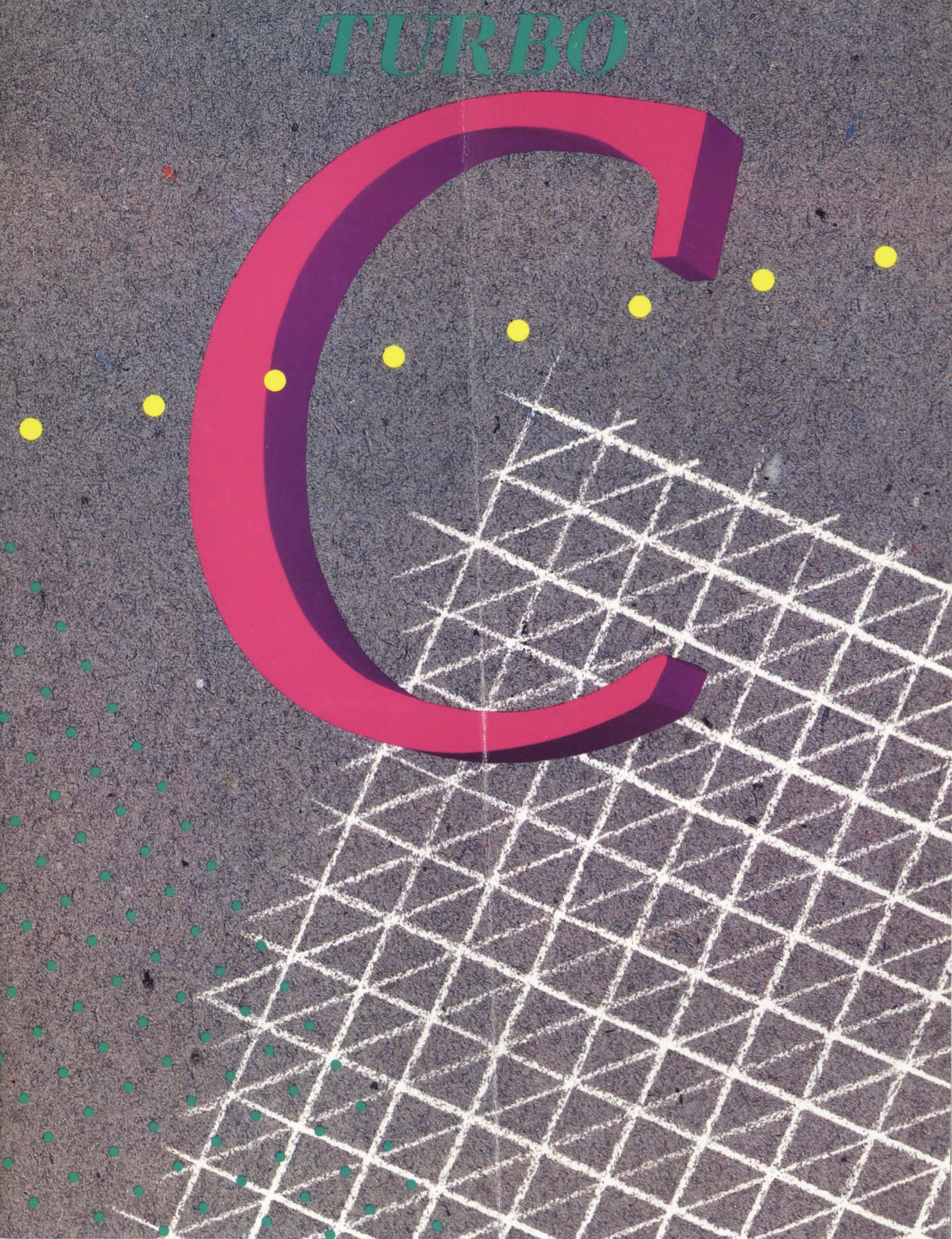
# TURBO TECHNIX™

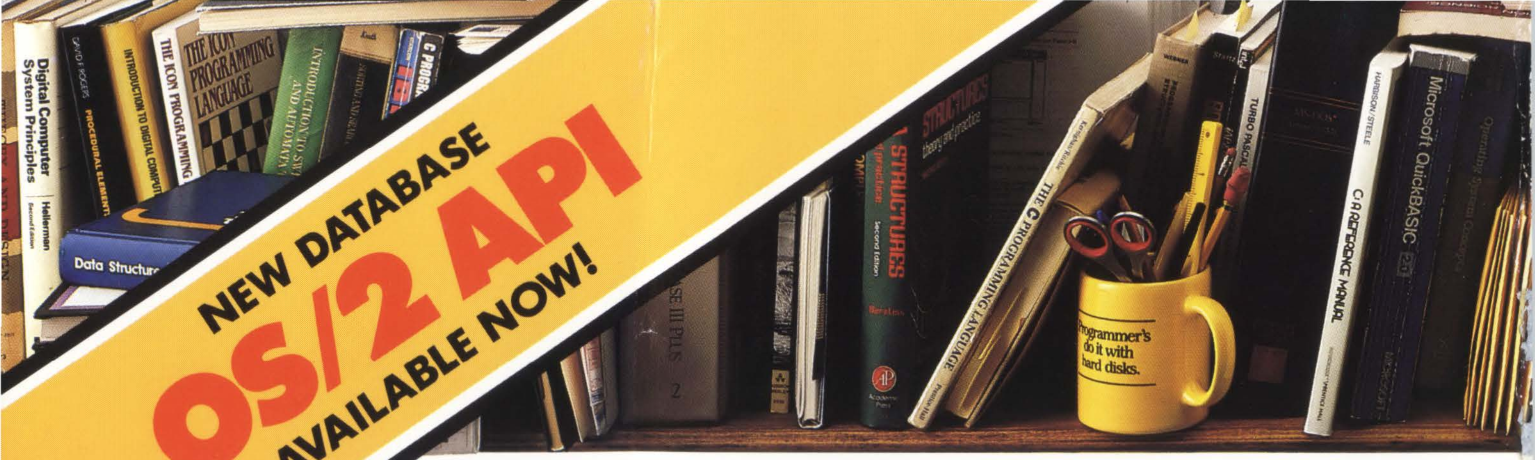
THE BORLAND LANGUAGE JOURNAL • JANUARY/FEBRUARY 1988 • VOLUME ONE NUMBER TWO

## FLOATING POINT IN TURBO C
## The Machinery Behind the Numbers

## Intercepting Keyboard Interrupts

## The Turbo Prolog Toolbox's Pulldown Predicate




Digital Computer System Principles
Hellerman
Data Structures
Introduction to Digital Computer
The Icon Programming Language
Structures theory and practice
The C Programming Language
Turbo Pascal
C: A Reference Manual
Harbison / Steele
Microsoft QuickBASIC
Programmer's do it with hard disks.

NEW DATABASE
OS/2 API
AVAILABLE NOW!

# 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.

The Norton On-Line Programmer's Guides are a quartet of pop-up reference packages that do the same things in four different languages.

Each package consists of two parts: A memory-resident instant access program. And a comprehensive, cross-referenced database crammed with just about everything you need to know to program in your favorite language.

## GUIDES DATA

**Instant Access Program**
- Memory-resident—uses just 71K.
- Full-screen or moveable half-screen view, with pull-down menus.
- Auto lookup and searching.
- Tools for compiling your own databases.

**ASSEMBLY (600K of data)**
- DOS Service Calls: All INT 21h services, interrupts, error codes, FCB and PSP fields, standard handles and more.
- ROM BIOS Calls: All ROM calls plus low RAM usage.
- Instruction Set: All 8088/86 instructions, addressing modes, flags, bytes per instruction, clock cycles and more.
- MASM: Pseudo-ops and assembler directives.
- Tables: ASCII chart, line-drawing charts, keyboard scan codes and more.

**BASIC (270K each database)**
- IBM BASICA, Microsoft QuickBASIC and TurboBASIC.
- Statements and Functions: Describes all statements and built-in library functions.
- Tables: Line-drawing characters, ASCII chart, keyboard codes, error codes, operators, etc.

**C (600K each database)**
- Microsoft C and Turbo C: Describes language, including statements, operators, data types and structures.
- Library Functions: Detailed descriptions of all functions, from abort () to write ().
- Preprocessor Directives: Describes commands, usage and syntax.
- Tables: ASCII chart, line-drawing characters, keyboard codes, error codes, operators, etc.

**PASCAL—Turbo (360K of data)**
- Language: Describes statements, syntax, operators, data types and records.
- Library: Describes the library procedures and functions.
- Tables: ASCII chart, line-drawing characters, keyboard codes, error codes, reserved words, etc.

(If you don't believe us, you might want to take a moment or two to examine the data box you just passed.)

You can, of course, find most of this

Designed for the IBM® PC, PC-AT and DOS compatibles. Available at most software

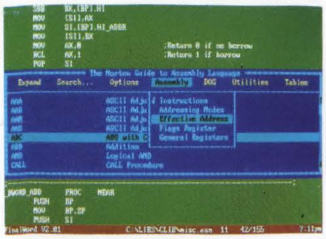
# gramming tool ate manual labor.

information 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.

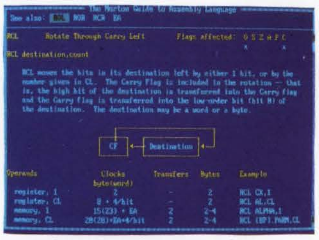
In seconds.

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



```
        SBB     BX,[BP].HI
        MOV     [SI],AX
        MOV     SI,[BP].HI_ADDR
        MOV     [SI],BX
        MOV     AX,0                    ;Return 0 if no borrow
        RCL     AX,1                    ;Return 1 if borrow
        POP     SI
============ The Norton Guide to Assembly Language ============
 Expand   Search...   Options   Assembly   DOS   Utilities   Tables

 AAA              ASCII Adju ✓ Instructions
 AAD              ASCII Adju   Addressing Modes
 AAM              ASCII Adju   Effective Address
 AAS              ASCII Adju   Flags Register
 ADC              ADD with C   General Registers
 ADD              Addition
 AND              Logical AND
 CALL             CALL Procedure

DWORD_ADD       PROC    NEAR
        PUSH    BP
        MOV     BP,SP
        PUSH    SI
FinalWord V2.01          C:\LIBS\CLIB\misc.asm  11  42/155        7:11pm
```

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



```
============ The Norton Guide to Assembly Language ============
 See also:  RCL  ROR  RCR  EA

RCL      Rotate Through Carry Left          Flags affected:  O S Z A P C
                                                             *         *
RCL destination,count

   RCL moves the bits in its destination left by either 1 bit, or by the
   number given in CL.  The Carry Flag is included in the rotation — that
   is, the high bit of the destination is transferred into the Carry flag
   and the Carry flag is transferred into the low-order bit (bit 0) of
   the destination.  The destination may be a word or a byte.

                  CF  <--  Destination  <--

Operands            Clocks        Transfers   Bytes    Example
                    byte(word)
  register, 1          2              -         2      RCL CX,1
  register, CL      8 + 4/bit         -         2      RCL AL,CL
  memory, 1         15(23) + EA       2        2-4     RCL ALPHA,1
  memory, CL       20(28)+EA+4/bit    2        2-4     RCL [BP].PARM,CL
```

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

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

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

And you're right.

But even Peter Norton can't think of everything.

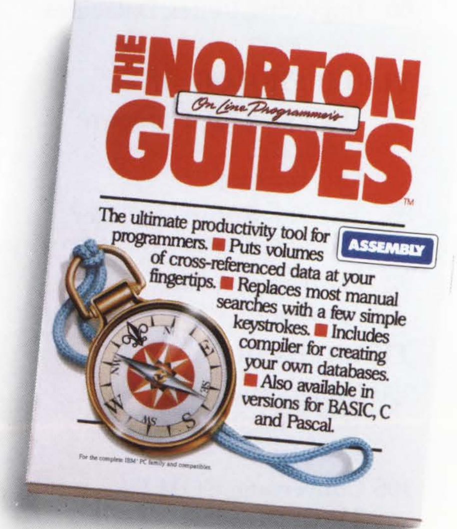
Which is why there's a built-in compiler for creating databases of your own.

And why all Guides databases are compatible with the instant access program in your original package.

So you can add more languages without spending a lot more money.

To get more information, call your dealer. Or call Peter Norton at 1-800-451-0303 Ext. 40.

And ask for some guidance.


THE NORTON
On-Line Programmer's
GUIDES
TM
The ultimate productivity tool for programmers. ■ Puts volumes of cross-referenced data at your fingertips. ■ Replaces most manual searches with a few simple keystrokes. ■ Includes compiler for creating your own databases. ■ Also available in versions for BASIC, C and Pascal.
ASSEMBLY
For the complete IBM® PC family and compatibles.

Peter Norton
COMPUTING

dealers, or direct from Peter Norton Computing, Inc., 2210 Wilshire Blvd. #186, Santa Monica, CA 90403. 213-453-2361. Fax 213-453-6398, MCI Mail: PNCI

**TURBO TECHNIX**
The Borland Language Journal
January/February 1988
Volume I, Number 2

## FEATURES

### TURBO PASCAL

### TURBO C

### TURBO PROLOG

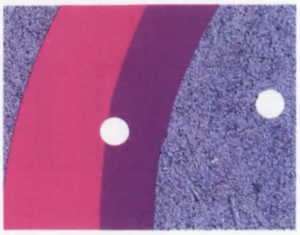
### TURBO BASIC

### BUSINESS LANGUAGES

## DEPARTMENTS

8

Adding new keystrokes to your keyboard (or taking some away!) requires replacing the PC keyboard interrupt with an interrupt handler of your own.



34

Expressing analog quantities in a digital world involves subtleties that the integer world neatly avoids. Math coprocessors further complicate matters for those who want their programs to run under all machine environments. Turbo C's answers to these problems are worth close study.

72

Fine-tuning the pulldown predicate from the Turbo Prolog Toolbox provides text menu versatility with relatively little effort.

100

Looking at a Turbo Basic disk file as a collection of fixed-length records divided into logical fields allows you to read any record at random—without reading sequentially through those that come before it.

*Cover:*
*The humble decimal point helps Turbo C express quantities in the real world. Photography by Mike Kirkpatrick.*

# More Magic from Blaise.

# Turbo C TOOLS™

Magic is easy with Turbo C TOOLS in your bag of tricks. New Turbo C TOOLS™ from Blaise Computing is a library of compiled C functions that allows you full control over the computer, the video environment, and the file system, and gives you the jump on building programs with Borland's new C compiler. Now you can concentrate on the creative parts of your programs.

The library comes with well-documented source code so that you can study, emulate, or adapt it to your specific needs. Blaise Computing's attention to detail, like the use of function prototyping, cleanly organized header files, and a comprehensive, fully-indexed manual, makes Turbo C TOOLS the choice for experienced software developers as well as newcomers to C.

Turbo C TOOLS provides the sophisticated, bullet-proof capabilities needed in today's programming environment, including removable windows, "sidekickable" applications, and general interrupt service routines written in C.

The functions contained in Turbo C TOOLS are carefully crafted to supplement Turbo C, exploiting its strengths without duplicating its library functions. As a result you'll get functions written predominantly in C, that isolate hardware independence, and are small and easy to use.

Turbo C TOOLS embodies the full spectrum of general purpose utility functions that are critical to today's applications. Some of the features in Turbo C TOOLS are:

◆ **WINDOWS** that are stackable and removable, that have optional borders and a cursor memory, and that can accept user input.

◆ **INTERRUPT SERVICE ROUTINE** support for truly flexible, robust and polite applications. We show you how to capture DOS critical errors and keystrokes.

◆ **INTERVENTION CODE** lets you develop memory resident applications that can take full advantage of DOS capabilities. With simple function calls, you can schedule a Turbo C function to execute either when a "hot key" is pressed or at a specified time.

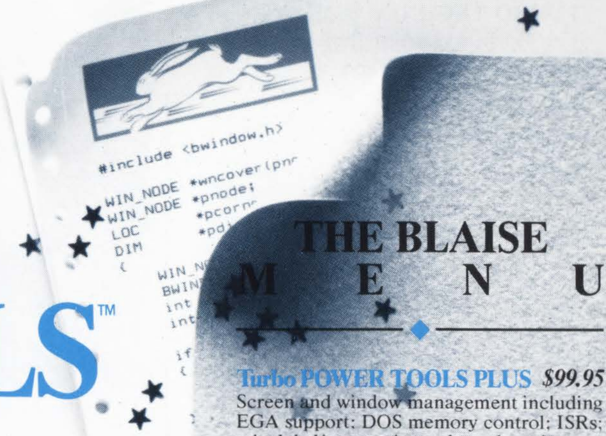
◆ **RESIDENT SOFTWARE SUPPORT** lets you create, detect, and remove resident utilities that you write with Turbo C TOOLS.

◆ **FAST DIRECT VIDEO ACCESS** for efficiency, and support for all monitors including EGA 43-line mode.

◆ **DIRECTORY AND FILE HANDLING** support let you take advantage of the DOS file structure, including volume labels and directory structure.

In addition to Turbo C TOOLS, Blaise Computing Inc. has a full line of support products for Microsoft, Lattice and Datalight C, Microsoft Pascal and Turbo Pascal. Call today for details, and make magic!

**Turbo C TOOLS supports the Borland Turbo C compiler, requires DOS 2.00 or later and is just $129.00**



## THE BLAISE MENU

**Turbo POWER TOOLS PLUS** *$99.95*
Screen and window management including EGA support; DOS memory control; ISRs; scheduled intervention code; and much more. For Turbo Pascal.

**Turbo POWER SCREEN**
COMING SOON! General screen management; paint screens; block mode data entry or field-by-field control with instant screen access. For Turbo Pascal.

**Turbo ASYNCH PLUS** *$99.95*
Interrupt driven support for the COM ports. I/O buffers up to 64K; XON/XOFF; up to 9600 baud; modem and XMODEM control. For Turbo 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.

**C TOOLS PLUS** *$175.00*
Windows; ISRs; screen handling; multiple monitors; EGA 43-line text mode; direct screen access; DOS file handling and more. For MS and Lattice C version 3.00 and later.

**LIGHT TOOLS** *$99.95*
Windows; ISRs; EGA 43-line text mode; direct screen access; DOS file handling and more. For the Datalight C compiler.

**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. For C or MS-Pascal.

**VIEW MANAGER** *$275.00*
General screen control; paint screens; block mode data entry or field-by-field control with instant screen access. For C or MS-Pascal.

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

**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.

**TO ORDER CALL TOLL FREE
800-333-8087
TELEX NUMBER-338139**

**YES! I want to make magic!**
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: ________

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

*Turbo C is a trademark of Borland International.*

**TURBO TECHNIX**
The Borland Language Journal
January/February 1988
Volume I Number 2

*Publisher*
Marcia Blake

*Editor in Chief*
Jeff Duntemann

EDITORIAL

*Managing Editor*
Michael Tighe

*Technical Editor*
Michael A. Floyd

TECHNICAL CONSULTANTS

Brad Silverberg
David Intersimone
Roger Schlafly

DESIGN & PRODUCTION

*Art Director*
Karen Miner

*Production Assistant*
Annette Fullerton

*Typesetting Manager*
Walter Stauss

*Typesetter/System Supervisor*
Jeffrey Schwertley

*Typesetters*
Ron Foster
Jeanie Maceri

*Photographer*
Bradley Ream

*Typesetting Traffic*
Charlene McCormick

ADMINISTRATION

*Purchasing*
Brad Asmus

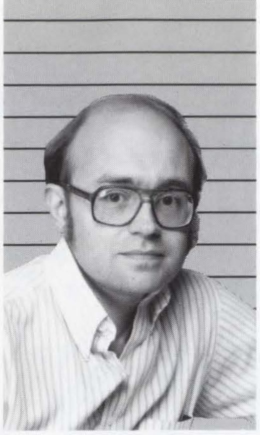
ADVERTISING

*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



# BEGIN

## DOS, The Understood

*Jeff Duntemann*

Let's not be so quick to bury DOS. It may well be the first and only operating system in history that we truly understand.

What is the real value of an operating system? Far more than multitasking or a standard user interface; it is the industry's level of knowledge about it. My rationale: What the OS can do, we know how to do; what the OS *can't* do, we know how to work around.

My experience with minicomputer operating systems provides a good counterpoint: Back in my Xerox days I had to deal with a Honeywell OS called CP/6. Like almost all minicomputer software, it was needlessly convoluted and virtually undocumented. With only a couple of hundred installations in the world, and only three at Xerox, there were no easily accessible gurus to provide help. Honeywell may have had some gurus tucked away somewhere, but they pointedly wouldn't tell us who they were. Let's not even think about a local equivalent to Ray Duncan's *Advanced MS-DOS.* So no matter how powerful CP/6 may in fact have been, the only way to gain access to that power was by trial and error after relentless hunting through the impenetrable manuals.

***DOS is easily the most documented operating system in history, where documentation is measured in insight, not shelf-feet.***

DOS is easily the most documented operating system in history, where documentation is measured in insight, not shelf-feet.The ten-million-plus installed base allows world-class explainers like Peter Norton and Ray Duncan to make a living explaining DOS by way of books, and now Peter Norton has distilled much of the tough stuff into a memory-resident green card called The

*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. Entire contents Copyright ©1988 Borland International, Inc. All rights reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the publisher. For a statement of our permission policy for use of listings appearing in the magazine, send a self-addressed stamped envelope to Permissions, *TURBO TECHNIX,* 4585 Scotts Valley Drive, Scotts Valley, CA 95066. 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. Third class postage pending at Santa Cruz, CA. 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.

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

Norton Guides. Having such reference works on hand, and having professional explainers publishing solid information on DOS all the time, I'm confident that we know exactly what DOS can do and what it can't.

Our hacker-level knowledge of DOS goes far deeper than that. The unappreciated liberty of taking a debugger to the OS (try *that* in the mainframe/mini world!) has allowed the wild-eyed among us to discover things, like the DOS BUSY flag, that take the edge off of DOS's famous reentrancy limitations. Careful program design won't make DOS reentrant, but it will allow us to design applications that *act* as though DOS were reentrant, which from the end user's perspective is the same thing.

Contributing to our knowledge of DOS is the fact that it rests on a hardware platform known equally well. If IBM had not published the schematics and BIOS source code for the original PC architecture, we would be years behind in discovering workarounds for DOS's weaknesses. Chaining onto hardware-related system interrupts like the timer tick (1CH) have allowed DOS wizards like Lane Ferris to build preemptive multitasking into DOS applications written in Turbo Pascal. (We'll be providing much more on Lane's multitasking kernel in a future issue.) The stellar LIM 4.0 Expanded Memory Specification would not have come to pass so well or so quickly if the PC's bus and memory architecture had remained hidden behind legal ramparts and 457-pin bed-of-nails custom ICs. LIM 4.0 goes about 80 percent of the way toward freeing DOS from the 640K barrier—again, not by breaking the barrier, but by making DOS *act* as though it were breaking the barrier.

None of this would have happened had DOS only sold into a few hundred thousand systems, or if Microsoft and IBM had been pathological secret-keepers from the beginning.

With all that in mind, I'll ask the question every industry pundit has been lately asking in print: How long will DOS last?

*It will outlast OS/2.*

My reasoning turns on the following issues:

- DOS can fake most of OS/2's important features even on 8088-based PCs. DOS can coexist with software that multitasks—anyone who has used BackComm or Lotus Express will have to admit that. LIM 4.0 and its successors will put as much memory as we can afford in the box, and DOS will be faked into using it for programs and data.

***What DOS can do, we know how to do. What DOS can't do, we know how to work around.***

  If the faking is seamless enough, who cares?
- OS/2 has been designed for a dead-end processor. The 80286 is an evolutionary side-trip; essentially the chip that Intel made while they were learning how to make the 80386. It's only about two-thirds there, with incomplete hardware memory management and an inability to virtualize the 8086/8088.

  By the time OS/2 comes into wide use, Intel will most likely be shipping their P9 CPU, which does for the 386 what the 8088 did for the 8086: embed a powerful CPU in a low-cost mass-market package. If the P9 costs the same as a 286, why bother with the 286? In six months that will be *the* question to answer.
- In a well-integrated 386-based 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. Since each task has its own copy of DOS, reentrancy ceases to be an issue. Virtual-86 partitions are limited to 640K, but products like Qualitas's 386-To-The-Max will take 32-bit extended memory and make it act like LIM 4.0 expanded memory. I have not yet tested Windows/386, but my sources indicate that it is as fully capable of providing a standard user interface as OS/2's Presentation Manager, and deadly fast to boot. As 386-power becomes more common, DOS and Windows/386 will gradually melt together; no one will buy one without the other.
- We will never know as much about OS/2 as we do about DOS. By its design, the kernel is a black box, and will be highly resistant to probings by curious hackers. Hardware memory protection is a devilish thing to defeat. My hunch is that the merely curious will stay home—and much that we know about DOS has come from the merely curious.

Furthermore, DOS may in fact outlast OS/3, or whatever the next generation protected-mode 386 OS happens to be, simply by riding in its hip pocket through the years. Knowing what we now know about the market's insistence on upward compatibility, no 386 OS worth its pound of silicon will go to market without being able to run or emulate DOS as a virtual-86 task.

Long after OS/2 has evolved into OS/3 or OS/4, you'll still be able to bring up Turbo Pascal under DOS. For an operating system, at least, a little learning is dangerous—and a lot may well mean eternal life. ■

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

# FOR THOSE WHO WOULD BE WRITERS

Your job is to weld ice and iron.

Effective writing about programming must span the conceptual gulf between high technology that defies description and the English language, which must describe it. The job is a dual one, requiring two sets of skills developed in two sadly divergent cultures. The seminal British thinker C. P. Snow despaired of a bridge between these two cultures, and while Snow may have been reaching higher than literate tutorials on technology, I feel that he despaired too soon. Bridging those cultures is what we do at *TURBO TECHNIX*. If you have the interest, we'd like you to join us. Before you do, here are some things to keep in mind:

***Keep your topic focused.*** If you can't get your mental arms around a concept, the concept is probably too broad. Broad articles can rarely be deep enough to serve the *TURBO TECHNIX* readership. "Programming the EGA" is too broad. "Loading and Saving EGA Text Fonts" is more like it. Zero in close and give us the whole story.

***Understand before you explain.*** If a concept proves difficult to explain, it may be because you don't fully understand it yet. Go back to the manual. Bring up the compiler and try a few twists on the idea. Talk to your friends. Know your topic inside and out, and the discussion will flow more easily.

***Study the work of those who succeed.*** What books and what writers have helped you most? Go back and read them again, not for their content but for their methods. How have they organized their material? What details have they included, and what have they omitted? What are their examples like? Do they use short sentences or long? What technical figures do they provide? There's no sin in imitating the successful.

***Use your own voice.*** The best technical writing carries the imprint of a human being. Present your explanations to the readership as though you were explaining it to a friend across a desk. You don't have to "ummm" and "ahhh," and the diction may be more formal than ordinary speech, but there's little profit in pushing it into third person passive. You wrote the program to make BIOS calls; don't say, "The program was written to make BIOS calls."

These are broad principles to ease your way into what we consider the technical writer's mindset. The details involving margins, word processor formats, code listings, and so on have been collected into *The TURBO TECHNIX Authors' Guide*, which is yours for the asking. Call or write for a copy. Read it thoroughly and mull it over.

That done, call and speak to one of the editorial technical staff about article concepts. We do read unsolicited manuscripts, but you have to keep in mind that we have an entire year's worth of issues in progress here at any given time. Someone else already may have sold us the concept you're thinking about. Certainly offer your ideas, but don't be offended if we have to say, "It's been done." Ask what we'd like to have but haven't assigned already. Somewhere we'll find a concept that generates some mutual excitement.

The goal is to make programming comprehensible, even to those who do not consider themselves programmers. Take up the challenge. Ice welded to iron, after all, produces light. ■

—*Jeff Duntemann*

# You do the creative stuff. We'll write the code.

## SYSTEM BUILDER™ $\$149^{95}$ & REPORT BUILDER™ $\$129^{95}$ automate Turbo Pascal programming.

**It's a state-of-the-art program generator that automatically builds a relational database application for you in just seconds. You just paint your screen and datafile layouts.**

**SO EASY. . . ideal for entry level "coders" to produce relational database systems without coding. (Entry level guide with sample On-disk systems is provided.) SO POWERFUL . . . it provides programming professionals with more flexibility and horsepower than any development tool on the market (guide is provided.)**

**"I think it's wonderful . . . prospective buyers should seriously consider DESIGNER even before dBASE III."** *Mr. Greg Weale*
*Corporate Accounts Manager, Computerland*

**"We used DESIGNER last year to program a major application. It saved our programmers so much time. We now use DESIGNER instead of dBASE III as our development standard."**
*Mr. Peter Barge, Director Services Division, Horwath & Horwath*

**"DESIGNER has resulted in significant time savings . . . We use it on classical database applications."**
*Mr. Andy Rudevics, Director Andrasoft Corporation*

### REPORT BUILDER CYCLE:

Key in the report parameters on screen

Print your listings
- New report format for reference
- Report element layout

**Press a key, wait 6 secs***

Key in the report data elements on screen

Report Builder automatically writes the program code and links it to your datafile

Print your listing
- Report program source code listings

Compile the report builder code using the Turbo Pascal™ compiler

Attach the new report module to your system menu

### SYSTEM BUILDER CYCLE:

Paint the menu screens

Paint the application screens

**Press a key, wait 6 secs***

Define the datafile(s) on the screen

System Builder automatically writes the program code and combines the datafiles into a relational database

Print your listings
• Program source code listing • Datafile layouts • Self-documenting program (includes screen schematics)

Compile the System Builder code using Turbo Pascal™ compiler

Start using the completed system

*System Builder will generate 2,000 lines of program code in approximately 6 seconds.

## REPORT BUILDER FEATURES:

- Automatically generates Indented, Structured Source Code ready for compiling Turbo Pascal (no programming needed)
- Automatically interfaces to a maximum of 16 Datafiles created with System Builder
- Supports Global Parameters such as Headings, Footers, Lines Per Page, Print Size and Ad Hoc Sorting
- Produces reports containing an unlimited number of Sub-Headings, Sub-Totals and Totals
- Page breaks on Sub-Totals
- Report Builder will generate Report Programs which can contain Report Elements not just restricted to Data Elements. Reports can also include Text Strings, Variables or Computed expressions containing references from up to 16 Datafiles
- Use range input screens produced by System Builder to allow End Users to select portions of a report as needed (i.e. specific account ranges can be requested)
- Produces standalone Report Modules
- Easy-to-use Interface Program to access dBase Files

### SYSTEM BUILDER PERFORMANCE
(Typical 10 screen 8 file/index application)

| TASK | SYSTEM BUILDER | DBASE III™ |
|---|---|---|
| Planning and Design | 60 minutes | 60 minutes |
| Screen Painting | 15 minutes | 3 hours |
| Programming | 2 minutes | 10 hours |
| **Elapsed time to completed system** | **1 hour and 17 minutes** | **14 hours** |

## SYSTEM BUILDER FEATURES:

- Automatically generates Indented, Structured, Copy Book Source Code ready for compiling with Turbo Pascal (no programming needed)
- Paint Application and Menu screens using Keyboard or Microsoft Mouse™
- Finished Application screens all use System Builder's In-Line machine code for exceptional speed
- Use fully prompted Screen Guidance Templates™ to define up to 16 Datafiles per application, each record having an Unlimited Number of fields
- Define up to 16 Index Keys per application database
- Paint functions include:
  —Center, copy, move, delete, insert or restore a line, Go straight from screen to screen with one keystroke
  —Cut and paste blocks of text screen to screen
  —Draw and erase boxes, Define colors and intensities
  —Access special graphic characters and character fill
- Supports an unlimited number of memory variables
- File Recovery Program Generator to make fixing of corrupted datafiles an automatic process
- Automatically modifies datafiles without loss of data when adding/deleting a field
- Menu Generator with unlimited Sub-Menu levels
- Experienced developers can modify the System Builder
- Develop systems for Floppy or Hard Disk
- Modify System Builder's output code to include External Procedures, Functions and Inline Code
- Easy-to-use Interface to access ASCII and dBase Files

**VARs, system integrators and dealers: Your inquires are always welcome.**

**Call us at the numbers shown on coupon.**

**ROYAL AMERICAN** *Technologies Corporation*™

**Royal American Technologies**
**320 Harris Ave, Suite A**
**Sacramento, CA 95838**

**(800) 654-7766**
**In California (800) 851-2555**

Please rush me ____ copies of SYSTEM BUILDER at $149.95 per copy and ____ copies of REPORT BUILDER at $129.95 per copy. I am enclosing $6.00 for postage and handling. Note: California residents please add 6% sales tax.

Name ____________________

Address ____________________

____________________

City ____________________

State ____________ Zip ____________

Phone ____________________

Payment: □ Check □ Money Order

□ Cashiers Check □ AMEX

□ VISA □ MASTERCARD

Expiration date ____________________

Card Number ____________________

Signature ____________________

**30-Day Money-Back Guarantee.** Not copy protected. $10 restocking fee if envelope is opened.

System Requirements—System Builder/Report Builder: IBM PC/XT/AT[1], or similar, with minimum 256K RAM, dual floppy drives, or hard disk, color or monochrome monitor, MS[2] or PC DOS[1] version 2.0 or later, Turbo Pascal Version 2.0 or later (Normal, BCD or 8087 versions).

[1] Trademarks of International Business Machines Corp.
[2] Trademark of Microsoft Corp.
™Turbo Pascal is a registered trademark of Borland International.
™dBASE is a registered trademark of Ashton-Tate.

# REPLACING THE KEYBOARD INTERRUPT

## Capture the keystrokes that DOS and BIOS throw away, and hide the ones your programs shouldn't see.

*Neil J. Rubenking*

WIZARD

When you hit a key on the PC keyboard, a microprocessor in the keyboard itself sends a signal to an I/O port in the PC. The PC interprets the signal and takes action. If you press a shift key, it notes the shift state. If you press a non-shift key, it puts information into the keyboard buffer. When your program reads a key, it gets key code information from this buffer. However, the keyboard signal contains information that you can't normally get. It sends signals when any key is pressed *or* released, and it sends signals for key combinations like Alt-Home that the PC's BIOS ignores. In order to get this information, you have to intercept that keyboard signal before the BIOS can get it.

When the keyboard sends a signal, it causes a *hardware interrupt*; whatever else the PC is doing, it spends a moment servicing the keyboard. Because this interrupt could happen in the middle of another process without waiting its turn, it is called an *asynchronous* interrupt. The Hardware Keyboard Interrupt is a routine in the BIOS that handles these signals from the keyboard. At the start of the PC's memory map, there is a table of vectors (addresses) for such BIOS routines called the *interrupt vector table.* The Hardware Keyboard Interrupt routine is number 9 in this table. When the keyboard sends a code, the PC transfers control to the address stored in the INT 9 vector. (When a program calls for keyboard input, it uses another keyboard interrupt, number 16H. In this article, the phrase *keyboard interrupt* always refers to Interrupt 9.)

