at hand. I have most of the tools necessary to work on my car, but I occasionally have to borrow an item from my neighbor's toolbox.

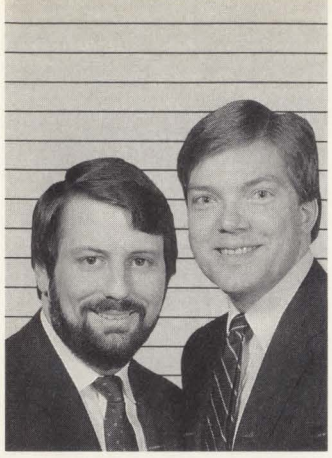
Likewise, Turbo Prolog is a powerful tool that opens the door to a wide variety of programming tasks. On occasion, however, I also reach into my collection of procedural tools. Together, this blend of procedural and declarative programming tools make up a toolbox whose creative possibilities are unequaled by either language alone. ■

Listings may be downloaded from CompuServe as PCSORT.ARC.

```
       /* Search a sorted list for a specified value */
void search_0(IntList *InList, int *key)
{
  int  n, *Array, *ptr;
  n = ListToArray(InList, &Array);
  ptr = (int *) bsearch(&key,&Array[0],n,sizeof(int),compare);
  if (ptr == 0)
  {
     fail_cc();
  }
}
```

LISTING 2: PSORT.PRO

```
DOMAINS
  list = integer*

DATABASE
  db(integer)

GLOBAL PREDICATES
  cpinit                          language c
  sortlist(list,list)  - (i,o) language c
  search(list,integer) - (i,i) language c

PREDICATES
  run
  repeat
  test_input(integer)
  find(list,integer)

GOAL
   cpinit,
   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 to sort list. */
    makewindow(2,7,7," In Turbo Prolog ",7,10,7,50),
    write(L),nl,
    readchar(_), clearwindow,
    write("Enter value to search for: "),
    readint(SVal),nl,
    find(L,SVal),        /* Search for element in list */
    readchar(_),
    removewindow, clearwindow.

  find(L,SVal):-
    search(L,SVal),  /* If search fails, backtrack to next clause */
    write(Sval," found in List").
  find(_,SVAL):-
    write(Sval," NOT found in List").

  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.
```



TALES FROM THE RUNTIME

Diving into printf

Bill Catchings and Mark L. Van Name

Two issues back, we added a wildcard expansion routine, **expwild**, to the Runtime. While from time to time you may want to add other routines, you may find that you also want to change existing ones. Changing an existing routine, however, can be much harder than adding a new one. This time around we'll demonstrate how to modify a Runtime routine by adding a new capability to an existing major function—**printf**.

printf is a familiar built-in function that produces formatted output, and requires two sets of arguments. The first set is a single quoted string that contains a series of literal constants and special formatting instructions. These formatting instructions typically consist of a percent sign (%) followed by a single letter, such as **%d** for an integer number. The second set of arguments contains the variables and constants to be formatted. **printf** matches each of these to a formatting instruction. Consider the **printf** statement:

```
printf( "This is a sample:  %d, %d.\n", i, j );
```

This statement outputs the characters up to the first format instruction (**%d**) as a literal string. It then applies the first **%d** to the contents of the integer variable **i**, and the next instruction (**%d**) to the contents of the integer variable **j**. It also prints the comma and spaces as literals.

If **i** is **1** and **j** is **2**, this **printf** statement produces the following line:

```
This is a sample:  1, 2.
```

ADDING A DATE FORMAT TO printf

The new capability that we'll add to **printf** is **%m**, a formatting instruction for dates. We chose a simple format that consists of a three-letter month abbreviation, a space, a two-digit day, a comma and space, and then a four-digit year. For example, the date 6/1/88 appears as "Jun 1, 1988." Because the date structure used by the Turbo C **getdate** routine returns a date that fits in a **long**, we made our **%m** format instruction require a **long** argument.

Listing 1 shows a sample **printf** that uses **%m**. The only unusual thing about this routine is the bit of type casting trickery that we went through to pass the contents of the **date** structure to **printf** as a **long**. Because C will not allow us to directly cast the **date** structure as a **long**, we start with the address of the structure, which is a pointer, and cast it as a pointer to a **long**. Then we get the **long** value addressed by that pointer.

By processing the date as a **long**, you can easily convert other dates to a format that our new **printf** will handle. You can also modify our new **printf** to work directly on the date structure. The DOS date function returns its values in two registers—the year is returned in one register, and the day and month are returned in the other. While you could change our code so that **%m** handles two arguments, each **printf** format directive traditionally matches one argument; we chose to follow that tradition.

Now that we've discussed how to use **%m**, let's look at the way it works. The source code for **printf** is in the file PRINTF.C in the CLIB subdirectory of the Runtime source directory structure that we set up in our first column (see "Tales From the Runtime," *TURBO TECHNIX*, November/December, 1987). So far, so good: PRINTF.C is even written in C. Look closer, however, and you'll find that PRINTF.C is a 326-byte shell that calls the function **_vprinter** (see Listing 2). If you snoop around the C language a bit, you'll find that **printf** is actually a family of three functions, which differ only with respect to where they send their output. **printf** writes to the standard output device, while **fprintf** sends output to a file, and **sprintf** puts its results into a string. All three functions use **_vprinter** for the real work. This design keeps all three **printf** versions consistent, and makes our job easier—when we change **_vprinter**, we're adding the **%m** function to all three **printf** functions.

INSIDE _vprinter

The source for **_vprinter** is located in the file VPRINTER.CAS, which contains 38K of primarily assembler code. Since VPRINTER is a .CAS file, it has the advantage of containing both C and assembler code. Because this is a C column, and because we prefer C to assembler, we wrote nearly all of our **_vprinter** changes in C.

In order to change **_vprinter**, you first need to understand how it works; that makes a dip into assembler unavoidable. The commented C code in

VPRINTER, however, explains the routine's structure nicely. We discuss some of the more interesting aspects here.

_vprinter begins with a few important compiler directives. The **#pragma inline** directive enables the use of inline assembly language and allows us to freely mix C and assembler. **_vprinter** also contains three **#include** statements. The first include file, **asmrules.h**, contains many macros that are very useful for inline assembler. The second file, **rules.h**, contains a number of general Runtime declarations. The final include file, **_printf.h**, provides the declarations of several support routines used by the **printf** family.

Because **_vprinter** is designed to support three functions that handle output in different ways, it has an unusual calling sequence. The basic format for a call to **_vprinter** is:

```
int pascal _vprinter( putnF *putter,
                      void *outP,
                      char *formP,
                      va_list argP)
```

The first unusual aspect of **_vprinter** is immediately apparent—**_vprinter** uses the Pascal calling conventions. Many internal Runtime routines use these calling conventions rather than the C rules, because the C calling sequence rules are built to handle a variable number of arguments. Routines that obey these sequence rules push parameters on the stack in right-to-left order, and then push the return address. The called routine must know how many arguments it needs, and be able to get those arguments from the stack as it needs them. In addition, the called routine cannot clean up the stack; the calling routine must clean up the stack after it regains control. Routines that obey the Pascal calling rules also push the arguments first and then the return address, but they push the arguments in left-to-right order. Because Pascal does not allow a variable number of arguments, the called routine cleans up the stack as it returns.

_vprinter follows the Pascal calling conventions and accepts a fixed number of arguments; in this case, it accepts four. The first argument, **putter**, is a pointer to a function to which **putter** passes the output string. This function prints output in the manner appropriate to the particular **printf** family member that makes this call. For example, both **printf** and **fprintf** use the function **_fputn** to write bytes to a file. The second argument, **outP**, is used by the first argument; **_vprinter** passes **outP** to the function **putter**. For **printf** and **fprintf**, **outP** is a file pointer for the file to which **_fputn** writes (**printf** places **stdout** in the address pointed to by **outP**).

The last two parameters contain the business part of **_vprinter**'s input. **formP** is a pointer to the format string, which is the first major parameter to **printf**. **_vprinter** parses this string and uses the string's directives to format the output. **argP** is a pointer to a list of the rest of the arguments; **_vprinter** retrieves each of these arguments in turn to match directives in the format string.

The basic operation of **_vprinter** is fairly simple. **_vprinter** reads the format string and outputs the literals it finds there until it encounters either a \ or

continued on page 142

LISTING 1: DATE.C

```
/* DATE.C -- routine to test our %m addition to printf. */

#include <stdio.h>
#include <dos.h>

main()
{
    struct date test;     /* the structure needed by getdate */

    getdate( &test );     /* pass it the address of that structure */
                          /* so that it can change that structure  */

    /* Our printf enhancement expects a long.  Cast the address   */
    /* of the structure as a pointer to a long, and then get that */
    /* long.  You have to go through this silliness because you   */
    /* cannot directly cast a structure as a long.                */

    printf( "%m %s\n", * (long *) &test, "-- everything works!" );
}
```

LISTING 2: PRINTF.C

```
/*  the printf family   last modified :- 18 Mar 87

Turbo C Runtime Library      version 2.0

Copyright (c) 1987 by Borland International Inc., All Rights Reserved.
*/

#include <stdio.h>
#include <_stdio.h>
#include <_printf.h>

cdecl    printf(char *fmt, ...)
    {
    return    _vprinter (_fputn, stdout, fmt, _va_ptr);
    }
```

LISTING 3: VPRINTER.CAS

```
/*      vprinter        last modified :- 18 Mar 87

Turbo C Runtime Library        version 2.0

Copyright (c) 1987 by Borland International Inc., All Rights Reserved.
*/

#pragma  inline

#include <asmrules.h>
#include <rules.h>
#include <_printf.h>

static  char    NullString[] = "(null)";

static  char    hexDigits [16] =
{
    '0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F',
};

/**** Begin addition ****/

/* Array of month names */

static char *months [12] = {
     "Jan", "Feb", "Mar", "Apr", "May", "Jun",
     "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
};

/**** End addition ****/

static  void    near pascal   Hex4 (unsigned  datum)
/*
  Convert 16 bit parameter to 4 hex digits at ES: [di].

  Note: TC does not realize that "scasb" implies DI, so DI is not
        pushed/popped.  That is nice, but one day it may cease to
        be true...
```

```
*/
{
asm     mov     dx, datum
asm     mov     cx, 0F04h
asm     mov     bx, offset hexDigits
asm     cld
asm     mov     al, dh
asm     shr     al, cl
asm     xlat
asm     stosb
asm     mov     al, dh
asm     and     al, ch
asm     xlat
asm     stosb
asm     mov     al, dl
asm     shr     al, cl
asm     xlat
asm     stosb
asm     mov     al, dl
asm     and     al, ch
asm     xlat
asm     stosb
    return;
}


/*
_vprinter is a table-driven design, for speed and flexibility.
There are two tables. The first table classifies all 7-bit ASCII
chars and then the second table is the switch table which points to
the function blocks which handle the various classes of characters.
    All characters with the 8th bit set are currently classed as
don't cares, which is the class of character also used for normal
alphabetics. All characters less than ' ' (0x20) are also classed
as don't cares.
*/
typedef enum
{
    _si,        /* sign fill ('+' or ' ') */
    _af,        /* alternate form       */
    _ar,        /* format (width or precision) by argument */
    _lj,        /* left justify         */

    _pr,        /* precision            */
    _nu,        /* numeral              */
    _lo,        /* long                 */
    _sh,        /* short                */
    _fz,        /* fill zeros           */

    _de,        /* decimal              */
    _oc,        /* octal                */
    _un,        /* unsigned decimal     */
    _he,        /* hexadecimal          */

    _pt,        /* pointer              */
    _fl,        /* float                */
    _ch,        /* char                 */
    _st,        /* string               */

    _ns,        /* number sent          */
    _zz,        /* terminator           */
    _dc,        /* dont care            */
    _pc,        /* percent              */

    _ne,        /* near pointer         */
    _fa,        /* far pointer          */
    _dt, /**** Date format   ****/
}
    characterClass;

/*  Here is the table of classes, indexed by character.
*/
static  ord8    printCtype [96] =
{
/* SP   !   "   #   $   %   &   '   (   )   *   +   ,   -   .   / */
  _si,_dc,_dc,_af,_dc,_pc,_dc,_dc,_dc,_dc,_ar,_si,_dc,_lj,_pr,_dc,

/* 0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ? */
   _fz,_nu,_nu,_nu,_nu,_nu,_nu,_nu,_nu,_nu,_dc,_dc,_dc,_dc,_dc,_dc,

/**** Change the _dc (don't care) for M to _dt (date) ****/
/* @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O */
   _dc,_dc,_dc,_dc,_de,_fl,_fa,_fl,_sh,_de,_dc,_dc,_dc,_dt,_ne,_dc,

/* P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _ */
   _dc,_dc,_dc,_dc,_dc,_dc,_dc,_dc,_he,_dc,_dc,_dc,_dc,_dc,_dc,_dc,

/**** Change the _dc (don't care) for m to _dt (date) ****/
/* `   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o */
   _dc,_dc,_dc,_ch,_de,_fl,_fl,_fl,_sh,_de,_dc,_dc,_lo,_dt,_ns,_oc,

/* p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~ DEL */
   _pt,_dc,_dc,_st,_dc,_un,_dc,_dc,_he,_dc,_dc,_dc,_dc,_dc,_dc,_dc,
};
```

FROM THE RUNTIME

continued from page 141

a %. If **_vprinter** finds a \, it gets the next character and handles the escape sequence.

The real work starts when **_vprinter** encounters a %, which indicates a format directive. **_vprinter** gets the type of the format directive and cases off that type into code that handles the directive. If **_vprinter** encounters an illegal format character, it abandons the format process and then prints the illegal format character, along with the rest of the format directive string, as literals. **_vprinter** follows this approach because it assumes that the format string and the subsequent arguments are not synchronized and can no longer be processed together.

ADDING THE FORMAT DIRECTIVE

To support our modification of **_vprinter** with a new format directive case, we've made several other, smaller changes. Listing 3 contains the source for the modified **_vprinter** routine; our changes are marked with comments that begin with **/****** and end with ******/**.

The first new code in **_vprinter** is a set of month names, in the form of a static array of three-character entries. We index into this array by using the month number minus 1 to find the name of the month we want. While all of our month names are the same length, the code handles names of different lengths, so you can also use full month names.

_vprinter represents a format directive as an enumerated data type containing both directives (**typedef enum characterClass**) and a lookup array of characters (**printCtype**). **printCtype** maps the actual format characters (such as **d**, **s**, **f**, and now **m**) to their corresponding internal directive cases (**_de**, **_st**, **_fl**, and now **_dt**, respectively). We changed both **characterClass** and **printCtype** to handle a new case for our date format, **_dt**, and added this new case to the end of the enumerated data type. We added entries for both lowercase and uppercase **m** into the character array **printCtype**. To support the **_dt** case, we also declared five new variables. Three of these variables—**year**, **month**, and **day**—are integers that hold the parts of a date. We also declared a character pointer to temporarily hold the **month** string, and another character pointer for the output string.

Before we get into our new code, we should mention something odd that happened as we worked on **_vprinter**. Our new code somehow caused a problem with some code (**_si**) in another part of the program. **_si**, which is the case that handles a sign directive, contains a **short** jump (limited to 128K) to **vpr_nextSwitch**. Since our new code doesn't fall between the jump and the destination, it should not have affected this jump; nevertheless, the **short** jump broke, probably due to the different alignment of the code. This break caused the assembly language code passed by Turbo C to MASM to generate a MASM error. The fix was easy: we turned the jump into a normal jump by removing **short** from it. If you make our other changes, be sure to make this one.

continued on page 144

```
int   pascal _vprinter (putnF  *putter,
                        void   *outP,
                        char   *formP,
                        va_list argP
       )
{

/*
The list of arguments *argP is combined with literal text in the
format string *formP according to format specifications inside
the format string.
    The supplied procedure *putter is used to generate the output.
It is required to take the string S, which has been constructed by
_vprinter, and copy it to the destination outP. The destination may
be a string, a FILE, or whatever other class of construct the caller
requires. It is possible for several calls to be made to *putter
since the buffer S is of limited size.

*putter is required to preserve  SI, DI.

The only purpose of the outP argument is to be passed through to
putter.
    The object at *argP is a record of unknown structure, which
structure is interpreted with the aid of the format string. Each
field of the structure may be an integer, long, double, or string
(char *). Chars may appear but occupy integer-sized cells. Floats,
character arrays, and structures may not appear.
    The result of the function is a count of the characters sent
to *outP.
    There is no error indication.  When an incorrect conversion spec
is encountered _vprinter copies the format as a literal (since it is
assumed that alignment with the argument list has been lost),
beginning with the '%' which introduced the bad format.
    If the destination outP is of limited size, for example a string
or a full disk, _vprinter does not know.  Overflowing the destination
causes undefined results.  In some cases *putter is able to handle
overflows safely, but that is not the concern of _vprinter.

The syntax of the format string is:

   format ::=  ([literal] ['%' conversion ])* ;

   conversion ::= '%' | [flag]* [width] ['.' precision] ['l'] type ;

   flag ::= '-' | '+' | ' ' | '#' | '0' ;

   width ::= '*' | number ;

   precision ::= '.' ('*' | number) ;

   type ::= 'd'|'i'|'o'|'u'|'x'|'n'|'X'|
            'f'|'e'|'E'| 'g'|'G'|
            'c'|'s'
            'p'|'N'|'F'
*/
#   define Ssize 80

    typedef enum
    {
        flagStage, fillzStage, wideStage, dotStage, precStage,
        ellStage, typeStage,
    }
        syntaxStages;

    typedef enum
    {
        altFormatBit = 1,       /* the '#' flag         */
        leftJustBit  = 2,       /* the '-' flag         */
        notZeroBit   = 4,       /* 0 (octal) or 0x (hex) prefix */
        fillZerosBit = 8,       /* zero fill width      */
        isLongBit    = 16,      /* long-type argument   */
        farPtrBit    = 32,      /* far pointers         */
         altOxBit    = 64,      /* '#' flag confirmed for %x format */
    }
        flagBits;
        ord16   aP;
        char    fc;         /* format char, from the format string */
        char    isSigned;   /* chooses between signed and unsigned
                               ints */
        int16   width;
        int16   precision;
        bits8   flagSet;
        char    plusSign;
        int     leadZ;
        ord16   abandonP;       /* posn of bad syntax in format
                                   string */
        char    tempStr [38];   /* longest _realcvt string */
        card16  totalSent = 0;  /* characters sent to putter */
        card8   Scount = Ssize; /* free space remaining in S */
        char    S [Ssize];      /* temporary constructed result
                                   buffer */

register        SI, DI;     /* prevent the compiler making its
                               own usage */
    int year, month, day;   /**** Holders for the parts of a
                               date ****/
        char *cP;       /**** Pointer to output string ****/
    char *monthptr;         /**** Temporary pointer ****/

#if 0   /* the remaining variables are held entirely in registers */

        char    hexCase;        /* choose upper or lower case Hex
                                   alphabet */
        long    tempL;
        syntaxStages    stage;  -- CH
        char            c;
        char            *cP;
        int             *iP;
#endif

/* General outline of the method:

First the string is scanned, and conversion specifications detected.
The preliminary fields of a conversion (flags, width, precision,
long) are detected and noted.
    The argument is fetched and converted, under the optional
guidance of the values of the preliminary fields. With the sole
exception of the 's' conversion, the converted strings are first
placed in the tempStr buffer.
    The contents of the tempStr (or the argument string for 's')
are copied to the output, following the guidance of the preliminary
fields in matters such as zero fill, field width, and justification.
*/

#if 0
/*  Warning: the following C code is comment only !  It has not
been tested.
*/

aP = 0;

#define PutToS(c) \
   (S[aP++] = c; \
    if (--Scount == 0) \
    { S[aP] = 0;  putter (S, Ssize, outP);  aP = 0;\
      totalSent += (Scount = Ssize); } )


vpNEXT:
    if ('\0' == (fc = *(formP++)))  /* the normal end */
    {
        if (Ssize - Scount)
        {
            totalSent += (Ssize - Scount);
           putter (S, Ssize - Scount, outP);
        }
        return totalSent;
    }

    if (('%' == fc) && ('%' != (fc = *(formP++))))  goto vpCONV;

    PutToS (fc);
    goto  vpNEXT;

vpCONV:
    abandonP = (unsigned) formP;
    width = -1;
    precision = -1;
    plusSign = '\0';
    leadZ = 0;
#if LDATA
    flagSet = farPtrBit;
#else
    flagSet = 0;
#endif
    stage = flagStage;
    cP = 1+tempStr;    /*tempStr [0] may be used for inserting '+'*/
    goto  vpDoSwitch;

vpNextSwitch:
    fc = *(formP++);
vpDoSwitch;
    if ((fc < ' ') || (fc & 0x80))  goto vpAbanton;
    switch (printCtype [fc-' '])
    {
        case (_af):     /* alternateForm */
            if (stage > flagStage)  goto vpAbandon;
            flagBits |= altFormatBit;
            goto vpNextSwitch;
        case (_lj):     /* leftJust */
            if (stage > flagStage)  goto vpAbandon;
            flagBits |= leftJustBit;
            goto vpNextSwitch;
        case (_si):     /* sign fill */
            if (stage > flagStage)  goto vpAbandon;
            if (plusSign != '+')  plusSign = fc;
            goto vpNextSwitch;

        case (_ne):     /* near pointer */
            if (stage > flagStage)  goto vpAbandon;
            flagBits &= ~farPtrBit;
            goto vpNextSwitch;
        case (_fa):     /* far pointer */
            if (stage > flagStage)  goto vpAbandon;
            flagBits |= farPtrBit;
            goto vpNextSwitch;
```

```
        case (_ar):     /* format by arg */
            temp = *(((int *) argP)++);
            if (stage < wideStage)
            {
                width = temp;     stage = wideStage + 1;
            }
            else
                if (stage == precStage)
                {
                    precision = temp;  stage ++;
                }
                else
                    goto vpAbandon;
            goto vpNextSwitch;

        case (_fz):     /* fillZeros */
            if (stage > flagStage)  goto vpr_NUMERAL;
            if (flagBits & leftJustBit)
                goto vpNextSwitch;              /* TB 12.may.87 */
            flagSet |= fillZerosBit;
            stage = fillzStage;
            goto vpNextSwitch;

        case (_pr):     /* precision '.' */
            if (stage >= precStage)  goto vpAbandon;
            stage = precStage;
            goto vpNextSwitch;

        case (_nu)      /* numeral */
vpr_NUMERAL:
            if (stage <= wideStage)
            {  width = (width < 0) ? fc - '0' : width*10+(fc - '0');
               stage = wideStage;
            }
            else
                if (stage != precStage)
                {
                    precision = precision * 10 + (fc - '0');
                    stage = precStage;
                }
                else
                    goto vpAbandon;
            goto vpNextSwitch;

        case (_lo):     /* long */
            flagSet |= isLongBit;   stage = ellStage;
            goto  vpNextSwitch;

        case (_sh):     /* short*/
            flagSet &= ~isLongBit;   stage = ellStage;
            goto  vpNextSwitch;

        case (_de) : radix = 10;  goto vpINT;
        case (_oc) : radix = 8;  goto vpUINT;
        case (_un) : radix = 10;  goto vpUINT;
        case (_he) : hexCase = fc - ('X' - 'a');
                     radix = 16;  goto vpUINT;
        case (_fl) : goto  vpFLOAT;

        case (_ch) :
           cP = (char *) (((int *) argP)++); cP [1] = 0; goto vpCOPY;
        case (_st) :
            cP = *(((char **) argP)++);  goto  vpCOPY;

        case (_ns) :            /* number sent */
            iP = *(((int *) argP)++);
            *iP = totalSent + Ssize - Scount;
            goto vpNEXT;

        case (_pt) : goto  vpPointer

        case (_zz) :
        case (_pc) :
        case (_dc) : goto  vpAbandon;
    }

vpUINT:
    isSigned = false;
    tempL = (flagSet & isLongBit) ? *(((long *)argP)++):
            *(((unsigned *) argP)++);
    goto  vpPUTINT;

vpINT:
    isSigned = true;
    tempL = (flagSet & isLongBit) ? *(((long *) argP)++) :
            *(((int *) argP)++);
vpPUTINT:
    notZero = tempL != 0L;
    cP = 1 + tempStr;         /* tempStr [0] reserved for sign */
    _longtoa (tempL, cP, radix, isSigned, hexCase);
    if (precision > 0)
    {
        if (precision > (len = strlen (cP) - (*cP == '-'))
            leadZ = precision - len;
        else
            precision = len;    /* ragged format is safer than lost
                                   digits */
```

FROM THE RUNTIME

continued from page 142

We've added the new case **_dt** after the last existing case that did any real work. We comment the code heavily, but a few points deserve further explanation.