The keyboard sends a signal telling exactly which key was pressed (the *make code*), and also sends a *break code* when a key is released. Every time you

*continued on page 10*

*Bradley Ream*

LISTING 1: ERROR.INC

```
(*ERROR.INC
  This is the error handler for ALL the INT9 front end
  demo programs*)
TYPE
  string2 = string[2];
  string4 = string[4];

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

  FUNCTION HexByte(B : Byte) : string2;
  BEGIN
    HexByte := HexDigit[B SHR 4]+HexDigit[B AND $F];
  END;

  FUNCTION Hex(I : Word) : string4;
  BEGIN
    Hex := HexByte(Hi(I))+HexByte(Lo(I));
  END;


{$F+}  PROCEDURE My_Error; {$F-}
  BEGIN
    SetIntVec(Kbd_Int, Kbd_vec); {restore OLD INT9}
    IF (ExitCode <> 0) OR (ErrorAddr <> NIL) THEN
      BEGIN
        Assign(Output,'');
        ReWrite(OutPut);
        WriteLn(#7);
        IF ExitCode = $FF THEN
          WriteLn('USER BREAK')
        ELSE
          BEGIN
            WriteLn('Critical Error # ',HEX(ExitCode));
            Write('AT PROGRAM LOCATION ');
            WriteLn(HEX(seg(ErrorAddr^)),':',Hex(ofs(ErrorAddr^)));
          END;
      END;
    ExitProc := Exit_Vec;        {restore previous ExitProc}
  END;
```

LISTING 2: ACCEL.PAS

```
PROGRAM Accel;
USES Crt,Dos;
  (* ====================================== *)
  (* This program demonstrates a method for  *)
  (* accelerating the motion of an arrow-key *)
  (* controlled character on the screen.     *)
  (* If a "direction" key is held down, the  *)
  (* character moves in larger and larger    *)
  (* jumps, up to a preset "Speed Limit".    *)
  (* It's easy to set the SPEED back down to *)
  (* 1 whenever a new direction is chosen -- *)
  (* the catch is to reset it when the       *)
  (* SAME direction key is RELEASED.         *)
  (* ====================================== *)
{============}
{BEGIN INCLUDE}
{============}
CONST
  KR : Boolean = False;{KeyReleased FLAG}
  Kbd_Int = 9;
VAR
  Kbd_Vec, Exit_Vec : Pointer;

  {$I ERROR.INC}
```

## INTERRUPTS

*continued from page 8*

press or release a key the Keyboard Interrupt receives a signal. In most cases, the INT 9 routine analyzes the key and puts the result in the keyboard buffer. However, the keyboard sends more information than the BIOS gives to your program. For example, the BIOS doesn't tell you when a non-shift key is released, nor does it directly tell you when a shift key is pressed. Also, it disregards quite a few logical and useful key combinations such as the Alt-Shift of the numeric keypad keys. To get this information, you have to change the Keyboard Interrupt Vector to point to an Interrupt Service Routine (ISR) of your own devising. There's no need to rewrite the whole BIOS routine, because you can pass control back to the original INT 9 when you've finished taking the information you want. The signal from the keyboard remains available until you send a particular Reset code back to the keyboard.

DOS offers standard services to fetch or change the value of any interrupt vector. Turbo Pascal 4.0 includes a pair of routines that call on these DOS services: **GetIntVec** and **SetIntVec**. You can create your own ISR in **INLINE** code, save the old vector using **GetIntVec**, and install your new ISR with **SetIntVec**. Your routine will intercept the keyboard's signals and process them, then pass the signals on to the BIOS. Your routine can even prevent some keys from reaching the BIOS, or take over some of INT 9's regular functions. The four sample programs included with this article demonstrate applications of this technique.

### THE DANGERS

Of course, as soon as you start calling interrupts and inserting **INLINE** machine code, you give up portability. The very purpose of a language compiler is to allow the programmer to write in a high-level language and not worry about what machine code the compiler generates. When you put **INLINE** machine code directly into a program, you lose this advantage. The possibilities for error are much greater at the machine-code level, and can have much wider consequences. Also, Terminate and Stay Resident programs (TSRs) such as SideKick often trap INT 9 themselves in order to be able to pop up on a particular keypress. There is a chance your program may be incompatible with some TSRs, but the results are worth the risk.

## WHAT YOU MUST DO

Any program that contains an interrupt handler has to be able to perform a number of functions. It must be able to change the appropriate interrupt vector to point to the address of the new ISR, and it must be able to retain the previous value of that vector. Your program absolutely *must* restore this value when it finishes. If a program quits without restoring the keyboard interrupt, the next time you press a key the PC will transfer control to the portion of memory you have just vacated, which now contains random bytes. At this point you'll probably have to turn off the computer, because the keyboard won't respond at all.

But what if your program crashes? You still must restore the interrupt even if the program fails. Turbo Pascal 4.0 provides the *exit procedure* facility for just such a problem. The exit procedure gets control when a program ends, even if it crashes with a runtime error. The exit procedure doesn't prevent a crash, nor does it allow you to fix things and return to your normal program logic, but it does allow you to perform some cleanup. If you restore the interrupt in your error handler, you

*continued on page 12*

```
  PROCEDURE CLI; INLINE($FA); {INLINE procedures are NICE!}
  PROCEDURE STI; INLINE($FB);

  PROCEDURE INT9_ISR(_Flags, _CS, _IP, _AX, _BX, _CX, _DX,
                     _SI, _DI, _DS, _ES, _BP:word);
  INTERRUPT;
  (* ======================================== *)
  (* This procedure gets ahead of the normal  *)
  (* interrupt 9 and checks if the current    *)
  (* character is a KEYPRESS code or a KEY    *)
  (* RELEASE -- if the latter, the typed      *)
  (* constant "KR" is set to TRUE (= 1).      *)
  (* ======================================== *)
  BEGIN
  Inline(
    $9C/               {PUSHF          ;Save flags}
    $E4/$60/           {IN   AL,$60  ;Read the keyboard port}
    $A8/$80/           {TEST AL,$80  ;Is the high bit set?}
    $74/$05/           {JZ   Press   ;If not, skip to "Press"}
    $C6/$06/>KR/$01/  {MOV  BYTE PTR [>KR],+$01 ;If so, make KR TRUE}
{Press:}
    (* ============================ *)
    (* CHAIN to the regular INT 9   *)
    (* ============================ *)
    $9D/               {POPF           ;Restore the flags}
    $A1/>KBD_VEC+2/    {MOV  AX,[>KBD_VEC+2] ;Old vector seg to AX}
    $8B/$1E/>KBD_VEC/ {MOV  BX,[>KBD_VEC]   ;Old vector ofs to BX}
    $87/$5E/$0E/       {XCHG BX,[BP+$0E] ;Swap ofs w/ return address}
    $87/$46/$10/       {XCHG AX,[BP+$10] ;Swap seg w/ return address}
    $89/$EC/           {MOV  SP,BP ;UNDO procedure's entry code}
    $5D/               {POP  BP}
    $07/               {POP  ES}
    $1F/               {POP  DS}
    $5F/               {POP  DI}
    $5E/               {POP  SI}
    $5A/               {POP  DX}
    $59/               {POP  CX}
    $CB);              {RETF ;in effect, JMP to old vector}
  END;

  FUNCTION KeyReleased : Boolean;
  (* ================================ *)
  (*  Returns the state of the flag   *)
  (*  KR and resets it to FALSE       *)
  (* ================================ *)
   BEGIN
     CLI; {Don't want it changing DURING this!}
     KeyReleased := KR;
     KR := False;
     STI; {OK, can change now}
   END;
{=============}
{END INCLUDE  }
{=============}


   PROCEDURE Do_Demo;
   (* ======================================== *)
   (* Here begins the DEMO procedure that uses *)
   (* the ISR above.  It responds to the four  *)
   (* arrows keys and to "U", "A", and "Q".    *)
   (* Move around with the arrow keys for a    *)
   (* while, and then hit "A" to engage the    *)
   (* Accellator.  "U" will Unaccelerate the   *)
   (* arrow keys, and "Q" is the signal to     *)
   (* Quit.                                    *)
   (* ======================================== *)
```

*continued from page 11*

avoid having a program crash become a complete system crash. Also, every unit in your program can have its own exit procedure, and all the exit procedures will execute when the program ends. The file ERROR.INC (Listing 1) contains an exit procedure error handler that all the example programs use.

The example programs I've devised all intercept the INT 9 vector and replace it with one pointing to a custom keyboard service routine. The program logic for each demo program runs as follows:

1. Save the old interrupt vector
2. Install the new ISR
3. Save the old ExitProc
4. Enable the new ExitProc
5. Demonstrate the ISR
6. Reinstall the old interrupt

The last demo program, **MoreKey**, is different from the others in that all the interrupt code is in a separate unit, but the sequence of events it follows is the same.

### SAMPLE USES

The simplest ISR just "tastes" the signal from the keyboard port before passing it on. For example, the ISR in the sample program **Accel** merely checks if the scan code is a break code. If so, it sets a flag. Then it passes control on to the original INT 9.

The ISR in **ShKey** does a little more work before handing over control. It compares the received scan code to the codes of the seven shift keys. If a code matches, it sets a flag. With this routine, you can say, "Press any key when ready," and really mean *any* key.

**NoReboot** prevents anyone from rebooting the computer with Ctrl-Alt-Del while the program is running by suppressing the Del

```
CONST
  UKey = #72;  {SCAN codes for the arrow keys}
  DKey = #80;
  LKey = #75;
  RKey = #77;
TYPE
  direction = (Up, Down, Left, Right);
VAR
  CRow, CCol            : Byte;
  accel                 : Boolean;
  CH, CH2, Last_Arrow   : Char;
  M, Speed              : Byte;
CONST
  Speed_Limit = 8;
  Mark        = #$E9;{theta character}
  unmark      = #$20;{space character}
  Arrows : SET OF Char = [UKey, DKey, LKey, RKey];

  PROCEDURE RevVideo;
  BEGIN
    TextColor(Black);
    TextBackground(White);
  END;

  PROCEDURE initialize;
  BEGIN
    TextBackground(black);
    ClrScr;
    RevVideo;
    Write('    MOVE with 4 arrow keys.');
    Write('  [A]ccel, [U]naccel, [Q]uit.');
    Write('              Speed:   ');
    TextBackground(Black);
    TextColor(White);
    Speed      := 1;
    CRow       := 12;
    CCol       := 40;
    Last_Arrow := #0;
    Accel      := False;
  END;

  PROCEDURE PutAChar(co, ro, fore, back : Byte; CH : char);
  (* ==================================== *)
  (* At location (co,ro), write character  *)
  (* CH with color specified by the fore-  *)
  (* and background attributes.            *)
  (* ==================================== *)
  BEGIN
    TextColor(fore);
    TextBackground(back);
    GoToXY(co, ro);
    Write(CH);
  END;

  PROCEDURE Move_Increment(D : direction);
  (* ======================================= *)
  (* Move the marker in the given direction  *)
  (* by as many spaces as the current SPEED. *)
  (* If we hit the edge, beep and set speed  *)
  (* back to one.                            *)
  (* ======================================= *)

    PROCEDURE beep;
    BEGIN
      Sound(1000); Delay(50);
      Sound(2000); Delay(50);
      NoSound;
    END;
```

key. If it detects a Del, it resets the keyboard without ever letting the BIOS see the Del keystroke.

The three examples above are subtle, nosing about the edges of the BIOS interrupt. They steal a little data, or prevent the BIOS from doing its job. **MoreKey**, the fourth sample, actually takes over the function of the BIOS and creates useful new key codes not provided by the standard keyboard interrupt.

## THE DEMO PROGRAMS

**Accel.** Listing 2 shows a sample program that puts an accelerator in your arrow keys. Many programs move a marker around the screen using these keys. It can be very tiresome to move from one edge of the screen to the other one space at a time. The automatic accelerator causes the cursor to move faster when the user holds down a key. You can implement this fairly easily by keeping a **speed** variable, and incrementing it every time the key pressed is the same as the previous key.

This almost works. However, you need to be able to *de*celerate when you get close to your destination. When the user takes their finger off the key, you need to set the speed back to minimum.

Every key produces a *make* code when you press it and a *break* code when you release it. The two codes are identical except for the highest bit, set to 0 for a make code and 1 for a break code. The ISR in **ACCEL** simply tests for a break code—one with the high bit set to 1—and sets the Boolean typed constant flag **KR** to **True** if it finds one.

Note the unusual procedure declaration for procedure **INT9_ISR**. This is an *interrupt procedure*, a new feature of Turbo Pascal 4.0. The keyword **INTERRUPT** tells 4.0 to save and restore all the registers at the start and end of this procedure. It also sets the DS register to the main program's Data Segment, so you have

*continued on page 14*

```
  BEGIN
    {FIRST blank the old location }
    PutAChar(CCol, CRow, white, black, unmark);
    CASE D OF
      Up    : CRow := CRow-1;
      Down  : CRow := CRow+1;
      Left  : CCol := CCol-1;
      Right : CCol := CCol+1;
    END;
    IF CRow < 2  THEN
      BEGIN CRow := 2;  speed := 1; beep; END;
    IF CRow > 24 THEN
      BEGIN CRow := 24; speed := 1; beep; END;
    IF CCol < 1  THEN
      BEGIN CCol := 1;  speed := 1; beep; END;
    IF CCol > 80 THEN
      BEGIN CCol := 80; speed := 1; beep; END;
    {NOW mark the new location }
    PutAChar(CCol, CRow, black, white, Mark);
  END;

BEGIN                         {procedure Do_Demo;}
  Initialize;
  PutAChar(CCol, CRow, black, white, Mark);
  REPEAT
    REPEAT
      CH := #0; CH2 := #0;
      REPEAT UNTIL KeyPressed OR KeyReleased;
      IF KeyPressed THEN
        BEGIN
          CH := ReadKey;
          IF (CH = #0) AND KeyPressed THEN
            CH2 := ReadKey
          ELSE CH := UpCase(CH);
        END
      ELSE  {A key was released}
        speed := 0;
    UNTIL ((CH IN ['A', 'U', 'Q']) OR (CH2 IN Arrows));
    IF CH = #0 THEN
      BEGIN
        IF Accel THEN
          IF CH2 = Last_Arrow THEN
            BEGIN
              {Key CH2 is being held down --
               increase speed!}
              IF Speed < Speed_Limit THEN
                Speed := Speed+1;
            END
             ELSE Speed := 1
           ELSE Speed := 1;
           GoToXY(79, 1); Write(speed);
           Last_Arrow := CH2;
           CASE CH2 OF
             UKey : FOR M := 1 TO speed DO
                      Move_Increment(Up);
             DKey : FOR M := 1 TO speed DO
                      Move_Increment(Down);
             LKey : FOR M := 1 TO speed DO
                      Move_Increment(Left);
             RKey : FOR M := 1 TO speed DO
                      Move_Increment(Right);
           END;
         END
      ELSE
        CASE CH OF
          'A' : BEGIN
                  Accel := True;
                  RevVideo;
                  TextColor(Black+Blink);
                  GoToXY(59, 1); Write('ACCELERATED');
                END;
          'U' : BEGIN
                  Accel := False;
                  RevVideo;
                  GoToXY(59, 1); Write('           ');
                END;
          'Q' : ;
        END;
    UNTIL CH = 'Q';
  END;

BEGIN
  CheckBreak := TRUE;
  GetIntVec(Kbd_Int, Kbd_Vec);   {save "old" INT9}
  SetIntVec(Kbd_Int, @INT9_ISR); {install new}
  Exit_Vec := ExitProc;          {save old ExitProc}
  ExitProc := @My_Error;         {install new}
  Do_Demo;                       {show yer stuff!}
  {The interrupt vector gets RESTORED in the ExitProc}
END.
```

LISTING 3: SHKEY.PAS

```
PROGRAM Shift_Key_Pressed;
uses crt, dos;
{=============}
{BEGIN INCLUDE}
{=============}
VAR
  Kbd_Vec, Exit_Vec : pointer;
CONST
  Kbd_Int = 9;

  {$I ERROR.INC}

  PROCEDURE CLI; INLINE($FA); {INLINE procedures are NICE!}
  PROCEDURE STI; INLINE($FB);

CONST
  (* Scan codes for seven shift keys *)
  SC_LeftShift   = 42;
  SC_RightShift  = 54;
  SC_CtrlShift   = 29;
  SC_AltShift    = 56;
  SC_NumLock     = 69;
  SC_ScrollLock  = 70;
  SC_CapsLock    = 58;
  SKP   : Boolean = False;{ShiftKeyPressed flag}
  which : Byte = 0;

  PROCEDURE INT9_ISR(_Flags, _CS, _IP, _AX, _BX, _CX, _DX,
                     _SI, _DI, _DS, _ES, _BP:word);
```

## INTERRUPTS

*continued from page 13*

access to program variables. Figure 1 shows the entry and exit code Turbo Pascal generates for an interrupt procedure.

```
50            PUSH   AX
53            PUSH   BX
51            PUSH   CX
52            PUSH   DX
56            PUSH   SI
57            PUSH   DI
1E            PUSH   DS
06            PUSH   ES
55            PUSH   BP
89E5          MOV    BP,SP
81ECxxxx      SUB    SP,LocalSize
B8yyyy        MOV    AX,SEG DATA
8ED8          MOV    DS,AX
{Body of procedure goes here}
89EC          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
CF            IRET
```

*Figure 1. Entry and exit code for interrupt procedures.*

There is one catch. We just want to peek at what the keyboard is sending and then chain to the old interrupt. This was a snap in Turbo Pascal 3.0, because we could store the old interrupt vector in the code segment by making it a typed constant. A 3.0 program always has one single code segment, so we always knew where the saved interrupt vector was kept. In 4.0, a program can have multiple code segments, and you can't store data in them. We can store the old interrupt vector in a variable of the new "generic pointer" type, but we need access to the data segment in order to locate that variable. Before we chain to the old interrupt, we have to restore all the registers; after doing this we no longer have access to the main program's data segment.

*continued on page 16*

# 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 3.0 and the reason why you need 4.0 today.

### *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 every byte of memory in your computer.

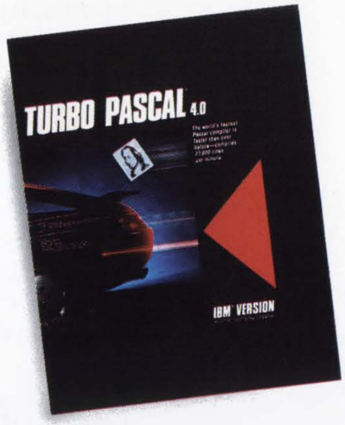
### *4.0 uses logical units for separate compilation*

Pascal 4.0 lets you break up the code gang into "units," or "chunks." These logical modules can be worked with swiftly and separately. Compiling and linking these separate units happens in a flash because your compiling horsepower is better than 27,000 lines a minute.* And 4.0 also includes an automatic project Make.

### *4.0 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

**60-Day Money-back Guarantee****



### *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.

### *4.0 gives you an integrated programming environment*

4.0's integrated environment includes pull-down menus and a built-in editor. Your program output is automatically saved and shown in the output window. You can Scroll, Pan, or Page through all your output and know where everything is all the time. Given 4.0's integration, you can edit, compile, find and correct errors—all from inside the integrated development environment. We even include a command line version of the compiler.

### *Compatibility with Turbo Pascal 3.0*

We've created Version 4.0 to be highly compatible with Version 3.0 and included a conversion program and compatibility units to help you convert 3.0 programs to 4.0.

### *4.0 is all yours for only $99.95*

### *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

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

For the dealer nearest you,or to order now,

Call (800) 543-7543

BORLAND
INTERNATIONAL

BORLAND

*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.

BI 1161B

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

Registered owners have been notified by mail. 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:

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

Name ____________________

Ship Address ____________________

City ____________________ State ________

Zip __________ Telephone (     ) __________

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

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. Please allow 4 to 6 weeks for delivery of Toolboxes.

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.

| Please check box(es) | ***Suggested Retail*** | ***Upgrade†*** | ***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 and handling for each product $ ________
Total amount enclosed $ ________

Please specify diskette size □ 5¼" □ 3½"

Payment: □ VISA □ MC □ Check □ Bank Draft

Credit card expiration date: ______/______

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

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

```
  INTERRUPT;
  BEGIN
    INLINE(
    $9C/                 {PUSHF}
    $E4/$60/             {IN  AL,$60 ;read keyboard port}
    $3C/<SC_CAPSLOCK/    {CMP AL,<SC_CAPSLOCK}
    $74/$1F/             {JZ  Was_Pressed}
    $3C/<SC_LEFTSHIFT/   {CMP AL,<SC_LEFTSHIFT}
    $74/$1B/             {JZ  Was_Pressed}
    $3C/<SC_RIGHTSHIFT/  {CMP AL,<SC_RIGHTSHIFT}
    $74/$17/             {JZ  Was_Pressed}
    $3C/<SC_CTRLSHIFT/   {CMP AL,<SC_CTRLSHIFT}
    $74/$13/             {JZ  Was_Pressed}
    $3C/<SC_ALTSHIFT/    {CMP AL,<SC_ALTSHIFT}
    $74/$0F/             {JZ  Was_Pressed}
    $3C/<SC_NUMLOCK/     {CMP AL,<SC_NUMLOCK}
    $74/$0B/             {JZ  Was_Pressed}
    $3C/<SC_SCROLLLOCK/  {CMP AL,<SC_SCROLLLOCK}
    $74/$07/             {JZ  Was_Pressed}
    (* ================================================ *)
    (* IF you didn't jump by now, it wasn't a shift key  *)
    (* ================================================ *)
    $C6/$06/>SKP/$00/    {MOV BYTE PTR [>SKP],+$00 ;set SKP FALSE}
    $EB/$08/             {JMP SHORT To_Normal}
{Was_Pressed:}
    $C6/$06/>SKP/$01/    {MOV BYTE PTR [>SKP],+$01 ;set SKP TRUE}
    $A2/>WHICH/          {MOV [>WHICH],AL ;remember WHICH key}
{To_Normal:}
    (* ========================== *)
    (* CHAIN to the regular INT 9  *)
    (* ========================== *)
    $9D/               {POPF          ;Restore the flags}
    $A1/>KBD_VEC+2/    {MOV  AX,[>KBD_VEC+2] ;Old vector seg to AX}
    $8B/$1E/>KBD_VEC/  {MOV  BX,[>KBD_VEC]   ;Old vector ofs to BX}
    $87/$5E/$0E/       {XCHG BX,[BP+$0E] ;Swap ofs w/ return address}
    $87/$46/$10/       {XCHG AX,[BP+$10] ;Swap seg w/ return address}
    $89/$EC/           {MOV  SP,BP ;UNDO procedure's entry code}
    $5D/               {POP  BP}
    $07/               {POP  ES}
    $1F/               {POP  DS}
    $5F/               {POP  DI}
    $5E/               {POP  SI}
    $5A/               {POP  DX}
    $59/               {POP  CX}
    $CB);              {RETF ;in effect, JMP to old vector}
  END;

  FUNCTION ShiftKeyPressed : Boolean;
  (* ====================================== *)
  (* Returns the value of flag variable SKP, *)
  (* and resets it to FALSE                  *)
  (* ====================================== *)
  BEGIN
    CLI; {Don't want it changing DURING this!}
    ShiftKeyPressed := SKP;
    SKP := false;
    STI; {OK, can change now}
  END;

  FUNCTION Read_SKP : Byte;
  (* ================================= *)
  (* Returns the value of flag variable *)
  (* "WHICH", and resets it to 0        *)
  (* ================================= *)
  BEGIN
    CLI; {Don't want it changing DURING this!}
    Read_SKP := which;
    which := 0;
    STI; {OK, can change now}
  END;
```

## INTERRUPT

*continued from page 14*

**INLINE** wizard Lane Ferris devised the solution to this problem. We play some tricks with the stack. When Turbo Pascal 4.0 encounters an interrupt procedure, it pushes all the registers starting with AX and BX. That means AX and BX are the last registers to get popped when the procedure ends. We copy the old interrupt vector into AX:BX, then exchange them with the segment and offset of the return address on the stack. The code that we use to simulate the interrupt procedure's exit code leaves AX and BX on the stack. Hence, when we do a RETF (far return), control passes to the old interrupt, and when it ends, control goes to the original return address.

***DOS keeps track of the current shift states using two bytes in low memory (addresses 0040:0017 and 0040:0018).***

**ShKey (Listing 3).** DOS keeps track of the current shift states using two bytes in low memory (addresses 0040:0017 and 0040:0018). Each bit in each of these bytes indicates whether a particular shift key is being held down, or whether a shift lock is active. When you press a shift key, the BIOS updates these shift-state bytes, but doesn't put anything in the keyboard buffer. Consequently, the Turbo function **KeyPressed** does not return **True** when you press a shift key. The ISR in **ShKey** checks each key code received against the codes for Ctrl, Alt, Left Shift, Right Shift, Caps Lock, Num Lock, and Scroll Lock.

If it makes a match, it sets one flag to say a key was pressed and another to say which key it was. As in **Accel**, after the program takes a peek at what the keyboard is sending, it passes control on to the regular keyboard interrupt vector.

The **ShiftKeyPressed** function reads the flag and automatically resets it to **False**. The **Read_SKP** function reads which key it was, and resets the **which** flag to zero. Shift key presses do *not* stack up in the keyboard buffer the way ordinary keys do. If you press four shift keys before the program is ready to recognize one, only the last will be recognized and acted upon.

Picture what would happen if the keyboard sent a key after the **ShiftKeyPressed** function had read the **SKP** flag but before **ShiftKeyPressed** had zeroed **SKP** out. Remember, this kind of interrupt is asynchronous, so it can happen any time, even between those two statements. Another key would come in through the ISR, and the **SKP** flag would be set to **True**, but the next program line would set it to **False**. The new shift key would get lost. In order to avoid this kind of problem, we disable interrupts during functions **ShiftKeyPressed** and **Read_SKP**. Turbo Pascal 4.0 has a new feature that makes this easy. Note the calls to procedures **CLI** and **STI**. These procedures are **INLINE** *directives*, and as such they are much like macros in a macro assembler. Wherever you use the name of an **INLINE** directive, 4.0 directly inserts the **INLINE** code it defines. In this case, the procedures simply disable interrupts at the start of the functions and enable them again at the end.

**NoReboot (Listing 4).** If your program is doing something important, like updating files for million-dollar transactions, you may want to prevent anyone from rebooting the computer. The ISR in this program reads the keyboard port and checks the result against the scan code of the Del key. When it finds a Del code, it resets the keyboard port just as the normal keyboard interrupt does when it's finished with a key. This opens the keyboard to receive the next key. In this case, the ISR has to handle all the housekeeping needed to end a hardware interrupt. It sends an End-of-Interrupt signal to the Interrupt Controller chip and lets the special Turbo Pascal 4.0 interrupt procedure code finish off the interrupt call. The regular keyboard interrupt never sees the Del keystroke.

To suppress the Del key thoroughly, you have to catch both its make code and its break code, since either code can initiate a reboot in combination with Ctrl and Alt. The two codes differ only in the highest bit; 1 for a break code, 0 for a make code. You could compare the received code against both codes, but I chose instead to ignore the highest bit and make only one comparison. Performing an arithmetic AND of the code with 01111111 binary (7FH) forces the highest bit to 0 and leaves the lower seven unchanged.

There's one problem with this technique: certain RAM-resident programs, such as SideKick, can prevent it from working. Even when **NoReboot** is running, you can reboot by bringing up SideKick first. Since the default activation key sequence for SideKick is Ctrl-Alt, Ctrl-Alt-Del will usually bring up SideKick and reboot the computer. On the other hand, SuperKey doesn't interfere. If you pop up SuperKey's macro editor over **NoReboot**, you'll find that you cannot use the Del key. If you seriously need to prevent rebooting, you'll have to make sure SideKick is not in the system.

**MoreKey (Listing 5).** There are quite a few logical key combinations that the BIOS keyboard interrupt simply ignores. For example, Ctrl-Left arrow is a valid combination, but Ctrl-Up is not. Alt-F1 works, but not Alt-Home.

*continued on page 18*

```
{=============}
{END INCLUDE  }
{=============}

  PROCEDURE Do_Demo;
  VAR
    CH : Char;
  BEGIN
    ClrScr;
    WriteLn('   KEYBOARD INTERRUPT DEMO "Shift Keys"');
    WriteLn('   ===================================');
    WriteLn;
    Write('   Press the various shift keys on the ');
    WriteLn('keyboard.  The normal "KeyPressed"');
    Write('   function doesn''t notice these keys.  ');
    WriteLn('But the new "ShiftKeyPressed"');
    WriteLn('   notices!  Hit <Ctrl><Break> to quit.');
    REPEAT
      REPEAT UNTIL KeyPressed OR ShiftKeyPressed;
      WHILE KeyPressed DO CH := ReadKey;
      CASE Read_SKP OF
        SC_LeftShift  : WriteLn('Left Shift');
        SC_RightShift : WriteLn('Right Shift');
        SC_CtrlShift  : WriteLn('Control Shift');
        SC_AltShift   : WriteLn('Alt Shift');
        SC_NumLock    : WriteLn('Num Lock');
        SC_ScrollLock : WriteLn('Scroll Lock');
        SC_CapsLock   : WriteLn('Caps Lock');
      END;
    UNTIL FALSE;
  END;

BEGIN
  CheckBreak := TRUE;
  GetIntVec(Kbd_Int, Kbd_Vec);    {save "old" INT9}
  SetIntVec(Kbd_Int, @INT9_ISR); {install new}
  Exit_Vec := ExitProc;           {save old ExitProc}
  ExitProc := @My_Error;          {install new}
  Do_Demo;                        {show yer stuff!}
  {Old interrupt is restored by ExitProc}
END.
```

LISTING 4: NOREBOOT.PAS

```
PROGRAM No_Reboot;
{=============}
{BEGIN INCLUDE}
{=============}
Uses Crt, Dos;
CONST
  D_Key = 83; (* SCAN code of the Del key *)
  Kbd_Int = 9;

VAR
  Kbd_Vec, Exit_Vec : Pointer;

{$I ERROR.INC}

  PROCEDURE INT9_ISR(_Flags, _CS, _IP, _AX, _BX, _CX, _DX,
                     _SI, _DI, _DS, _ES, _BP:word);
  INTERRUPT;
  (* ======================================= *)
  (* This routine suppresses the <Del> key.  *)
  (* If it detects either a "make" or a      *)
  (* "break" from the <Del> key, it simply   *)
  (* resets the keyboard.  Without <Del>     *)
  (* there's no way to enter <Ctrl><Alt><Del> *)
  (* so you can't reboot.                    *)
  (* ======================================= *)
```

## INTERRUPTS

*continued from page 17*

**MoreKey** enables thirteen useful new keys, all previously unrecognized Alt-key combinations.

When you press an ordinary key, the BIOS keyboard interrupt inserts two bytes into the keyboard buffer. These bytes are the ASCII code for that key and the scan code that produced it. For keys that don't have ASCII equivalents, like the function keys and arrow keys, it inserts ASCII code 0 followed by an extended scan code.

***When you press an ordinary key, the BIOS keyboard interrupt inserts two bytes into the keyboard buffer.***

Appendix E in the *Turbo Pascal 4.0 Owner's Handbook* lists the key codes returned by many special key combinations. Unfortunately, the list is not entirely correct. Any key on that chart with a code greater than 132 is *only* valid if SuperKey is loaded. The BIOS ignores them. **MoreKey** uses the codes from this list for ten of its new keys and extrapolates the list for the other three. **MoreKey** builds on the techniques introduced in the other examples, and adds the ability to insert key codes in the keyboard buffer, just as the BIOS INT 9 does.

**MoreKey**'s new keys are the Alt-Shift of the 13 keypad keys—the nine-key numeric pad itself, plus Ins, Del, and the gray + and – keys. However, there's a catch—normally you would use Alt plus the keypad to enter special ASCII codes. If you press Alt, type a number on the keypad, and release Alt, the ASCII character corresponding to that number appears. This process interferes with using the Alt keypad another way. **MoreKey** solves the problem as SuperKey does, by requiring that you press Left Shift-Alt for those special ASCII codes.

The first thing the **MoreKey** ISR does is check the shift states in the BIOS data area. The byte at address 0040:0017 contains this information. In this byte, the eighth bit reflects the Alt state and the second bit the Left Shift state; if the bit is 1, the corresponding shift state is *on*. If Alt is off, it immediately passes control to the regular keyboard interrupt. If both the Alt and Left Shift states are on, it also hands over control. And if the received signal is a keyboard break code, or if it's less than the code for Home or greater than the code for Del, it gives control to the regular interrupt right away.

Any keyboard signals that made it through these tests are Alt plus keypad codes. When one of these codes is received, the keyboard is reset so it can receive more keys and deal with the received code. The *Turbo Pascal 4.0 Owner's Handbook* extended codes for these keys are the scan codes plus 67H, so that's what you put into the keyboard buffer, with a zero for the ASCII code.

The code that does the buffer-filling is almost identical to the corresponding code in the BIOS, except that it does not check for a full buffer. It does the following: puts the two bytes of key code into the keyboard buffer at the location marked by the buffer pointer **Tail**; advances **Tail** by two bytes; and if **Tail** points to the end of the buffer, the code resets **Tail** to the beginning. That's it.