We save the current position in the format string, and the format character itself, in the variables **formP** and **fc** (respectively), because the rest of the **_vprinter** cases save them. We've used assembler code because the values were already in registers, and assembler provides the quickest way to access these values.

The next block of code takes the **long** argument from **argP** that contains the date, and unpacks that date into the **year**, **day**, and **month** variables. The first word contains **year**; although DOS says that dates start from 1980, the year is actually counted from zero so we don't need to convert it. We extract the year with a bit of casting trickery: First, we cast the **argP** pointer to an integer pointer so that we can access the integer **year**. Next, we get the value of **year**, and then increment the pointer **argP**. Notice that we increment **argP** only after we have cast it as an integer, so that **argP** is incremented correctly by two bytes. We use similar tricks to retrieve the day and month, but because they are one byte each, we get them as characters and then cast them to integers. The final cast is unnecessary because C would do the conversion for us automatically, but we do it explicitly to make the intent of the code clear.

We then set our string pointer to the work area pointer, **tempStr**, that is used by the rest of the code in **_vprinter**.

From here on, our code is a fairly typical number-to-character conversion exercise. We index into our **month** array to find the month name and move that name, along with a trailing space, into the work string. We then convert the day to a character string and add the string, a comma, and a space to the work string. Finally, we convert the year to a character string and add it to our work string.

The only unusual code here involves the function **_longtoa**. This function, as its name implies, translates a **long** into an ASCII string. Its source code is in the file LTOA.CAS in the CLIB subdirectory. **_longtoa** requires five arguments. The first two arguments—the number to be translated and the destination string—are the main ones. The third parameter specifies the radix (**_longtoa** handles any radix from 2 to 36). The final two arguments specify how the sign should be handled and whether to capitalize letters if the output is hexadecimal; since we don't need either of them here, we set them both to zero. Because **_longtoa** does not update the destination string pointer to point to the end of the result, we handle that step manually after converting the day.

The last bit of our code is in assembler so that we can easily jump to the code labeled **vpr_COPY**, which is used by all of the **_vprinter** cases to handle the final output and cleanup. Our code places a

continued on page 147

```
    }
    goto  vpNUMERIC;

vpPointer:
    isSigned = false;
    cP = tempStr;

    tempL = *(((unsigned *) argP)++);
    if (flagSet & farPtrBit)
    {
        Hex4 (cP, *(((unsigned *) argP)++);
        cP += 4;
        *(cP++) = ':';
    }
    Hex4 (cP, tempL);
    tempL = (flagSet & farPtrBit) ? *(((long *) argP)++) :
            *(((unsigned *) argP)++);

    notZero = false;        /* suppress check for 0/0x/0X prefixing */
    *(cP += 4) = '\0';
    len = cP - tempStr;
    cP = tempStr;
    precision = MAX (len, precision);
    goto  vpCOPY;

vpFLOAT:
    cP = 1 + tempStr;             /* tempStr [0] reserved for sign */
    _realcvt (((double *) argP)++,
              (precision > 0) ? precision : 6,
              cP, fc, altFormat);
    notZero = false;       /* suppress check for 0/0x/0X prefixing */
    goto  vpCOPY;

vpNUMERIC:
    if (plusSign && (*cP != '-'))   *(--cP) = plusSign;

vpCOPY:
    len = strlen (cP);
    if (altFormat & notZero)
    {
        if ((fc == 'o') && (leadZ <= 0))  leadZ = 1;
        if ((fc == 'x') || (fc == 'X'))
        {
             flagSet |= altOxbit;
             width -= 2;
             if ((leadZ -= 2) < 0)       /* DM : 05/11/87 */
                  leadZ = 0;              /* DM : 05/11/87 */
        }
    }
    if (! leftJust)
        while (width > (len + leadZ))
        {
            width --;  PutToS (' ');
        }
    if (flagSet & altOxBit)
    {
        PutToS ('0');  PutToS (fc);
    }
    if (leadZ > 0)
    {
        len -= leadZ;
        width -= leadZ;
        if (((c = cP*) == '-') || (c == ' ') || (c == '+'))
            if (len > 0)  { width--;  len --;  PutToS (*(cP++)); }
        while (leadZ-- > 0)  PutToS ('0');
    }
    width -= len;
    while (len -- > 0)  PutToS (*(cP++));
    if (leftJust)
        while (width-- > 0)  PutToS (' ');
    goto  vpNEXT;

#endif



asm     push    ES
asm     cld

asm     lea     di, S
asm     mov     aP, di

/* This paragraph is arranged to give in-line flow to the most
   frequent case, literal transcription from *formP to *outP. */

vpr_NEXTap:
asm     mov     di, aP
vpr_NEXT:                     /* loop to here when DI still valid */
asm     LES_    si, formP

vpr_nextCh:                     /* resume here from this literal/
                                   space scan section */
asm     lods    BYO (ES_ [si])
asm     or      al, al
asm     jz      vpr_respondJmp

asm     cmp     al, '%'       /* The '%' character begins a
                                 conversion */
asm     je      vpr_CONV
vpr_literal:                  /* but "%%" is just a literal '%'. */
asm     mov     SS_ [di], al
asm     inc     di
asm     dec     BYO (Scount)
asm     jg      vpr_nextCh
        _SimLocalCall_
asm     jmp     vpr_CallPutter
asm     jmp     short   vpr_nextCh


vpr_respondJmp:
asm     jmp     vpr_respond


/* If arrived here then a conversion specification has been
   encountered. */

vpr_CONV:
asm     mov     abandonP, si   /* abandon will print from here */

asm     lods    BYO (ES_ [si])
asm     cmp     al, '%'
asm     je      vpr_literal

asm     mov     aP, di         /* keep the result pointer safe. */

asm     sub     cx, cx         /* CH is flagStage */
asm     mov     leadZ, cx
#if LDATA
asm     mov     BYO (flagSet), farPtrBit
#else
asm     mov     flagSet, cl
#endif
asm     mov     plusSign, cl
asm     mov     WO (width), -1
asm     mov     WO (precision), -1
asm     jmp     short   vpr_doSwitch

vpr_nextSwitch:               /* loop to here when scanning flags */
asm     lods    BYO (ES_ [si])

vpr_doSwitch:                 /* this is the major switch. */
asm     cbw
asm     mov     dx, ax        /* save original char in DL */
asm     xchg    bx, ax
asm     sub     bl, ' '
asm     cmp     bl, 128 - ' '
asm     jae     vpr_jmpAbandon
asm     mov     bl, BYO (printCtype [bx])
        switch  (_BX)         /* ===>  clobbers AX, BX  <=== */
{

vpr_jmpAbandon:               /* Extend local jump range */
asm     jmp     vpr_abandon


case (_af):                   /* when '#' was seen */
asm     cmp     ch, flagStage
asm     ja      vpr_jmpAbandon
asm     or      BYO (flagSet), altFormatBit
asm     jmp     short   vpr_nextSwitch

case (_lj):                             /* when '-' was seen */
asm     cmp     ch, flagStage
asm     ja      vpr_jmpAbandon
asm     or      BYO (flagSet), leftJustBit
asm     jmp     short   vpr_nextSwitch

case (_si):                           /* when ' ' or '+' was seen */
asm     cmp     ch, flagStage
asm     ja      vpr_jmpAbandon
asm     cmp     BYO (plusSign), 2Bh  /* '+' */
asm     je      vpr_nextSwitch       /* ' ' ignored if '+' already */
asm     mov     plusSign, dl
asm     jmp     vpr_nextSwitch       /**** Eliminate the short ****/

case (_ne):                           /* near pointer */
asm     cmp     ch, flagStage
asm     ja      vpr_abandonJmp
asm     and     BYO (flagSet), NOT farPtrBit
asm     jmp     vpr_nextSwitch

case (_fa):                             /* far pointer */
asm     cmp     ch, flagStage
asm     ja      vpr_abandonJmp
asm     or      BYO (flagSet), farPtrBit
asm     jmp     vpr_nextSwitch


case (_fz):                 /* leading width '0' acts as a flag */
asm     cmp     ch, flagStage
asm     ja      case_nu                 /* else it is just a digit */
asm     test    BYO (flagSet), leftJustBit      /* TB 12.may.87 */
asm     jnz     short   vpr_nextSwitchJmp       /* TB 12.may.87 */
asm     or      BYO (flagSet), fillZerosBit
asm     mov     ch, fillzStage          /* but it must be part of
                                           width */
asm     jmp     short   vpr_nextSwitchJmp
```

```
vpr_abandonJmp:                 /* Extend local jump range */
asm     jmp     vpr_abandon


case (_ar):                             /* when '*' was seen */
#if LDATA
asm     push    ES
#endif
asm     LES_    di, argP
asm     mov     ax, ES_ [di]        /* it causes the next argument */
asm     add     W0 (argP), 2        /*     to be taken, */
#if LDATA
asm     pop     ES
#endif
asm     cmp     ch, wideStage           /* depending on the stage, */
asm     jnb     vpr_argPrec

asm     mov     width, ax               /* as the width, */
asm     mov     ch, wideStage + 1
vpr_nextSwitchJmp:
asm     jmp     vpr_nextSwitch

vpr_argPrec:
asm     cmp     ch, precStage
asm     jne     vpr_abandonJmp
asm     mov     precision, ax           /* or as the precision. */
asm     inc     ch
asm     jmp     short   vpr_nextSwitchJmp


case (_pr):                             /* when '.' is seen */
asm     cmp     ch, precStage
asm     jnb     vpr_abandonJmp
asm     mov     ch, precStage         /* a precision should follow */
asm     jmp     short   vpr_nextSwitchJmp


/* When a numeral is seen, it may be either part of a width, or */
/*      part of the precision, depending on the stage. */

case (_nu):                             /* when 0..9 seen */
case_nu:
asm     xchg    ax, dx                  /* move char back into AL */
asm     sub     al, '0'
asm     cbw
asm     cmp     ch, wideStage
asm     ja      vpr_precNumeral

asm     mov     ch, wideStage
asm     xchg    ax, width
asm     or      ax, ax          /* is this the first width digit ? */
asm     jl      vpr_nextSwitchJmp    /* default width was -1 */
asm     mov     dx, 10
asm     mul     dx
asm     add     width, ax
asm     jmp     short   vpr_nextSwitchJmp

vpr_precNumeral:
asm     cmp     ch, precStage
asm     jne     vpr_abandonJmp

asm     xchg    ax, precision
asm     or      ax, ax       /* is this the first precision digit ? */
asm     jl      vpr_nextSwitchJmp       /* default precision was -1 */
asm     mov     dx, 10
asm     mul     dx
asm     add     precision, ax
asm     jmp     vpr_nextSwitchJmp

case (_lo):                             /*        was seen */
asm     or      BY0 (flagSet), isLongBit
asm     mov     ch, ellStage
asm     jmp     vpr_nextSwitchJmp

case (_sh):                             /* if 'h' was seen */
asm     and     BY0 (flagSet), not isLongBit
asm     mov     ch, ellStage
asm     jmp     vpr_nextSwitchJmp


/* The previous cases covered all the possible flags. Now the
   following cases deal with the different argument types.

   The first group of cases is for the integer conversions. */

case (_oc):                             /* octal        */
asm     mov     bh, 8
asm     jmp     short   vpr_NoSign

case (_un):                             /* unsigned     */
asm     mov     bh, 10
asm     jmp     short   vpr_UINT


case (_he):                             /* hex          */
asm     mov     bh, 10h
asm     mov     bl, 'A' - 'X'
asm     add     bl, dl
vpr_NoSign:
asm mov  BY0 (plusSign), 0
/*      jmp     short   vpr_UINT        */
```

```
vpr_UINT:
asm     mov     BY0 (isSigned), false
asm     mov     fc, dl          /* remember the type character. */

asm     LES_    di, argP

asm     mov     ax, ES_ [di]            /* fetch the argument.w0 */
asm     sub     dx, dx                  /* zero extend by default */
asm     jmp     short   vpr_toAscii


case (_de):                     /* decimal      */
asm     mov     bh, 10
vpr_INT:
asm     mov     BY0 (isSigned), true
asm     mov     fc, dl          /* remember the type character. */

asm     LES_    di, argP

asm     mov     ax, ES_ [di]            /* fetch the argument.w0 */
asm     cwd                             /* sign-extend by default */

vpr_toAscii:
asm     inc     di
asm     inc     di                      /* advance past arg.w0 */

asm     mov     formP, si      /* remember progress through format */

asm     test    BY0 (flagSet), isLongBit    /* short or long int ? */
asm     je      vpr_shortInt
asm     mov     dx, ES_ [di]
asm     inc     di
asm     inc     di
vpr_shortInt:
asm     mov     argP, di
asm     push    dx
asm     push    ax              /* (tempL */
asm     or      ax, dx
asm     jz      vpr_doLtoA
asm     or      BY0 (flagSet), notZeroBit
vpr_doLtoA:
#if LDATA
asm     push    SS
#endif
asm     lea     di, tempStr [1]
asm     push    di              /*      , cP == 1+tempStr */
asm     mov     al, bh
asm     cbw
asm     push    ax              /* AL ==        , radix  */
asm     mov     al, isSigned
asm     push    ax              /*                  , isSigned */

asm     push    bx              /* BL ==                , hexCase) */
asm     call    EXTPROC (_longtoa)    /* returns pointer to string */

asm     push    SS
asm     pop     ES                      /* ES_ [di] = cP == 1+tempStr */
                                        /* ES is needed in all models */
asm     mov     dx, precision
asm     or      dx, dx
asm     jg      vpr_countActualJmp
asm     jmp     vpr_testFillZeros
vpr_countActualJmp:
asm     jmp     vpr_countActual


/*
  The 'p' conversion takes either a near or a far pointer and puts
  it out in the usual Intel xxxx:xxxx hex style.
*/
case (_pt):                     /* pointer      */
asm     mov     fc, dl          /* remember the type character. */
asm     mov     formP, si       /* remember progress through
                                   format */
asm     lea     di, tempStr

asm     LES_    bx, argP
asm     push    ES_ [bx]         /* fetch the argument.w0 */
asm     inc     bx
asm     inc     bx
asm     mov     argP, bx

asm     test    BY0 (flagSet), farPtrBit
asm     jz      vpr_ptrLSW

asm     push    ES_ [bx]                /* fetch the argument.w1 */
asm     inc     bx
asm     inc     bx
asm     mov     argP, bx
asm     push    SS
asm     pop     ES
asm     call    Hex4
```

```
/*      add     di, 4                   Hex4 does this          */

asm     mov     al, ':'
asm     stosb

vpr_ptrLSW:
asm     push    SS
asm     pop     ES
asm     call    Hex4

/*      add     di, 4                   Hex4 does this          */
asm     mov     BYO (SS_ [di]), 0

asm     mov     BYO (isSigned), false
asm     and     BYO (flagSet), NOT notZeroBit

asm     lea     cx, tempStr
asm     sub     di, cx
asm     xchg    cx, di          /* CX = len,  DI = tempStr      */

asm     mov     dx, precision
asm     cmp     dx, cx
asm     jg      vpr_ptrEnd
asm     mov     dx, cx

vpr_ptrEnd:
asm     jmp     vpr_testFillZeros


/* The 'c' conversion takes a character as parameter. However, note
   that the character occupies an (int) sized cell in the argument
   list. */

case (_ch):                     /* char         */
asm     mov     formP, si       /* remember progress through format */
asm     mov     fc, dl          /* remember the type character */

asm     LES_    di, argP
asm     mov     ax, ES_ [di]
asm     add     WO (argP), 2

asm     push    SS
asm     pop     ES
asm     lea     di, tempStr [1]
asm     mov     ah, 0           /* terminate the temporary string. */
asm     mov     ES_ [di], ax
asm     mov     cx, 1
asm     jmp     vpr_CopyLen


/* The 's' conversion takes a string (char *) as argument and copies
   the string to the output buffer. */

case (_st):                     /* string       */
asm     mov     formP, si       /* remember progress through format */
asm     mov     fc, dl          /* remember the type character */

asm     LES_    di, argP
asm     test    BYO (flagSet), farPtrBit
asm     jnz     vpr_farString
#if __HUGE__
asm     jmp     vpr_abandonJmp  /*DS can't be assumed in HUGE model*/
#else
asm     mov     di, ES_ [di]    /* [di] = (DS:char *) *(argP++) */
asm     add     WO (argP), 2
asm     push    DS
asm     pop     ES
asm     or      di, di                  /*$$*/
asm     jmp     short   vpr_countString
#endif
vpr_farString:
asm     les     di, ES_ [di]    /* ES: [di] = (char *) *(argP++) */
asm     add     WO (argP), 4
asm     mov     ax, es                  /*$$*/
asm     or      ax, di                  /*$$*/

vpr_countString:
asm     jnz     NotaNullPtr             /*$$*/
asm     push    DS                      /*$$*/
asm     pop     ES                      /*$$*/
asm     mov     di,offset NullString    /*$$*/

NotaNullPtr:                            /*$$*/
        _SimLocalCall_
asm     jmp     vpr_strlen      /* CX = strlen (ES: [di]) */
asm     cmp     cx, precision
asm     jna     vpr_CopyLenJmp
asm     mov     cx, precision   /* precision may truncate string. */
vpr_CopyLenJmp:
asm     jmp     vpr_CopyLen


/* All real-number conversions are done by _realcvt. */

case (_fl):                     /* float        */
asm     mov     formP, si       /* remember progress through format */
asm     mov     fc, dl          /* remember the type character */
```


continued from page 144

string pointer in ES:DI (the standard place to store a string pointer), which points to the output string we create. Because **tempStr** is a local variable, we know that it is in our stack segment, so we set **ES** to **SS**. (We use the trick of pushing **SS** and then popping the contents into **ES** to get around the fact that Intel architectures before the 80386 treat these two registers as less than full, and will not let you move from **SS** to **ES**.) We finish by putting the address of **tempStr** into **DI** and jumping to **vpr_COPY**.

That's it! All of the **printf** family members can now handle dates. When it comes to working with existing routines, the bulk of the effort goes into understanding how the code works. Adding the new code is simple once you master the original code.

We've not yet selected the topic for our next column, and are considering everything from text processing goodies to BIOS-independent screen handling code. Do you have any suggestions? Like some radio stations, we welcome your requests. Write to us care of *TURBO TECHNIX*, and we'll see what we can do. Until then, have fun as you continue to work with the Runtime and Turbo C. ■

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 PRINTF.ARC.

```
asm     LES_    di, argP

asm     mov     cx, precision
asm     or      cx, cx          /* is precision defaulted ? */
asm     jnl     vpr_cvtReal
asm     mov     cx, 6

vpr_cvtReal:
#if LDATA
asm     push    ES
#endif
asm     push    di                      /* (valueP */
asm     push    cx                      /*     , ndec */
#if LDATA
asm     push    SS
#endif
asm     lea     bx, tempStr [1]
asm     push    bx                      /*       , cP */
asm     push    dx                      /*       , formCh */
asm     mov     ax, altFormatBit
asm     and     al, BYTO (flagSet)
asm     push    ax                      /*        , altFormat) */
asm     call    EXTPROC (_realcvt)

asm     add     WO (argP), 8            /* ((double *) argP) ++ */

asm     push    SS
asm     pop     ES
asm     lea     di, tempStr [1]      /* ES_ [di] = cP == 1+tempStr */

vpr_testFillZeros:
asm     test    BYTO (flagSet), fillZerosBit
asm     jz      vpr_NUMERIC
asm     mov     dx, width
asm     or      dx, dx
asm     jng     vpr_NUMERIC

vpr_countActual:                        /* ES must be well defined! */
        _SimLocalCall_
asm     jmp     vpr_strlen              /* CX = strlen (ES: [di]) */

asm     sub     dx, cx                  /* DX = leadZ */
asm     jng     vpr_NUMERIC
asm     mov     leadZ, dx


/* If arrived here, then tempStr contains the result of a numeric
   conversion. It may be necessary to prefix the number with a
   mandatory sign or space. */

vpr_NUMERIC:                    /* ES must be well defined !    */
asm     mov     al, plusSign
asm     or      al, al
asm     jz      vpr_COPY
asm     cmp     BYTO (ES: [di]), '-'
asm     je      vpr_COPY
asm     dec     di
asm     sub     WO (leadZ), 1
asm     adc     WO (leadZ), 0    /* don't allow negative leadZ */
asm     mov     ES: [di], al     /* *(--cP) = plusSign */

/* If arrived here then ES: [di] = cP points to the converted string,
   which must now be padded, aligned, and copied to the output. */

vpr_COPY:
        _SimLocalCall_
asm     jmp     vpr_strlen          /* CX = strlen (ES: [di]) */

vpr_CopyLen:                        /* comes from %c or %s section */
asm     mov     si, di              /* cP == ES: [si] */
asm     mov     di, aP

asm     mov     bx, width           /* BX = width */

asm     mov     al, notZeroBit + altFormatBit
asm     and     al, BYTO (flagSet)
asm     cmp     al, notZeroBit + altFormatBit
asm     jne     vpr_doLead
asm     mov     ah, fc

asm     cmp     ah, 'o'
asm     jne     vpr_maybeAltHex
asm     cmp     WO (leadZ), 0   /* alternate mode with octal format*/
asm     jg      vpr_doLead      /* requires there to be at least */
asm     mov     WO (leadZ), 1   /* one leading zero. */
asm     jmp     short   vpr_doLead

vpr_maybeAltHex:
asm     cmp     ah, 'x'          /* alternate mode with 'x' or 'X' */
asm     je      vpr_isAltHex     /*   format requires sending a  */
asm     cmp     ah, 'X'          /*    "0x" or "0X" prefix.      */
asm     jne     vpr_doLead
vpr_isAltHex:
asm     or      BYTO (flagSet), altOxBit
asm     sub     bx, 2                   /* width -= 2;  */
asm     sub     WO (leadZ), 2           /* leadZ -= 2;  */
asm     jnl     vpr_doLead              /* DM : 05/11/87 */
asm     mov     WO (leadZ), 0           /* DM : 05/11/87 */
```

```
vpr_doLead:
asm     add     cx, leadZ               /* CX = len + leadZ */

asm     test    BYTO (flagSet), leftJustBit  /* is result to be left
                                                justified? */
asm     jnz     vpr_checkOx
asm     jmp     short   vpr_nextJust   /* (! leftJust) == leftFill */

vpr_justLoop:
asm     mov     al, ' '
        _SimLocalCall_
asm     jmp     vpr_PutToS
asm     dec     bx

vpr_nextJust:
asm     cmp     bx, cx
asm     jg      vpr_justLoop

vpr_checkOx:
asm     test    BYTO (flagSet), altOxBit
asm     jz    vpr_checkLeadZ

asm     mov     al, '0'
        _SimLocalCall_
asm     jmp     vpr_PutToS
asm     mov     al, fc
        _SimLocalCall_
asm     jmp     vpr_PutToS

vpr_checkLeadZ:                 /* is leading zero fill required ? */
asm     mov     dx, leadZ
asm     or      dx, dx
asm     jng     vpr_actualCopy

asm     sub     cx, dx                  /* len -= leadZ */
asm     sub     bx, dx                  /* width -= leadZ */

asm     mov     al, ES: [si]            /* any leading sign must be */
asm     cmp     al, '-'                 /*   copied before the */
asm     je      vpr_leadSign            /*     leading zeroes. */
asm     cmp     al, ' '
asm     je      vpr_leadSign
asm     cmp     al, '+'
asm     jne     vpr_signedLead
vpr_leadSign:
asm     lods    BYTO (ES: [si])
        _SimLocalCall_
asm     jmp     vpr_PutToS
asm     dec     cx
asm     dec     bx                      /* anticipates actualCopy */

vpr_signedLead:
asm     xchg    cx, dx              /* leading zeroes follow sign */
asm     jcxz    vpr_leadDone

vpr_leadZero:
asm     mov     al, '0'
        _SimLocalCall_
asm     jmp     vpr_PutToS
asm     loop    vpr_leadZero

vpr_leadDone:
asm     xchg    cx, dx


/* Now we copy the actual converted string from tempStr to output. */

vpr_actualCopy:
asm     sub     bx, cx          /* width -= len; */
asm     jcxz    vpr_copied

vpr_copyLoop:                /*this is the high-point of _vprinter!*/
asm     lods    BYTO (ES: [si])
asm     mov     BYTO (SS_ [di]), al
asm     inc     di
asm     dec     BYTO (Scount)
asm     jg      vpr_loopTest
        _SimLocalCall_
asm     jmp     vpr_CallPutter
vpr_loopTest:
asm     loop    vpr_copyLoop


vpr_copied:

/* Is the field to be right-filled ? */

asm     or      bx, bx          /* any remaining width ? */
asm     jng     vpr_done
asm     mov     cx, bx
vpr_rightLoop:
asm     mov     al, ' '
        _SimLocalCall_
asm     jmp     vpr_PutToS
asm     loop    vpr_rightLoop
```

```
/*If arrive here, the conversion has been done and copied to output*/

vpr_done:
asm     jmp     vpr_NEXT


case (_ns) :                    /* number sent */
asm     mov     formP, si       /* remember progress through format */

asm     LES_    di, argP
asm     test    BYO (flagSet), farPtrBit
asm     jnz     vpr_farCount
#if __HUGE__
asm     jmp     vpr_abandonJmp  /*DS can't be assumed in HUGE model*/
#else
asm     mov     di, ES_ [di]    /* [di] = (DS:char *) *(argP++) */
asm     add     WO (argP), 2
asm     push    DS
asm     pop     ES
asm     jmp     short   vpr_makeCount
#endif
vpr_farCount:
asm     les     di, ES_ [di]    /* ES: [di] = (char *) *(argP++) */
asm     add     WO (argP), 4

vpr_makeCount:

asm     mov     ax, Ssize
asm     sub     al, Scount
asm     add     ax, totalSent
asm     mov     ES: [di], ax
asm     jmp     vpr_NEXTap

/**** Begin addition ****/

/* Code for date processing */

case (_dt) :
asm     mov     formP, si       /* Save progress through format */