**MoreKey** differs from the other examples in that *all* of its interrupt

*continued on page 20*

```
  BEGIN
    INLINE(
    $FB/               {STI                ;Allow interrupts}
    $9C/               {PUSHF              ;Save the flags}
    $E4/$60/           {IN   AL,$60        ;READ the keyboard port}
    $24/$7F/           {AND  AL,$7F        ;Mask off "break bit"}
    $3C/<D_KEY/        {CMP  AL,<D_KEY ;Is it a "Del" key?}
    $74/$18/           {JZ   GetOut        ;If so, throw it away}
    (* ============================ *)
    (* CHAIN to the regular INT 9   *)
    (* ============================ *)
    $9D/               {POPF             ;Restore the flags}
    $A1/>KBD_VEC+2/    {MOV  AX,[>KBD_VEC+2] ;Old vector seg to AX}
    $8B/$1E/>KBD_VEC/  {MOV  BX,[>KBD_VEC]   ;Old vector ofs to BX}
    $87/$5E/$0E/       {XCHG BX,[BP+$0E] ;Swap ofs w/ return address}
    $87/$46/$10/       {XCHG AX,[BP+$10] ;Swap seg w/ return address}
    $89/$EC/           {MOV  SP,BP ;UNDO procedure's entry code}
    $5D/               {POP  BP}
    $07/               {POP  ES}
    $1F/               {POP  DS}
    $5F/               {POP  DI}
    $5E/               {POP  SI}
    $5A/               {POP  DX}
    $59/               {POP  CX}
    $CB/               {RETF ;in effect, JMP to old vector}
{GetOut:}
    $E4/$61/           {IN   AL,$61  ;Read Kbd controller port}
    $88/$C4/           {MOV  AH,AL}
    $0C/$80/           {OR   AL,$80  ;Set the "reset" bit and}
    $E6/$61/           {OUT  $61,AL  ;  send it back to control}
    $86/$C4/           {XCHG AH,AL   ;Get back control value}
    $E6/$61/           {OUT  $61,AL  ;  and send it too}
    $9D/               {POPF         ;Restore the flags}
    $FA/               {CLI          ;No interrupts }
    $B0/$20/           {MOV  AL,+$20 ;Send an EOI to the}
    $E6/$20);          {OUT  $20,AL  ;  interrupt controller }
  END;

{=============}
{END INCLUDE  }
{=============}

  PROCEDURE Do_Demo;
  VAR
    L : STRING[80];
  BEGIN
    ClrScr;
    WriteLn('KEYBOARD INTERRUPT DEMO "REBOOT PROHIBITED"');
    WriteLn('===========================================');
    WriteLn;
    Write('IF SideKick is not loaded, you ');
    WriteLn('cannot reboot from within');
    Write('this program.  Try it!  You can ');
    WriteLn('enter text, but you cannot');
    WriteLn('reboot.  Enter a blank line to quit.');
    WriteLn;
    REPEAT
      ReadLn(L);
      WriteLn(L);
    UNTIL L = '';
  END;

BEGIN
  CheckBreak := TRUE;
  GetIntVec(Kbd_Int, Kbd_Vec);   {save "old" INT9}
  SetIntVec(Kbd_Int, @INT9_ISR); {install new}
  Exit_Vec := ExitProc;          {save old ExitProc}
  ExitProc := @My_Error;         {install new}
  Do_Demo;                       {show yer stuff!}
  {Interrupt vector is RESTORED in the ExitProc}
END.
```

LISTING 5: MOREKEY.PAS

```
PROGRAM More_Keys;
(* ============================================= *)
(*  IN this example, the interrupt handler code    *)
(*  is completely contained in the UNIT called     *)
(*  "MOREKEYU".  The unit's initialization part    *)
(*  installs the new interrupt, and its ExitProc   *)
(*  restores the original interrupt.  This is      *)
(*  totally invisible to your program -- just      *)
(*  USE the unit and that's all!                   *)
(* ============================================= *)

USES Crt,Dos,morekeyU;

  PROCEDURE Do_Demo;
  VAR
    CH, DH : Char;
  BEGIN
    ClrScr;
    WriteLn('KEYBOARD INTERRUPT DEMO "More Keys"');
    WriteLn('===================================');
    WriteLn;
    Write('Press various keys and combinations.  ');
    WriteLn('The <Alt> plus keypad combinations');
    Write('now work as in Appendix K of the TURBO ');
    WriteLn('3.0 manual.  ALSO, the <Alt>+number');
    Write('combinations are still available -- you ');
    WriteLn('must press <Alt><LeftShift>+number.');
    WriteLn('Hit <Esc> to end demo.');
    WriteLn;
    REPEAT
      DH := #0;
      CH := ReadKey;
      IF (CH = #0) AND KeyPressed THEN
        BEGIN
          DH := ReadKey;
          CASE DH OF
            #174 : WriteLn('<Alt><Home>');
            #175 : WriteLn('<Alt><Up>');
            #176 : WriteLn('<Alt><PgUp>');
            #177 : WriteLn('<Alt><GreyMinus>'); {*}
            #178 : WriteLn('<Alt><Left>');
            #179 : WriteLn('<Alt><Center>');    {*}
            #180 : WriteLn('<Alt><Right>');
            #181 : WriteLn('<Alt><GreyPlus>');  {*}
            #182 : WriteLn('<Alt><End>');
            #183 : WriteLn('<Alt><Down>');
            #184 : WriteLn('<Alt><PgDn>');
            #185 : WriteLn('<Alt><Ins>');
            #186 : WriteLn('<Alt><Del>');
            (* ================================ *)
            (* NOTE: The three keys marked with a *)
            (* {*} do NOT appear in the list in   *)
            (* Appendix K.  However, the scan     *)
            (* codes are logical in relation to   *)
            (* those that do appear.              *)
            (* ================================ *)
          END;
        END
      ELSE
        CASE CH OF
          #8  : Write(#8,' ',#8);
          #13 : WriteLn;
          #27 : ; {our QUIT signal}
          ELSE Write(CH);
        END;
    UNTIL (CH = #27) AND (DH = #0);
    {i.e., until you press <Esc> }
  END;

BEGIN
  Do_Demo;                              {show yer stuff!}
END.
```

*continued from page 19*

handling code is contained in a unit. The unit is called **MoreKeyU** (Listing 6), and it takes care of everything. Any code you put between a **BEGIN..END** pair at the end of a unit is executed automatically at the start of any program that **USES** the unit. This initialization section is where we put the code to install the new ISR. The exit procedure gets executed at the end of any program that **USES** the unit. We put the code to restore the original interrupt in the exit procedure. Hence the main program **MoreKey** only needs to put **MoreKeyU** in its **USES** statement. Without any further work, the ISR will be installed at the start and removed at the end of the program.

***The initialization section is where we put the code to install the new ISR.***

### USING THESE ROUTINES IN YOUR PROGRAMS

To incorporate the INT 9 ISR routines from the sample programs into your own programs, follow these steps:

1. Put the lines of the main program body that precede **Do_Demo** before the start of your main program.
2. Mark the code between the **BEGIN INCLUDE** and **END INCLUDE** comments as a block and press Ctrl-KW to write it to a file under a name of your choosing. These comments bracket the ISR routine itself and any of its essential declarations. **$INCLUDE** the resulting file in your program.

3. Modify your own exit procedure if you have one. Note: The file ERROR.INC is **$INCLUDE**ed inside this block (Turbo Pascal 4.0 allows nested include files). If you already have an exit procedure in your program, eliminate the line that **$INCLUDE**s ERROR.INC and put the line

   ```
   SetIntVec(Kbd_Int, Kbd_vec);
   ```

   at the end of your exit procedure. This absolutely essential statement restores the computer's original INT 9 vector before your program terminates. If you fail to do this, your system will come down hard as soon as the next key is pressed.

If you follow these instructions, the special keyboard functions added by the ISR will be available in your program. Of course, if you choose to use **MoreKey**, it's much simpler. Just put **MoreKeyU** in your **USES** statement, and that's all you need do. You can convert the other examples to units too, if you wish.

**Do be careful**. The ISRs shown in this article *should* be quite safe, but there may be interactions with RAM-resident programs or other parts of your own program. Test them carefully and satisfy yourself that the routines work correctly in your program. If you make a mistake in the **INLINE** code, the results may be drastic. The keyboard may not respond, or you may get the message "Memory Allocation Error." If this happens, reboot and double check your code.

## WRITING YOUR OWN ISRs

If you're not familiar with assembly language, the safest way to write a new ISR is to modify one shown in the listings. You may want to install a hardware reset switch on your computer before starting to work with ISRs, because almost every error in an ISR requires that you power down the computer. A reset switch allows you to do the equivalent of

*continued on page 22.*

LISTING 6: MOREKEYU.PAS

```
UNIT MoreKeyU; {More_Keys UNIT}
  (* ==================================== *)
  (* Demonstrates a method for enabling   *)
  (* handy key combinations that the BIOS *)
  (* normally throws away.                *)
  (* ==================================== *)
Interface
USES Crt,Dos;
  (* ======================================== *)
  (* There's nothing at all in the INTERFACE  *)
  (* portion of this unit.  It's completely   *)
  (* self-contained.  The initialization      *)
  (* code at the end loads the new Interrupt  *)
  (* Service Routine and the ExitProc puts    *)
  (* back the old interrupt.                  *)
  (* ======================================== *)
Implementation

VAR
  Kbd_Vec, Exit_Vec : Pointer;

CONST
  ROM_Data = $0040; {Segment for ROM data about keyboard }
  KB_Flag  = $0017; {Offset for shift states             }
  Head     = $001A; {Offset for Kbd. buffer HEAD pointer }
  Tail     = $001C; {Offset for Kbd. buffer TAIL pointer }
  KeyBuf   = $001E; {Offset for Keyboard buffer itself   }
  BufEnd   = $003E; {Offset for end of keyboard buffer   }
  Kbd_Int  = 9;

{$I error.inc}

  PROCEDURE INT9_ISR(_Flags, _CS, _IP, _AX, _BX, _CX, _DX,
                     _SI, _DI, _DS, _ES, _BP:word);
  INTERRUPT;
  (* ======================================== *)
  (* This ISR first checks if the <Alt> key   *)
  (* is pressed and the <LeftShift> is NOT    *)
  (* pressed.  IF so, it grabs the scan code *)
  (* waiting in the keyboard and checks if    *)
  (* it is a KEYPAD key.  IF so, it clears    *)
  (* the keyboard and stuffs the keyboard     *)
  (* buffer with the value corresponding      *)
  (* to that key combination as listed in     *)
  (* Appendix K of the TURBO 3.0 manual.      *)
  (*                                          *)
  (* If none of the special cases apply, it   *)
  (* is a normal key, to be given to the      *)
  (* normal keyboard interrupt.               *)
  (*                                          *)
  (* Sounds complicated, but the end result   *)
  (* is that you can use the <Alt>+Keypad     *)
  (* combinations in a program.  If you want *)
  (* <Alt><Number> combinations (e.g., to     *)
  (* get char 219), you use <Alt><LeftShift> *)
  (* <Number>, just as with SuperKey.         *)
  (* ======================================== *)
  BEGIN
  INLINE(
    $FB/              {STI ;Allow interrupts}
    $9C/              {PUSHF ;Save the flags}
    $1E/              {PUSH DS ;Save the Turbo DSeg}
    $E4/$60/          {IN   AL,$60 ;Read the keyboard port}
    $88/$C1/          {MOV  CL,AL}
    $B8/>ROM_DATA/    {MOV  AX,ROM_DATA}
    $8E/$D8/          {MOV  DS,AX ;Set DS to ROM_DATA segment}
    $A0/>KB_FLAG/     {MOV  AL,[>KB_FLAG]}
    $A8/$08/          {TEST AL,$08 ;The 8 bit is ALT}
    $74/$14/          {JZ   Norm_Key ;IF not alt, normal}
    $A8/$02/          {TEST AL,$02 ;The 2 bit is L-Shift}
```

```
    $75/$10/            {JNZ  Norm_Key ;If L-shifted, normal}
    $88/$C8/            {MOV  AL,CL}
    $3C/$80/            {CMP  AL,$80 ;Is it a key-release?}
    $73/$0A/            {JNB  Norm_Key ;If so, treat as normal}
    $3C/$47/            {CMP  AL,$47 ;Below Home is normal}
    $72/$06/            {JB   Norm_Key}
    $3C/$53/            {CMP  AL,$53 ;Above Del is normal}
    $7F/$02/            {JG   Norm_Key}
    $EB/$19/            {JMP  SHORT Special_Key}
{Norm_Key:}
    {If it's not a special key, just CHAIN to the old interrupt}
    $1F/                {POP  DS        ;Restore TURBO DSeg}
    $9D/                {POPF           ;Restore the flags}
    $A1/>KBD_VEC+2/     {MOV  AX,[>KBD_VEC+2] ;Old vector seg to AX}
    $8B/$1E/>KBD_VEC/   {MOV  BX,[>KBD_VEC]   ;Old vector ofs to BX}
    $87/$5E/$0E/        {XCHG BX,[BP+$0E] ;Swap ofs w/ return address}
    $87/$46/$10/        {XCHG AX,[BP+$10] ;Swap seg w/ return address}
    $89/$EC/            {MOV  SP,BP ;UNDO procedure's entry code}
    $5D/                {POP  BP}
    $07/                {POP  ES}
    $1F/                {POP  DS}
    $5F/                {POP  DI}
    $5E/                {POP  SI}
    $5A/                {POP  DX}
    $59/                {POP  CX}
    $CB/                {RETF ;in effect, JMP to old vector}
{Special_Key:}
    $50/                {PUSH AX ;Save the key we got}
    $E4/$61/            {IN   AL,$61  ;Read Kbd controller port}
    $88/$C4/            {MOV  AH,AL}
    $0C/$80/            {OR   AL,$80  ;Set the "reset" bit and}
    $E6/$61/            {OUT  $61,AL  ;  send it back to control}
    $86/$C4/            {XCHG AH,AL   ;Get back control value}
    $E6/$61/            {OUT  $61,AL  ;  and send it too}
    $58/                {POP  AX}
    $04/$67/            {ADD  AL,$67  ;+67h makes it SuperKey code}
    $B4/$00/            {MOV  AH,+$00 ;0 for Scan Code}
    $86/$C4/            {XCHG AH,AL}
    $8B/$1E/>TAIL/      {MOV  BX,[>TAIL]}
    $89/$07/            {MOV  [BX],AX ;Put key in buffer}
    $81/$C3/$02/$00/    {ADD  BX,+$02 ;Advance tail pointer}
    $81/$FB/>BUFEND/    {CMP  BX,>BUFEND ;IF at end of buffer}
    $7C/$03/            {JL   BufOK}
    $BB/>KEYBUF/        {MOV  BX,>KEYBUF ;  set back to beginning}
{BufOK:}
    $89/$1E/>TAIL/      {MOV  [>TAIL],BX}
    $80/$26/>KB_FLAG/$F7/  {AND BYTE PTR [>KB_FLAG],$F7}
                           {Turn off ALT flag }
    $1F/                {POP  DS        ;Restore TURBO DSeg}
    $9D/                {POPF           ;Restore the flags}
    $FA/                {CLI            ;No interrupts }
    $B0/$20/            {MOV  AL,+$20 ;Send an EOI to the}
    $E6/$20);           {OUT  $20,AL  ;  interrupt controller }
  END;

  (* ========================================= *)
  (* You _can_ end a UNIT with just an "END."  *)
  (* statement, but if you end it with a       *)
  (* "BEGIN..END." pair, the code between      *)
  (* that pair will be executed automatically  *)
  (* at the beginning of any program that      *)
  (* USES the UNIT.                            *)
  (* ========================================= *)
BEGIN
  CheckBreak := TRUE;
  GetIntVec(Kbd_Int, Kbd_Vec);    {save "old" INT9}
  SetIntVec(Kbd_Int, @INT9_ISR); {install new}
  Exit_Vec := ExitProc;           {save old ExitProc}
  ExitProc := @My_Error;          {install new}
END.
```

*continued from page 21*

a power-down reboot without actually turning off the power, thereby avoiding electrical stress on your system. When you're developing a new ISR, *always* save your code before you run it, or else you may lose your work.

You might think it's easier to write your routines in Turbo Pascal itself, rather than using **INLINE**. In fact, it is *dangerous*. The 4.0 Runtime Library is not completely reentrant, though it is more so than 3.0. DOS itself is not reentrant, so any routines that call on DOS services are not safe in an ISR. The safest way to avoid reentrancy problems is to stick to **INLINE** code.

***When you're developing a new ISR,* always *save your code before you run it, or you may lose your work.***

The PC keyboard sends a lot of information to the BIOS, but the BIOS throws some of it away. Using Interrupt Service Routines gives you access to this information before the BIOS does. Use it to your advantage. By keeping your ISRs simple you avoid interfering with the BIOS while gaining information that would not normally be available to your program. ■

*Neil Rubenking is a professional Pascal programmer and writer. He can be found daily on Borland's CompuServe Forum answering Turbo Pascal questions.*

*Listings may be downloaded from CompuServe as KEYINT.ARC.*

# FORWARD DECLARATIONS IN TURBO PASCAL

## When chicken calls egg and egg calls chicken, Pascal will call foul—unless you use a forward declaration.

*Allen J. Friedman*

SQUARE ONE

What is forward declaration, and why use it?

The technique is controversial, running contrary to the style and spirit of the Pascal language, but it can be very handy. In this article we will look into the nature of forward declaration, as well as discuss some reasons for limiting the use of this technique.

Pascal was developed in reaction against the common programming practices widely used in other older languages like FORTRAN and COBOL. Source code could appear anywhere and subroutines could be in any order, global variables could be created and destroyed at will, data could be freely converted from one data type to another, and nested chains of GOTO statements snaked lazily around huge programs.

These practices made life easy for some programmers, but they also caused maintenance and reliability nightmares. Pascal, as it was originally defined, was supposed to be pure, without the potential for such abuse. It forced programmers to document, to declare, to keep things in their proper order and therefore preserve some logical sense throughout a program. But it was also a difficult language in which to do useful work.

### FORWARD REFERENCING

*Forward declaration* was introduced as a way of satisfying what the Turbo Tutor documentation calls the Great Underlying Rule of Pascal: All identifiers must be declared before they are used. If your program logic requires calling procedure **P**, but you have not yet declared procedure **P**, you must use a *forward reference* or the program will not compile.

The classic example of forward reference is the circular recursion problem. In short, *circular recursion* means a situation in which two procedures call one another. This is distinct from ordinary (and more common) recursion, in which a single procedure calls itself (see Figure 1). At first glance it looks like both instances are infinite loops, but for simplicity's sake other parts of the required logic are not shown. In a real situation, there must be a Boolean test before each recursive or circularly recursive call so that there is some way to halt the process when it has gone on long enough.

In Figure 2 the two procedures **P1** and **P2** each call the other, and there is no way of arranging them in the source file in order to satisfy the Great Underlying Rule. Pascal demands that both **P1** and **P2** be declared before they are used. The program as it is given in Figure 2 will *not* compile, much less run. Declaring **P2** with the reserved word **FORWARD**, as is done in Figure 3, solves this paradox and makes the program compilable.

Because space is limited, this is not a particularly compelling example of the use of forward declarations. However, in real-world programming there are cases where circular recursion is the best way to go, and where the logic is complex enough to require the use of forward declarations. In such situations, the only alternative to forward declaration is another programming language.

### DEFINING DECLARATION

Note the technique used in forward declaration. We actually declare **P2** twice, once before it is first called (by **P1**) as a forward reference, and then later when we specify the procedure logic. The forward declaration contains the full procedure or function header, with the reserved word **FORWARD** added. The second, or *defining* declaration contains only the procedure name in its header, followed by the type and variable declarations and the code block itself. Note that the procedure's parameters are *not* declared a second time in the definition declaration. This is similar to what is done in specifying a procedure that exists in a separately compiled unit, where the interface declaration of a procedure contains the

*continued on page 24*

## DECLARATIONS

*continued from page 23*

parameter list, whereas the implementation declaration does not. The first declaration allows the compiler to accept **P1**'s call to **P2**, as long as the forward reference is eventually resolved later in the same program block.

Programmers who learned Pascal before encountering Turbo Pascal will probably feel a bit uneasy about all this. It seems like having your cake and eating it too, and we know about free lunches. Forward reference is one of those things, like **GOTO**, that have been included in the language because someone somewhere may really need them, but you really don't expect it to happen to you.

### IMPLEMENTING FORWARD REFERENCES

Forward reference raises potential program maintenance issues because of having two declarations, usually widely separated in the source file. There can also be problems due to the restricted header in the actual declaration. When the code block references the procedure's parameters, it must use them exactly as they appear in the header of the forward declaration. Of course, good programming style and programmer discipline can overcome these objections, just as in the use of **GOTO**.

Forward references do have some implementation restrictions under Turbo Pascal 4.0. A forward-declared subprogram cannot be an **INLINE** subprogram, nor an interrupt procedure. The defining declaration, however, may be a machine-code external subprogram. Finally, the defining subprogram definition may not be another forward declaration.

We have examined forward declarations in some detail, and have suggested situations in which they might be useful. There are some situations involving circular recursion where clarity of the code is actually enhanced by the use of forward declaration. There are potential problems in maintenance and debugging caused by forward references, but when used with care and discipline, they can be a valuable tool for solving certain kinds of programming problems. ■

*Allen J. Friedman is an independent software consultant and freelance writer living in Maine.*

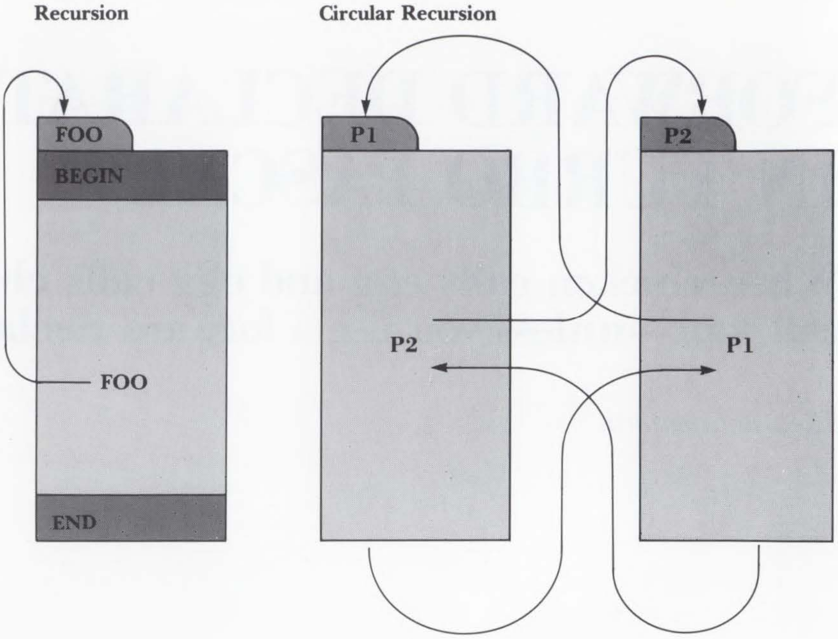


*Figure 1. Recursion and circular recursion.*

```
{ This program WILL NOT compile }
PROGRAM  Example_1;

VAR
   y : integer;


PROCEDURE P1(VAR x : integer);

BEGIN
  { program logic that changes x }
  If x > 0 Then P2(x);
END; { P1 }


PROCEDURE P2(VAR x : integer);

BEGIN
  { program logic that changes x }
  If x < 0 Then  P1(x);
END; { P2 }


BEGIN { Example_1 }
   { ... program logic to set y }
   P1(y);
END.  { Example_1 }
```

*Figure 2. Circular recursion without forward declaration.*

```
{ This program WILL compile... }
PROGRAM Example_2;

VAR
   y : integer;


PROCEDURE P2(VAR x : integer);
                        Forward;


PROCEDURE P1(VAR x : integer);

BEGIN
  If x > 0 Then P2(x);
  { program logic that changes x }
END; { P1 }


PROCEDURE P2;

BEGIN
  { program logic that changes x }
  If x < 0 Then  P1(x);
END; { P2 }


BEGIN { Example_2 }
   { ... program logic to set y }
   P1(y);
END.  { Example_2 }
```

*Figure 3. Circular recursion with forward declaration.*

# SKYDIVING AND THE NUMERICAL METHODS TOOLBOX

## Free-fall into easy numerical solution of terminal velocity with a little help from Messrs. Newton and Raphson.

*Victor Mansfield*

PROGRAMMER

Gabardine sleeves snapping against a rush of wind, sunlight sparkling on a colorful helmet and the fading roar of an airplane engine may not sound very close to computer programming, but a skydiving analysis easily demonstrates the simple yet elegant numerical algorithms in the Numerical Methods Toolbox.

### SKYDIVING EQUATIONS

Figure 1 shows a schematic of the forces acting on you, the skydiver, falling under the gravity force, *mg*, where *m* is the skydiver's mass and *g* is the acceleration due to gravity (32 feet/second/second or 9.81 meters/second/second). This force acts in what is chosen to be the negative direction. The air rushing past you provides a viscous friction force opposing your downward motion. Experiments have shown this friction force is approximated by $kV^2$ where *V* is your velocity relative to the air and *k* is a constant depending upon the surface area you present to the air. Constant *k* changes if you go into spread-eagle position, increasing your friction, or if you pull your arms and legs in (greatly reducing your friction) and dive down head first.

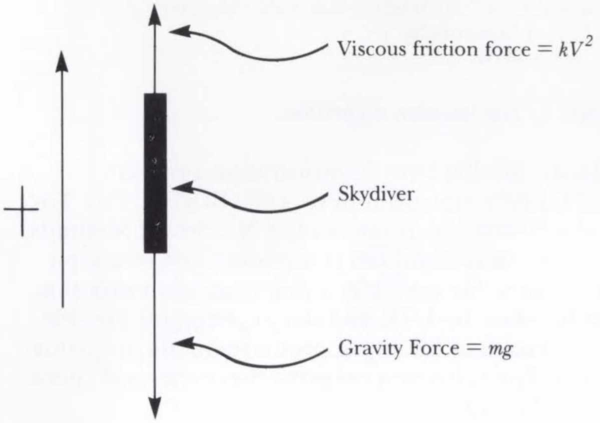


*Figure 1. The dynamics of skydiving.*

Skydiving dynamics are governed by Newton's beloved Second Law: $F = ma$, where *F* is the sum of the external forces, *m* is the mass of the object, and *a* is the acceleration. Acceleration is the time derivative of the velocity, therefore:

$$F = ma = m\frac{dV}{dt}$$

If the forces are written explicitly with the proper signs this becomes:

$$kV^2 - mg = m\frac{dV}{dt} \tag{1}$$

Since the viscous friction force grows with the square of the velocity, there is a critical *terminal velocity* for which the gravitational force is exactly balanced by the friction force. This means $kV^2 = mg$ and $dV/dt = 0$, or *V* is constant at the terminal velocity, $V_t$. In other words, when the forces cancel there is no acceleration and *V* is constant at $V_t$. At terminal velocity, the left side of equation (1) is zero so we can say:

$$mg = kV_t^2 \tag{2}$$

Combine equation (2) with equation (1) to eliminate the constant *k*. The resulting equation can be easily integrated over time to give:

$$V = V_t\,\frac{\exp\left(-\frac{2gt}{V_t}\right) - 1}{\exp\left(-\frac{2gt}{V_t}\right) + 1} \tag{3}$$

Although I don't do it here, you can complete the analysis by integrating equation (3) over time to get the distance fallen with time. For the present, it is enough to examine $V(t, V_t)$. Notice as time *t* goes to infinity that equation (3) shows *V* going to $-V_t$. In

*continued from page 25*

other words, after enough time (we will find out what "enough" means below), the viscous friction forces build up enough to balance gravity and leave you falling blissfully at the constant terminal velocity $V$ going to $-V_t$.

We must have a measurement to find $V_t$. An exhilarating downward velocity of $-90$ miles/hour $= -40$ meters/second is measured after five seconds of falling. Knowing this we plug $V = -40$ m/s, $g = 9.81$ m/s/s, and $t = 5$ seconds into equation (3) to get

$$-40 = V_t \frac{\exp\left(-\frac{98.1}{V_t}\right) - 1}{\exp\left(-\frac{98.1}{V_t}\right) + 1} \tag{4}$$

### ROOTS TO EQUATIONS IN ONE VARIABLE: BISECTION

Equation (4) cannot be solved analytically for $V_t$, but it is a snap numerically. To start the process define a function $F(V_t)$:

$$F(V_t) = V_t \left( \frac{\exp\left(-\frac{98.1}{V_t}\right) - 1}{\exp\left(-\frac{98.1}{V_t}\right) + 1} \right) + 40 \tag{5}$$

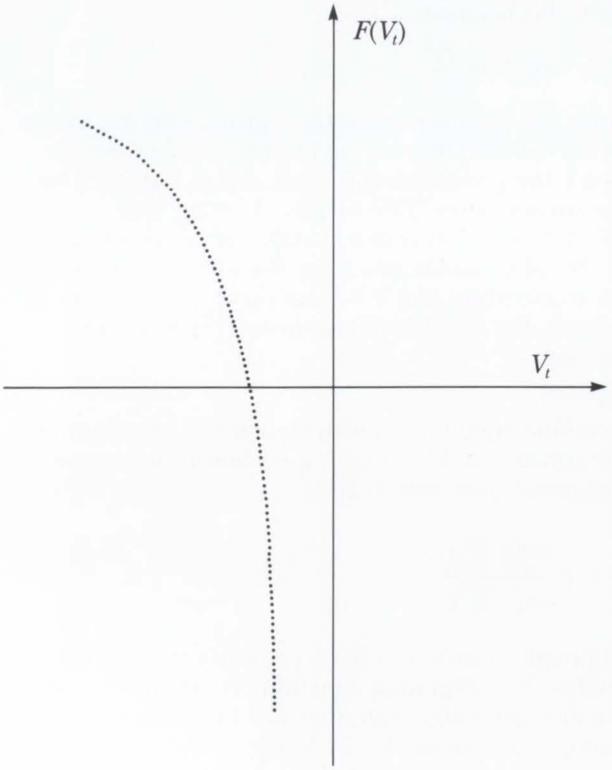


*Figure 2.* F(V$_t$) *versus* V$_t$.

When $F(V_t) = 0$ a root to equation (5) has been found. This root is the numerical value of $V_t$. Figure 2 shows a plot of $F(V_t)$ versus $V_t$. As we will see, it is good practice to get a rough plot of the function before attempting to find its roots or zeros.

The simplest root-finding method is the *bisection iteration method.* All the methods for finding roots in Chapter 2 of the Numerical Methods Toolbox are iterative. Although bisection is slow it cannot fail, and for that reason alone it is valuable. From Figure 2 we see that the function must pass through zero over some interval because it changes sign. The bisection algorithm boils down to this: Evaluate the function at the midpoint of the interval and examine its sign. Replace whichever limit has the same sign as the function at the midpoint with the midpoint. Each iteration reduces by half the interval containing the root. A formal statement of the algorithm to solve equation (5) is shown in Figure 3. This simple algorithm is implemented in Listing 1, taken from the Numerical Methods Toolbox file BISECT.INC.

```
PURPOSE: Find a root for a user-specified function,
F(x), within a user-specified interval, [LeftEnd,
RightEnd], where F(LeftEnd) and F(RightEnd) are of
opposite signs. The user supplies the desired tolerance,
Tol, to which the root is found.
INPUT: LeftEnd, RightEnd, Tol, MaxIter
OUTPUT: Answer, fAnswer, Iter, Error
Step 1: Set Iter = 1. { Iteration variable. }
Step 2: While Iter < MaxIter do Steps 3-6.
   Step 3: MidPoint = (LeftEnd + RightEnd)/2.
   Step 4: If F(MidPoint) = 0 or (RightEnd - LeftEnd)/
             2 < Tol then OUTPUT(Answer =
             Midpoint, fAnswer = F(Answer),
             Iter, Error);
             STOP. {Successful completion.}
Step 5: Iter = Iter + 1.
Step 6: If F(LeftEnd) F(MidPoint) > 0
                        then LeftEnd = MidPoint
                        else RightEnd = Midpoint.
Step 7: OUTPUT(Bisection failed after MaxIter)
        { Unsuccessful. }
         STOP.
```

*Figure 3. The bisection algorithm.*

In the Toolbox the demonstration program BISECT.PAS (not given here) calls BISECT.INC. For all the 70-odd algorithms in the Numerical Methods Toolbox, each algorithm is implemented in a separate include file called by a demonstration program that handles the I/O. The demo programs provide for keyboard or file input, and screen, file, or printer output. They also process error messages and check for legal input.

To see just how simple it is to call the bisection routines, I have included a simplified version of BISECT.PAS, called BISECT2.PAS, in Listing 2. This has all the bells and whistles removed so that it is easier to see the essential root-finding mechanisms at work.