asm     mov     fc, dl          /* Save the type character */

/* Unpack the components of the date.  Year - int, day - char,
   and month - char.  Casting is the key here.  */

    year = *(((int *) argP)++);
    day = (int) *(((char *) argP)++);
    month = (int) *(((char *) argP)++);

    cP = tempStr;  /* Get a pointer to the destination */

/* Move the appropriate month name into the destination string.
   Follow it by a space.  */

    monthptr = months[month-1];
    while ((*cP = *monthptr++) != '\0') cP++;
    *cP++ = ' ';

/* _longtoa converts a long into an ASCII string.  The first
   argument is the long, the second the destination, the third
   the radix.  The remaining two are not used here.  Convert
   the day into ASCII.  */

    _longtoa ((long) day, cP, 10, 0, 0);

/* _longtoa does not update the string pointer, so we do that by
   hand.  Add a comma and a space for the sake of neatness.  */

    cP = tempStr + strlen (tempStr);
    *cP++ = ',';
    *cP++ = ' ';

/* Convert the year to ASCII.  */

    _longtoa ((long) year, cP, 10, 0, 0);

/* vpr_COPY takes care of all the applying of arguments to
   the string we created (such as maximum length, right
   justification, etc.) and then calls the function to output
   the resulting string.  vpr_COPY requires that ES:DI point
   to the string to process, so we set it up before calling. */

asm push SS
asm pop  ES
asm lea  di, tempStr
asm jmp  vpr_COPY

/**** End addition ****/

case (_zz):  /* \0 characters, unexpected end of format string */
case (_dc):  /* ordinary "don't care" chars in the wrong position */
case (_pc):  /* '%' percent characters in the wrong position */
             /* goto vpr_abandon     */
}            /* end switch */

/* If the format goes badly wrong, then copy it literally to the
   output and abandon the conversion. */
```

```
vpr_abandon:
asm     mov     si, abandonP
#if LDATA
asm     mov     ES, W1 (formP)
#endif
asm     mov     di, aP
asm     mov     al, '%'

vpr_abandLoop:
        _SimLocalCall_
asm     jmp     vpr_PutToS
asm     lods    BYO (ES_ [si])
asm     or      al, al
asm     jnz     vpr_abandLoop


/* If arrived here then the function has finished (either correctly
   or not). */

vpr_respond:
asm     cmp     BYO (Scount), Ssize   /* anything waiting to be
                                         written? */
asm     jnl     vpr_end
        _SimLocalCall_
asm     jmp     vpr_CallPutter

vpr_end:
asm     pop     ES

    return totalSent;


/**     local, nested functions are placed here         **/
/*
        clobbers AX.  ES *must* be defined in all models.
*/
vpr_strlen:                     /* scan string ES: [DI] up to \0 */
asm     push    di
asm     mov     cx, -1          /*   count the string length. */
asm     mov     al, 0
asm repne       scasb
asm     not     cx              /* (not CX) == (-1 -CX) */
asm     dec     cx              /* scasb overshoots */
asm     pop     di
asm     pop     ax
asm     add     ax, 3         /* skip the jmp after _SimLocalCall_  */
asm     jmp     ax              /* CX = string length */
/*      RETNEAR         */

/***    Put character to next postion in S, check for S full    ***/
/*
        clobbers AX.
*/
vpr_PutToS:
asm     mov     BYO (SS_ [di]), al
asm     inc     di
asm     dec     BYO (Scount)
asm     jng     vpr_CallPutter
asm     pop     ax
asm     add     ax, 3         /* skip the jmp after _SimLocalCall_  */
asm     jmp     ax              /* CX = string length */
/*      RETNEAR         */


/**     call *putter to flush S         **/
/*
        clobbers AX.
*/
vpr_CallPutter:
asm     push    bx
asm     push    cx
asm     push    dx
asm     push    ES
asm     lea     ax, S
asm     sub     di, ax          /* count chars in S */

    putter (S, _DI, outP);

    Scount = Ssize;
    totalSent += _DI;

asm     lea     di, S
asm     pop     ES
asm     pop     dx
asm     pop     cx
asm     pop     bx
asm     pop     ax
asm     add     ax, 3         /* skip the jmp after _SimLocalCall_ */
asm     jmp     ax              /* CX = string length */
/*      RETNEAR         */


/***    end of embedded functions       ***/
}
```

ARCHIMEDES' NOTEBOOK

Choosing the most cost-effective lens design is easy with Eureka!

Milton C. Kurtz

Newton postulated that light, as a stream of particles, travels in straight lines. This theory makes optical problems solvable using simple geometry, or more specifically, by applying *geometrical optics* (see accompanying sidebar). One application particularly well-suited to geometrical optics is the lens problem. We will examine the behavior of a thin lens, and use Eureka to determine the most cost-effective way to design a thin lens for a given application.

DESCRIBING A THIN LENS

First, we will present some definitions to describe our lens.

Index of Refraction. A light ray is bent, or *refracted*, when it passes from one medium, such as air, into another medium of different density, such as glass. The ability of a medium to bend light rays is its *index of refraction*, which is a ratio of the velocity of light in a medium compared to the velocity of light in a vacuum. (This subject has been investigated thoroughly by Snell, and the refraction of light follows Snell's Law.)

Thin Lens. A *thin lens* is a lens whose thickness is small compared with its other features (e.g., focal length).

Focal Point. The *focal point* of a lens is a point on the axis having the property such that any ray coming from it, or proceeding to it, travels parallel to the axis after refraction.

Focal Length. The *focal length* of a lens is the distance between the focal point on the axis of the lens and the center of the lens.

Axis. The *axis* of the lens is a straight line passing through the geometrical center of the lens, perpendicular to the radii of curvature.

A lens has two focal points; each focal point is equidistant from the center of a symmetrical lens. The focal point defined above is the *primary focal point*; the other focal point is the *secondary focal point*. Light traveling parallel to the axis will, after refraction, proceed toward—or appear to emanate from—the secondary focal point.

If we designate f as the primary focal length in a symmetrical lens, and f' as the secondary focal length, then $f = f'$. These conditions are shown in Figure 1.

If we know the focal length of a thin lens and the position of the object being viewed through the lens, we can find the position of the object's image by one of three methods: graphical construction, experimentation, or use of the lens formula.

THE LENS FORMULA

The lens formula is readily derived from the geometry of Figure 2. The diagram shows two rays leading from the object of height y

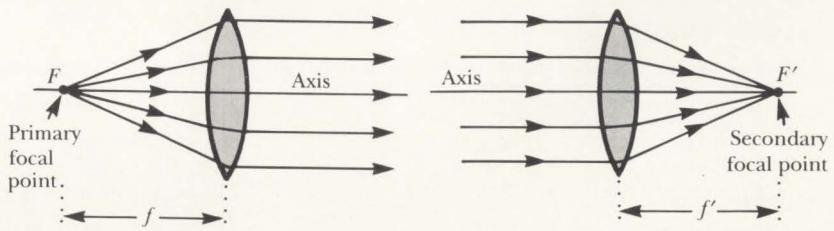
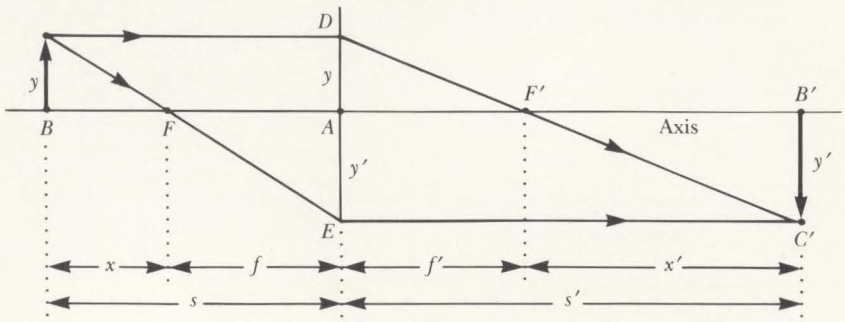


Figure 1. Two views of a thin lens with rays from and to F and F′, respectively.



to the image of height y'. Let s represent the object distance from the center of the lens and let s' represent the image distance from the center of the lens. x represents the object distance from the focal point F, and x' represents the image distance from focal point F'.

From similar triangles $C'DE$ and $F'DA$, the proportionality between corresponding sides gives:

$$\frac{y - y'}{s'} = \frac{y}{f'} \qquad (1)$$

$y - y'$ is used instead of $y + y'$ because y', by the convention of signs, is a negative quantity. Also, from the similar triangles CDE and FAE, we can derive the relationship:

$$\frac{y - y'}{s} = \frac{-y'}{f} \qquad (2)$$

The sum of these two equations yields:

$$\frac{y - y'}{s} + \frac{y - y'}{s'} = \frac{y}{f'} - \frac{y'}{f} \qquad (3)$$

Since $f = f'$, the two terms on the right may be combined and $y - y'$ canceled out, resulting in the equation:

$$\frac{1}{s} + \frac{1}{s'} = \frac{1}{f} \qquad (4)$$

This equation is the lens formula, where s is the object distance, s' is the image distance, and f is the focal length of the lens. Object distances are positive if the object lies to the left of its reference point A, and image distances are positive if the image lies to the right of reference point A.

Now that we know the relative positions of the object and image, it's easy to determine their sizes. The lateral magnification is:

$$m = \frac{y'}{y} = -\frac{s'}{s} \qquad (5)$$

When s and s' are both positive, the negative sign of the magnification denotes an inverted image.

The images formed by the lens in this exercise are real in that they form a visible image on a screen. *Real images* are formed when the rays of light are actually brought to focus in the plane of the image. Other conditions will form a virtual image, or one that cannot be formed on a screen. In a *virtual image*, the rays from a given point on the object do not come together at the corresponding point in the image. They must be projected backward to find the image plane. Virtual images are produced by converging lenses when the object is placed between the focal point and the lens. This condition is shown in Figure 3.

The power of a thin lens (in units known as diopters) is given as the reciprocal of the focal length expressed in meters.

One other point of interest is the determination of the power of a lens. The *power* of a thin lens (measured in units known as diopters [D]) is given as the reciprocal of its focal length (expressed in meters).

$$P = \frac{1}{f} \qquad (6)$$

For example, a lens with a focal length of +25 centimeters has a power of 1/0.25 meters or +4.0 D. These units are used in optometry, and can be found on your eyeglass prescription or stamped on the inside of the temple bow on reading glasses.

CHOOSING A LENS

We now have the formulas and background information that we need in order to choose a lens for use in applications such as those involving the observation of an object located at a fixed distance from the lens. A number of different focal lengths are possible and the image distance varies, so we'll set up the lens formula in Eureka to determine the image distances that pertain to a range of focal lengths. Of course, we could set up this problem in a language such as Turbo Basic, where all system constraints, as well as the variables to be considered, are called out. However, Eureka allows us to quickly model the lens problem

continued on page 152

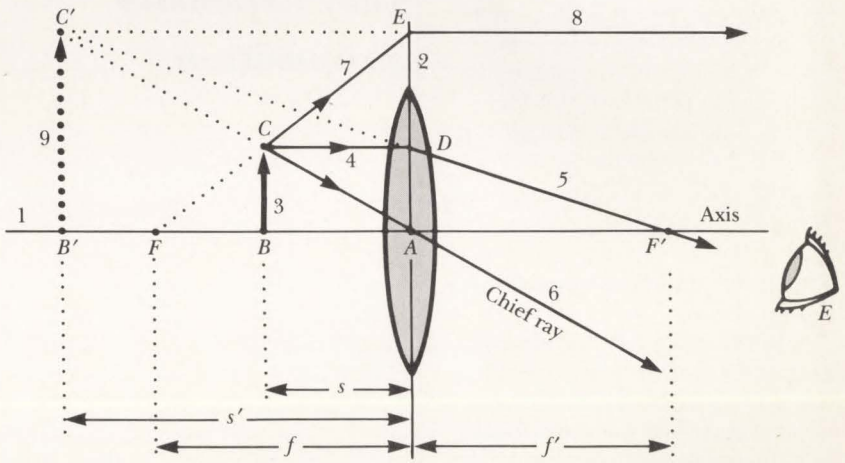


Figure 3. Placement of an object between the focal point and the lens, creating a virtual image.

continued from page 151

without worrying about programming details.

As a more concrete example, let's use an optical inspection instrument to observe a row of slots that are machined into a rod. We want to determine the width of the slots, along with the distance between them. (For simplicity's sake, we'll look only at the optics here, disregarding the usual mechanisms that are involved in holding and supporting an object.) The only constraint we'll put on the system is that we cannot get closer than six inches from the slots because of interfering mechanisms. We now have a two-part problem to solve—we need to select the best focal length in order to project an image of the slots onto an appropriate sensor, and we need to determine the location of the sensor.

This is a simple statement of the problem, but additional information must be considered (such as the resolution that we want in the measurement). In addition, only a finite number of focal length lenses is available from suppliers; others must be specially ordered (a very expensive consideration). Thus, we want to choose our lens from a standard catalog.

SOLVING IT WITH EUREKA

To set up the problem in Eureka, we'll slightly rewrite Equation 4. Let p represent the object distance, q represent the image distance, and f represent the focal length. The new lens equation is:

$$\frac{1}{p}+\frac{1}{q}=\frac{1}{f} \qquad (7)$$

This is the standard form of the lens equation.

After loading Eureka, choose **E**dit and enter the lens formula. Add the constraint **p** = 25. Next, select a series of standard focal lengths from a supplier's catalog and set f to those values. Table 1 shows a list of image distances for given values of the focal length.

Focal Length f	Image Distance q
7.0 mm	9.72 mm
8.0 mm	11.75 mm
10.0 mm	16.67 mm
12.7 mm	25.81 mm
14.2 mm	32.87 mm
15.0 mm	37.25 mm
18.0 mm	64.29 mm

Table 1. List of image distances for given values of the focal length.

```
1/p + 1/q = 1/f

p = 25

f = 7
```

```
Solution:

Variables        Values

f          =     7.0000000

p          =     25.000000

q          =     9.7222222
```

Figure 4. Solution generated by Eureka to find the image distance for a given focal length.

Only a finite number of focal length lenses is available from suppliers; others must be specially ordered, which is a very expensive proposition.

Choose the first focal length listed in Table 1, and set **f** = 7.

Now, escape from the **E**dit mode and choose **S**olve. As Eureka solves the lens equation, we find that **q** is easily obtained for each value of **f**. When **f** = 7, the image distance **q** is approximately 9.72 millimeters. Figure 4 shows a solution generated from this run.

We can easily choose an image distance for a given focal length, but how do we choose the right image distance? Equation 5 shows that the ratio of the image distance to the object distance provides a measure of the magnification of the system. Let's assume that it would be desirable to have the image be approximately two times the size of the object. Set up that ratio in Eureka, without specifying a focal length **f**:

```
q/p = 2
```

Given an object distance of 25 millimeters, we see that the ideal focal length is 16.67 millimeters, and the image distance is 50 millimeters. However, because this is a nonstandard focal length, it's not desirable. The two standard

focal lengths of 15 and 18 millimeters are close, so let's examine them more carefully.

We specified earlier that we could get no closer than 25 millimeters from the object. When we set the focal length to 15 at a magnification of 2, the object distance is 22.5 millimeters and the image distance is 45 millimeters.

Equation 5 shows that the ratio of the image distance to the object distance provides a measure of the magnification of the system.

Again, this does not meet our requirement that the object distance be at least 25 millimeters. When we change the focal length to 18 millimeters, the object distance becomes 27 millimeters, and the image distance becomes 54 millimeters. Clearly, the 18 millimeter lens meets all the requirements of the system.

The short history of optics included in the accompanying sidebar, along with this simple geometrical optics application, should help make optics a less formidable area. We did not touch on physical optics, where Eureka can play a much greater role in easing the workload of the system designer who deals with the complex equations of that field. ■

Milton C. Kurtz is a 1946 graduate of the University of Maryland. He has spent most of his career in applied science and instrumentation development. Now retired, he is a director of Emkay Engineering, Inc., of Saratoga, California.

THE DUALITY OF LIGHT

The fundamental picture of the way we view light has changed significantly during the last 300 years. Isaac Newton defined light as a stream of particles in his basic treatise *OPTICKS*, printed in 1704. Because of his stature as a scientist, his hypotheses were supported by his contemporaries and successors. This *corpuscular theory of light* was generally accepted for almost a century. However, Newton also noticed that light exhibits some "wave-like characteristics"—called "Newton's Rings"—during his experiments with glass plates and thin films. These "Rings" cast considerable doubt upon the corpuscular theory as the sole answer for the understanding of light.

In 1803, Thomas Young observed that a monochromatic light beam passing through two pinholes produces an interference pattern much like the pattern of waves in water. His observation lent support to the *wave theory* of light, which was expressed earlier by Christian Huygens. Another fact already known about the behavior of light was that two kinds of wave propagation exist in a medium—longitudinal waves, which consist of compressions and rarefactions in that medium as exhibited by sound waves; and transverse waves, as demonstrated by wave propagation in water and electromagnetic waves. Augustin Jean Fresnel and Dominique Francois Arago reviewed this information and other work to date and clearly demonstrated that light must consist of transverse waves oscillating at right angles to their direction of propagation.

This important development fit James Clerk Maxwell's *electromagnetic theory* of light, which was postulated later in the 1800s. Maxwell described light as a "rapid variation in the electromagnetic field surrounding a charged particle, the variations in the field being generated by the oscillation of the particle."

In Maxwell's theory, light takes its place along with other forms of radiant energy as an aspect of the fundamental phenomenon of electromagnetism. The study of physics over the last 100 years has indicated that while the fundamental association of light with electromagnetism has held firm, the nature of the understanding of that association has undergone some changes. Light, even though it demonstrates such wave-like phenomena as interference and polarization, also interacts with matter as if the light itself consists of a stream of individual bundles—called photons—with their own energy and momentum. So, what is light? Except for minor differences, light and matter are essentially the same—both are made up of particles that exhibit wave-like characteristics. We must keep this *duality* in mind when considering the applications of optics. ■

—Milton C. Kurtz

CRITIQUE

MACH 2 FOR TURBO BASIC

MicroHelp, Inc.
2220 Carlyle Drive
Marietta, Georgia 30062
(404) 973-9272
$69.95

Speed is of the essence in commercial programming. And speed is something that BASIC programmers have traditionally forsaken in exchange for BASIC's easy coding and testing. To boost the notoriously slow speed of IBM PC BASIC, MicroHelp, Inc., developed Mach 2, a set of assembly language subroutines; Mach 2 is now available for Turbo Basic. Even though Turbo Basic is much faster than the BASICA interpreter, it still benefits greatly from Mach 2's speed injection.

Mach 2's two diskettes contain .INC files for the subroutines and a copious set of example programs, plus a master demo program. The manual is organized functionally, with a separate chapter for each type of Mach 2's subroutine (screen handling, window management, etc.).

To use Mach 2, you copy the .INC files to a working directory and use the **$INCLUDE** compiler directive to incorporate them into your Turbo Basic program. Most of the .INC files are inline subroutines, consisting of unadorned lines of hexadecimal constants, and cannot be readily modified by the user. Assembler source code for the routines is available separately, but since the routines work as documented in the manual, the source code is unnecessary for most users.

Mach 2 provides more than 34 routines that handle a variety of functions, including screen management (reading and writing characters directly to screen memory), window management, file and device management, user input, sorting, and menus. In particular, the string input routine is unusually fast and flexible, with 16 calling options. Using this single input routine, the programmer can control the cursor's shape (with different shapes for insert and typeover modes), text colors, fill characters, field length, and field exit criteria.

Some of Mach 2's more unusual (and useful) functions include large character displays (multiline characters made out of the PC's line-draw characters, useful for titles and warnings); Soundex codes (hashing a string into its phonetic equivalent, useful for searching for names that sound alike but are spelled differently); and Lotus-Intel-Microsoft (LIM) Expanded Memory handling. Mach 2 also provides routines for storing strings in *reserved memory*, which consists of dynamic arrays that are allocated from within a Turbo Basic program and used to hold strings. This use of reserved memory for strings works around Turbo Basic's 64K string space limitation, and could prove a godsend to programmers writing large applications. The master demo program demonstrates each subroutine's capabilities to good effect.

Even when using all the subroutines provided, programmers who write business applications will still need a file access manager (such as the Turbo Basic Database Toolbox) in order to develop complete business applications.

Mach 2's documentation is good, and offers many hints on improving the performance of Turbo Basic code and using the Mach 2 subroutines. Handy reference sections list the subroutine calls, and the sample programs are well documented.

While the performance of Mach 2's assembly language subroutines is striking, there is a price to be paid—the interface to them is more fragile than that of subroutines written in Turbo Basic alone. For example, single- and double-precision floating point numbers cannot both be passed interchangeably to Mach 2's routines, so careful declaration of a variable's type is necessary. Nor can constants or dynamic arrays be passed. The type and number of parameters to each **CALL** statement must exactly match the documented parameters for that statement, otherwise what the Mach 2 manual euphemistically calls "unpredictable results" will occur. Finally, careful initialization and termination of the routines is required. The manual clearly spells out these limitations, but novice programmers might have trouble at first, since Mach 2 requires more discipline than is needed for normal Turbo Basic programming.