For example, to use BISECT.INC to solve the present problem, we only need to replace the Pascal function **TNTargetF** as originally given in

BISECT.PAS, with a Pascal function that evaluates $F(V_t)$ of equation (5). This is done in BISECT2.PAS. Then run the modified program to get the value of $V_t$. Figure 4 shows a sample session running BISECT2.PAS to solve our problem.

```
B:\>bisect2
Enter LeftEndpoint RightEndpoint separated by a space.
17 -75

Enter the tolerance (1E-8 suggested): 1e-8
Enter maximum number of interations
(100 suggested): 100

Error = 0
                 left endpoint:  1.7000000000E+01
                right endpoint: -7.5000000000E+01
                     Tolerance:  1.0000000000E-08
Maximum number of iterations: 100

    Number of iterations:  28
         Calculated root: -5.8201114371E+01
   Value of the function
  at the calculated root: -7.6135620475E-08

B:\>
```

*Figure 4. A sample session using BISECT2.PAS.*

Knowing $V_t = -58.20$ m/s we can plot equation (3) as a function of time. Figure 5 shows how the skydiver's velocity evolves with time. Notice that after about 15 seconds this velocity levels off to a value indistinguishable from $V_t$. Although I will not do it here, it

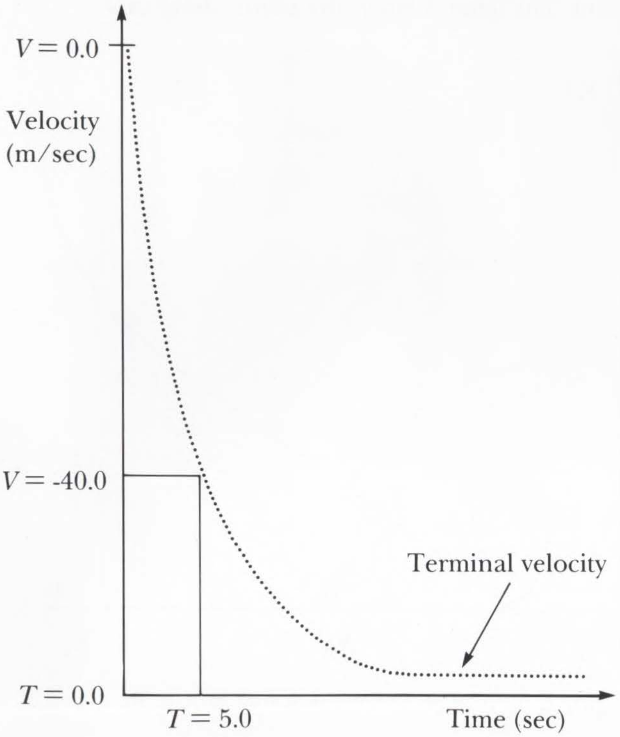


*Figure 5. How the skydiver's velocity evolves over time.*

*continued on page 28*

LISTING 1: BISECT.INC

```
PROCEDURE Bisect(LeftEnd : Real;
                 RightEnd : Real;
                 Tol : Real;
                 MaxIter : Integer;
                 VAR Answer : Real;
                 VAR fAnswer : Real;
                 VAR Iter : Integer;
                 VAR Error : Byte);

  {-------------------------------------------------------------}
  {-                                                           -}
  {-      Turbo Pascal Numerical Methods Toolbox               -}
  {-      (C) Copyright 1986 Borland International.            -}
  {-                                                           -}
  {- Input:  LeftEnd, RightEnd, Tol, MaxIter                   -}
  {- Output: Answer, fAnswer, Iter, Error                      -}
  {-                                                           -}
  {- Purpose: This unit provides a procedure for finding a root -}
  {-        of a user specified function, for a user specified  -}
  {-        interval, [a,b], where f(a) and f(b) are of opposite -}
  {-        signs.  The algorithm successively bisects the      -}
  {-        interval, closing in on the root.  The user must    -}
  {-        supply the desired tolerance to which the root should -}
  {-        be found.                                          -}
  {-                                                           -}
  {- Global Variables: LeftEnd  : real    left endpoint        -}
  {-                   RightEnd : real    right endpoint       -}
  {-                   Tol      : real    tolerance of error   -}
  {-                   MaxIter  : real    max number iterations -}
  {-                   Answer   : real    root of TNTargetF    -}
  {-                   fAnswer  : real    TNTargetF(Answer)    -}
  {-                                      (should be close to 0) -}
  {-                   Iter     :integer  number of iterations -}
  {-                   Error    : byte    error flags          -}
  {-                                                           -}
  {-           Errors: 0: No error                             -}
  {-                   1: maximum number of iterations exceeded -}
  {-                   2: f(a) and f(b) are not of opposite signs -}
  {-                   3: Tol <= 0                             -}
  {-                   4: MaxIter < 0                          -}
  {-                                                           -}
  {-     Version Date: 26 January 1987                         -}
  {-                                                           -}
  {-------------------------------------------------------------}

CONST
  TNNearlyZero = 1E-015;    { If you get a syntax error here, }

  { you are not running TURBO-87.            }
  { TNNearlyZero = 1E-015 if using the 8087   }
  {                       math co-processor.  }
  { TNNearlyZero = 1E-07 if not using the 8087 }
  {                       math co-processor.  }

VAR
  Found : Boolean;

  PROCEDURE TestInput(LeftEndpoint : Real;
                      RightEndpoint : Real;
                      Tol : Real;
                      MaxIter : Integer;
                      VAR Answer : Real;
                      VAR fAnswer : Real;
                      VAR Error : Byte;
                      VAR Found : Boolean);

    {---------------------------------------------------------}
    {- Input:  LeftEndpoint, RightEndpoint, Tol, MaxIter     -}
    {- Output: Answer, fAnswer, Error, Found                 -}
```

*continued on page 28*

```
  {-                                                         -}
  {- This procedure tests the input data for errors.  If     -}
  {- LeftEndpoint > RightEndpoint, Tol <= 0, or MaxIter < 0, -}
  {- then an error is returned.  If one the of the endpoints -}
  {- (LeftEndpoint, RightEndpoint) is a root, then Found=TRUE -}
  {- and Answer and fAnswer are returned.                    -}
  {-----------------------------------------------------------}

VAR
  yLeft, yRight : Real; { The values of function at endpoints. }

BEGIN
  yLeft := TNTargetF(LeftEndpoint);
  yRight := TNTargetF(RightEndpoint);

  IF Abs(yLeft) <= TNNearlyZero THEN
    BEGIN
      Answer := LeftEndpoint;
      fAnswer := TNTargetF(Answer);
      Found := True;
    END;

  IF Abs(yRight) <= TNNearlyZero THEN
    BEGIN
      Answer := RightEndpoint;
      fAnswer := TNTargetF(Answer);
      Found := True;
    END;

  IF NOT Found THEN          { Test for errors }
    BEGIN
      IF yLeft*yRight > 0 THEN
        Error := 2;
      IF Tol <= 0 THEN
        Error := 3;
      IF MaxIter < 0 THEN
        Error := 4;
    END;
END;                         { procedure Tests }

PROCEDURE Converge(VAR LeftEndpoint : Real;
                   VAR RightEndpoint : Real;
                   Tol : Real;
                   VAR Found : Boolean;
                   MaxIter : Integer;
                   VAR Answer : Real;
                   VAR fAnswer : Real;
                   VAR Iter : Integer;
                   VAR Error : Byte);

  {-----------------------------------------------------------}
  {- Input:  LeftEndpoint, RightEndpoint, Tol, MaxIter       -}
  {- Output: Found, Answer, fAnswer, Iter, Error             -}
  {-                                                         -}
  {- This procedure applies the bisection method to find a   -}
  {- root to TNTargetF(x) on the interval [LeftEndpoint,     -}
  {- RightEndpoint]. The root must be found within MaxIter   -}
  {- iterations to a tolerance of Tol. If root found, then it -}
  {- is returned in Answer, the value of the function at the -}
  {- approximated root is fAnswer (should be close to        -}
  {- zero), and the number of iterations is returned in Iter. -}
  {-----------------------------------------------------------}

VAR
  yLeft : Real;
  MidPoint, yMidPoint : Real;

  PROCEDURE Initial(LeftEndpoint : Real;
                    VAR Iter : Integer;
                    VAR yLeft : Real);
```

## SKYDIVING

*continued from page 27*

would be easy to integrate the function $V(t)$ from 0 to 15 seconds to see how far the skydiver falls in 15 seconds. This can be done with equal ease, either analytically or with the Numerical Methods Toolbox. In this time period the skydiver must fall around 600 meters (1970 feet) before reaching terminal velocity. Most of the time you free-fall at terminal velocity rather than accelerate.

### FASTER CONVERGENCE: NEWTON'S METHOD

Even though this problem hardly strains the computer, 28 iterations were needed to get the required accuracy using bisection. (The number of iterations increases with the size of the interval, **RightValue – LeftValue**, or with a smaller tolerance, **TOL**.) In more involved problems, this much calculation could be a problem. Although numerical methods for finding roots to equations is still an active research topic, Isaac Newton developed (without a PC!) one of the most popular and efficient methods used today—*Newton's method.* (This method is also called the *Newton-Raphson method.*) It has a simple geometric interpretation.

Figure 6 shows a function, $F(X)$, which has a root at $X = R$. Let an estimate of the root be $X_n$, which equals the line segment $OB$. The value $F(X_n)$ is labeled $C$ and equals the line segment $BC$. The slope of $F(X)$ at $X_n$ is the tangent to the curve $F(X)$ at that point. The letter $A$ labels the point where this

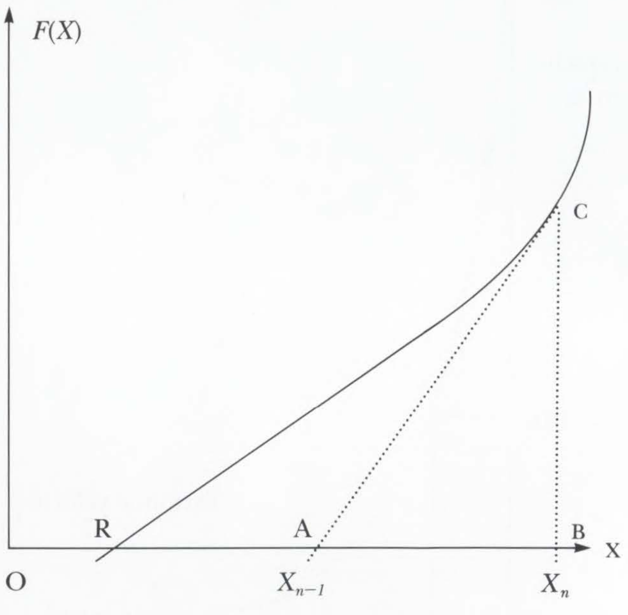


*Figure 6. A function* F(X) *with a root at* X ± R.

tangent line intersects the X-axis. Let this intersection point be the next estimate of the root, $X_{n+1}$ which equals the line segment OA. This is obviously a better estimate of $R$ than $X_n$. From the geometry of Figure 6 we have

$$OA = OB - AB$$

or equivalently:

$$X_{n+1} = X_n - AB \tag{6}$$

By the definition of the tangent function we have:

$$\tan(CAB) = \frac{BC}{AB} \tag{7}$$

From Figure 6 we know that $BC = F(X)$, and from elementary calculus we know that the slope of the line at point $C$ equals $\tan(CAB)$. Thus equation (7) becomes

$$F'(X_n) = \frac{F(X_n)}{AB} \tag{8}$$

where $F'(X)$ is the standard notation for the X-derivative of $F(X)$. Finally, we can eliminate $AB$ from equations (6) and (8) to get

$$X_{n+1} = X_n - \frac{F(X_n)}{F'(X_n)} \tag{9}$$

This is the heart of Newton's method. A more formal presentation of the algorithm is shown in Figure 7.

```
PURPOSE: Finding a root for a user-specified function,
F(x), with a user-specified initial approximation, Guess.
The method uses the derivative of the function to
rapidly converge to an approximate solution whose tolerance is specified by Tol.

INPUT:   Guess, Tol, MaxIter
OUTPUT: Root, Value { = F(Root)}, Iter, Error
Step 1: Set Iter = 1. { Iteration variable. }
  Step 2: While Iter < MaxIter do Steps 3-6.
  Step 3: $\mathbf{Root}_{n-1}$ $\mathrm{Root}_n$ $-F(\mathbf{Root}_n)/F'(\mathrm{Root}_n)$.
  Step 4: If $|\mathbf{Root}_{n+1} - \mathbf{Root}_n| < \mathbf{Tol}$ then
            OUTPUT(Root, Value, Iter, Error);
            STOP.           {Successful completion.}
  Step 5: Iter = Iter + 1.
  Step 6: $\mathbf{Root}_n = \mathrm{Root}_{n+1}$ .
```

*Figure 7. Algorithm for Newton's Method.*

Newton's method has much faster convergence than the bisection method. In the present example, with equivalent starting guesses, Newton's method required six iterations while bisection required 28. When it is approaching a root, the number of significant digits found with Newton's method may double at each iteration! However, it has obvious problems where the function has a zero derivative. For example, imagine trying to find the root $R$ shown in

*continued on page 30*

```
    { Initialize variables. }
  BEGIN
    Iter := 0;
    yLeft := TNTargetF(LeftEndpoint);
  END;                          { procedure Initial }


FUNCTION TestForRoot(X, OldX, Y, Tol : Real) : Boolean;
{-----------------------------------------------------------------}
{-  These are the stopping criteria.  Four different ones are    -}
{-  provided.  If you wish to change the active criteria, simply -}
{-  comment off current criteria (including the appropriate OR)  -}
{-  and remove the comment brackets from the criteria (including -}
{-  the appropriate OR) you wish to be active.                   -}
{-----------------------------------------------------------------}
BEGIN
  TestForRoot :=                          {----------------------}
    (ABS(Y) <= TNNearlyZero)              {- Y = 0              -}
                                          {-                    -}
          OR                              {-                    -}
                                          {-                    -}
    (ABS(X - OldX) < ABS(OldX * Tol))     {- Relative change in X -}
                                          {-                    -}
                                          {-                    -}
(*        OR                           *) {-                    -}
(*                                     *) {-                    -}
(*  (ABS(X - OldX) < Tol)              *) {- Absolute change in X -}
(*                                     *) {-                    -}
(*        OR                           *) {-                    -}
(*                                     *) {-                    -}
(*  (ABS(Y) <= Tol)                    *) {- Absolute change in Y -}
                                          {----------------------}

{-----------------------------------------------------------------}
{- The first criteria simply checks to see if the value of the   -}
{- function is zero. You should probably always keep this        -}
{- criteria  active.                                             -}
{-                                                               -}
{- The second criteria checks relative error in x. This criteria -}
{- evaluates the fractional change in x between interations. Note -}
{- x has been multiplied through the inequality to avoid divide  -}
{- by zero errors.                                               -}
{-                                                               -}
{- The third criteria checks the absolute difference in x        -}
{- between iterations.                                           -}
{-                                                               -}
{- The fourth criteria checks the absolute difference between    -}
{- the value of the function and zero.                           -}
{-----------------------------------------------------------------}

END; { function TestForRoot }


  BEGIN                         { procedure Converge }
    Initial(LeftEndpoint, Iter, yLeft);
    WHILE NOT(Found) AND (Iter < MaxIter) DO
      BEGIN
        Iter := Succ(Iter);
        MidPoint := (LeftEndpoint+RightEndpoint)/2;
        yMidPoint := TNTargetF(MidPoint);
        Found :=TestForRoot(MidPoint, LeftEndpoint, yMidPoint, Tol);
        IF (yLeft*yMidPoint) < 0 THEN
          RightEndpoint := MidPoint
        ELSE
          BEGIN
            LeftEndpoint := MidPoint;
            yLeft := yMidPoint;
          END;
      END;
    Answer := MidPoint;
    fAnswer := yMidPoint;
    IF Iter >= MaxIter THEN
      Error := 1;
  END;                          { procedure Converge }

BEGIN                           { procedure Bisect }
  Found := False;
  TestInput(LeftEnd, RightEnd, Tol, MaxIter, Answer,
  fAnswer, Error, Found);
  IF (Error = 0) AND (Found = False) THEN { i.e. no error }
    Converge(LeftEnd, RightEnd, Tol, Found, MaxIter,
    Answer, fAnswer, Iter, Error);
END;                            { procedure Bisect }
```

LISTING 2: BISECT2.PAS

```
PROGRAM Bisect2;

{--------------------------------------------------------------}
{-                                                            -}
{- Purpose: This program demonstrates the bisection routine with  -}
{-          a bare minimum of calling code.                   -}
{-          No I/O options or error checking code.            -}
{-                                                            -}
{-                                                            -}
{- Include files:   BISECT.INC     procedure Bisect           -}
{-                                                            -}
{-    Version July 1987                                       -}
{--------------------------------------------------------------}


VAR
  LeftEndpoint, RightEndpoint : Real; { Endpoints of the region }
  Answer, yAnswer : Real;             { Root of F(X) }
  Tol : Real;                         { Tolerance }
  Iter, MaxIter : Integer;            { Number of iterations }
  Error : Byte;                       { Flags something wrong}


  {----- HERE IS THE FUNCTION TO FIND A ROOT OF ------}

  FUNCTION TNTargetF(X : Real) : Real;
  BEGIN
    TNTargetF := -40.0+X*(1.0-Exp(-98.1/X))/(1.0+Exp(-98.1/X));
  END; { function TNTargetF }

  {-------------------------------------------------}


  {$I BISECT.INC}  { Load procedure Bisect }


BEGIN
  { Get necessary input. }
  Error := 0;
  WriteLn('Enter LeftEndpoint RightEndpoint seperated by a space.');
  ReadLn(LeftEndpoint, RightEndpoint);
  WriteLn;
  Write('Enter the tolerance (1E-8 suggested): ');
  ReadLn(Tol);
  Write('Enter maximum number of interations (100 suggested): ');
  ReadLn(MaxIter);

  Bisect(LeftEndpoint, RightEndpoint, Tol, MaxIter,
         Answer, yAnswer, Iter, Error);

  { Give resulting output. }
  WriteLn;
  WriteLn('Error = ', Error);
  WriteLn('left endpoint: ':30, LeftEndpoint);
  WriteLn('right endpoint: ':30, RightEndpoint);
  WriteLn('Tolerance: ':30, Tol);
  WriteLn('Maximum number of iterations: ':30, MaxIter);
  WriteLn;
  WriteLn('Number of iterations: ':26, Iter:3);
  WriteLn('Calculated root: ':26, Answer);
  WriteLn('Value of the function  ':26);
  WriteLn('at the calculated root: ':26, yAnswer);
END.


BEGIN
  IF Abs(Slope) <= TNNearlyZero THEN
    Error := 2;            { Slope is zero }
END;                       { procedure CheckSlope }

PROCEDURE Initial(Guess : Real;
                  Tol : Real;
                  MaxIter : Integer;
                  VAR OldX : Real;
                  VAR OldY : Real;
                  VAR OldDeriv : Real;
                  VAR Found : Boolean;
                  VAR Iter : Integer;
                  VAR Error : Byte);


  {----------------------------------------------------------}
  {- Input:  Guess, Tol, MaxIter                            -}
  {- Output: OldX, OldY, OldDeriv, Found, Iter, Error       -}
  {-                                                        -}
  {- This procedure sets the initial values of the above    -}
  {- variables. If OldY is zero, then a root has been       -}
  {- Found and Found = TRUE.  This procedure also checks    -}
  {- the tolerance (Tol) and the maximum number of iterations -}
  {- (MaxIter) for errors.                                  -}
  {----------------------------------------------------------}
```

*continued from page 29*

Figure 8 with an initial estimate of $X_0$. The first attempt at improving the estimate of the root brings you to $X_1$ then to $X_2$. From there you go off into the wild blue yonder. The minimum of $F(X)$ at $X_{min}$ will force the algorithm to oscillate about that point or go off into infinity. (Now you can see why it is good practice to get a rough plot of the function before finding its roots.) In the Numerical Methods Toolbox implementation (Listing 3, RAPHSON.INC) we have an error message that appears if the derivative is approaching zero, and an upper limit, **MaxIter**, to the number of iterations that can be attempted. If these conditions occur, appropriate error messages are generated.

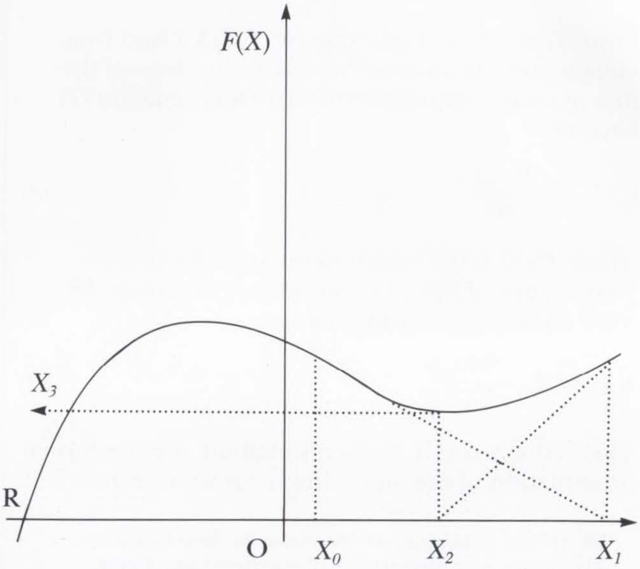


*Figure 8. A function* F(X) *with a zero derivative at* $X_2$.

As with the bisection method, there is a simple demo program for Newton's method included with the Numerical Methods Toolbox. Program RAPHSON2.PAS can be easily modified to solve the function given in equation (5). Listing 4, RAPHSON2.PAS, shows the modified program. Again, the function **TNTargetF** must be changed to evaluate the $F(V_t)$. Additionally, a second function must be provided, **TNDerivF**. This function evaluates $F'(Vt)$, the derivative of the function given in equation (5). A sample session running RAPHSON2 is given in Figure 9.

## MORE FROM CHAPTER 2 OF NUMERICAL METHODS TOOLBOX

As powerful as Newton's method is, there are many situations requiring other techniques. The Numerical Methods Toolbox implements several other algorithms that can handle most of the broad range of root-finding problems encountered in the real world. For example, in situations where it is difficult to calculate the derivative of a function (as Newton's

*continued on page 32*

# Sophisticated User Interfaces in Minutes!

**Put magic in your programs with**

New!
Version 2.0

## The World's Best Code Generator!

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

Users and critics say it all!...

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

*"... the best screen generator on the market."* George Kwascha, TUG Lines, Nov/Dec 87

*"... the Cadillac of prototyping tools for Turbo Pascal. ... Unlike the others, turboMAGIC is extremely flexible. ... [it] clearly offers the greatest variety of options."*
Jim Powell, Computer Language, Jun 87

*"Fast automatic updating of dependent fields adds flair to your input screens. ... turboMAGIC will be a blessing for programmers who would rather not write the user interface for every program."* Neil Rubenking, PC Magazine, 24 Feb 87

*"I was impressed with the turboMAGIC package. ... the procedures created by turboMAGIC are well commented and easy to add to your own code."*
Kathleen Williams, Turbo Tech Report, May/Jun 87

*"... definitely a recommended program for any Turbo Pascal programmer, novice or expert."*
Terry Lovegrove, Library Hi Tech News, Oct 87

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

**sophisticated**
**software**

6586 Old Shell Road, Mobile, AL 36608

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

## SKYDIVING

*continued from page 30*

```
  BEGIN
    Found := False;
    Iter := 0;
    Error := 0;
    OldX := Guess;
    OldY := TNTargetF(OldX);
    OldDeriv := TNDerivF(OldX);
    IF Abs(OldY) <= TNNearlyZero THEN
      Found := True
    ELSE
      CheckSlope(OldDeriv, Error);
    IF Tol <= 0 THEN
      Error := 3;
    IF MaxIter < 0 THEN
      Error := 4;
  END;                        { procedure Initial }

  FUNCTION TestForRoot(X, OldX, Y, Tol : Real) : Boolean;

{---------------------------------------------------------------}
{- These are the stopping criteria.  Four different ones are   -}
{- provided.  If you wish to change the active criteria, simply -}
{- comment off the current criteria (including the preceding OR) -}
{- and remove the comment brackets from the criteria (including -}
{- the following OR) you wish to be active.                    -}
{---------------------------------------------------------------}

  BEGIN

    TestForRoot :=                       {--------------------------}
      (ABS(Y) <= TNNearlyZero)           {- Y = 0                  -}
                                         {-                        -}
            or                           {-                        -}
                                         {-                        -}
      (ABS(X - OldX) < ABS(OldX*Tol))    {- Relative change in X   -}
                                         {-                        -}
(*          or                        *) {-                        -}
(*                                    *) {-                        -}
(*    (ABS(OldX - X) < Tol)           *) {- Absolute change in X   -}
(*                                    *) {-                        -}
(*          or                        *) {-                        -}
(*                                    *) {-                        -}
(*    (ABS(Y) <= Tol)                 *) {- Absolute change in Y   -}
                                         {--------------------------}

{-----------------------------------------------------------------}
{- The first criteria simply checks to see if the value of the   -}
{- function is zero.  You should probably always keep this       -}
{- criteria active.                                              -}
{-                                                               -}
{- The second criteria checks relative error in X. This criteria -}
{- evaluates the fractional change in X between interations.     -}
{- Note that X has been multiplied throught the inequality to    -}
{- avoid divide by zero errors.                                  -}
{-                                                               -}
{- The third criteria checks the absolute difference in X        -}
{- between iterations.                                           -}
{-                                                               -}
{- The fourth criteria checks the absolute difference between    -}
{- the value of the function and zero.                           -}
{-                                                               -}
{-----------------------------------------------------------------}

  END;                        { procedure TestForRoot }

  BEGIN                         { procedure Newton_Raphson }
    Initial(Guess, Tol, MaxIter, OldX, OldY, OldDeriv,
    Found, Iter, Error);

    WHILE NOT(Found) AND (Error = 0) AND (Iter < MaxIter) DO
      BEGIN
        Iter := Succ(Iter);
        NewX := OldX-OldY/OldDeriv;
        NewY := TNTargetF(NewX);
        NewDeriv := TNDerivF(NewX);
        Found := TestForRoot(NewX, OldX, NewY, Tol);
        OldX := NewX;
        OldY := NewY;
        OldDeriv := NewDeriv;
        IF NOT(Found) THEN
          CheckSlope(OldDeriv, Error);
      END;
    Root := OldX;
    Value := OldY;
    Deriv := OldDeriv;
    IF NOT(Found) AND (Error = 0) AND (Iter >= MaxIter) THEN
      Error := 1;
  END;                          { procedure Newton_Raphson }
```

```
B:\>raphson2
Initial Approximation to the root: -17

Tolerance (> 0, suggested 1E-6): 1e-6

Maximum number of iterations (>= 0, suggested 100): 100


Error = 0
        Number of iterations:   6
             Calculated root: -5.8201114685E+01
       Value of the function
                 at the root:  0.0000000000E+00
     Value of the derivative
      of the function at the
             calculated root: -2.4257986428E-01

B:\>
```

*Figure 9. A sample session using RAPHSON2.PAS.*

method requires), the Secant method can often be used. It is similar to Newton's method, but slower.

There are also cases where the roots of real polynomials are needed. Polynomials have multiple roots, making it more efficient to factor out the root from the polynomial after it is found. This reduces the degree of the polynomial (the highest power of the variable) and allows for more accurate and faster root finding of the next root. The factoring (called *deflation*) is carried out for each root as it is found. Sometimes the functions are complex, i.e., involving the square root of -1. Muller's method can find a possibly complex root to a complex function. Finally, it is often necessary to find complex and real roots to a complex polynomial. The powerful and reliable Laguerre's method (which also uses deflation) does this.

Each of these algorithms is carefully implemented in Chapter 2 of the Numerical Methods Toolbox and is called by a simple demonstration program that handles all I/O and error checking. The demo programs are written so that very little coding is needed to adapt it to your particular problem.

### F(GERONIMO)

Even where roots are known to exist, it is not always possible to find them with purely analytical methods. Among other things, the Turbo Pascal Numerical Methods Toolbox provides numerical methods for finding roots. For any given problem, at least one of the methods will be appropriate. Finding your terminal velocity after jumping from an airplane is only a single vivid example, if not an especially practical one—unless your lapheld computer skydives with you. ■

*Victor Mansfield, a professor of physics and astronomy at Colgate University, headed the team that built the Turbo Numerical Methods Toolbox. His principle research interests are in theoretical astrophysics and the philosophy of quantum mechanics.*

*Listings may be downloaded from CompuServe as SKYDIV.ARC.*

LISTING 3: RAPHSON.INC

```
PROCEDURE Newton_Raphson(Guess : Real;
                         Tol : Real;
                         MaxIter : Integer;
                         VAR Root : Real;
                         VAR Value : Real;
                         VAR Deriv : Real;
                         VAR Iter : Integer;
                         VAR Error : Byte);

{----------------------------------------------------------------}
{-                                                              -}
{- Turbo Pascal Numerical Methods Toolbox                       -}
{- (C) Copyright 1986 Borland International.                    -}
{-                                                              -}
{-         Input: Guess, Tol, MaxIter                           -}
{-        Output: Root, Value, Deriv, Iter, Error               -}
{-                                                              -}
{- Purpose: This unit provides a procedure for finding a single -}
{-          real root of a user specified function with a known -}
{-          continuous first derivative, given a user           -}
{-          specified initial guess.  The procedure implements  -}
{-          Newton-Raphson's algorithm for finding a single     -}
{-          zero of a function.                                 -}
{-          The user must specify the desired tolerance         -}
{-          in the answer.                                      -}
{-                                                              -}
{- Global Variables:                                            -}
{-    Guess   : real;    user's estimate of root                -}
{-            Tol     : real;    tolerance in answer            -}
{-            MaxIter : integer; maximum number of iterations   -}
{-            Root    : real;    real part of calculated roots  -}
{-            Value   : real;    value of the polynomial at root -}
{-            Deriv   : real;    value of the derivative at root -}
{-            Iter    : real;    number of iterations it took   -}
{-                               to find root                   -}
{-            Error   : byte;    flags if something went wrong  -}
{-                                                              -}
{-    Errors: 1: Iter >= MaxIter                                -}
{-            2: The slope was zero at some point               -}
{-            3: Tol <= 0                                       -}
{-            4: MaxIter < 0                                    -}
{-                                                              -}
{-  Version Date: 26 January 1987                               -}
{-                                                              -}
{----------------------------------------------------------------}

CONST
  TNNearlyZero = 1E-015; { If you get a syntax error here, you are }
  { not running TURBO-87.                  }
  { TNNearlyZero = 1E-015 if using the 8087    }
  {                       math co-processor.   }
  { TNNearlyZero = 1E-07 if not using the 8087 }
  {                          math co-processor. }

VAR
  Found : Boolean;            { Flags that a root has been Found }
  OldX, OldY, OldDeriv,
  NewX, NewY, NewDeriv : Real; { Iteration variables }

  PROCEDURE CheckSlope(Slope : Real;
                       VAR Error : Byte);

    {-----------------------------------------------}
    {- Input:  Slope                               -}
    {- Output: Error                               -}
    {-                                             -}
    {- This procedure checks the slope to see if it is -}
    {- zero.  The Newton Raphson algorithm may not be  -}
    {- applied at a point where the slope is zero.     -}
    {-----------------------------------------------}
```

LISTING 4: RAPHSON2.PAS

```
PROGRAM Raphson2;

{-----------------------------------------------------------------}
{                                                                -}
{  Turbo Pascal Numerical Methods Toolbox                        -}
{-     (C) Copyright 1986 Borland International.                 -}
{-                                                               -}
{-     Purpose: This sample program demonstrates the             -}
{-              Newton-Raphson algorithm.  This program is very  -}
{-              bare-boned; it contains no I/O checking.         -}
{-                                                               -}
{-  Include Files: RAPHSON.INC          procedure Newton_Raphson  -}
{-                                                               -}
{-       Version Date: 26 January 1987                           -}
{-                                                               -}
{-----------------------------------------------------------------}

VAR
  InitGuess : Real;            { Initial approximation }
  Tolerance : Real;            { Tolerance in answer }
  Root, Value, Deriv : Real;   { Resulting roots and other info }
  Iter : Integer;              { Number of iterations to find root }
  MaxIter : Integer;           { Maximum number of iterations }
  Error : Byte;                { Error flag }
  OutFile : Text;              { Output file }


  {------- HERE IS THE FUNCTION ----------}
  FUNCTION TNTargetF(X : Real) : Real;
  BEGIN
    TNTargetF := -40.0+X*(1.0-Exp(-98.1/X))/(1.0+Exp(-98.1/X));
  END;                         { function TNTargetF }
  {--------------------------------------}

  {------- HERE IS THE DERIVATIVE --------}
  FUNCTION TNDerivF(X : Real) : Real;
  VAR EE : Real;
  BEGIN
    EE := Exp(-98.1/X);
    TNDerivF := (1.0-EE*(1.0+98.1/X))/(EE+1.0)+
                98.1*EE*(EE-1.0)/(X*Sqr(EE+1.0));
  END;                         { function TNDerivF }
  {--------------------------------------}

  {$I RAPHSON.INC}             { Load procedure Raphson }

BEGIN                          { program Newton_Raphson }
  Write('Initial Approximation to the root: ');
  ReadLn(InitGuess);
  WriteLn;
  Write('Tolerance (> 0, suggested 1E-6): ');
  ReadLn(Tolerance);
  WriteLn;
  Write('Maximum number of iterations (>= 0, suggested 100): ');
  ReadLn(MaxIter);
  WriteLn;

  Newton_Raphson(InitGuess, Tolerance, MaxIter,
                 Root, Value, Deriv, Iter, Error);

  WriteLn;
  WriteLn('Error = ', Error);
  WriteLn('Number of iterations: ':30, Iter:3);
  WriteLn('Calculated root: ':30, Root);
  WriteLn('Value of the function':28);
  WriteLn('at the root: ':30, Value);
  WriteLn('Value of the derivative':28);
  WriteLn('of the function at the':28);
  WriteLn('calculated root: ':30, Deriv);
END.                           { program Newton_Raphson }
```

# FLOATING POINT IN TURBO C

## Describe an analog world in a digital fashion—here's what happens beneath the surface.

*Roger Schlafly*

WIZARD

While computers see information only in black and white, the world around us exists in endless shades of gray. Mapping the real world onto long lines of ones and zeros is a difficult business, made all the more difficult by a need to ration the binary bits that represent an analog quantity. Small is fast, large is expensive, and as in many realms, there are compromises to be made. The most effective tool for mapping the analog onto the digital is the floating point number.

Computers use *floating point* numerals to represent the values that fall between the whole numbers. For example: 2.3, -3.14159, and 1.02e10. Internally, such numbers are represented as a mantissa and an exponent. The number 20403.23 becomes $2.0403 \times 10^4$, or 0.000012407 is $1.2407 \times 10^{-5}$. Thus the decimal point "floats" to where the significant digits begin. The big advantage to floating point is that it allows a fixed amount of storage (usually four or eight bytes) to represent a wide range of real numbers.

### THE IEEE FLOATING POINT STANDARD

The IEEE standard for representing floating point numbers specifies a 4-byte format, an 8-byte format, and a 10-byte format. These are the formats used by the Intel math coprocessors 8087, 80287, and 80387. They are also the formats used by Turbo C, Turbo Basic, and Turbo Pascal 4.0, as well as Turbo-87 Pascal 3.0. (This article refers to the 8088 and 8087 for simplicity, but the information in it holds true for the 8086, 80186, 80286, 80386, 80287, and the 80387.)

The floating point types in Turbo C are **float** and **double**. Variables of type **float** are stored as IEEE four-byte numbers. These are commonly referred to as "single precision" in FORTRAN and other languages. The type **double** is for double-precision variables, and they are stored as IEEE eight-byte numbers. Certain internal calculations are done in still higher precision, storing temporary values in the IEEE 10-byte format. The precisions of these formats are summarized in Table 1.

| FORMAT | BYTES | BITS | SIGN | MANTISSA | EXPONENT |
|---|---|---|---|---|---|
| Single | 4 | 32 | 1 bit | 23 bits | 8 bits |
| Double | 8 | 64 | 1 bit | 52 bits | 11 bits |
| Extended | 10 | 80 | 1 bit | 64 bits | 15 bits |

| FORMAT | DECIMAL PRECISION | RANGE |
|---|---|---|
| Single | about 7 digits | $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$ |
| Double | about 16 digits | $\pm 1.8 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$ |
| Extended | about 19 digits | $\pm 1 \times 10^{-4932}$ to $\pm 1 \times 10^{4932}$ |

*Table 1. Floating point formats and their precisions.*

### TURBO C AND THE ANSI C STANDARD

Turbo C is a nearly complete implementation of the ANSI C draft standard for the C language. The treatment of floating point in the ANSI C standard is not much different from the way people have always used floating point in C, the main differences being in prototypes, unary pluses, and long doubles.

**Prototypes.** In pre-ANSI C compilers, all floating point numbers were converted to doubles in expressions. Function arguments were always converted to doubles before being pushed onto the stack. Even if you tried to define a function taking an argument of type **float,** as in

```
void foo(x)
float x;
{ ... }
...
foo(3.4);
```

the compiler would treat the argument as a double anyway. The call **foo(3.4)** would push the eight-byte representation of 3.4, and the function **foo** would access it on the stack as an eight-byte double.

In ANSI C and Turbo C, functions are allowed to take four-byte floating point arguments on the stack. The above code is written:

```
void foo(float x)
{ ... }
...
foo(3.4);
```

The prototype for **foo** declares that its argument is of type **float**. In the subsequent call to **foo**, the compiler knows that the argument 3.4 is to be treated as a float constant, and not a double constant. As long as you put the necessary prototypes at the beginning of your file, which in this case is:

```
void foo(float x);
```

Turbo C checks all of the argument types. If they do not match exactly, it either does the necessary conversions or reports an error.