There is one niggling flaw in Mach 2—it requires interrupt 66H for passing information between the Turbo Basic program and the assembly language routines. Some way to alter Mach 2's interrupt number should be provided.

But when Mach 2 is set up correctly, the results can be dramatic. If you need the functions it provides, and you don't mind writing your Turbo Basic code with a bit more discipline than usual, this assembly language subroutine package comes highly recommended. ■

—Marty Franz

C-SCAPE WITH LOOK AND FEEL

The Oakland Group, Inc.
675 Massachusetts Avenue
Cambridge, MA 02139
(800) 233-3733 or (617) 491-7311
$99.00

One of the major differences between desktop computers and their mainframe brethren is that the small machines must cater to naive users. This creates a programming challenge—many desktop computer users who have grown up with applications such as Paradox and WordStar also expect easy-to-use, friendly interfaces for custom or inhouse software.

In C, this user interface challenge poses a problem, because the C language supports no standard user interface other than the operating system's command line. Sooner or later, most serious C software developers must face the task of writing a library of C functions to get input from, and to present output to, the user. When that time arrives, "canned" libraries of complete and tested user interface functions can prove useful—C-scape from The Oakland Group is such a product.

C-scape contains a library of functions that implement a user interface, and a screen designer program, called Look and Feel, that automates the coding of screens, borders, and input fields. The Turbo C programmer can use C-scape to incorporate menus, windows, and input fields into a program with a minimum of fuss. C-scape is available for most popular PC-based C compilers; the Turbo C version, "turbo priced" at $99, includes the function library object modules, the Look and Feel utility, and bulletin board support. It is shipped as a 500-page perfect bound manual and five diskettes. A $180 upgrade to source code and full telephone support is also available.

To use C-scape, you place the appropriate **#include** statements and function calls into your Turbo C program, and then compile and link with the C-scape header files and libraries. As an alternative to coding these statements manually, the Look and Feel screen designer can be used to design the screens interactively, and then to automatically generate the calls to the appropriate C-scape library routines. As a third option, C-scape can import screens created with Dan Bricklin's Demo Program.

The Look and Feel screen designer uses the C-scape user interface, and provides a good example of what can be done with C-scape in a production program. The screen designer allows you to type characters and lines in true WYSIWYG (What-You-See-Is-What-You-Get) fashion, to draw borders and backgrounds, and to create menus. Once you're happy with what you see, you can generate source code from the completed screen. Commands are activated from a menu that pops up when F10 is pressed; or through hot keys (a necessary option, given the infrequent-then-intense use that this type of program receives). Look and Feel also contains a complete help facility. A nice feature of Look and Feel that is not found in many other screen designers is the ability to create screens larger than 80 × 25 with support for horizontal and vertical scrolling. This feature allows developers to support screen formats such as the EGA 43-line and VGA 50-line formats; and to support large text screens like the Micro Display Systems Genius VHR 66-line display.

The C-scape library is built in two levels. The higher level of the library has functions for entering fields of text, phone numbers, currency, dates, times, and so forth. An applications programmer will probably use these functions most frequently. Several different types of menus are supported at this level, including the familiar Lotus-style "moving bar," pulldown menus, and pop-up menus. A help facility can be incorporated into programs to allow the creation of text files for display when the user presses the F1 key. C-scape's sophisticated help facility contains highlighted keywords with cross references to other screens. Miscellaneous utility functions handle such activities as word-wrapping text in a string.

The lower level of the C-scape library controls the software objects that the higher level uses. These objects include menus, fields, and "seds," which are screen windows with customizable properties such as borders and titles. The lower-level functions are most useful for hardcore product developers who want a unique "look and feel" for their product. The C-scape library makes heavy use of pointers to functions; the lower level of the library supports replacement of these functions to handle custom field validation and keystroke translation. Another portion of C-scape's lower-level functions is a screen driver, which can be customized to handle nonstandard PC displays. (It's illuminating to run the RAM screen driver and the BIOS-only screen driver consecutively to compare their performance!)

The C-scape library works well in actual use. The higher-level functions are rich; in fact, many Turbo C programmers will never need to use the lower-level functions. The **menufunPrintf()** function in particular is nicely done—strings similar to those used by **printf()** handle cursor positioning, color changes, and printable and nonprintable characters with a powerful, if concise, syntax. Since the C-scape functions all religiously use the screen driver, they're portable among various types of PC hardware. In addition, when the RAM driver is used, C-scape's functions are fast. The product's documentation is well-written and fully indexed, and describes each call in some detail.

C-scape suffers from a few drawbacks. Its construction, while flexible, may be too complex for modification by beginning or casual C programmers. Also, C-scape's documentation describes the lower-level functions before discussing the higher-level ones; this could prove initially overwhelming to less-advanced programmers.

But building a good user interface is what C-scape is for, and that's what it accomplishes. Professional C programmers and consultants would do well to evaluate C-scape before undertaking their next screen-intensive project. ■

—Marty Franz

BOOKCASE

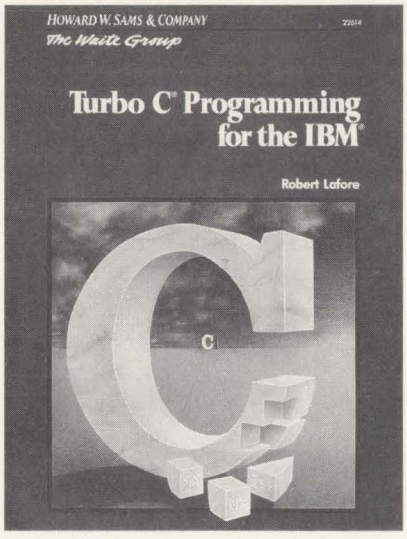
TURBO C PROGRAMMING FOR THE IBM

Robert Lafore, Howard W. Sams & Company, Indianapolis, IN, ISBN 0-672-22614-6, 585 pages, softcover, $22.95.

The first half of this unusually comprehensive book offers an excellent introduction to the Turbo C environment and to C programming in general. The second half of the text covers a wealth of progressively more advanced Turbo C topics (such as memory models, pointers, and advanced variables), which provide interesting reading for both the novice and the more experienced Turbo C programmer.

Chapter 1 presents a step-by-step guide to Turbo C's integrated development environment, and walks the reader through the process of creating a simple Turbo C program. The author then discusses the different types of files (header, library, runtime, math, and programmer-generated) that are required to build a C program. In this chapter, Lafore provides easily understood explanations of basic concepts, and doesn't make the common mistake of assuming that the reader understands the linking process and the use of header files.

The second chapter introduces the building blocks of Turbo C programs: variables, I/O functions, and operators. In the following three chapters, Lafore uses these building blocks to explore the control statements of Turbo C in an informative discussion of loops, decisions, and functions.



The next two chapters focus on those areas where C differs most from other compiled languages. Array declaration, content, initialization, size, and sorting are presented in Chapter 6, along with a discussion of string variables, initialization, and I/O functions. The explanation on copying strings into an array of strings is particularly interesting.

The discussion on pointers in Chapter 7 skillfully explains what is often regarded as one of the most difficult concepts to master in C. Lafore starts with a lucid introduction to the need for pointers in Turbo C, and moves on to cover the mechanics of pointer usage. This chapter is required reading even for the very experienced Turbo Pascal programmer, because C handles pointers differently than does Turbo Pascal.

The last seven chapters of the book address more advanced programming topics such as extended keyboard codes, cursor control, command-line arguments, structures, unions, ROM BIOS routines, memory and the character display, color graphics, and disk I/O operations. Lafore devotes a chapter to the creation of larger programs in Turbo C, and covers separate compilation, conditional compilation, and memory models. The book's final chapter examines advanced variables (register variables, enumerations, and storage classes).

Throughout the text, Lafore emphasizes a practical, hands-on approach to programming concepts, and includes a wealth of clear and concise program examples. Important facts are visually highlighted in gray boxes throughout each chapter. All chapters end with a summary, followed by a set of exercises that is designed to help the reader apply his/her newfound knowledge (answers are included in the back of the text). Appendices include references, hexadecimal numbering, a bibliography, and an ASCII chart.

As an added plus, Lafore often provides helpful solutions to problems that a C programmer will typically encounter. For example, the discussion of color graphics introduces direct memory access by presenting relevant examples such as the use of ROM routines to plot points in graphics modes. The "Files" chapter presents an impressive discussion of text mode versus binary mode.

This book offers an excellent presentation of programming concepts and example programs for both beginning and more advanced C programmers, and is a valuable addition to any Turbo C programmer's book shelf. ■

—Robert Alonso

ARTIFICIAL INTELLIGENCE PROGRAMMING WITH TURBO PROLOG

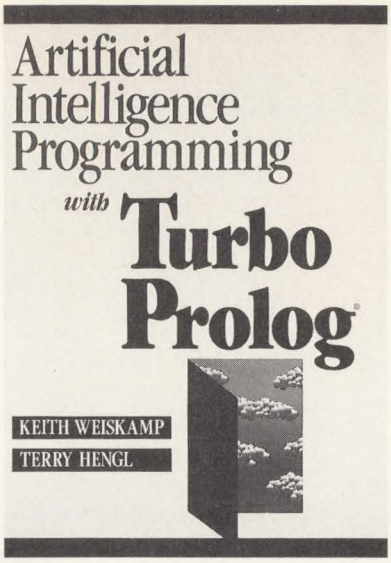
Keith Weiskamp and Terry Hengl, John Wiley & Sons, Inc., New York, NY: 1988, ISBN 0-471-62752-6, 262 pages, soft cover, $22.95, disk $24.95.

If you are one of the many newcomers to the field of Artificial Intelligence programming, you may be discovering that learning Turbo Prolog and reading books about basic Prolog concepts are not enough to get you started on developing AI applications. Many programmers who have explored Turbo Prolog and even written a few short programs may still feel handicapped when trying to apply that knowledge to the development of a serious AI application.

Until now, this scenario could have been attributed, at least in part, to the absence of books on AI fundamentals as they relate to Turbo Prolog. Most books on the market today emphasize Prolog programming theories and techniques, rather than the world of Artificial Intelligence and related concepts. None of these books has attempted to illustrate the fundamentals of AI using Turbo Prolog—that is, until *Artificial Intelligence Programming with Turbo Prolog*, written by Keith Weiskamp and Terry Hengl. This book has a very specific goal—to show you how to use Turbo Prolog to develop AI applications. The authors seem to have spent a great deal of effort in staying with this game plan.

OPENING THE WINDOW

The book has a pleasant, readable style. The authors begin by "Opening the AI Window" to let you peek into the world of AI applications. The first two chapters in the book also cover features of Turbo Prolog. However, instead of regurgitating the information in the *Turbo Prolog Owner's Handbook*, the authors focus on the way that Turbo Prolog works and cover such topics as the Resolution Principle, unification, backtracking, and nondeterministic programming.



Chapter 3 provides an introduction to software design using Turbo Prolog. Here, the authors illustrate techniques for developing an AI toolbox—a collection of subroutines (predicates) that are essential to developing AI applications. For example, if you have some experience with a procedural language like Pascal or C, you know that you need to develop some routines for controlling program flow. One such control structure that is essential in large scale applications—the **repeat/fail** loop—is covered in the book. Similarly, the authors provide predicates to process various Prolog data structures, such as characters, strings, and lists. A section is also devoted to the role of recursion in Turbo Prolog programming.

LAYING THE GROUNDWORK

The remaining four chapters in the book develop the groundwork for AI programming. In Chapter 4, the authors discuss the development of an inference engine, including the fundamentals of reasoning as a process of both categorizing information in the form of known facts and rules (knowledge), and creating new facts and rules. Perhaps the most important concept in this chapter is that of formal reasoning using propositional and predicate calculus. The authors follow the discussion on propositional calculus with an example (the Translate program) that illustrates how Turbo Prolog could be used with formal propositional logic. The chapter concludes by explaining forward and backward chaining, which are the control strategies used in building an inference engine, along with the actual construction of the core of an inference engine (a scheduler, a rule interpreter, and the knowledge interface.)