**Unary pluses.** Like prototypes, *unary pluses* are also new with ANSI C. In a statement such as

```
x = a + (b - c);
```

where **x**, **a**, **b**, and **c** are all doubles, the definition of C explicitly allows the C compiler to ignore the parentheses and evaluate in a different order, such as:

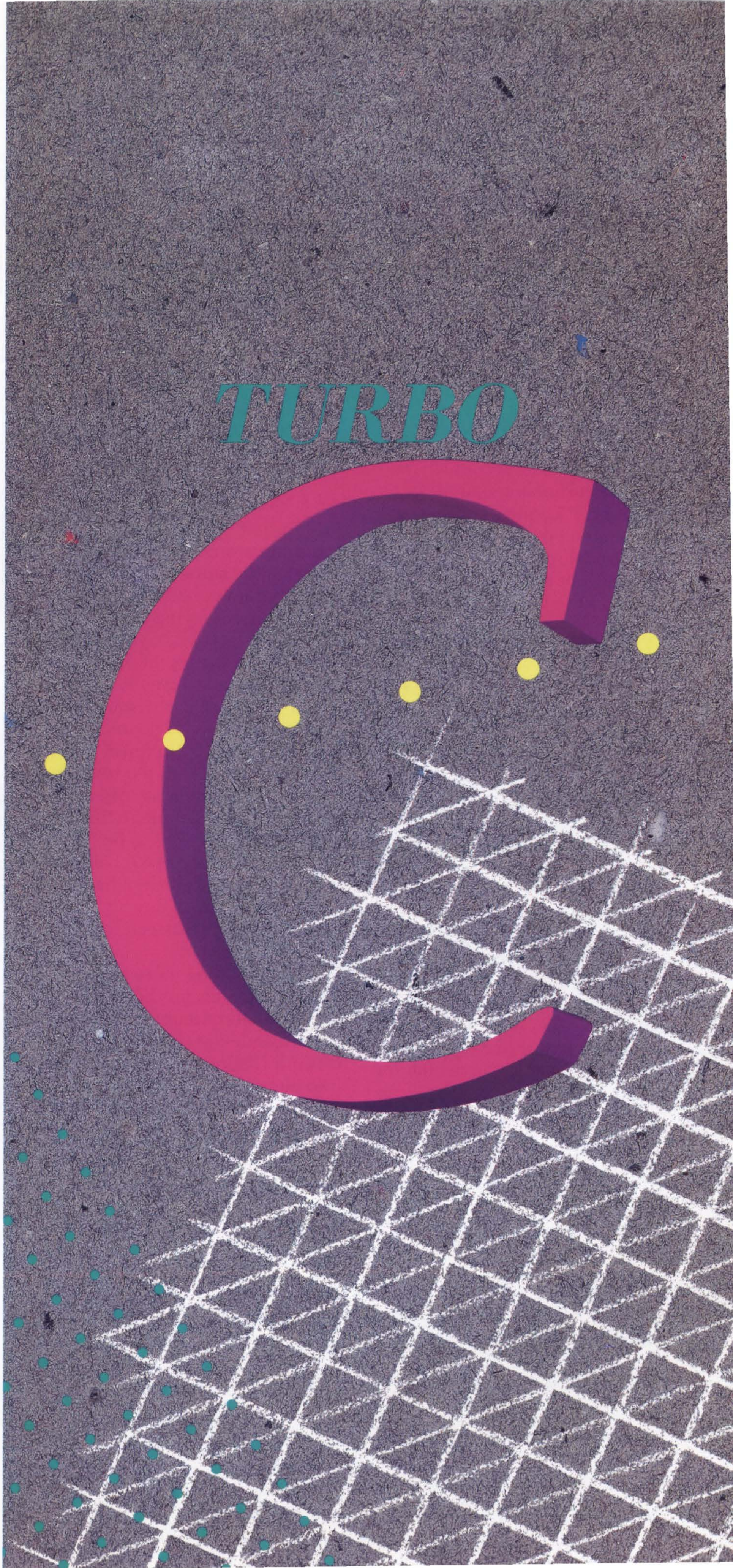
```
x = (a + b) - c;
```

or even,

```
x = (a - c) + b;
```

This flexibility lets the compiler do some optimization. The above three statements would be mathematically equivalent if infinite precision were available, but they are not equivalent when evaluated on the limited precision of computers. In cases where the values being manipulated are very large, precision can be lost trying to express the larger intermediate results, even if the final value is fully expressible in the available precision. If **b** and **c** are nearly equal the programmer might place **b** — **c** in parentheses, on the assumption that it would be evaluated first, thus minimizing the intermediate round-off errors. This would have no effect, since the chosen order of evaluation cannot be guaranteed under UNIX C. With ANSI C, however, the programmer can guarantee it

*continued on page 36*



*continued from page 35*

by using a unary plus before the left parenthesis, as in:

```
x = a + +(b - c);
```

**Long doubles.** The third ANSI C innovation is the **long double** type. The exact size or format of the floating point types are not specified in the standard, but **long double** is a floating point type at least as big as **double**. The idea is to implement floats as IEEE 4-byte single precision, doubles as IEEE 8-byte double precision, and long doubles as IEEE 10-byte extended reals. Thus long doubles are used for intermediate results where high accuracy is required. In Turbo C 1.5, long doubles are implemented as IEEE 8-byte double precision, the same as type **double**, but future versions are likely to use the 10-byte format.

## ASSEMBLY LANGUAGE INTERFACE

Real-valued functions in Turbo C have a different calling convention from Microsoft C, Lattice C, and all of the other C compilers. Since this is the biggest incompatibility between Turbo C and the other C compilers on an object code level, it is worth explaining what the differences are and why they exist.

In Lattice C, doubles are returned in the registers AX, BX, CX, and DX. Microsoft C and Turbo C functions may be declared as **cdecl** or **pascal** depending on whether you want C-like or Pascal-like calling conventions. **cdecl** is the default for both compilers.

In Microsoft C, a double **cdecl** function returns a pointer to the double in the DX:AX registers, or the DS:AX registers, depending on the memory model. A double **pascal** function also has an extra pointer on the stack when called, and the function must remove it on exit. It is unclear why the extra pointer is there; the other **pascal** functions don't have it and there is nothing in the Microsoft documentation that discusses it.

In contrast, Turbo C returns doubles on the top of the 8087 stack. The 8087 stack is assumed to be clean when the call takes place, and it is the responsibility of the calling function to remove the real value from the 8087 and put it wherever it belongs. **pascal** and **cdecl** functions return doubles the same way. The Turbo C convention is much more efficient than that of Microsoft C or Lattice C, because the 8087 is the natural place to keep floating point numbers. Consider this fairly typical code:

```
double square(double x)
{  return  x*x;}

....

y = square(x) / 3.;
```

In Turbo C, **y** is computed by pushing **x** onto the 8088 stack, and calling **square**. Then **square** loads **x** into the 8087, squares it, and returns, leaving the result on the 8087 stack. Next, **y** is obtained by dividing the top of the 8087 stack by 3 and storing it.

In Microsoft C, the **square** function would have to do an **FWAIT** to allow the 8087 to finish the multiply, then store the result in a static area of memory. After returning, the double in that static area of memory has to be reloaded into the 8087. Lattice C is even more inefficient because there is no way to directly transfer a real number between the 8088 and 8087 registers.

Besides being more efficient, returning reals on the 8087 stack gives greater accuracy. All of the transcendental functions in the Turbo C library are computed to 80 bits of precision. An expression such as the conditional

```
if (sin(x)*cos(x)-sqrt(x) > 0.) ...
```

is calculated to the full 80 bits of precision. Future versions of Turbo C will have 10-byte IEEE long doubles, so the full precision may be saved in named variables as well.

If no 8087 is present, it is emulated, and double-precision quantities are returned on the emulator stack. If you only write programs in C, you don't need to know how it works. However, if you want to write Turbo C programs that call assembly language routines to do floating point arithmetic, and if you want your programs to work automatically whether there is an 8087 installed in the machine or not, then you need to understand how the emulator works.

## THE 8087 EMULATOR

The *emulator* optimizes floating point programs by imitating the 8087. Nearly everyone who has a PC and cares about floating point speed and accuracy buys an 8087. Floating point programs should therefore have inline instructions to make efficient use of the 8087. However, you also want your programs to be usable by people who *don't* have one. Therefore, Turbo C generates code in such a way that the 8087 can be emulated if it is not there. At runtime, a library routine tests for the presence of the 8087. If one is found, inline code is used; if no 8087 is found, the 8087 instructions are emulated. There is some overhead in using an emulator, but it is a penalty paid only by those who don't have an 8087.

So how can we have inline floating point instructions and still have programs that work without an 8087? If things had been planned properly, the 8088 would always be aware of the presence of an attached coprocessor. If it tried to execute an 8087 instruction when the chip is not there, then it would trigger a restartable exception. Then the operating system could have been written so that at boot time it would determine if the 8087 is present, and if not, load an appropriate emulator. The emulator would then be a software exception handler that is capable of mimicking all of the 8087 instructions. When a program encounters a floating point instruction, then it would be executed inline if the 8087 is present, otherwise it would jump to the exception handler, mimic the instruction, and return to the program to execute the next instruction.

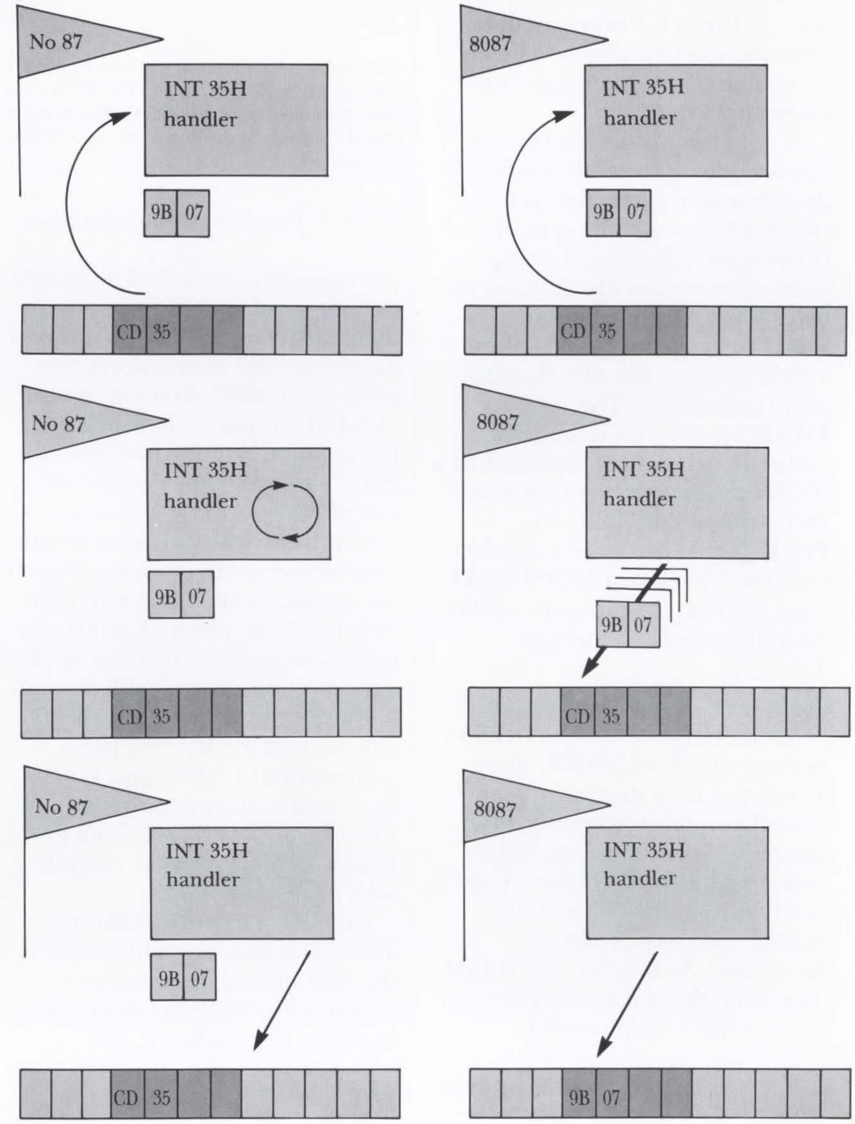
Unfortunately, the 8088 is not smart enough to behave reasonably if the 8087 is absent, and there is no floating point exception handler software in DOS. The 80286 was designed with the capability to jump to an interrupt handler if there was no attached 80287, but by the time the 80286 hit the market, the IBM PC and DOS had become the standard that software was obliged to support.

Runtime detection and emulation of the math coprocessor orginated with Microsoft, who wanted its languages to work efficiently both with and without the math coprocessor. The scheme they developed built on Intel's earlier system, which involved replacing the first two bytes of a floating point instruction with a software interrupt instruction. The interrupt pointed to a handler that actually emulated 8087 instructions.

**Interrupt handling.** In overview, the Microsoft emulation works like this: At compile/link time, floating point instructions are generated with software interrupt opcodes in place of the first two inline opcode bytes. At runtime, a suite of emulation interrupt handlers is installed. Then, a library routine runs before the application code itself takes control, and tests for the presence or absence of the 8087, posting a flag somewhere with the results. The interrupt handler checks the flag, and if an 8087 is present it patches the two software interrupt opcode bytes *back* to the equivalent 8087 inline floating point instructions. Then it returns control to the beginning of the floating point instruction it just patched, executing the instruction inline. Alternately, if the 8087 is *not* present, the interrupt handler mimics the 8087.

The result is fairly efficient if the 8087 is present, as the first time each generated floating point instruction is executed there is the extra overhead of going through the interrupt handler, but each subsequent execution of the same instruction will execute the inline 8087 code and *not* jump to the interrupt handler. This dual sequence of events is illustrated in Figure 1.

Now, in detail: Microsoft allocates interrupts 34H through 3EH for use by floating point emulation. DOS does not use them. During code generation, Turbo C can follow two paths, depending on whether the programmer chooses to compile for inline 8087 code (thus requiring a math coprocessor at runtime) or for runtime emulation. If inline code is the desired output, (using the **-f87** switch for the Turbo C command-line compiler, or the **O**ptions/**C**ompiler/**C**ode **G**eneration/**F**loating point toggle set to 8087/80287 in the Turbo C programming environment) ordinary 8087 opcodes (beginning with 9BH) are generated for floating point instructions.



*If no 8087 is detected in the system, a software interrupt transfers control to the interrupt handler, which emulates the floating point operation. Control returns to the instruction* following *the software interrupt instruction.*

*If an 8087* is *detected in the system, control is passed to the software interrupt the* first *time the code is executed. The software interrupt handler patches the calling code, replacing the software interrupt call with the equivalent native 8087 instruction. Any future execution of that code is done directly by the coprocessor.*

*Figure 1. Floating point coprocessor emulation.*

*continued on page 38*

## FLOATING POINT

*continued from page 37*

For inline 8087 code, this is all that happens. However, if emulation is selected, the compiler adds special fixup records to the .OBJ file so that the linker will convert the floating point instruction opcodes to software interrupt opcodes at link time. The fixups are associated with public symbols. A Turbo C library module contains the definitions. The correct values for these fixups are given in Table 2.

If a floating point instruction has a segment override prefix, then sometimes two fixups are needed for one floating point instruction. All of the floating point instructions are at least two bytes long, except when an **FWAIT** instruction stands alone and is not the start of a floating point instruction. The lone **FWAIT** instruction is actually emitted by the code generator as a **NOP FWAIT** and fixed up to an **INT 3DH** at link time with **FIWRQQ**. At runtime, it is either converted back to a **NOP FWAIT** if an 8087 is present, or to a **NOP NOP** if there isn't one (see Table 3).

| Floating point mnemonics | | Opcodes emitted | |
|---|---|---|---|
| | | Without emulation | With emulation |
| FLD | dword ptr [bx] | 9BD907 | CD3507 |
| FLD | dword ptr [bx+100H] | 9BD9870001 | CD35870001 |
| FLD | qword ptr [bx] | 9BDD07 | CD3907 |
| FLD | qword ptr [bx+100H] | 9BDD870001 | CD39870001 |
| FLD | qword ptr [si] | 9BDD04 | CD3904 |
| FLD | qword ptr [si+100H] | 9BDD840001 | CD39840001 |
| FLD | qword ptr es:[si] | 9B26DD04 | CD3CDD04 |
| FLD | qword ptr es:[si+100H] | 9B26DD840001 | CD3CDD840001 |
| FADDP | | 9BDEC1 | CD3AC1 |
| FMULP | | 9BDEC9 | CD3AC9 |
| FSTP | ST(0) | 9BDDD8 | CD39D8 |
| FINIT | | 9BDBE3 | CD37E3 |
| FWAIT | | 909B | CD3D |

*To emulate a floating point instruction, the first two bytes of each instruction must be replaced by a software interrupt. Standalone* **FWAIT** *is a special case. The 9BH byte is the* **FWAIT** *8087 instruction. Because the code for* **INT CDH** *is two bytes long, the code generator must precede standalone* **FWAIT** *instructions with a* **NOP** *(code 90H) to allow room to replace the* **FWAIT** *with a two-byte software interrupt call.*

*Table 3. Emulation code generation.*

**MASM and the 8087.** For pure C programming there is no need to bother with these details, since Turbo C does it all for you, but what if you have assembly language files or inline assembler code in your Turbo C files? Currently you are dependent on MASM, Microsoft's Macro Assembler. As it turns out, MASM does everything you need with an inadequately documented command-line switch. The MASM default is *not* to recognize any 8087 instructions. If you want it to assemble 8087 instructions, you must put a **.8087** directive in your .ASM file or put a /R switch on the command-line when assembling your file. (For the one or two 80287-specific instructions, you must use the **.287** directive in your source code, because there is no corresponding command-line switch.) If you put a /E switch on the command line instead, MASM generates the fixups, with the public names shown in Table 2, that will convert the floating point instructions to interrupts. If you have inline assembler in your .C files, Turbo C will EXEC out to MASM from within the compiler, with the /E switch set.

```
            public    FIARQQ, FIDRQQ, FICRQQ, FIERQQ, FISRQQ,
FIWRQQ
FIARQQ            equ  0FE32h          ; fwait / ds:
FICRQQ            equ  00E32h          ; fwait / cs:
FIDRQQ            equ  05C32h          ; fwait / esc
FIERQQ            equ  01632h          ; fwait / es:
FISRQQ            equ  00632h          ; fwait / ss:
FIWRQQ            equ  0A23Dh          ; nop   / fwait

            public    FJARQQ, FJCRQQ, FJSRQQ
FJARQQ            equ  04000h          ; esc nn --> ds:nn
FJCRQQ            equ  0C000h          ; esc nn --> cs:nn
FJSRQQ            equ  08000h          ; esc nn --> ss:nn
```

*Table 2. Emulator from the Turbo C Runtime Library Source.*

An alternative to MASM for Turbo C inline assembler code is the A86 shareware assembler, which is available for downloading from CompuServe. A86 supports emulated floating point instructions, and is many times faster than MASM.

If you have an 8087 and you do not want the linker to change your floating point code to interrupts, you can create an assembler file just like Table 2, but with the public symbols equated to zero.

**At runtime.** Each Turbo C-generated .EXE program tests for the presence of the 8087 at startup, and installs interrupt handlers for interrupts 34H through 3EH. The **exit()** function restores these interrupts to their previous vectors before returning to DOS. If you exit a program without going through the **exit()** function, such as with the Break key, or by aborting during a critical error, then the interrupts will not be restored. It doesn't matter in most everyday situations, because DOS does not use these interrupts. The scenario for trouble would be this: A Turbo C program does a spawn to another program that has floating point code and no critical error or control-break handler; the user breaks out of the spawned program back to the parent program, and then the parent program attempts to execute floating point instructions. The results would be unpredictable (but almost certainly fatal to the parent program) since the spawned program's interrupt handlers would still be in place. You can avoid this situation by calling **_fpreset** after the spawn but before executing any floating point instructions.

When first loaded into memory, the floating point instructions in the application all begin with software interrupts in the range 34H through 3EH. The *first* time any instruction is executed, the software interrupt passes control to the interrupt handler. The interrupt handler checks a flag that indicates whether or not the initial test discovered an 8087 in the system. If no 8087 is installed, the interrupt handler then emulates the floating point instruction, and returns control to the first instruction *after* the modified floating point instruction. However, if an 8087 does exist in the system, the interrupt handler "unpatches" the software interrupt instruction that invoked it *back* to the original 8087 inline floating point instruction. Then, it returns control to the just-patched floating point instruction, which executes inline.

This process is repeated for the first execution of any floating point instruction in the application. However, once a floating point instruction is patched, it executes inline for all subsequent executions, providing the full speed of 8087 code execution. The overhead of invoking the software interrupt handler happens only once per instruction, so it is relatively slight. The only disadvantage is that code generated in this way cannot be written into ROM or EPROM.

## LIMITATIONS OF THE EMULATOR

The floating point emulator does not support everything the coprocessor does, and there are certain limitations on its use, explained below.

**Denormals.** These are special numbers that are smaller than the smallest normal number but still larger than zero. Denormals are expressed with a special bit encoding and must be handled differently than normal numbers. The smallest normal double that is greater than zero is about $1.8 \times 10^{-308}$, but there are denormal doubles as small as about $1.0 \times 10^{-324}$. These denormals are like fixed-point numbers with the decimal point at approximately $1.0 \times 10^{-308}$, so the smaller they are, the fewer bits of precision they have.

The advantage of having denormals is demonstrated by the program in Listing 1. Both **x** and **y** are normal and have full precision in the double-precision format, but they are very small and very close to each other, and their difference is about $1.0 \times 10^{-315}$, which is smaller than can be normally represented. If this situation occurred on an older computer that did not use the IEEE floating point format, you would have the unpleasant situation in which the difference would underflow to zero even though the numbers are not equal. As a result, code like

```
if (x != y) z = x / (x - y);
```

could fail to produce the expected results. Using denormals, the IEEE standard requires that two numbers be equal if and only if their difference is actually zero.

The emulator supports denormals for intermediate calculations, but not for end results, because denormals are converted to zeros. Thus the above code would work properly, but in the program given in Listing 2, **z** should be a denormal. If run on a machine with an 8087, **z** will be a denormal, but on a machine without an 8087, **z** will underflow to zero.

**Floating point registers.** The 8087 has eight floating point registers that are intended to be used as a stack, and that is how Turbo C uses them. Normally, numbers are pushed onto the top of the stack, operations take place at the top of the stack, and numbers are popped off. For example, the function in Listing 3 computes an expression and leaves the result on the top of the stack.

The 8087 also has instructions that are not stack-oriented for moving registers around. An example is

```
FXCH   ST(3), ST
```

which exchanges the contents of register 3 with the top of the stack

*continued on page 40*

# GRAPHICALLY YOURS, TURBO C 1.5

The first major update of Turbo C contains a number of important enhancements. The most visible of these is Turbo C 1.5's completely new video support. This support closely parallels the video support provided with Turbo Pascal 4.0, and includes an important new subsystem: The Borland Graphics Interface (BGI), a device-independent graphics library with loadable drivers for most major graphics display devices.

Supported display devices include the CGA, EGA, VGA, MCGA, Hercules, ATT 400-line Graphics Adapter, and the 3270 PC Graphics Adapter. The BGI is viewport-based, and all drawing coordinates are viewport-relative. Most BGI parameters (size of viewport, last point addressed, etc.) can be queried as to their current state. Drawing functions include line, rectangle, arc, circle, ellipse, polygon, bar (filled rectangle), pie-slice, and filled polygon. An 8 by 8 bit-mapped text font is provided, as well as several scaleable stroke fonts.

Turbo C 1.5 provides enhanced text video support as well, with text windows, text block transfers between screen and memory buffers, text mode state query, and complete attribute control.

Also new to 1.5 is Streams, a highly portable means of handling text and binary files that is similar to file handling in Turbo Pascal. A grep utility, and TLIB, an object librarian, are among the new utilities providing support for the compiler and linker.

A 175-page bound addendum to the *Turbo C Reference Guide* summarizes these and other changes, all intended to make Turbo C the most powerful system development tool you can buy. ■

*continued from page 39*

(register 0). It does the expected thing whether or not register 3 is empty. The emulator, however, implements the registers as a true stack, and empty registers do not exist. Exchanges only work properly on nonempty registers. Register wraparound is also not supported in the emulator.

**Precision exception.** The 8087 sets an exception flag if the result of an operation is imprecise, which includes the majority of floating point calculations. For example, 2.0 × 3.0 is calculated on the 8087 with complete accuracy, but computing 0.2 × 0.3 gives a round-off error. Since 0.2 and 0.3 cannot be represented with complete accuracy in a binary format anyway, everyone expects that the last bit may have required rounding. The emulator only sets the flag bit when a floating point number in the emulator is being stored as an integer and rounding is necessary.

The TC library function **_status87()** returns the status word of the 8087 chip, or the status word of the emulator if no 8087 is present. Bit 5 of the status word is the precision exception. The following code will print a message if the 8087 is present, but not otherwise:

```
double x;
x = 0.2;
x *= 0.3;
if (_status87() & 0x20)
  puts("Loss of precision.");
```

**Infinities.** Turbo C only supports projective infinity. This is of no importance to the vast majority of programmers, but the 8087 can be put into a special mode called *affine infinity*, which distinguishes between positive infinity and negative infinity. Future releases of Turbo C are likely to support affine infinity, since projective infinity has been dropped from the IEEE standard.

**FWAIT.** An **FWAIT** instruction must precede each floating point instruction, since an instruction without the **FWAIT**, such as **FNINIT**, cannot be emulated. The **FWAIT** provides room in the floating point instruction to patch in the emulator's software interrupt invocation, as described earlier. The Microsoft assembler puts these in automatically. They are not really necessary with the 80287, but if you want your programs to work interchangeably on the 8087, using them is a good idea.

### ASSEMBLY LANGUAGE EXAMPLE

Most people think that only a masochist would write floating point code in assembly language. For the most part I agree, but there are cases when it is worthwhile. Many floating point programs spend most of their time in a few tight loops, so it might make sense to write these tight loops in assembly language. However, a tight floating point loop could still be spending most of its time in a library routine. In that case you can't speed it up very much without rewriting the library routine. This option will soon be available to the programmer, since the Turbo C Runtime Library source is to be released as a separate product.

With Turbo C, you can improve both speed and accuracy by using the floating point registers. Gains will be achieved whether the 8087 is present or not. Consider a function to take the dot product of two real vectors, each of length *n*. The dot product of **x[]** and **y[]** is defined by:

```
x[0] * y[0] + ... + x[n-1] * y[n-1]
```

(Remember that all arrays start at 0 in C, so a length *n* array has an index running from 0 to *n*-1, not 1 to *n*.) A typical C function to calculate it is given in Listing 4. Listing 5 shows a slightly smaller way to code it that is no more efficient and perhaps less readable.

On an 8 MHz IBM AT with an 80287, either of these functions takes about 11 milliseconds to take the dot products of vectors of length 100. This can be coded with inline assembler in Turbo C (for the tiny- , small- , and medium-memory models) as shown in Listing 6.

This last version (Listing 6) is more efficient because the variable **sum** is kept on the floating point stack instead of in memory. It takes about two-thirds the time of the earlier versions (Listings 4 and 5) if a coprocessor is present, or about three-quarters the time if

```
#include <float.h>

unsigned int fpstatus;
main(){
        /* reset the 8087, leaving all exceptions masked */
        _control87(MCW_EM,MCW_EM);
        /* do floating point work here */
                ...
        /* now, check for masked exceptions before proceeding */
        fpstatus = _status87();
        if (fpstatus & SW_INVALID)
                puts("Floating point error: invalid operation.");
        if (fpstatus & SW_ZERODIVIDE)
                puts("Floating point error: zero divide.");
        if (fpstatus & SW_OVERFLOW)
                puts("Floating point error: overflow.");

        /* reset 8087 before doing more work */
        _clear87();

        /* do more floating point work here */
                ...
}
```

*Figure 2. A simple floating point exception handler.*

a coprocessor is not present. It is also more accurate, because the floating point registers have 10-byte precision and doubles only have eight. As a result, there is less reason to be concerned that round-off errors will accumulate to a significant level.

If some future version of Turbo C implements long doubles as 10-byte reals, then the extra precision could be obtained in C (without resorting to assembler) by merely declaring the variable **sum** to be a long double.

## FLOATING POINT EXCEPTIONS

The C language does not specify what happens with a floating point exception (such as dividing by zero); it is up to the compiler implementer. The possible floating point exceptions that are caught by the 8087 are:

- Invalid operation
- Denormal operand
- Zero divide
- Overflow
- Underflow
- Inexact result

Of these, Turbo C traps only the invalid operation, zero divide, and overflow exceptions. (An early release also trapped the denormal operand and underflow exceptions.) If a program ever divides by zero, or if a computation produces a number too large to be represented in the eight bytes allotted an IEEE real, an error message appears on the screen and the program aborts.

Such an exception need not be fatal, though. The IEEE standard provides for arithmetic with Infinity and Not-a-Number (NaN), so you can just let the 8087 divide by zero and produce an infinity if you wish. Figure 2 shows a program that does this. Note that the code in Figure 2 is not complete and is given as an example only.

The other exceptions are not worth checking for in most cases. If a computation underflows, i.e., if the evaluated quantity gets too close to zero to be distinguished from zero within the eight bytes of

*continued on page 42*

LISTING 1: DENORMAL.C

```
/* Listing 1: */

double x,y;
main(){
x = 1.e-300;
y = x *(1. + 1.e-15);
if (x == y) puts("x == y"); else puts("x != y");
if (x - y == 0.) puts("x - y = 0"); else puts("x - y != 0");
}
```

LISTING 2: DENORMAL2.C

```
/* Listing 1: */

double x,y;
main(){
x = 1.e-300;
y = x *(1. + 1.e-15);
if (x == y) puts("x == y"); else puts("x != y");
if (x - y == 0.) puts("x - y = 0"); else puts("x - y != 0");
}
```

LISTING 3: SUMPROD.C

```
/* Listing 3: */

double sum_of_prod(double a, double b, double c, double d)
/* returns  a*b + c*d  */
{
asm   fld    qword ptr a
asm   fmul   qword ptr b
asm   fld    qword ptr c
asm   fmul   qword ptr d
asm   fadd
}
```

LISTING 4: DOT.C

```
/* Listing 4: */

double dot(int n, double *x, double *y)
{
     int i;
     double sum = 0.;
     for (i = 0; i < n; ++i)
          sum = sum + x[i] * y[i];
     return sum;
}
```

### LISTING 5: DOT2.C

```
/* Listing 5: */

double dot(int n, double *x, double *y)
{
     double sum = 0.;
     while (n--)
          sum += *x++ * *y++;
     return sum;
}
```

### LISTING 6: DOT3.C

```
/* Listing 6: */

double dot(int n, double *x, double *y)
{
asm  fldz
     _SI = x;
     _DI = y;
     while (n--)
     {
asm  fld  qword ptr [si]
asm  add  si,8
asm  fmul qword ptr [di]
asm  add  di,8
asm  fadd
     }
}
```

### LISTING 7: MATHERR.C

```
/* Listing 7: */

#include <math.h>
int cdecl matherr(struct exception *e)
{ return 1;}
```

## FLOATING POINT

*continued from page 41*

an IEEE real, the 8087 just converts it to a zero as the programmer presumably wants. The inexact result exception occurs on almost every floating point operation, because there is almost always some round-off error. Denormals should not be a concern in most programs either; they have less precision than normals but they are so close to zero it hardly matters.

Those Turbo C Runtime Library math functions having arguments out of range or other such errors call the **__matherr()** function to handle them. **__matherr()** in turn, calls **matherr()**, which returns a zero in its default incarnation, causing **__matherr** to print a simple message and abort to DOS. To perform custom error handling, you simply rewrite **matherr()** to handle the error and return a nonzero value to **__matherr**. On a nonzero value from **matherr()**, **__matherr()** continues execution without aborting. If you do not want an error message on your screen just because you tried to take the square root of a negative number, add the function in Listing 7 to one of your files. More sophisticated error handling can be obtained by examining the struct whose address is passed to **matherr()**.

### CONCLUSION

Floating point numbers are most often used as a method of describing a thoroughly analog world in digital fashion. A Turbo C programmer who is aware of the underlying theory of floating point numerics will make the best use of the considerable powers of the 86-family of CPUs and their math coprocessors. ■

*Roger Schlafly is in charge of scientific and engineering products at Borland. He is the author of Eureka: The Solver and worked on floating point support for Turbo C.*

*Listings may be downloaded from CompuServe as CFLOAT.ARC.*

# THINKING IN TURBO C

## Until you learn to think in a language, you cannot hope to become a virtuoso.

*Bruce F. Webster*

SQUARE ONE

It's been argued that Pascal and C are fairly similar. This contention is usually supported by noting that both languages support loops, **IF/THEN** statements, subroutines (with parameters), pointers, records, and user-defined data types. Some of the recent versions of BASIC are lumped in there, too, since they often provide similar features. Therefore, it should be very easy for someone proficient in one of these languages to pick up another.

Nonsense. All three languages come from three different directions, and whatever the superficial similarities may be, the underlying philosophical differences remain profound.

The goal of this article is to help you to understand the mindset behind the C programming language. If you're a Pascal (or BASIC) programmer, and you approach C as being merely Pascal (or BASIC) with different syntax, you're in for a lot of frustration. If, on the other hand, you learn to think in C, you'll be pleasantly surprised at just how effective a C programmer you can be.

### FREEDOM OVER SECURITY

Programming in BASIC is like carving soap with a plastic knife: there's little chance of hurting yourself, but it takes a lot of work to get the job done. Programming in Pascal is like using a table knife: the work is easier and faster, and the chances of slipping and cutting yourself aren't that much greater. Programming in C is like using a double-edged razor blade: the work is quick and intricate, and there's a much greater chance of bloody fingers.

C was developed by Dennis Ritchie at Bell Laboratories, in conjunction with his efforts to create the UNIX operating system. C itself descended from a chain of languages, including B, BCPL, and CPL. The goal through this development chain was to create a language that offered the structure and portability of high-level languages (such as FORTRAN and ALGOL), yet allowed the user the low-level system access and freedom of assembly language.

The result—as documented in *The C Programming Language* by Ritchie and Brian Kernighan—was a language that, while appearing to be high-level, gave you the freedom to hang yourself. Type and parameter checking were nonexistent, implicit and explicit type casting were allowed, heavy use of pointers was almost essential, and any memory location was at your disposal. On top of all that, the C compilers did only minimal syntax checking; if you wanted to do a thorough check of your program, you ran the source code through a separate style- and syntax-checking program called **lint**.

This was precisely the freedom needed for systems programming, particularly for writing an operating system. And that freedom is still the source of much of the praise and most criticism of C. For novice (and not-so-novice) C programmers, this freedom can be deadly and frustrating, leading to bugs that are difficult to track down. For skilled C programmers, it can be the life-saving difference, allowing them to get in and do exactly what they need to do without the language (and any arbitrary restrictions thereof) getting in the way.

The C standard evolved as an attempt to increase security while not limiting freedom. One example is the function prototype, similar to a **FORWARD** declaration in Pascal, which allows the compiler to check the number of parameters and their types. And some C compilers are doing considerably more error checking during compilation, eliminating the need for a separate **lint** program. Turbo C, for exam-

*continued on page 44*

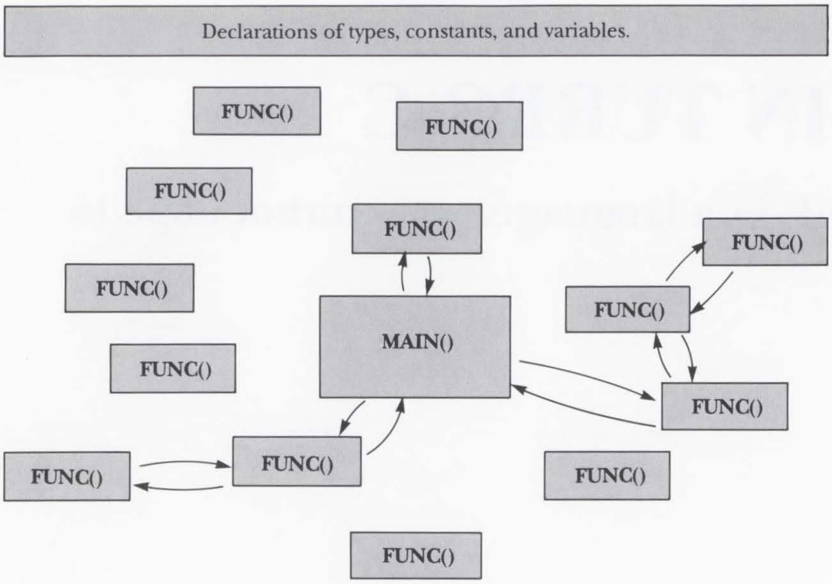


*Figure 1.* **main()** *and its functions.*

## THINKING IN TURBO C

*continued from page 43*

ple, does extensive checking, not only for errors but also for potential errors—statements that are syntactically correct but which may have different results than those intended.

### CONCISENESS OVER CLARITY

C programs lend themselves to conciseness. This is due to the rich, extensive set of operators, and the loose definition of what constitutes a statement. Briefly put, C treats any expression followed by a semicolon as a statement. The expression doesn't have to *do* anything, or it may perform several tasks. Furthermore, you can insert these expressions in unexpected places, such as in the three sections of a **for** statement.

Because of this flexibility, C programmers can (and often do) cram considerable work onto a single line. Here's a simple example. Suppose you have three variables, **a**, **b**, and **c**, and you want to assign to **c** the maximum of **a** and **b**. In Pascal, you'd probably write this:

```
if a > b
  then c := a
  else c := b;
```

In C, you could write a very similar construct:

```
if (a > b)
  c = a;
else
  c = b;
```

But there's a more concise way to write this in C:

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

The construct

```
<exp1> ? <exp2> : <exp3>
```

is known as the *conditional operator.* The operator evaluates <exp1>; if the result is "true" (nonzero), then it evaluates <exp2> and assumes that value; otherwise, it evaluates <exp3> and assumes that value.

As another example, consider the following piece of code:

```
indx = 0;
while ((line[indx++] =
  toupper(getc(infile))) != EOF);
```

The **while** statement reads in a series of characters from a previously opened file, converting the alphabetic characters to uppercase, and storing all characters (whether converted or not) into **line**, presumably an array of type **char**. This continues until the end-of-file is reached. Note that the **while** *expression* does all the work; there's no actual statement being executed in the **while** loop, as would be required to accomplish anything in Pascal.

These are simple examples, but others can be found in any good (and many bad) C programs. The resulting compactness has a major advantage: it allows you to see more of your program at a time . on your screen. It also has a major disadvantage: it can make your program very hard to read.

### FLEXIBILITY OVER FORM

Standard Pascal is a form-driven language. A program follows a fixed format: program header; sections for labels, constants, types, and variables; procedure and function declarations, and finally, the main program body. Procedures and functions replicate the main program format and are often called subprograms for that very reason. All identifiers (labels, constants, types, variables, procedures, and functions) must be declared before they are used. Turbo Pascal relaxes some of those restrictions, allowing the declaration sections (everything between the program headers and the main body) to be in any order and to occur multiple times. In either case, execution starts at the first statement of the main body and continues until the last statement of the main body (or until a **Halt** or **Exit** statement is executed).

Standard BASIC (whatever that is) has very little form: it's just a collection of numbered statements. Execution starts with any statement and freely flows to any other statement. Variables can be "declared" just by using them in a statement. Some newer versions of BASIC (such as Turbo Basic) eliminate line numbers, and add form with control structures, true subroutines, and alphabetic labels.

C is different from both Pascal and BASIC. A C program is a collection of declarations and functions. Constants, variables, and data types must be declared before they are used, but there is no inherent ordering beyond that. Functions may be called by other functions before being declared, though that prevents parameter checking. The main body of the program is just another function; it can appear anywhere in the program and is distinguished by having the special name **main()**. Think of functions as islands floating in a sea of declarations: no one island is in control of any other, but you need a starting point, which is **main()**. See Figure 1.

There is also form within functions. A function consists of a function header (including parameter declarations) and a body. The header declares the function, giving its data type, the function name, and the names and types of its parameters. The body is just a compound statement: a left (opening) brace, zero or more local declarations, zero or more statements, and a right (closing) brace. Pascal users will note that the left and right braces correspond to **BEGIN** and **END** in Pascal.

Consider the sample program in Listing 1. Execution of this program begins with the function **main()** being called by the operating system. **main()**, in turn, calls three other functions: **lsort**, **dumplist**, and **lmax**. The function **lsort**, in turn, calls yet another function, **swap**. Each time a function ends, control returns to the function that called it. The program ends when **main()** is done, that is, when its last call to **printf()** returns.

### ADAPTABILITY OVER CONSISTENCY

After you write a program in Turbo Pascal or Turbo Basic, you compile it and run it. Occasionally, directives may include additional files, or enable and disable certain compiler options, but the source code that the compiler uses remains pretty much as you wrote it.

In contrast, the C language is closely tied to the concept of a *preprocessor.* The purpose of the preprocessor is to create a copy of your source code for the compiler to use; the text is modified according to the preprocessor commands it contains. The **#include <file>** command should be familiar (and self-explanatory), but the other commands provide capabilities that you might not expect, given a background in other languages.

The fundamental idea of preprocessor commands is *macro substitution.* A *macro* is a piece of text that you can define and use throughout your program. When the preprocessor massages your text immediately prior to compilation, it substitutes the macro definition for the macro name.

Suppose you had the following preprocessor command near the start of your file:

```
#define    NULL     0
```

This defines the macro **NULL** to be equivalent to the text string **0**. Later in your program, you might use **NULL** like this:

```
if (result == NULL) { ... }
```

When the preprocessor massages your code it makes a literal substitution, so that the compiler sees the following:

```
if (result == 0) { ... }
```

If that were all that macros did, it would be nice but not terribly exciting. Macros, however, do much more. Consider the following:

```
#define
max(i,j) ((i) > (j) ? (i) : (j))
```

If you used this macro in your program, it might look like this:

```
c = max(a+2,b);
```

which the preprocessor would convert to

```
c = (a+2) > (b) ? (a+2) : (b);
```

assigning to **c** the maximum of **a2** and **b**. If you look at the file STDIO.H on your Turbo C distribution disk, you'll find a number of such macros already defined for you. Keep in mind that while **max(i,j)** looks like a function, it is *not* a function but a single C statement incorporating the conditional operator explained earlier.

### THINGS TO WATCH FOR

There are a number of common pitfalls for novice C programmers. Many of them are listed in the *Turbo C User's Guide*, but they bear repeating here.

Beware of confusing the expressions **a = b** and **a == b**. The first expression assigns the value of **b** to **a**, then **a** assumes the value of **b**; the second compares **a** and **b** and yields a 0 (**false**) if **a** and **b** are not equal or a 1 (**true**) if **a** and **b** are equal. So, the statement

```
if (a b)..
```

is valid, but doesn't do what you might think.

Identifiers in C are case significant, so that the variable names **indx**, **Indx**, and **INDX** are all separate and distinct identifiers.

All simple **type** parameters (**int**, **float**, **char**, etc.) are *pass-by-value.* If you want to do a *pass-by-address*, (similar to a **VAR** parameter in Pascal) then you must do it literally: pass the address, using the address-of operator "**&**" on the actual parameters. You must then define the formal parameters as pointers. For example, look at the **swap()** function in Listing 1, then look at the call to **swap()** inside of **lsort()**.

Speaking of which...when you read values into variables using **scanf()**, be sure to use the address-of operator "**&**" as needed.

It's very easy to get confused

*continued on page 46*

## THINKING IN TURBO C

*continued from page 45*

with pointers, especially when you drag in arrays and strings, or when you pass parameters through several levels of function calls. When in doubt, make diagrams.

There are two ways to define string: as a pointer to **char** or as a **char** array. The first method does *not* set aside any memory for the string, but it does allow direct assignment of string literals. The second does set aside memory but does not allow direct assignment; you have to use the built-in function **strcpy() instead.**

Make sure you understand the use of the "++" and "−−" operators before using them. Beware of using them with a variable that appears two or more times in an expression. Also, if you use one with a pointer variable, be sure you know whether it's incrementing the pointer address or the value to which the pointer points.

### DIVING IN

This is all well and good, but how do you take that first step, dipping your toes into C? Coming from BASIC or Pascal, you may find C intimidating. For starters, write a few simple programs. Don't use any global declarations (i.e., those written outside of any function); just set up your main function

```
main()
{
   ...some C declarations...
   ...some C statements...
}
```

and see what you can do. Declare variables of different data types. Practice reading data in from the keyboard and writing it back out to the screen. Try out each of the statement types (assignment, **for**, **while**, **do..while**, **if**, **switch**, etc.). Become familiar with the compiler warnings and error messages that you will invariably generate the first few times.

Next, write a few functions

LISTING 1: SAMPLE.C

```
/*---------------------------------------------------------*/
/*       Demo sort program for THINKING IN TURBO C         */
/*                                                         */
/*                      by Bruce Webster                   */
/*                                                         */
/*                                  Turbo C 1.5            */
/*                                  Last modified 11/20/87 */
/*---------------------------------------------------------*/

#define LISTSIZE  100                  /* define constant */
typedef int       numlist[LISTSIZE];   /* define data type */


int     lmax(numlist list, int count); /* declare functions */
void    swap(int *i, int *j);
void    lsort(numlist list, int count);
void    dumplist(numlist list, int count);

/*---------------------------------------------------------*/
/*                          main()                         */
/*---------------------------------------------------------*/

main()
{
   numlist      list;                  /* declare array */
   int          count,i;               /* declare variables */

   count = 0;
   do {                                /* get value from 1 to 100 */
      printf("Enter # of items (1..%d):  ",LISTSIZE);
      scanf("%d",&count);
   } while (count < 1 && count > LISTSIZE);

   for (i=0; i<count; i++)             /* initialize array with */
      list[i] = rand();                /* random values         */

   lsort(list,count);                  /* sort array            */
   dumplist(list,count);               /* print array on screen */
   i = lmax(list,count);               /* get max value in array*/
   printf("The maximum value in the list is %d\n",i);
   return(0);
}
/*---------------------------------------------------------*/
/*                      end of main()                      */
/*---------------------------------------------------------*/


/*---------------------------------------------------------*/
/*   lmax() -- Returns the maximum value in an array       */
/*             Called only by main()                       */
/*---------------------------------------------------------*/

int lmax(numlist list, int count)
{
   int  i,max;
```

(outside of **main()**, that is). Call them from **main()**, and have them call other functions. Practice passing parameters: **int**, **float**, **char**, pointers, strings, arrays, structures, and so on. Find out what you need (and don't need) to do in order to pass-by-value. When you get code that works, print out a copy of it so that you can refer to it later. Now, start using more of the runtime library functions. There are over 300 predefined functions for you to use, all documented in the *Turbo C Reference Guide*. Do some exploring; select functions at random and write programs that use them. If you're really ambitious, keep writing programs until you've used each of the functions at least once.

Finally, start writing your own libraries. Collect useful functions of your own devising into one file, create a header file for it, and store it where you can get to it when you need to use it. At this point, you'll realize just how comfortable you feel in C, and you'll wonder why you were avoiding it in the first place.

## GO FOR IT

C is a powerful, popular language that is fast becoming the standard development language on mini- and microcomputers. Proficiency in C is a highly marketable skill, but to gain that proficiency, you need to understand the language well. This article provides a start, but the only sure method is to write lots of code (and *good* code) in C, on the language's own terms. Good luck, and happy coding. ■

*Bruce Webster is a computer mercenary living in the Rockies. He can be reached at Jadawin Enterprises, P.O. Box 1910, Orem, UT 84057; via MCI MAIL (as Bruce Webster) or on BIX (as bwebster.)*

*Listings may be downloaded from CompuServe as THINKC.ARC.*

```
   max = list[0];
   for (i=1; i<count; i++)
      if (list[i] > max)
         max = list[i];
   return(max);
}

/*------------------------------------------------------------*/
/*   swap() -- Swaps two integer values                       */
/*             Called only by lsort()                         */
/*------------------------------------------------------------*/

void swap(int *i, int *j)
{
   int  temp;

   temp = *i; *i = *j; *j = temp;
}


/*------------------------------------------------------------*/
/*   lsort() -- Sorts a numeric array in ascending order      */
/*              Uses selection sort algorithm                 */
/*              called only by main()                         */
/*------------------------------------------------------------*/

void lsort(numlist list, int count)
{
   int top,k,min;

   for (top=0; top < count-1; top++) {
      min = top;
      for (k = top+1; k < count; k++)
         if (list[k] < list[min])
            min = k;
      if (min != top)
         swap(&list[top],&list[min]);
   }
}


/*------------------------------------------------------------*/
/*    dumplist() -- displays a numeric array on the screen    */
/*                  called only by main()                     */
/*------------------------------------------------------------*/

void dumplist(numlist list, int count)
{
   int  i;

   for (i=0; i<count; i++)
     printf("%8d",list[i]);
   printf("\n");
}
```

# USING TURBO C

## Roll up your sleeves and take a practical course in power programming.

*Reid Collins*

■ SQUARE ONE

Borland International's Turbo C has a dual personality. For learning and tinkering, it offers TC.EXE, an integrated editor-compiler-linker that resembles the integrated environment used by Turbo Pascal. For developers of large applications, Turbo C can also be used in a more traditional command-line mode as TCC.EXE, permitting compiler and linker operations to be controlled by batch command scripts and automatic program maintainers. Both versions of the compiler are part of the Turbo C package. This package includes an enhanced MAKE program and other utilities that give the serious program developer complete control over the program development process.

To demonstrate Turbo C, we will develop two small programs. The first, HELLO, is a single module program and the second, CONCAT, is contained in two modules. CONCAT may be used as a substitute for the DOS TYPE command. First, we will examine Turbo C as an integrated environment, then as a collection of separate but related programs.

### A VERSATILE COMPILER

Turbo C is an optimizing compiler with numerous options that permit developers to customize operation for specific requirements and personal preferences. Resulting object modules may be optimized for speed or size. Other options available in the integrated environment and on the standalone compiler's command line afford the developer wide latitude in the selection of error-handling procedures, memory model, target processor, and code-generation features.

Turbo C supports the use of inline assembly language code for those who need to program the "bare metal." It also permits inter-language calling with Turbo Prolog and assembler routines. In addition, you may select the standard Pascal parameter-passing sequence to speed up function-calling times.

With the de facto C standard expressed in Kernighan and Ritchie's *The C Programming Language* and the ANSI draft standard at its roots, Turbo C is very compatible with the major C compilers currently available for the IBM PC. Code written for Turbo C can be compiled by Microsoft C, Lattice C, and most other C compilers with little or no change to the source code. The reverse is also true. Exceptions to painless portability are restricted primarily to low-level, machine-dependent routines that are not constrained by any standards, de facto or otherwise.

### WHOLE > SUM (PARTS)

The primary components of the Turbo C integrated development environment are contained in the TC.EXE program file. By combining the functions of a full-screen editor, an optimizing C compiler, a fast linker, and a project-management utility into a single program, Turbo C provides the sort of operational synergy that comes of having all the right tools instantly at hand.

Because of the close relationship of the environment's components—the compiler and the editor, for example—easy exchange of information and a high level of interaction are possible. When errors are detected by the compiler, information that identifies source lines containing the errors is fed back to the editor. You can quickly locate and correct offending source lines with the help of "point-and-shoot" control from an error list. After selecting an error from the list, you are returned to the editor with the cursor at the problem line. Switching among the Turbo C components is quick and easy.

Before using the integrated Turbo C environment, you will need to do a small amount of configuration work. The process is fairly painless because the TCINST program and TC do all the bookkeeping.

You just select options and type a few responses. As a minimum you should run TCINST to specify Turbo C's default directory. Then start TC (Figure 1 shows the main screen) and use the **O**ptions-**E**nvironment menu to set up the default paths to header files and libraries. TCINST modifies the TC.EXE file to reflect your preferences.

TC attempts to read the Turbo C configuration file, TCCONFIG.TC, at program start up. If the file does not exist or if you wish to change it, you can make selections under the TC **O**ptions menu. You provide values for various compiler, linker, and environment variables, then select the **S**tore options entry to preserve the values.

You will probably want to add the Turbo C directory to your PATH environment variable so that the programs in it can be run from anywhere in the directory hierarchy. This is best done by editing the line containing the PATH command in AUTOEXEC.BAT. After saving the change, run AUTOEXEC from the DOS command line or reboot the system. Either method will write the new path specification into the DOS environment.

### HELLO THERE

To test the Turbo C installation and configuration, create a simple program source file, compile it, and run it. Listing 1 is a variation on the HELLO program, which is the *de rigueur* C test program. This version of the program uses the low-level **write** function from the standard runtime library instead of the usual **printf**. The **sizeof** operator calculates the number of bytes in the message. Using **write** results in a smaller executable program (2120 bytes vs. 5760 bytes), because it doesn't drag unneeded formatting code into the executable file.

Creating and editing the source file with the built-in editor is easy. To edit the file, go to the **F**ile menu and select the **L**oad entry. Respond with the name HELLO.C, which the editor will adopt as the name of the current file. The Turbo C editor is immediately familiar to anyone who has used other Borland language products. If the default commands don't suit you, change them by using the **O**ptions-**E**ditor menu.

After the source code has been keyed in, save the file (Alt-F, then **S**, for **S**ave) and then compile it (Alt-C to get the **C**ompile menu and select the **M**ake EXE file option). The default is a small-model program optimized for speed, but you can instruct the compiler to use other models and optimizations from the **O**ptions-**C**ompiler menu.

***The Turbo C editor is immediately familiar to anyone who has used other Borland language products.***

The resulting program can be tested by using the **R**un selection of the main menu (Alt-R). Because a permanent copy of the program is saved to disk, you can also leave Turbo C (Alt-F, then **Q**, for **Q**uit) and run the HELLO program from the DOS command line.

### A TRIP TO THE PROJECTS

Our first example (HELLO.C) is a simple, single-module program. The program is contained in one source file and the compiling and linking steps are triggered by a single command (**C**ompile-**M**ake EXE file). More complex programs are often divided into modules, with each module containing one function or a small collection of closely related functions. How do we handle such programs? Certainly there will be additional work to do to keep track of all the pieces of a program. What happens if a change is made to one of the source files? Will it be necessary to recompile the entire program? No, because "project making" takes care of the details.

The Turbo C integrated development environment is as well suited to the task of preparing multi-module programs as it is to preparing single-module programs. In fact, you still start the process of compiling and linking with a single command (Alt-R). However, you become directly involved in project making because you create a project-control file that lists the names of all modules that comprise the program. You only have to do this once, (unless, of course, you change the organization of the program).

Project making takes the tedium out of building complex programs. When you invoke the **P**roject feature of Turbo C directly (Alt-P) or indirectly by selecting **C**ompile or **R**un, Turbo C takes a list of module names from the project-control file and makes sure that each object file is current with respect to its related source file. When you edit a source file, the modification date and time in the file's directory entry is set to the date and time when the changes are saved on disk. Assuming your system clock is correct, Turbo C compares the object file time to the source file time and recompiles the source file if it's newer than the object file. Only source files that have been modified since they were last compiled need to be recompiled.

The following multi-module example should help to clarify the project-making process. Refer to Figure 2 as you read this description.

The program CONCAT, short for concatenate, can be used to display the contents of ASCII text files and to combine the contents of several files into one by using output redirection. The program is produced from two source files, CONCAT.C (Listing 2) and FILECOPY.C (Listing 3). In addition to typing in the two source files, you must create the file CONCAT.PRJ, which is the list of filenames on which the program is based. Listing 4 contains the text of

*continued on page 50*

LISTING 1: HELLO.C

```
#define STD_OUT 1

main()
{
        static char msg[] = { "Hello, World!\n" };

        write(STD_OUT, msg, sizeof(msg));
        exit(0);
}
```

LISTING 2: CONCAT.C

```
/*********************************************************************
 *  C O N C A T
 *
 *  Concatenate files.  For each file named as an argument, CONCAT
 *  writes the contents of the file to standard output.  Command-line
 *  redirection may be used to collect the contents of multiple files
 *  into a single file.  This program is adapted for DOS from the
 *  "cat" program presented in "The C Programming Language", by
 *  Kernighan and Ritchie, Prentice-Hall, 1978.  Modifications include
 *  argv[0] processing for DOS and improved error handling.
 *
 *  Exitcodes (DOS ERRORLEVEL):
 *      0       success
 *      1       error opening a named file
 *      2       I/O error while copying
 *      3       error closing a file
 ********************************************************************/

#include <stdio.h>

/* function prototype */
extern int filecopy(FILE *, FILE *);

main(argc, argv)
int argc;
char *argv[];
{
    int i;              /* loop index */
    FILE *fp;           /* input file pointer */

    static char progname[] = { "CONCAT" };      /* program name */

    /*
     *  Be sure that argv[0] is a useful program name.  Under DOS 3.x
     *  and later, argv[0] is the program name.  The program name is
     *  not available under earlier versions of DOS and is presented
     *  as a null string ("") by Turbo C.
     */
    if (argv[0][0] == '\0')
        argv[0] = progname;

    /* if no filenames are given, use standard input */
    if (argc == 1) {
        if (filecopy(stdin, stdout) == EOF) {
            perror(argv[0]);    /* display the system error message */
            exit(2);
```

## USING TURBO C

*continued from page 49*

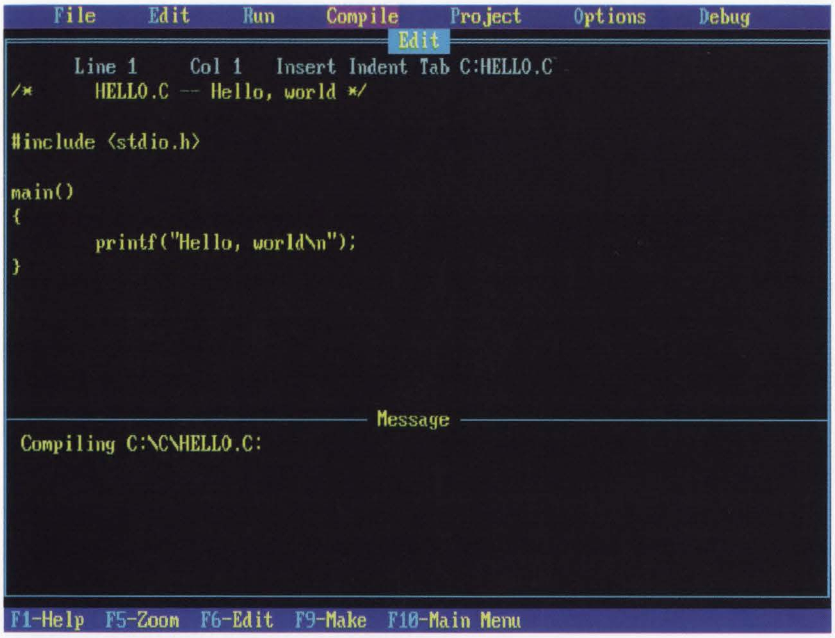
CONCAT.PRJ. Each filename in the project-control file may be typed with or without a .C extension.

Tell Turbo C to make the CONCAT project by going to the **P**roject menu, selecting **P**roject name, and entering CONCAT.PRJ, then pressing Enter. Invoking either the **R**un or **C**ompile-**M**ake EXE file menu executes the project-make feature. The results of the project-make operation are updated object files and an executable program file.

The integrated linker, which is responsible for combining all the components of a program, requires a set of instructions to tell it the program name, the linkable objects and libraries needed to produce the program, and whether to produce a MAP file. The program name is taken from the name of the project file. The instructions are prepared by the Turbo C project maker and are stored in the file TPROJ.LNK (Listing 5).

The first object in the list of linkable objects is the C0S.OBJ file (small model—other models use different names). This is the run-time startup module that each program must have to set up correct operating conditions and get information from the DOS command line. On successive lines, the link file lists two additional object files, CONCAT.OBJ and FILECOPY.OBJ, that must be linked. A "+" at the end of a line indicates that the object list continues onto the next line.

The next two lines provide the program and map file base names to which the linker appends .EXE and .MAP, respectively. The last line is a list of library files that are to be searched for modules that resolve names found in the program's object modules. For example, calls to **fopen**, **exit**, and other functions are satisfied by the linker when it includes copies of the object code for the named modules into the executable module. The linker only copies code that is needed from the



```
     Line 1     Col 1   Insert Indent Tab C:HELLO.C
/*      HELLO.C -- Hello, world */

#include <stdio.h>

main()
{
        printf("Hello, world\n");
}
```

```
 Compiling C:\C\HELLO.C:
```

*Figure 1. A Turbo C integrated development environment display.*

libraries in an attempt to keep the final program size to a minimum.

We have just used the integrated development environment provided by Turbo C's TCC.EXE program to produce a multi-module program. A contrasting view of the world of C program development is that each component of the C programming system should be a separate entity. One program is used for editing, another for compiling, and yet another for linking. Still other programs are used to manage program development and maintenance tasks. With Turbo C, you can have it your way. Let's see how to use the command-line mode.

### THE N COMMANDMENTS

Moses had it easy. He only had to deal with ten command(ment)s. The Turbo C command-line compiler presents you with a mind boggling array of command-line options that effectively force (or at least request) differing behavior of the C compiler system. But fear not, because you usually can get by remembering just a few of them. Most of the time, the options are cast into commands in batch files or *makefiles* where they are invoked automatically.

The primary advantage of using separate programs for editing and compiling is that each can be easily replaced by another that performs the same function in a way that better suits a particular programmer's needs. For example, you can use your favorite program editor instead of struggling to learn the peculiarities of the one provided with a vendor's C system. Although the vendor's editor may be perfectly acceptable, it may be markedly different from the one you know, making it uninviting to use. With separate programs, you can choose your editor.

The TCC program is the heart of the Turbo C system. TCC is comparable to the **cc** command under UNIX. It is the control program that can be used either to compile a single C source file or to orchestrate the process of compiling multiple modules. TCC's project management facilicy handles calling the linker to combine the modules into an executable program. We will, however, use TCC as a module compiler and use a separate program, MAKE, to manage the program preparation. MAKE is described in the next section.

The TLINK program is a standalone linker. It is compatible with the DOS linker, but because it has few optional features, it is both smaller and faster than most other linkers. It produces standard DOS .EXE files. When used to link tiny model programs (all code and data in a single 64K segment), the resulting .EXE file can be converted to .COM format. This makes it possible to write DOS device drivers and ROMable code in C instead of assembler.

Some configuration of the Turbo C command-line system is appropriate. Listing 6, TURBOC.CFG, contains a single line that is read by TCC and TLINK. It identifies the directories in which to find header files and libraries.

*continued on page 52*

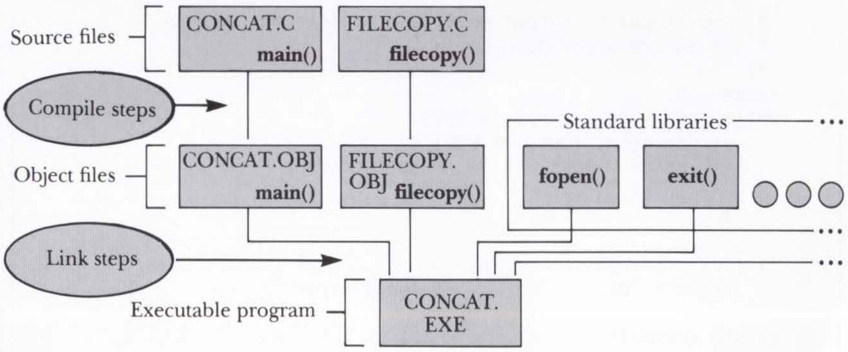


*Figure 2. Producing a multi-module program.*

```
    }
}
else
    /* process the named files one at a time */
    for (i = 1; i < argc; ++i) {
        /* attempt to open the source file */
        fp = fopen(argv[i], "r");
        if (fp == NULL) {
            /* unable to open the file */
            fprintf(stderr, "%s: cannot open %s\n",
                argv[0], argv[i]);
            continue;       /* look for more files */
        }
        else {
            /* copy the current file to the standard output */
            if (filecopy(fp, stdout) == EOF) {
                perror(argv[0]);
                exit(2);
            }
            /* close the current file */
            if (fclose(fp) == EOF) {
                fprintf(stderr, "%s: error closing %s\n",
                    argv[0], argv[i]);
                exit(3);
                }
            }
        }

    exit(0);
}
```

LISTING 3: FILECOPY.C

```
/********************************************************************
*   F I L E C O P Y
*
*   Copy the input stream to the output stream.  Return 0 if the
*   copy is successful or EOF for any I/O error.
*******************************************************************/

#include <stdio.h>

int
filecopy(fin, fout)
FILE *fin;      /* input stream pointer */
FILE *fout;     /* output stream pointer */
{
    int ch;     /* holds ASCII characters and EOF */
    int rcode;  /* return code */

    /*
     *  Copy input to output until end of file is reached
     *  or an I/O error occurs.
     */
    rcode = 0;
    while ((ch = getc(fin)) != EOF)
        if (putc(ch, fout) == EOF) {
            rcode = EOF;        /* output error */
            break;
        }

    if (ferror(fin))
        rcode = EOF;            /* input error */

    return (rcode);
}
```

## USING TURBO C

*continued from page 51*

You should use the correct names for your setup. The Turbo C manuals suggest putting all files in directories starting with TURBOC (e.g., C:\TURBOC), although nothing in the software requires it. I used C:\TC instead (less typing!) with no ill effects.

### COOKING WITH MAKE

I say "cooking" with MAKE because MAKE is a utility program that reads a "recipe" for a program, mixes and matches ingredients, and produces a product. Using MAKE, a separate program that does much the same job as the project-make feature of the integrated development environment, you could become the "Galloping Gourmet" of programmers. MAKE implements a series of instructions in an external data file, usually called MAKEFILE, to control the compilation of C sources, the assembly of assembly language sources, and the linking of objects and library modules to produce an executable program. MAKE employs file date and time stamps to determine which components of a program need to be remade.

To re-MAKE a program, simply type MAKE at the DOS prompt in the directory containing MAKEFILE. By default, MAKE looks for the name MAKEFILE unless told explicitly to use some other filename.

Listing 7 is the makefile for the CONCAT program. Any text that follows a pound sign (#) on a line is a comment. The MAKEFILE begins with a statement of the rule used by the TCC program to produce an object file from a C source file. Then some symbolic names are defined. Note that the MDL name (memory model) is required in the compiler rule but has not yet been defined when the rule is read. That's okay with the Turbo C MAKE command because it will rescan the file as needed to take care of such forward references.

The remaining instructions tell

MAKE to run the linker on the specified objects and libraries. The line

```
concat.exe:    $(OBJS)
```

tells MAKE that the executable file depends on the object files. **$(OBJS)** expands to **concat.obj filecopy.obj** when the line is scanned. If either object file is out of date (older than its related source file), it will be recompiled. If CONCAT.EXE is older than either of the object files on disk (or if CONCAT.EXE does not exist), the program will be relinked.

***It is not necessary with the Turbo C* MAKE *program to describe the dependencies of each file independently.***

The last line shows the dependencies of object files on source files. It is not necessary with the Turbo C MAKE program to describe the dependencies of each file independently as it is with the Microsoft MAKE program, for example.

When developing major applications, you will probably find it best to use the integrated development environment (TC) for initial design and experimentation. You can then switch to the command line mode for intense development and project maintenance. At that point, MAKE becomes an indispensable tool in the quest to keep a program up to date without a lot of manual bookkeeping and wasted time. ■

*Reid Collins is a computer programmer for a firm in the aerospace industry.*

*Listings may be downloaded from CompuServe as USETC.ARC.*

**LISTING 4: CONCAT.PRJ**

```
concat
filecopy
```

**LISTING 5: TPROJ.LNK**

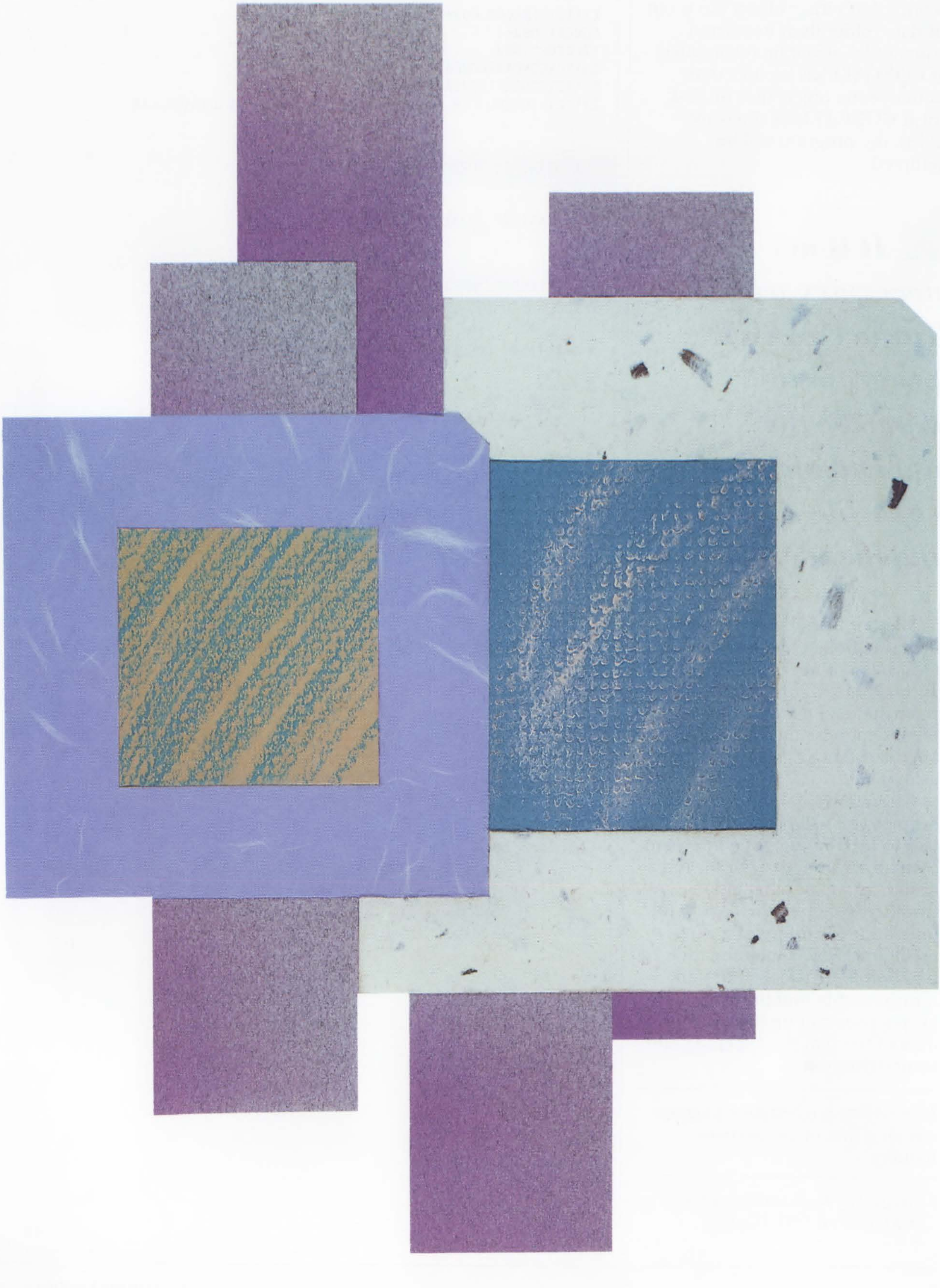
```
C:\TC\LIB\COS.OBJ+
CONCAT.OBJ+
FILECOPY.OBJ
C:\TC\CONCAT\CONCAT
C:\TC\CONCAT\CONCAT/N/D/C/X
C:\TC\LIB\EMU.LIB C:\TC\LIB\MATHS.LIB C:\TC\LIB\CS.LIB
```

**LISTING 6: TURBOC.CFG**

```
-Lc:\tc\lib -Ic:\tc\include
```

**LISTING 7: MAKEFILE**

```
# makefile: program builder for CONCAT

# rules
.c.obj:
    tcc -c -m$(MDL) $<

# symbolic constants
MDL = s
LIB = c:\tc\lib
SRCS = concat.c filecopy.c
OBJS = concat.obj filecopy.obj

# instructions
concat.exe:     $(OBJS)
    tlink $(LIB)\c0$(MDL) $(OBJS), $*, , $(LIB)\c$(MDL)

$(OBJS):        $(SRCS)
```



# WHICH PROCESSOR?

## Let your software determine what sort of engine is under the hood.

*Juan E. Jimenez*

WIZARD

The success of Intel's microprocessors, in conjunction with the IBM Personal Computer, has resulted in a proliferation of machines with different but related microprocessors, all running the same DOS operating system. This creates a problem for developers who want to take advantage of the newer CPUs such as the 80386, but still maintain compatibility with the 8088 and the 8086.