Chapter 5 provides an in-depth look at natural language processing. The reader is presented with natural language processing techniques, including pattern matching. The discussion of transition networks includes both augmented and recursive transition networks. Example programs illustrate each of these techniques.

One of the most critical factors in the development of expert systems is the representation of facts or knowledge, since knowledge comprises the actual data in a Prolog program. Chapter 6 explores various techniques used in AI programs for representing knowledge, including specific knowledge representation features in Turbo Prolog. The chapter also includes a discussion of knowledge representation with frames and provides an example of such a representation in Turbo Prolog.

The book's final chapter takes you through the various development stages of an expert system. The authors explain various characteristics and types of expert systems, and provide hints about techniques for improving the example expert system.

Artificial Intelligence Programming with Turbo Prolog provides you with basic Turbo Prolog concepts, and helps you build AI tools, by primarily focusing on the groundwork needed for programming AI applications. It is not (necessarily) a beginner's book about Turbo Prolog, and doesn't attempt to replace your *Owner's Handbook* or a tutorial/reference guide. For users who are ready to make the jump into real-world AI applications, *Artificial Intelligence Programming with Turbo Prolog* should prove quite valuable. ■

— Sanjiva Nath

TURBO RESOURCES

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 megabytes of utilities and source code in all Borland languages.

Subscribing to CompuServe can be done through the coupon enclosed with every Borland product (which also includes $15 worth of free online time for your first month) or by calling CompuServe at (800) 848-8199. You'll need a modem and communications software that supports XMODEM file transfers.

Learning your way around CompuServe takes some time and practice, but good books have been written about it, including Charles Bowen's and David Peyton's *How To Get The Most Out Of CompuServe* and *Advanced CompuServe for IBM Power Users* (New York: Bantam Computer Books, 1986.) Howard Benner's TAPCIS shareware utility can help you automate sessions and minimize connect time. It's available for downloading on CompuServe from DL 12 of the WordPerfect Support Group Forum (**GO WPSG**). The TAPCIS file is 239,297 bytes long—plan to spend some hours downloading it.

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 are 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.

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 containing all the individual .ARC files for that issue. You can therefore download listings for individual articles, or download the entire issue's listings at once.

The all-issue files follow a naming convention, such as NVDC87.ARC (which contains all listing archives from the November/December, 1987 issue), and JAFB88.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** 12-month subscription to *TURBO TECHNIX* is yours for the asking when you register any of the Borland languages (including Quattro, Paradox, Eureka, and Sprint) or language toolboxes. 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. If your copy of a Borland language product was shipped without the subscription request card, you can also use the subscription services card bound into this 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 to provide updated information. If possible, attach the old mailing label to the card.

NATIONAL USER GROUPS

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-tos, and product reviews. Extensive public-domain utility and source code libraries are available to members. An optional multiuser BBS with file uploading/downloading, messaging and teleconferencing is available to the public. Membership dues are $22.00 US/year ($23.72 in Washington State); $26.00 Canada and Mexico; $38.00 overseas.

TUG
PO Box 1510
Poulsbo, WA 98370
BBS: (206) 697-1151

TPro Users

TPro Users was founded specifically to support Turbo Prolog programming. Their bimonthly newsletter contains technical articles, application stories, tips and techniques, and more. TPro also maintains an electronic bulletin board for source code downloading and message posting. Dues are $25.00 US/year; $35.00 overseas.

TPRO USERS
3109 Scotts Valley Drive, Suite 138
Scotts Valley, CA 95066
BBS: (408) 438-6506

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 convenient 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) 353-9312
One Center Plaza
Boston, MA 02108

CAPITAL PC USER GROUP (DC)
4520 East-West Highway, Suite 550
Bethesda, MD 20814
C SIG: Fran Horvath
AI/Prolog SIG: Dick Strudeman
BASIC SIG: Don Withrow

CHICAGO COMPUTER SOCIETY
Information: (312) 942-0705
BBS: (312) 942-0706
Pascal SIG: Bill Todd (312) 439-3774
C SIG: Ed Keating (312) 438-0027
AI/Prolog SIG: Jim Reed
(312) 935-1479
Basic SIG: Hank Doden
(312) 774-5769

HAL/PC (HOUSTON)
Information: (713) 524-8383
BBS: (713) 847-3200 or
(713) 442-6704
Pascal SIG: Charles Thornton
(713) 467-1651
C SIG: Odis Wooten (713) 974-3674
Compiled BASIC SIG: Larry
Krutsinger (713) 784-9216
AI SIG (Prolog): George Yates
(713) 448-7621

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
Pascal SIG: Jeffrey Watson
(314) 481-4239
C/Assembler SIG: David Rogers
(314) 968-8012
BASIC SIG: Dennis Dohner
(314) 351-5371

TWIN CITIES PC USER GROUP
Information: (612) 888-0557
BBS: (612) 888-0468
PO Box 3163
Minneapolis, MN 55403

Independent CBBS systems with programming orientation

Questor Project (703) 525-4066	Washington, DC 24Hr $
Illinois BBS (312) 885-2303	Chicago, IL 24Hr $
PC-TECH BBS (408) 435-5006	Santa Clara, CA 24Hr

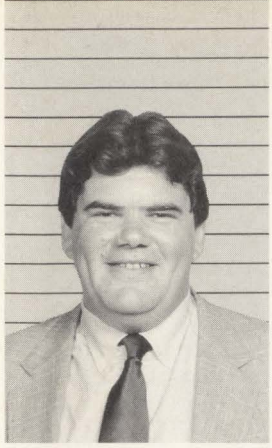
$ = membership fee required

C:>CLASS.ADS

C:>CLASS.ADS is *TURBO TECHNIX* 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—$300 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 prepaid and submitted in camera-ready form (black and white PMT or Velox) to:

C:>CLASS.ADS
TURBO TECHNIX
4585 Scotts Valley Drive
P.O. Box 660001
Scotts Valley, CA 95066-0001

For additional information, please call Production Assistant Annette Fullerton at (408) 438-9321.



PHILIPPE'S TURBO TALK

The new age of software craftsmanship.

Philippe Kahn

Does a compiler become better because it's packaged in a huge box and sold by weight? Maybe you've heard that if some companies know that a retailer must carry their product, they make the package as big as possible to push other vendors off the shelf. The user community is smart enough to see through this. Still, some people get caught by surprise.

Good documentation isn't the same as fat documentation. Documentation writers should not get paid by the word any more than software engineers should get paid by the line of code. It's content that counts, of course, and we all know that.

ARE BENCHMARKS USEFUL TO REAL USERS?

Should we compiler vendors optimize for benchmarks? At Borland, we make the conscious decision *not* to do so, but rather to set compiler default settings for convenience and efficiency. However, we've all heard about software optimized for benchmarks. How about "sieve recognizers?" Maybe that's a joke, but it's true that some business software, and even some compilers, are now written so that they make the "standard benchmarks" look good. Who cares about the user? The user never runs benchmarks. The only real measure of software performance is how quickly it gets the user's job done. It's hard to write a benchmark that takes the real world into account. But it should be done, or the benchmarks aren't saying anything useful. In the meantime, the users will, by their natural common sense, separate the hype from the reality and get their work done faster. The real benchmark is, "Am I more productive, and is the quality of my work higher when I use this tool?"

DO FAST AND SMALL STILL COUNT?

For years we've noticed that memory is getting cheaper and that processors are getting faster. So we're all excited about the wonderful applications that we are going to be able to build with this increased computing power. We all know that software tends to use up all available memory, and that it also has a tendency to grab all the processor time it can. Until now, we have all tried to be very careful to use hardware resources as efficiently as possible. Yet, although we remember great software that ran in 64K machines running 8-bit processors, today we consider it almost normal for software to require a 32-bit processor and several megabytes of RAM. Now, is this because this particular piece of software uses the computer's resources less efficiently? Or is it because the people who wrote it were thinking, "After all, memory is cheap and processor speeds are faster, so who cares?" With more memory in almost every machine, there's a tendency to write what I call "sloppy software." Who cares if "Hello, world" takes 400K if I still have megabytes left?

Then there's the old hype line that says, "This machine is so fast, it doesn't matter that the operating system is written in interpreted BASIC." Guess how fast that machine would run if it had a *real* operating system?

Why is this sort of logic wrong? After all, with faster processors and cheap memory, do code size and execution speed matter? Under DOS, of course they do. But how about multitasking operating systems with dynamic memory management? Think about it: slow applications will steal precious time from other tasks and slow them down. And memory hogs will force the OS to constantly swap the other tasks to and from disk. It's even worse than under DOS, where a big, slow application only penalizes itself. In a multitasking environment, slow and fat applications penalize all the other tasks. Deja vú! Small, efficient code matters more than ever before!

Now that the 640K barrier is about to be broken, new programs will grow to fit available memory and processing speed. A color paint program on the Mac that I was just playing with requires 1.5 megabytes. The first MacPaint ran in 128K. Is the new program ten times better than the original? I don't think so.

ENTER THE NEW AGE OF SOFTWARE CRAFTSMANSHIP

Remember the scene on the sinking Titanic, when the passenger yelled to the drink steward, "Yes, I know I rang for ice, but this is ridiculous!" Size and speed, quality documentation, and "real world optimizations" will matter more and more. If we keep that in mind, we are going to witness a new age of software craftsmanship. ■

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

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 • Saguaro Technical Services • Cave Creek, AZ

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

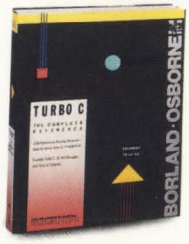
Visa, Amex, C.O.D., Check or P.O.
60-day satisfaction, money-back guarantee
Demo Diskettes and brochures available
Out of U.S. add postage

Nostradamus®

Borland·Osborne/McGraw-Hill Presents

The Official Books on TURBO C®, TURBO BASIC®, & TURBO PASCAL®

"The technical depth and timeliness of books in the Borland•Osborne/McGraw-Hill Programming Series provide excellent supplementary support for Borland's best-selling compilers. Users at all levels can turn to these books to help them get the most out of Turbo Pascal, Turbo C, and Turbo Basic." *Philippe Kahn, Chairman & CEO, Borland International*



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., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series

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.

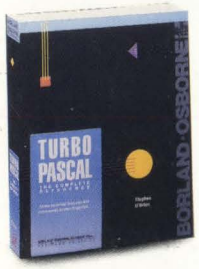
$22.95 Paperback, **ISBN: 0-07-881280-1**, 397 pp., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series

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., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series



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., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series

Turbo Pascal® 4: THE POCKET REFERENCE

by Kris Jamsa

This little booklet puts all essential Turbo Pascal Version 4 features and commands at your fingertips.

$5.95 Paperback, **ISBN: 0-07-881379-4**, 120 pp., 4¼ x 7
Borland•Osborne/McGraw-Hill Programming Series

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., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series

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.

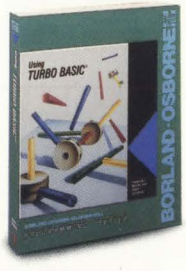
$21.95 Paperback, **ISBN: 0-07-881355-7**, 416 pp., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series

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., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series



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., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series

Turbo C® PROGRAMMER'S LIBRARY

by Kris Jamsa

This powerful collection of Turbo C programming routines enhances the producivity and efficiency of all Turbo C programmers.

$22.95 Paperback, **ISBN: 0-07-881394-8**, 650 pp., 7⅜ x 9¼
Borland•Osborne/McGraw-Hill Programming Series

ORDER TODAY! ***Call Us Toll-Free 800-227-0900*** *We accept Visa, MasterCard, and American Express.*
In Canada, contact McGraw-Hill Ryerson, Ltd. ***Phone 416-293-1911.***

Osborne McGraw-Hill
2600 Tenth Street
Berkeley, California 94710

BORLAND·OSBORNE