You can solve this problem by writing code that only uses the standard 8086 instruction set, if such a chip is resident in the computer where the program is running. However, if the computer has a more advanced microprocessor installed, then your program should make use of an extended instruction set.

But how do you identify the type of microprocessor in the computer? A few routines have appeared in the past, but most of them were dependent on obscure features and old chip design errors, and were unnecessarily complex. Basing a software decision on a bug in a chip design is risky business indeed—no CPU manufacturer should be expected to maintain bugs in chip mask redesigns for the benefit of programmers using the bugs as though they were features. The methods I describe below all depend on documented CPU features that are unlikely to change as Intel refines its chip designs. I am indebted to Mr. Jose Sanders, Intel Field Engineering Representative for Puerto Rico and the Caribbean, for providing the information upon which this article is based.

I have written a simple routine called GETCPU that identifies the 8088/86, 80188/186, 80286 or 80386 CPUs. Two different versions of GETCPU are presented here: Listing 1 is for Turbo C, and Listing 2 is for Turbo Pascal 4.0. The two versions differ in the calling requirements of the host high-level language; the actual MASM logic that tests the CPU is identical. Listing 3, WHATCPU.C, is a short program that invokes GETCPU and prints out the name of the CPU identified in the system. Listing 4, WHATCPU.PAS, is an identical program written in Turbo Pascal 4.0. Listing 5, WHATCPU.PRJ, is a Turbo C project file for recompiling and relinking WHATCPU.C. (In a sidebar to this issue's "Language Connections" column, Gary Entsminger describes how to interface GETCPU to Turbo Prolog; Bruce Tonkin describes the interface to Turbo Basic, in "Converting .COM Files to $INCLUDE Files.")

### IDENTIFYING THE 80286/80386 CHIPS

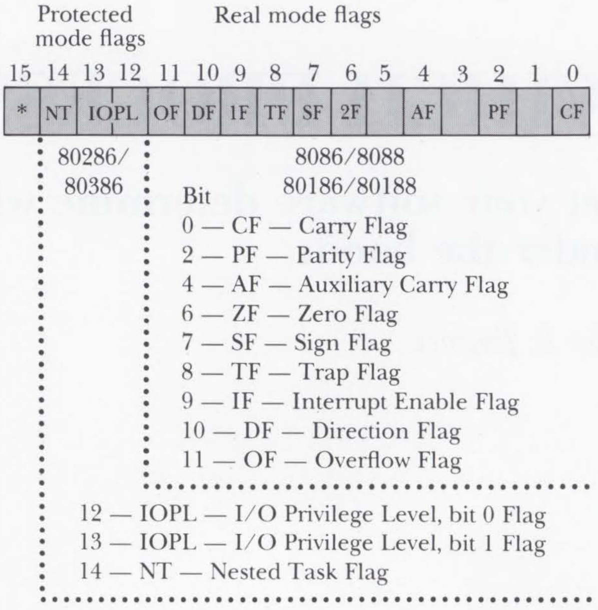
Here's how it works. The first test checks to see if we are working with an 8088/86/188/186 chip, or one of the more advanced 80286 or 80386 designs. The test examines the flag register of the CPU (see Figure 1). If you look over Intel's technical specifications for these chips, you will see that bit 15 of the flag register is undefined in all of these processors. However, Intel documentation states that this bit is always 1 in the case of the 8088/86/188/186 CPUs, and 0 in the case of the 80286 and 80386. It cannot be forced to the opposite state. In the first portion of the routine we try to set the upper bit of the flag register to 1; if we succeed, we know we have an 8088/86 or 80188/86. If we don't, we know we have either an 80286 or an 80386.

If we determine that the CPU is *not* an 8088/86 or an 80188/86, the second portion of GETCPU determines whether we are running an 80286 or 80386. This is done by attempting to set bits 12, 13, and 14 of the flag register to 1s. These bits represent the nested task flag (bit 14) and I/O privilege level (bits 12 and 13). After a RESET, all three of these bits are set to 0. On the 80286 these bits remain 0s, and cannot be set to 1s while in real mode. However, on the 80386 these bits can be set to 1s while in real mode, though doing so will have no effect. So, we try to set the three bits to 1s, and if we can't we know we have an 80286. If the bits can be set successfully, we know we have an 80386. In either case we are done and

*continued on page 56*

LISTING 1: C_GETCPU.ASM

```
        name    C_GETCPU
        page    55,132
        title   "C_GETCPU -- Determines Which INTEL CPU is Installed"
;-------------------------------------------------------------------
; This program determines which INTEL CPU is being used in the
; machine, whether it is an 8088/86, 80188/186, 80286 or 80386.
; It uses documented and supported differences in flag register bit
; configurations to determine whether the CPU is an 80286 or 80386,
; and differences in shifting using CL to determine if it is an
; 8088/86 or 80188/186. It is intended to be used as an external
; routine from Turbo C, and returns an integer result in the form
; the last three digits of the processor type, as depicted in the
; table below.  This code is designed for the TINY/SMALL modem.
; See page 254 of the Turbo C User's Guide for information on how
; to modify this routine for other memory models.
;
; If the processor is     The routine returns
; -------------------     -------------------
;
;      80386                   386
;      80286                   286
;      80188/186               186
;      8088/86                  86
;
;-------------------------------------------------------------------
; Declaration of the routine in Turbo C is:
;
; int C_GETCPU();
;
;-------------------------------------------------------------------
; To assemble:
;
; MASM C_GETCPU,,,;
;
;-------------------------------------------------------------------
; Code segment begins here
;-------------------------------------------------------------------
                        ; Required by Turbo C for small memory model
_TEXT   segment byte public 'CODE'
        assume cs:_TEXT         ; Ditto
;-------------------------------------------------------------------
; Actual ID routine begins here
;-------------------------------------------------------------------
        public  _C_GETCPU       ; Make sure Turbo C can get here
_C_GETCPU proc  near            ; Entry point for the subroutine
        pushf           ; Save flag registers, we use them here
        xor     ax,ax   ; Clear AX and...
        push    ax      ; ...push it onto the stack
        popf            ; Pop 0 into flag registers (all bits to 0),
        pushf           ; attempting to set bits 12-15 of flags to 0's
        pop     ax              ; Recover the save flags
        and     ax,08000h       ; If bits 12-15 of flags are set to
        cmp     ax,08000h       ; zero then it's 8088/86 or 80188/186
        jz      _8x_18x
;-------------------------------------------------------------------
; It is either an 80286 or an 80386, let's find out which...
;-------------------------------------------------------------------
        mov     ax,07000h       ; Try to set flag bits 12-14 to 1's
        push    ax              ; Push the test value onto the stack
        popf                    ; Pop it into the flag register
        pushf                   ; Push it back onto the stack
        pop     ax              ; Pop it into AX for check
        and     ax,07000h       ; if bits 12-14 are cleared then
        jz      _286            ; the chip is an 80286
```



*Always 1 for 8086/88/186/188 Always 0 for 80286/386

*Figure 1. The 86-family flag register*

## WHICH PROCESSOR

*continued from page 54*

can exit. Note that this may or may not work correctly in a protected mode environment like OS/2; the methods described in this article were designed and tested for use under DOS in real mode *only.*

### DISTINGUISHING THE "STANDARD CHIPS"

Discriminating between an 8088/86 or 80188/86 is a little more difficult. In this case, we use the SHL (shift left) and CL with a count command. As it turns out, the 8088/86 uses all the bits in CL to perform the shift, allowing a shift value of up to 255. However, the 80188/86 only use the lowest five bits in CL for the shift (as do the 80286 and 80386). Therefore, we set AX to all 1s, then try to shift AX left 33 times. In the 8088/86, we get a full 33-bit shift, leaving a value of 0 in AX. In newer CPUs, though, attempting to shift by 33 (100001 binary) amounts to a shift by only 1, since bits 5 and higher in CL are ignored.

We simply check AX to see if it contains a value of 0. If it does, we have an 8088/86 and we can exit. If we find a nonzero value, we have an 80188/86.

### RETURNING THE PROCESSOR NAME

GETCPU passes the result back to the host language as an integer. The integer value is 86 if we have an 8088/86, 186 for an 80188/186, 286 for an 80286, and 386 for an 80386. Note that no effort is made to distinguish between an 8088 and an 8086, or between an 80188 and an 80186. The differences between the members of these two closely related pairs of chips lie almost entirely in the bus structure that interfaces the CPU to the outside world. The 8088 and 80188 are essentially 8-bit parts with a 4-byte prefetch queue, while the 8086 and 80186 are 16-bit parts

```
;--------------------------------------------------------------------
; Ok, we know it's an 80386 now, tell the user about it!
;--------------------------------------------------------------------
        mov     ax,386  ; It's not a 286, so it must be an 80386
        jmp     DONE    ; (at least until the 80486 comes out...)
;--------------------------------------------------------------------
; Tell the user it's an 80286
;--------------------------------------------------------------------
_286:   mov     ax,286          ; Get the msg ready
        jmp     DONE            ; Bye
;--------------------------------------------------------------------
; We know it is either an 8088/86 or 80188/86, but which one is it?
;--------------------------------------------------------------------
_8x_18x:
        mov     ax,0FFFFh       ; Set AX to all 1's
        mov     cl,33   ; Now we try to shift left 33 time. If it's
        shl     ax,cl   ; an 808x it will shift it 33 times, if it's
                        ; an 8018x it wil only shift one time
        jnz     _18x    ; Shifting 33 times would have left all 0's
                        ; if any 1's are left it in an 80188/186
        mov     ax,86   ; No 1's, it's an 8088/86
        jmp     DONE
;--------------------------------------------------------------------
; It's an 80188 or 80186...
;--------------------------------------------------------------------
_18x:   mov     ax,186  ; Found a 1 in there somewhere, it's an 8018x

;--------------------------------------------------------------------
; All done, let's go back...
;--------------------------------------------------------------------
DONE:   popf            ; Restore the flag registers
        ret
;--------------------------------------------------------------------
; End of code and segment
;--------------------------------------------------------------------
_C_GETCPU endp
_TEXT   ends
        end     _C_GETCPU
```

LISTING 2: GETCPU.ASM

```
   name GETCPU
   page 55,132
   title    'GETCPU.BIN --- Determines which INTEL CPU is installed'
;--------------------------------------------------------------------
; This program determines which Intel CPU is being used in the
; machine, whether it is an 8088/86, 80188/186, 80286 or 80386.
; It uses documented and supported differences in flag register bit
; configurations to determine whether the CPU is an 80286 or 80386,.
; and differences in shifting using CL to determine if it is an
; 8088/86 or 80188/186. It is intended to be used as an external
; routine from Turbo Pascal, and returns an integer result in the
; form of the last three digits of the processor type,
; as depicted in the table below.
;
; If the processor is    The routine returns
; -------------------    -------------------
;     80386                     386
;     80286                     286
;     80188/186                 186
;     8088/86                    86
;
```

*continued on page 58*

with a 6-byte prefetch queue.

The instruction sets of the 8088 and the 8086 are identical, as are the instruction sets of the 80188 and 80186. Although the members of the two pairs can be distinguished from one another by testing the size of the prefetch queue, there is little point in it for the programmer if the instruction sets are the same.

In the best of all worlds, a CPU should be able to tell an application program what it is in response to a suitable query. The 386 (and, according to Intel, all future 8086-family chips) have chip type and other information placed in the general registers upon a hardware reset. However, no BIOS that we know of saves that information for later reference by application software, so we cannot get at it for the time being. I would encourage developers of future BIOS software to consider the needs of programmers whose code must run on all compatible chips, and keep this chip-identifying information in a safe place after a reset. ■

*Juan Jimenez is an independent computer consultant, programmer, systems analyst, and hacker. He can be reached at P.O. Box 9811, Santurce Station, Santurce, PR 00907.*

*Listings may be downloaded from CompuServe as GETCPU.ARC.*

*continued from page 58*

```
;--------------------------------------------------------------------
; Declaration of the routine in Pascal V4.0 is:
;
;      {$L GETCPU}
;      function GETCPU : integer; external;
;
;--------------------------------------------------------------------
; To assemble:
;
; MASM GETCPU;
;--------------------------------------------------------------------
; Code segment begins here
;--------------------------------------------------------------------
code     segment   para public 'CODE'
    assume    cs:code
    public    getcpu
;--------------------------------------------------------------------
; Actual id routine begins here
;--------------------------------------------------------------------
getcpu   proc near
    pushf                ; Save the flag registers, we use them here...
    xor  ax,ax           ; Clear AX and push it onto the stack
    push ax
    popf            ; Pop 0 into flag registers (all bits to 0),
    pushf           ; attempting to set bits 12-15 of flags to 0's
    pop  ax         ; Recover the saved flags
    and  ax,08000h ; If bits 12-15 of flags are set to zero then
    cmp  ax,08000h ; cpu is 8088/86 or 80188/86
    jz   _8x_18x
;--------------------------------------------------------------------
; It is either an 80286 or an 80386, let's find out which...
;--------------------------------------------------------------------
    mov  ax,07000h ; Try to set flag bits 12-14 to 1's
    push ax         ; Push the test value onto the stack
    popf            ; Pop it into the flag register
    pushf                ; Push it back onto the stack
    pop  ax         ; Pop it into AX for check
    and  ax,07000h ; If bits 12-14 are cleared then the chip is
    jz   _286       ; an 80286
;--------------------------------------------------------------------
; Ok, we know it's an 80386 now, tell the user about it!
;--------------------------------------------------------------------
    mov  ax,386          ; It's not a 286, so it must be a 386
    jmp  DONE
;--------------------------------------------------------------------
; Tell the user it is an 80286
;--------------------------------------------------------------------
_286:    mov  ax,286          ; Get the msg ready
    jmp  DONE
;--------------------------------------------------------------------
; We know it is either an 8088/86 or 80188/86, but which one is it?
;--------------------------------------------------------------------
_8x_18x:
    mov  ax,0ffffh ; Set AX to all 1's
    mov  cl,33           ; Now we try to shift left 33 times. If it's
    shl  ax,cl           ; an 808x it will shift it 33 times, if it's
                    ; an 8018x it will only shift one time.
    jnz  _18x       ; Shifting 33 times would have left all 0's.
                    ; If any 1's are left it's an 80188/186
    mov  ax,86           ; No 1's, it's an 8088/86
    jmp  DONE
;--------------------------------------------------------------------
; It's an 80188 or 80186...
;--------------------------------------------------------------------
_18x:    mov  ax,186     ; Found a 1 in there somewhere, it's an 80188
                         ; or an 80186
;--------------------------------------------------------------------
; All done, let's go back...
;--------------------------------------------------------------------
DONE:    popf            ; Restore the flag registers
    ret
;--------------------------------------------------------------------
; End of code and segment
;--------------------------------------------------------------------
getcpu   endp
code     ends
    end  getcpu
```

LISTING 3: WHATCPU.C

```
/* ----------------------------------------------------------------- */
/* WHATCPU.C - Turbo C program to show example of how to use the     */
/*             C_GETCPU assembly language module.                    */
/* ----------------------------------------------------------------- */

#include <stdio.h>

int C_GETCPU();

main()
{
  int   CPU_Type;                       /* Receives result        */
  CPU_Type = C_GETCPU();                /* Call the function      */
  printf("Processor is [80");           /* Print common msg       */
  switch (CPU_Type)                     /* Depending on result    */
  {                                     /* Print rest of msg      */
    case 386:  printf("386");
               break;
    case 286:  printf("286");
               break;
    case 186:  printf("188/186");
               break;
    case  86:  printf("88/86");
               break;
  }
  printf("]\n");                        /* Terminate msg string   */
}
```

LISTING 4: WHATCPU.PAS

```
PROGRAM WHATCPU;

VAR
   CPUTYPE: integer;

{$L GETCPU}
FUNCTION  GetCPU : integer; external;

BEGIN
     CPUTYPE := GetCPU;
     write('Processor is [80');
     CASE CPUTYPE of
          386: writeln('386]');
          286: writeln('286]');
          186: writeln('188/86]');
          86 : writeln('88/86]');
     END
END.
```

LISTING 5: WHATCPU.PRJ

```
whatcpu
c_getcpu.obj
```

# Turbocharge Your Programming With Turbo Basic!

> ❝ Borland International's Turbo Basic is unquestionably an outstanding software product. It provides an efficient and comprehensive BASIC programming environment at a very affordable price.
>
> An excellent BASIC development system with enhancements that allow more effective programming.
>
> *Giovanni Perrone, PC Week*

> Turbo Basic sets a standard for programming languages on PCs that is the equivalent of the first running of the four-minute mile.
>
> Corporate users of BASIC will find Turbo Basic a tool worth many times its cost and a quantum improvement over anything they have ever used.
>
> *William Zachmann, Computerworld* ❞

Turbo Basic® is the BASIC compiler you've been waiting for! It's a complete development environment with an amazingly fast compiler, a full-screen windowed editor, pull-down menus, and a trace debugging system. We've also added many innovative features including binary disk files, true recursion, and several new compiler directives to give you more control at compile time. And your program size isn't limited by 64K—you can use all available memory!

> ❝ I'm extremely impressed with Turbo Basic. It's fast, it cooperates with resident keyboard handlers . . . it offers a wealth of important new features, and it costs only $99.
>
> *Ethan Winer, PC Magazine* ❞

### *A technical look at Turbo Basic*

- Context-sensitive help
- Full recursion supported
- Customizability of user interface and editor
- Full 64K for strings
- Standard IEEE floating-point format
- Floating-point support, with full 8087 (math coprocessor) integration. Software emulation if no 8087 present
- Program size limited only by available memory (no 64K limitation)
- EGA and CGA support
- IBM Personal System/2 VGA and MCGA 2- and 16-color support in 640 x 480 resolution
- Full integration of the compiler, editor, runtime libraries, and executable program, with separate windows for editing, messages, tracing, linker libraries, user interface, and execution in one compiler file
- Compile, runtime, and I/O errors place you in source code where error occurred
- Access to local, static & global variables
- New long integer (32-bit) data type
- Full 80-bit precision
- Pull-down menus
- Full window management

Giovanni Perrone quote, reprinted from PC Week, May 5th, 1987. Copyright 1987 Ziff Communications Company. Ethan Winer quote, reprinted from PC Mag, May 12th, 1987. Copyright 1987 Ziff Communications Company. William Zachmann quote, reprinted from Computerworld, May 4th, 1987 with permission.

BORLAND
INTERNATIONAL

For the dealer nearest you or to order by phone

**Call (800) 255-8008**

in CA (800) 742-1133
in Canada (800) 237-1136

BORLAND

# IMPORTING REFLEX DATABASES

## Hook into a Reflex database to give your Turbo C application added flexibility

*Kent Porter*

Reflex, like most major application systems (Lotus 1-2-3, dBASE, Paradox, etc.) has proprietary file structures. Instead of simply writing data to a file, Reflex stores control information at the start of the file, followed by the data itself. Later, when the file is read back in, Reflex uses the control information to reconstruct labels, sort, and display the data in its proper format. Thus, Reflex databases are self-describing entities.

That is, they are if the receiving program knows how to interpret the control information. In the absence of this knowledge, the file is just a bunch of gobbledygook; with it, the program rapidly and efficiently processes the file's contents. This article takes us on a journey through the most important Reflex data control structures, providing practical information as well as a close look at the subtleties of a major application system. We'll also develop a program that prints out a Reflex database structure, and show how to extract data from it, all using Turbo C.

### THE BIG PICTURE

A Reflex database consists of fixed and variable-length structures in a preset order. As we'll see later, a system of pointers glues the whole thing together and provides software paths for getting from place to place. Here's the overall structure of a Reflex database file:

1. Header record
    a. General information
    b. Section descriptors
    c. Empty space to complete 512 bytes
2. Field directory
    a. Sort specifications
    b. Field name pool
    c. Field descriptor table
3. Enumerated ("repeating") text pools by field
4. Master record
5. Data records
6. Other sections (not of interest to programmers)

Some of the terms in this list might not be familiar. Don't worry; you'll know what they mean by the time we've finished, and how each contributes to the overall structure.

### GO TO THE HEAD OF THE FILE

Most of the Reflex file control information can be described in terms of Turbo C structures. We'll present them as we go along so that you can see what each one contains. Later we'll assemble all of them into a file called REFLEX.H, which you can **#include** at the start of any C program that accesses Reflex databases.

The very first component of the file is a 66-byte *header record* containing general information about the database and the Reflex version that created it. Figure 1 lists the header record structure.

Note that some of information is useful to programmers and some is not. For example, the three "ver" fields (**verViews**, etc.) give version numbers for the Reflex software components that created the database. These will help future releases of Reflex cope with old files but are of no value to your application. On the other hand, some of the fields are important.

The **stamp[]** field identifies the file as a Reflex database. It contains the null-terminated character sequence

```
3Q.!&@#$!&&
```

which is fixed in all databases. Your program should check this field for a match against a constant; if unequal, you are not reading a Reflex file. The **fRecalc** field normally contains a zero, indicating that the file was recalculated before being saved. If it contains a nonzero value, some of the data in the file might not be correct, as in the case of a file being modified and saved with **Recalc** turned off.

```
#define RXID      "3Q.!&@#$!&&"                      /* Reflex identifier */

typedef struct {                                 /* file header structure */
  int     hdrSz;                             /* size of file header = 512 */
  char    stamp[12];                /* identification = "3Q.!&@#$!&&" */
  int     dirty;                      /* non-zero implies corrupt file */
  int     verViews;                         /* view info version level */
  int     verModels;                 /* modeling system version level */
  int     verData;                           /* raw data version level */
  int     fRecalc;                 /* non-zero implies recalc necessary */
  char    screenType;            /* active screen type at file creation */
  char    checkSum;                                  /* file checksum */
  char    reserved[38];                     /* reserved for future use */
  int     sectionCt;               /* number of sections of type DFDESC */
  DFDESC dfSection[];                            /* section descriptors */
} DFHDR;
```

*Figure 1. Reflex header record structure described in Turbo C.*

```
typedef struct {       /* data file section descriptor */
   int dfType;         /* section type code */
   long dfAddr;        /* start address in file (bytes) */
   long dfLen;         /* length (bytes) */
   } DFDESC;
```

*Figure 2. Descriptor node structure in Turbo C.*

```
typedef struct {                                   /* field sort spec */
  unsigned  fldType      : 7;             /* used internally by Reflex */
  unsigned isAscending  : 1;                          /* sorting order */
  unsigned  fieldID      : 8;          /* field ID: index to field defin */
} FLDSORTSPEC;
```

*Figure 3. Turbo C field structure for sorting.*

The **sectionCt** field contains a value indicating how many section descriptors follow the header record.

**Section Descriptors.** A Reflex file contains up to 12 sections—major subdivisions—in addition to the header record. The *section descriptors* identify what and where they are. Only three of the 12 are mandatory and thus useful to programmers, with the rest (the View Manager state, global filter, etc.) being internal to Reflex.

Figure 2 lists the structure of the descriptor node. One such 10-byte node exists for each section present in the file; they begin immediately after the **sectionCt** field in the header record. We need to pay attention to nodes pertaining to the data control and content sections of the file, as shown in Table 1.

| dfType | SECTION |
|---|---|
| 2 | Field directory |
| 9 | Database master record |
| 1 | Data records |

*Table 1. The three mandatory section descriptors in Reflex.*

The three descriptors in Table 1 appear in every Reflex database. Any of the other nine may or may not be present.

**Empty Space.** The header node occupies 66 bytes and each descriptor occupies 10. Therefore the entire header record, which describes the database in general, uses somewhere between 96 and 186 bytes, but its physical length is 512. What's in the rest? Nothing. It's reserved space.

## DESCRIBING FIELDS

The *field directory* section contains almost everything you need to know about the file except its actual data contents. It consists of three major segments:

- Global sort specifications
- Field names
- Field descriptors

**Global Sort Specs.** Reflex lets you sort on up to five fields, independently, in either ascending (A to Z) or descending (Z to A) order. A precedence is assigned to each field (key), such that the order of Key 1 prevails over the order of Key 2, and so on. The *global sort specs*, located in the first 12 bytes of the file starting at offset 512, contain this information.

A 16-bit Turbo C bit field structure exists for each of the five possible sort keys, and Figure 3 shows how it's arranged. A sixth structure in the file contains the sort spec terminator, which sets the first two bit fields to binary 1s. A terminator also appears after the last valid sort spec. When sorting on two keys, the first two structures show valid sort specs, a terminator appears in the third and sixth structures, and the fourth and fifth contain garbage.

Precedence is indicated by the order in which fields appear in the specs. If Key 1 is field 4 and Key 2 is field 2, then the first record's **fieldID** contains 4 and the second contains 2. The third is a terminator. The **fieldID** component, then, is an index to the appropriate entry in the field directory table.

Up to this point, all the nodes of the file have been of fixed length, which makes for tidy programming because you know with certainty where things are. Not so from now on. Enter the variable-length record, which makes the programmer's life more interesting and guarantees job security to the venerable pointer.

**Field Names.** The *field directory table* consists of three variable-length nodes. Each begins with a word indicating how many bytes the node contains, followed immediately by its contents. These nodes are, in order, an index directory, the name pool, and the field descriptor table.

The *index directory* contains an offset to the field name string

*continued on page 62*

## REFLEX DATABASES

*continued from page 61*

within the pool. Since the field descriptors (described later) contain the same information, the index directory is redundant. Consequently, as you'll see in the SHOWRXD.C program later, it's unnecessary to remember the location of this index; simply use the node length to skip past the index to the actual name pool.

The *field name pool* contains ASCIIZ (null-terminated text) strings giving the names of the Reflex data fields. The maximum allowable length of a field name is 73 bytes plus the null terminator. Consistent with the overall architecture of this section of the database, the first word of the pool indicates how many bytes the pool occupies, including null terminators.

Within Reflex, a data field is identified by a relative number starting at 0 and working upward towards the total number of fields in the database. The names within the pool appear in field order, as do the field descriptors.

When retrieving data from the file, the text name of the field is probably irrelevant and certainly less important than knowing the field number, but it's vital when examining the database structure.

**Field Descriptors.** The *field descriptor table* is an array of fixed-length structures describing each field in reference-number order. While each descriptor occupies a known length of 16 bytes, the number of fields varies from one file to another. As a result the overall size of the table is variable. That's why the first word of the table gives its length, which is calculated as

```
n * sizeof (FLDDESC)
```

where *n* equals the number of fields in the database. You can work the length expression backwards (divide the table length by the field descriptor size) to find out how many fields there are, as in:

```
fread (&tablen, sizeof(int), 1, db);
nflds = tablen / sizeof (FLDDESC);
```

Figure 4 lists the format of each field descriptor within the table.

```
typedef struct {                                /* text table master structure */
  HANDLE    index;                                          /* pointer to index */
  HANDLE    pool;                                       /* pointer to text pool */
} ETREC;

typedef struct {                                      /* field descriptor record */
  unsigned nameOffset;                  /* offset of field name in name pool */
  char     dataType;                                            /* field type */
  char     precision;                          /* decimal precision = 5 bits,
                                                        field formats = 3 bits */
  unsigned fldOffset;                              /* offset of field in record */
  ETREC    etr;                                         /* repeating text info */
  unsigned isDescend : 1;                                     /* descending flag */
  unsigned sortPos : 7;                                       /* sort precedence */
  char reserved;                                                     /* not used */
} FLDDESC;
```

*Figure 4. Field descriptor master structure in Turbo C.*

Note that this structure includes the substructure **ETREC**, which applies to enumerated or "repeating text" fields.

Within the field descriptor, the **nameOffset** field gives an index to the name pool pinpointing the start of the ASCIIZ string that contains the field name. It is this field that makes the name pool index redundant, as discussed earlier.

The **dataType** member of the **FLDDESC** structure is an eight-bit integer indicating the data type of the field. Its permissible values are listed in Table 2.

| VALUE | TYPE | REMARKS |
|---|---|---|
| 0 | Untyped | Field type not yet determined |
| 1 | Text | Stored in record |
| 2 | Enum text | Offset into enumerated text pool |
| 3 | Date | 16-bit Julian date |
| 4 | Numeric | 64-bit IEEE (Turbo C) double |
| 5 | Integer | 16-bit signed integer |

*Table 2. Data types for the FLDDESC structure.*

The **precision** component is actually a bit field, but it cannot be defined as such because Turbo C automatically allocates 16 bits to bit fields, while **precision** is an eight-bit element. Consequently, "bit-fiddling" is necessary to derive values from this element. The first five bits indicate the decimal precision for floating point formats, while the remaining three define the field format.

In Reflex, *precision* refers to the number of digits following the decimal point in a floating point number. Reflex has options with regard to floating point output, in the format XXX.YYY...where the Xs are some number of significant digits and the Ys are fractional values. Except for the general numeric format, **precision** says how many Ys can appear in Reflex output. This has no bearing on the internal file-retained value of floating point numbers, which are 64-bit IEEE-standard formats compatible with Turbo C's double type. Legal precision values are 0 through 15.

The **format** component (low three bits) of the precision field refers to the display format for a field. The meaning depends on the field's data type. Table 3 lists values for dates as well as other numeric data.

*Date Fields*

| VALUE | MEANING |
|---|---|
| 0 | Default MM/DD/YY |
| 1 | MM/DD/YY |
| 2 | MM/YY |
| 3 | DD-Mon-YY |
| 4 | Mon-YY |
| 5 | Month DD, YYYY |

*Numeric Fields*

| VALUE | FORMAT | DISPLAY AS |
|---|---|---|
| 0 | General | (See 3 below) |
| 1 | Fixed | −XXX.YY |
| 2 | Scientific | −X.YYYe+ZZ |
| 3 | General | Fixed or Scientific for width |
| 4 | Currency | ($X,XXX.YY) |
| 5 | Financial | (X,XXX.YY) |

*Table 3. Format component for the display field.*

The **fldOffset** member gives the byte offset to the field within the data record, which we will discuss in more detail presently. For now, you should know that a data record contains a four-byte header followed by a variable number of elements actually containing data. The **fldOffset** is then calculated as four plus the sum of the sizes of all preceding fields within the data record. You can ignore the **isDescend** and **sortPos** elements, which merely confirm the global sort specifications discussed earlier.

Three words appear after the last field descriptor and before the enumerated text fields. Pay no attention to them; they are for internal Reflex use and contain the values 0x0013, 0x0001, and 0x0000. We mention them here only for those who would reverse the process described in this article and write a Reflex file from a Turbo C application.

Now let's talk about enumerated text fields.

## REPEATING TEXT

Reflex is unusual among database management packages in that it lets you define text fields that contain a predefined data set. An example might be a personnel application, in which everyone belongs to some department: Marketing, Engineering, Sales, Accounting, etc. You could set up these selections in advance and then pick from among them using the F10 (choices) key as you create a new record. Reflex refers to them externally as *Repeating Text* (RT) and internally as *Enumerated Text* (ET) fields. They get special treatment in a Reflex database.

ET data are stored in reverse order by field, after the three constants following the field descriptors. For example, if fields 3 and 7 contain Repeating Text according to the /RF display in Reflex, then the ET selections appear as field 7, followed by field 3 in the database file. Since each such field contains its own set of selections, a separate text pool exists for every ET field. You can locate these nodes using the **ETREC** pointers in the field descriptors.

Consistent with the spirit of Reflex's variable-length records, an ET node consists of:

- An integer showing the length of the index.
- A variable-length index of integers giving the offset of each repeating text string within the following pool, relative to the start of the pool. For example, if the pool contains the entries

```
Marketing\0    (length 10)
Engineering\0  (length 12)
Accounting\0   (length 11)
```

  then the index length is 6 because each of the three items is two bytes long. The contents of the index are 0, 11, and 24.
- An integer giving the length of the ET pool itself plus three bytes for each entry (this is required by Reflex). The total length in this case is (10 + 3) + (12 + 3) + (11 + 3) = 42 bytes.
- The text data comprising the ET pool.

When there are no repeating text fields in a database (which is most common), no space is set aside for them.

The major section following the enumerated text field selections is the database master record.

## MASTER RECORD

Despite its exalted name, the *master record* is the simplest structure in the entire Reflex database. It contains two integers, and can be found by using the **dfAddr** field in the section descriptor (**DFDESC** structure) whose **dfType** field contains nine integers.

The first field in the master record shows the total number of data records in the file. The second, which is of interest only to Reflex itself, indicates the number of records that have passed the most recently applied filter. Figure 5 lists a C structure describing the master record.

## DATA RECORDS: HOW MUCH CONTROL INFORMATION IS THERE?

There are a couple of ways to determine the amount of control information. The easiest is to treat the unused 326 bytes following the section descriptors as part of the control information. Thus, the header record occupies a fixed 512 bytes, the sort specs a fixed 12 bytes, and everything beyond that is variable.

The second way is to find the total size of the control information by inspecting the **dfAddr** field in the data records' section descriptor (**dfType** = 1), which is in the third descriptor record. This tells where the data records begin as an offset from the start of the file and thus accurately reflects the size of all the control information.

In the current releases of Reflex, this value is a long, located at offset 5CH (decimal 92) within the file. You can use **fseek()** or **lseek()**, depending on which Turbo C file access method you select, to move the file pointer, then read it into a variable that will be used to allocate the required space. After moving the pointer, don't forget to do two things: recast the long read from the file into an int or unsigned for the call to **malloc()**; and reset the file pointer to the start of the file before attempting another read.

There's a simple way to make all this control information instantly accessible to the program: put it on the heap. Calculate the size of the node as described, allocate the space using **malloc()**, and read that many bytes from the file into the node. When you finish doing this,

*continued on page 64*

```
typedef struct {                          /* data base master record */
  unsigned totalRecs;            /* total number of records in file */
  unsigned filtRecs;    /* # recs passing most recent global filter */
} MASTREC;
```

*Figure 5. Reflex master record structure in Turbo C.*

LISTING 1: REFLEX.H

```
/* reflex.h: structure definitions for Reflex data bases */

#define HANDLE   void far *                                /* 32-bit pointer */
#define RXID     "3Q.!&@#$!&&"                          /* Reflex identifier */

typedef struct {                              /* data file section descriptor */
  int  dfType;                                          /* section tytpe code */
  long dfAddr;                                /* start addr in file (bytes) */
  long dfLen;                                              /* length (bytes) */
} DFDESC;

typedef struct {                                     /* file header structure */
  int    hdrSz;                                /* size of file header = 512 */
  char   stamp[12];                      /* identification = "3Q.!&@#$!&&" */
  int    dirty;                               /* non-zero implies corrupt file */
  int    verViews;                                 /* view info version level */
  int    verModels;                          /* modeling system version level */
  int    verData;                                   /* raw data version level */
  int    fRecalc;                         /* non-zero implies recalc necessary */
  char   screenType;                    /* active screen type at file creation */
  char   checkSum;                                          /* file checksum */
  char   reserved[38];                             /* reserved for future use */
  int    sectionCt;                       /* number of sections of type DFDESC */
  DFDESC dfSection[];                                  /* section descriptors */
} DFHDR;

typedef struct {                                           /* field sort spec */
  unsigned  fldType     : 7;                     /* used internally by Reflex */
  unsigned isAscending  : 1;                                 /* sorting order */
  unsigned  fieldID     : 8;           /* field ID: index to field defin */
} FLDSORTSPEC;

typedef FLDSORTSPEC SORTSPEC[6];                           /* sort specs array */

typedef struct {                                 /* text table master structure */
  HANDLE    index;                                        /* pointer to index */
  HANDLE    pool;                                     /* pointer to text pool */
} ETREC;

typedef struct {                                   /* field descriptor record */
  unsigned nameOffset;                   /* offset of field name in name pool */
  char     dataType;                                             /* field type */
  char     precision;                       /* decimal precision = 5 bits,
                                                        field formats = 3 bits */
  unsigned fldOffset;                           /* offset of field in record */
  ETREC    etr;                                        /* repeating text info */
  unsigned isDescend : 1;                                   /* descending flag */
  unsigned sortPos : 7;                                     /* sort precedence */
  unsigned reserved : 8;                                           /* not used */
} FLDDESC;

typedef struct {                                   /* data base master record */
  unsigned totalRecs;                       /* total number of records in file */
  unsigned filtRecs;       /* # recs passing most recent global filter */
} MASTREC;
```

# REFLEX DATABASES

*continued from page 63*

the file pointer has advanced to the start of the data records themselves, and all the control information is on the heap where the program can address it directly. The only thing that still needs doing is to initialize pointers to the various elements of the node.

## SETTING UP THE POINTERS

The control information for a Reflex database has seven major entry points that must be assigned pointers, plus a number of structure types that have already been discussed. It's also useful to have a couple of work variables (**work** and **temp**) for performing pointer arithmetic. The **#include** file REFLEX.H, shown in Listing 1, defines the structures and declares all the global variables necessary to process Reflex database control information.

It takes a lot of pointer arithmetic to initialize the seven pointers to the control structure elements.

The **#include** file INPTRRXD.I in Listing 2 shows the processes for initializing the pointers. Note that this is an inline file; simply place the directive

```
#include <inptrrxd.i>
```

in your source file wherever you want to initialize the control structure pointers. Before invoking it, you must satisfy the following conditions:

- Include REFLEX.H
- Allocate a node on the heap for the control information
- Read the control information into the node
- Set variables **base** and **head** to point to the start of the node

The algorithms in INPTRRXD.I set the requisite global pointers; you can then use them in your application.

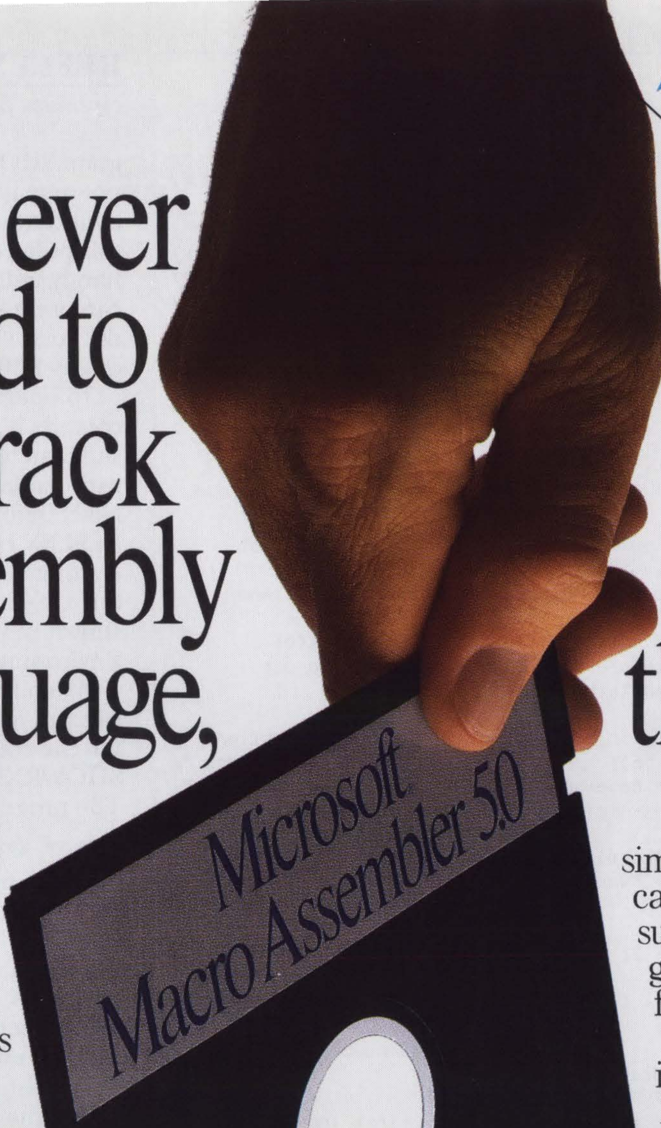
## DOCUMENTING A REFLEX DATABASE

Reflex doesn't include a utility for printing out the characteristics of a database. Consequently the pro-

*continued on page 66*

Works with Turbo C.

# If you ever wanted to take a crack at assembly language, now's the time.



You probably already know that assembly language subroutines are the smartest way to get the fastest programs.

But if the complexities of working in assembler made you think twice, here's some good news. We've made Microsoft® Macro Assembler Version 5.0 a lot easier to use.

We eased the learning process by giving you the best support around. We completely revised our documentation. The new Mixed Language Programming Guide gives you step by step instructions for linking your assembly code with Microsoft QuickBASIC, C, FORTRAN, Pascal and other languages. And you get a comprehensive reference manual with listings of the instruction set and examples of each instruction. We didn't stop there, though. You also get an on-disk collection of templates and examples.

We've also dramatically simplified the high-level language interface. In just a few simple steps, you can be calling Macro Assembler subroutines from programs written in your favorite language.

Now that you're writing the fastest programs, Microsoft is giving you the fastest way to debug them. For the first time, we've added our CodeView® debugger to Macro Assembler.

With source code and comments on your screen, Microsoft CodeView makes debugging programs containing assembly language subroutines a snap.

And you'll be glad to know that you don't sacrifice any speed for all the ease of use.

We took the fastest Macro Assembler on the market and made it even faster.

So what are you waiting for? Get your hands on Microsoft Macro Assembler and see what it's like to break your personal speed limit.

**Microsoft**®

For more information or for the name of your nearest Microsoft dealer, call (800) 426-9400. In Washington State and Alaska, (206) 882-8088. In Canada, call (416) 673-7638.

Microsoft, the Microsoft logo and CodeView are registered trademarks of Microsoft Corporation.

```
/* GLOBALS */
FILE            *lst;                                          /* printer */
unsigned        nflds;                                /* # field descriptors */
unsigned        base, temp;  /* point of reference for control info */
int             *work;                   /* for getting lengths from file */
DFHDR           *head;                                   /* header record */
DFDESC          *descr;                       /* section descriptor table */
SORTSPEC        *sort;                         /* global sort specs table */
char            *pool;                                 /* field name pool */
FLDDESC         *dtable;                        /* field descriptor table */
int             *etpool;                           /* enumerated text pool */
MASTREC         *mast;                                   /* master record */
```

LISTING 2: INPTRRXD.I

```
/* inptrrxd.i: initializes pointers to Reflex data base control
   info entry points.
     Assumes reflex.h has been #included and globals 'base' and
   'head' have been initialized for stack operation.
     This file is #included inline after loading the control
   info onto the heap and before the application tries to use
   the control info to do anything */

                          /* ---- initialize pointers to fixed sections */
  descr = (DFDESC*)(base + 66);                    /* section descriptors */
  sort = (SORTSPEC*)(base + head->hdrSz);                   /* sort specs */
                          /* ---- initialize ptrs to variable sections */
                                                       /* field name pool */
  work = (int*)((unsigned)(sort) + 11);
  temp = (unsigned)(work) + *work + 4;
  pool = (char*) temp;
                                                /* field descriptor table */
  temp = (unsigned)(pool) - 2;
  work = (int*)(temp);
  temp += (unsigned)(*work + 4);
  dtable = (FLDDESC*) temp;
                                            /* calculate number of fields */
  work = (int*)(unsigned)(dtable) - 1;
  nflds = *work / sizeof (FLDDESC);
              /* enumerated text pool (point to first index length */
  temp += (unsigned) *work + 8;
  etpool = (int*) temp;
                                                         /* master record */
  mast = (MASTREC*)(unsigned)((descr+1)->dfAddr + base - 1);
```

## REFLEX DATABASES

*continued from page 64*

gram SHOWRXD.C in Listing 3 not only illustrates a practical application of the discussion up to this point, but also provides a handy utility for Reflex users. You can use it to create a hardcopy document explaining the structure of any Reflex database.

To operate SHOWRXD.C, first compile it with Turbo C in the small memory model, making the executable file SHOWRXD.EXE. Note: before doing this, place REFLEX.H and INPTRRXD.I (Listings 1 and 2, respectively) in the \INCLUDE subdirectory shown in your Turbo C **O**ptions/**E**nvironment menu.

When the .EXE file exists, you can run it for any Reflex database. On the command line, type SHOWRXD and hit the Enter key. The program asks:

```
Name of Reflex database?
```

Type the filename, preceded by the drive and path if necessary. If you omit the .RXD filename extension associated with Reflex databases, the program automatically appends it using Turbo C's **fnsplit()** and **fnmerge()** functions.

The listing in Figure 6 shows the output of SHOWRXD.C for Reflex database CUSTLIST.RXD, a sample file included on the Reflex package distribution disk.

### FETCHING DATA FROM A REFLEX FILE

Unlike SHOWRXD.C (Listing 3), which reports the structure of any Reflex database irrespective of its actual contents, fetching data is application-dependent. That is, you need to know in advance which database you want to process, and which field(s) you want to read.

The reasons are clear based on the preceding discussion: The structure of a Reflex database is variable, and the contents of a given field in one database might be very different from those in the same position in another database.

As an example, Figure 7 contrasts two of the Reflex sample files: MAILLIST.RXD and CUSTLIST.RXD. You can't use the

```
Control information for Reflex
data base CUSTLIST.RXD:
  Total records      35
  Filtered records   35
  Number of fields   8

Header record contents:
  Size of header        512
  Reflex identifier     3Q.!&@#$!&&
  Corruption indicator  Clean
  View version level    7
  Modeling version lev. 4
  Raw data version lev. 3
  Recalc necessary      No
  Screen type           IBM CGA
  File checksum         25
  Reserved field        (38 bytes)
  Section descriptors   12

Global sort specifications by
precedence:
  Descending:  Field Address

8 Field Descriptors:
  Field name:      Date
  Data type:       Date
  Format:          MM/DD/YY
  Field offset     4

  Field name:      Rep
  Data type:       Text
  Field offset     6

  Field name:      Name
  Data type:       Text
  Field offset     8

  Field name:      Address
  Data type:       Text
  Field offset     10

  Field name:      City
  Data type:       Text
  Field offset     12

  Field name:      State
  Data type:       Text
  Field offset     14

  Field name:      Zip
  Data type:       Text
  Field offset     16

  Field name:      Total Sales
  Data type:       Numeric
  Precision        2
  Format           Fixed (-XXX.YY)
  Field offset     18
```

*Figure 6. Sample output of SHOWRXD, which documents the CUSTLIST database.*

same application program to pull mailing information from both databases for two reasons:

1. The corresponding fields are in different positions (i.e., Name is field 0 in MAILLIST and field 2 in CUSTLIST).
2. The corresponding fields are of different data types (i.e., **ZIP**

*continued on page 68*

LISTING 3: SHOWRXD.C

```
/* showrxd.c: Displays fixed information about Reflex data bases */
/* Written for small model of Turbo C by K. Porter */

/* INCLUDE FILES */
#include <stdio.h>
#include <string.h>
#include <dir.h>
#include <reflex.h>       /* Separate Reflex structure definitions */

/* DEFINE CONSTANTS */
#define  OUTDEV    "PRN"                           /* output device */
#define  EJECT     12                         /* printer page eject */

/* LOCAL FUNCTION PROTOTYPES */
void     showMast (char name[], MASTREC *mast);
void     showHead (DFHDR *head);
void     showSort (SORTSPEC *srt);
void     showName (unsigned offset);
void     showField (unsigned nf, FLDDESC *fld);

main ()
{
char           fname[MAXPATH],                          /* filename */
               drive[MAXDRIVE], dir[MAXDIR],          /* components */
               name[MAXFILE], ext[MAXEXT];
long           fpos;                               /* file position */
unsigned       d;                                  /* misc variable */
FILE           *db;                                /* database file */

/* OPEN FILES */
  d = 0;
  lst = fopen ( OUTDEV, "w" );                       /* open output */
  cputs ( "\nName of Reflex data base? " );
  gets ( fname );                                   /* get filename */
  fnsplit ( fname, drive, dir, name, ext );       /* split filename */
  if ( !strlen ( ext ))
     strcpy ( ext, ".RXD" );     /* if no extension, make it ".RXD" */
  fnmerge ( fname, drive, dir, name, ext );       /* and reassemble */
  db = fopen ( fname, "r" );
  if ( db != NULL ) {
    setvbuf ( db, NULL, _IONBF, 0 );        /* make file unbuffered */
           /* ---- verify that file is open and a Reflex data base */
    fseek ( db, 88L, SEEK_SET );  /* point to size of control info */
    fread ( &fpos, sizeof (long), 1, db );                /* get it */
    fseek ( db, 0L, SEEK_SET );                 /* repoint to start */
                               /* ---- put control info on the heap */
    base = (unsigned) malloc ((unsigned) fpos);    /* allocate node */
    head = (DFHDR*) base;                     /* set header pointer */
    d = fread (((char*)(base)), sizeof (char), ((int)(fpos)), db);
  }
  if ( db == NULL || d == 0 ) {                    /* error handler */
    printf ( "Error accessing file %s\n", fname );
    printf ( "File is %s open, items read = %u\n",
        (db==NULL ? "not" : ""), d);
    exit (1);
  } else
    if ( strcmp ( head->stamp, RXID ) != 0 ) {
      printf ( "\n\nFile %s is not a Reflex data base\n", fname );
      exit (1);                         /* exit with condition code */
    }
/* INITIALIZE POINTERS TO CONTROL INFO */
#include <inptrrxd.i>

/* SHOW INFORMATION ABOUT DATA BASE */
  showMast (fname, mast);                    /* show master record */
  showHead (head);                           /* list header record */
  showSort (sort);                       /* show global sort specs */
  showField (nflds, dtable);             /* show field descriptors */

/* END OF JOB */
  putc ( EJECT, lst );                               /* eject page */
  close ( lst );                                  /* close printer */
  free ( head );                         /* deallocate heap space */
} /* --------------- End of main() ---------------------------- */

void  showMast (char name[], MASTREC *mast)  /* show master record */
{
  fprintf ( lst, "Control information for Reflex data base %s:\n",
      name );
  fprintf ( lst, "  Total records      %d\n",
      mast->totalRecs );
  fprintf ( lst, "  Filtered records   %d\n",
      mast->filtRecs );
  fprintf ( lst, "  Number of fields   %d\n", nflds );
} /* ----------------------- */

void  showHead (DFHDR *head)                 /* list header record */
{
  fputs ( "\nHeader record contents:\n", lst );
  fprintf ( lst, "  Size of header            %d\n", head->hdrSz );
  fprintf ( lst, "  Reflex identifier         %s\n", head->stamp );
  fprintf ( lst, "  Corruption indicator      %s\n",
      (head->dirty) ? "Corrupt" : "Clean" );
  fprintf ( lst, "  View version level        %d\n", head->verViews );
  fprintf ( lst, "  Modeling version level    %d\n", head->verModels );
  fprintf ( lst, "  Raw data version level    %d\n", head->verData );
  fprintf ( lst, "  Recalc necessary          %s\n",
      (head->fRecalc) ? "Yes" : "No" );
  fputs ( "  Screen type               ", lst );
  switch ( head->screenType ) {
    case 0: fputs ( "IBM CGA", lst ); break;
    case 1: fputs ( "Hercules", lst ); break;
    case 2: fputs ( "IBM 3270 PC APA", lst ); break;
    case 3: fputs ( "IBM EGA", lst ); break;
    case 4: fputs ( "IBM PGC", lst ); break;
    case 5: fputs ( "AT&T 6300 Series", lst ); break;
    case 6: fputs ( "Sigma 400", lst ); break;
    case 7: fputs ( "STB SuperRes 400", lst ); break;
    default: fputs( "(Unknown)", lst );
  }
  putc ( '\n', lst );
  fprintf ( lst, "  File checksum             %X\n", head->checkSum );
  fputs ( "  Reserved field         (38 unused bytes)\n", lst );
  fprintf ( lst, "  Section descriptors       %d\n", head->sectionCt );
} /* ----------------------- */
```

*continued from page 67*

```
Maillist:
Fld# Name     Type
---- -------- ----
  0  Name     Text
  1  Address  Text
  2  City     Text
  3  State    Text
  4  ZIP      Numeric

Custlist:
Fld# Name     Type
---- -------- ----
  0  Date     Text
  1  Rep      Text
  2  Name     Text
  3  Address  Text
  4  City     Text
  5  State    Text
  6  ZIP      Text
  7  Sales    Numeric
```

*Figure 7. Structures of two dissimilar Reflex files.*

is numeric in MAILLIST and text in CUSTLIST).

Consequently, you have to write programs to process specific databases, and the only variability is among Reflex databases that have identical field attributes within the domain of the application. That is, if two files are identical within the first *n* fields and the application only processes those *n* fields (or some subset of them), then the database is sufficiently generalized to process both files.

That's a hair-splitting distinction. The general rule is that you need a separate program for extracting data from every Reflex file unless you specifically know otherwise.

First, set the file pointer to the data records. The section descriptor field **(descr + 2)−>dfAddr** gives the offset of the data records from the start of the file. The records section itself begins with an unsigned word used internally by Reflex, with the first data record following immediately. Thus, use **fseek()** to point into the file at the location given by:

```
(descr+2)->dfAddr + 2
```

A data record consists of four sections: a word indicating the length of the record (not counting itself), a fixed-format record header structure, a fixed-length data section, and a variable-length text pool.

The four-byte record header

contains mostly reserved Reflex information. Only the fourth byte is potentially meaningful; it shows how many fields in the record actually contain data. Ordinarily you can simply skip the header and go directly to the fixed-length data section.

The term *fixed length* is somewhat misleading. The data section's length is fixed for each record within a given file, but its length varies from one file to another based on the number of fields per record.

Every field is 16 bits long except for untyped fields and numeric (floating point), which is a 64-bit IEEE-standard format compatible with Turbo C's double type. All fields, except untyped fields, have two special values representing null and error. A null value shows up as a blank cell on the Reflex display, while an error value causes Reflex to display the word ERROR in the cell. The format specifications for each type are listed in Table 4.

| TYPE | REPRESENTATION |
|---|---|
| 0 (untyped) | No data stored |
| 1 (text) | 16-bit unsigned giving offset into text pool at end of record, measuring from start of record header.<br>Null: 0<br>Error: 1 |
| 2 (ET) | 16-bit unsigned giving offset into ET pool in field directory.<br>Null: 0<br>Error: 1 |
| 3 (date) | 16-bit unsigned: number of days since 12/31/1899<br>Null: 0<br>Error: 0xFFFF |
| 4 (numeric) | 64-bit IEEE floating point<br>Null: MSW = 0x7FFF<br>Error: MSW = 0x7FF0 |
| 5 (integer) | 16-bit signed integer<br>Null: 0x8000<br>Error: 0x8001 |

*Table 4. Format for fixed-length data in a record.*

Thus, each field's position within the data section is offset from the start of the header record by four bytes plus the aggregate length of all preceding fields (see Table 5).

*continued on page 70*

```
void  showSort (SORTSPEC *srt)            /* show global sort specs */
{
unsigned  n, p;
FLDSORTSPEC  *spec;
FLDDESC      *fld;

  fputs ( "\nGlobal sort specifications by precedence:\n", lst );
  for ( n = 0; n < 6; n++ ) {
    spec = srt + n;                        /* point to next sort spec */
    if ( spec->fldType == 0x7F && spec->isAscending ) {
      if ( spec == srt )
        fputs ( "  (None)\n", lst );
      break;                             /* quit on terminator (0xFF) */
    } else
        if ( spec->fieldID <= nflds ) {                  /* show order */
          fprintf ( lst, "  %s  Field ", (spec->isAscending)
              ? "Ascending: " : "Descending:" );

                      /* Note: We have to follow a chain of references
                         to find the field name. spec->fieldID gives
                         the field descriptor record #, whose first
                         field is an offset into the fieldname pool,
                         from which we can print the name */

          fld = dtable + spec->fieldID;    /* point to field descr */
          p = fld->nameOffset;            /* get name offset in pool */
          showName ( p );                                /* print it */
        }
  }
} /* ------------------------ */

void  showName (unsigned offset)   /* print pool field name to lst */
{
char   *name;

  name = (char *) pool + offset;          /* node address of string */
  fputs ( name, lst );
  putc ( '\n', lst );
} /* ------------------------ */

void  showField (unsigned nf, FLDDESC *fld)
{                                          /* list field descriptors */
FLDDESC   *f;
unsigned  n;

  fprintf ( lst, "\n%u Field Descriptors:", nf );
  for ( n = 0; n < nf; n++ ) {      /* loop thru table for nf items */
    f = fld + n;                       /* point to next descriptor */
    fputs ( "\n  Field name:      ", lst );
    showName ( f->nameOffset );                   /* list field name */
    fputs ( "  Data type:       ", lst );
    switch ( f->dataType ) {                       /* show data type */
      case 0: fputs ( "Untyped\n", lst ); break;
      case 1: fputs ( "Text\n", lst ); break;
      case 2: fputs ( "Repeating text\n", lst ); break;
      case 3: fputs ( "Date\n", lst );
              fputs ( "  Format:          ", lst );
              switch ( f->precision & 0x07 ) {       /* date format */
                case 0:
                case 1: fputs ( "MM/DD/YY\n", lst ); break;
                case 2: fputs ( "MM/YY\n", lst ); break;
                case 3: fputs ( "DD-Mon-YY\n", lst ); break;
                case 4: fputs ( "Mon-YY\n", lst ); break;
                case 5: fputs ( "Month DD, YYYY\n", lst );
```

*continued from page 69*

| FIELD TYPE | OFFSET | LENGTH |
|---|---|---|
| date | 4 | 2 |
| untyped | 6 | 2 |
| numeric | 8 | 8 |
| text | 16 | 2 |

*Table 5. Field position within the data section of a record.*

For example, if you want to read only the numeric field from each record, first get the word length at the start of the record, then jump eight bytes and read the field into a variable of type **double**. Skip to the next record by adding the record length to the file pointer, then doing an **fseek()** on that result.

The data text pool is at the end of the record. It exists only when the record contains nonnull and nonerror text fields. If the user has typed "Mary Smith" into the only text field of the record, then Mary's name appears as an ASCIIZ string with the "M" in the first byte position after the end of the data section. The text field within the data section contains a number indicating how many bytes the "M" is offset from the start of the record header.

Note that there are no sequencing rules regarding the placement of data text strings in the pool. If a record has two text fields, the string for field=2 might precede that for field=1. Also, no string is referenced by more than one field. For every valid (nonnull, nonerror) text field, there is a unique ASCIIZ string in the variable-length pool, even if two or more have identical content.

### PUTTING IT TO WORK

Now let's turn talk into action with a program that reads the Reflex sample database CUSTLIST.RXD and prints a report showing total sales by customer. We call it CUSTLIST.C and it is in Listing 4.

The first part of the program is similar to SHOWRXD.C (Listing 3). The chief differences are that it declares fewer variables and functions, and it opens a constant filename.

```
            }
            break;
    case 4: fputs ( "Numeric\n", lst );
            fprintf ( lst, "  Precision        %d\n",
                (( f->precision & 0xF8 ) >> 3 ));
            fputs ( "  Format             ", lst );
            switch ( f->precision & 0x07 ) {   /* numeric format */
              case 0: fputs ( "General\n", lst ); break;
              case 1: fputs ( "Fixed (-XXX.YY)\n", lst ); break;
              case 2: fputs ( "Scientific (-X.YYe+ZZ)\n", lst );
                      break;
              case 3: fputs ( "General\n", lst ); break;
              case 4: fputs ( "Currency ($X,XXX.YY)\n", lst );
                      break;
              case 5: fputs ( "Financial (X,XXX.YY)\n", lst );
            }
            break;
    case 5: fputs ( "Integer\n", lst ); break;
  }
  fprintf ( lst, "  Field offset     %u\n", f->fldOffset );
 }
} /* ----------------------- */
```

LISTING 4: CUSTLIST.C

```
/* custlist.c: reports name and total sales from custlist.rxd */
/* Written for small model of Turbo C by K. Porter */

/* INCLUDE FILES */
#include <stdio.h>
#include <string.h>
#include <dir.h>
#include <reflex.h>       /* Separate Reflex structure definitions */

/* DEFINE CONSTANTS */
#define  outdev    "PRN"                /* report output device */
#define  dbname    "CUSTLIST.RXD"      /* Reflex file to process */
#define  EJECT     12                     /* printer page eject */

/* LOCAL FUNCTION PROTOTYPE */
void   report ( void );

/* GLOBAL INPUT FILE */
FILE           *db;

main ()
{
long           fpos;                            /* file position */
unsigned       d;                               /* misc variable */

/* OPEN FILES */
  d = 0;
  lst = fopen ( outdev, "w" );                    /* open output */
  cputs ( "\nGenerating sales report from Reflex data base:\n" );
  db = fopen ( dbname, "r" );
  if ( db != NULL ) {
    setvbuf ( db, NULL, _IONBF, 0 );      /* make file unbuffered */
          /* ---- verify that file is open and a Reflex data base */
    fseek ( db, 88L, SEEK_SET );  /* point to size of control info */
    fread ( &fpos, sizeof (long), 1, db );               /* get it */
    fseek ( db, 0L, SEEK_SET );                /* repoint to start */
                            /* ---- put control info on the heap */
    base = (unsigned) malloc ((unsigned) fpos);   /* allocate node */
    head = (DFHDR*) base;                    /* set header pointer */
    d = fread (((char*)(base)), sizeof (char), ((int)(fpos)), db);
```

Once the program has initialized the control pointers as described earlier, it calls the **report()** function. This subprogram implements the discussion in the preceding section.

In particular, note the manipulation of the **fptr** variable, which serves as a point of reference for the start of the record header. During each iteration of the loop—which repeats for every record—**fptr** is first set to point at the record length. After performing **fseek()** and reading the length, **fptr** advances two bytes so that it points to the start of the record header. All subsequent pointer arithmetic offsets from the value of **fptr**. The last instruction in the loop advances **fptr** by the current record length, so that it moves to the next record and repeats the process.

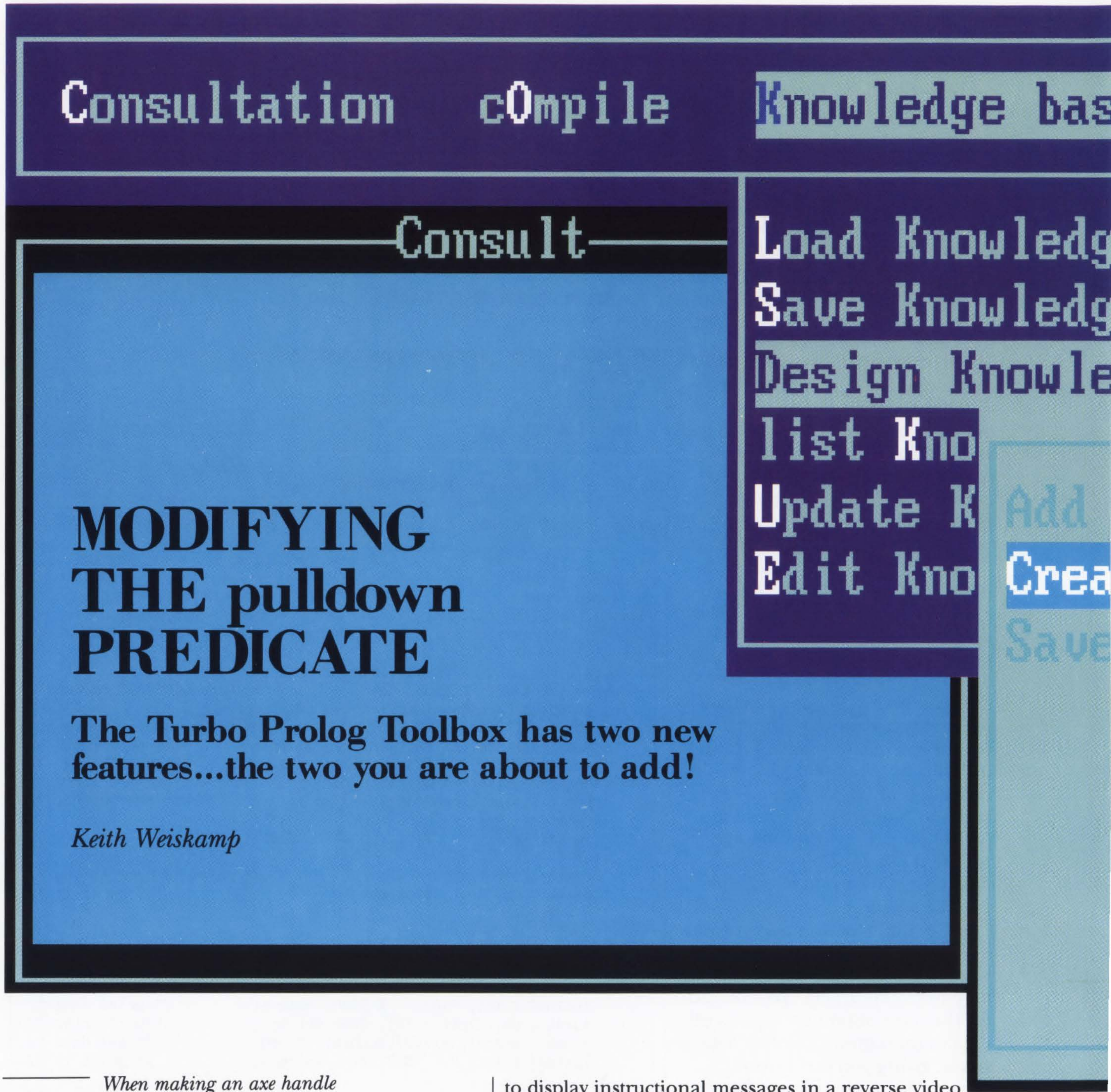
To run the program after compiling, just type CUSTLIST. (It expects to find CUSTLIST.RXD in the current directory.) The program then prints out a report with 35 line items (records) showing the customer name and the total sales. If you want to redirect the output to some other medium (a disk file, for example), change the definition of **OUTDEV** and recompile. Similarly, you can add pathname information to the definition of **DBNAME** when the database is known to exist in a specific subdirectory.

It's not difficult to find your way around a Reflex database once you know how the pieces fit together. This article takes the lid off the most important aspects of data organization in Reflex databases, giving you the Turbo C tools to document files and extract data from them. ■

*Kent Porter is a professional writer specializing in software. His latest book,* Stretching Turbo Pascal *(Simon & Schuster/Brady), is written for experienced Turbo Pascal programmers. He's now working on a similar book for Turbo C users that will appear next spring.*

*Listings may be downloaded from CompuServe as READRX.ARC.*

```
  }
  if ( db == NULL || d == 0 ) {                   /* error handler */
    printf ( "Error accessing file %s\n", dbname );
    printf ( "File is %s open, items read = %u\n",
        (db==NULL ? "not" : ""), d);
    exit (1);
  } else
    if ( strcmp ( head->stamp, RXID ) != 0 ) {
      printf ( "\n\nFile %s is not a Reflex data base\n", dbname );
      exit (1);                        /* exit with condition code */
    }

/* INITIALIZE POINTERS TO CONTROL INFO */
#include <inptrrxd.i>

/* PRODUCE REPORT BY READING DATA BASE */
  report ();

/* END OF JOB */
  putc ( EJECT, lst );                               /* eject page */
  close ( lst );                                  /* close printer */
  free ( head );                          /* deallocate heap space */
} /* --------------- End of main() ----------------------------- */

void  report (void)                             /* generate report */
{
long     fptr, text;          /* file pointers: main and text pool */
double   sales;                                 /* total sales field */
unsigned recs;                       /* loop counter for # records */
int      reclen, tofs, n;            /* record length, text offset */
char     ch, name[80];        /* character, string for text output */

  fputs ( "     SALES REPORT FROM CUSTLIST.RXD:\n", lst );
  fptr = (descr+2)->dfAddr + 2;           /* start of first record */
  for ( recs = 0; recs < mast->totalRecs; recs++ ) {
    fseek ( db, fptr, SEEK_SET );          /* point to next record */
    fread ( &reclen, sizeof(int), 1, db );           /* get length */
    fptr += 2;                       /* advance to start of header */
    fseek ( db, (fptr + 8), SEEK_SET );      /* skip to name field */
    fread ( &tofs, sizeof(int), 1, db );       /* get text offset */
    if ( tofs < 2 )
      fprintf ( "\n     %s", (tofs) ? "ERROR" : "NULL");
    else {
      text = fptr + tofs;                  /* point to name string */
      fseek ( db, text, SEEK_SET );                    /* go to it */
      n = 0;
      do {
        fread ( &ch, sizeof(char), 1, db );       /* get next char */
        name[n++] = ch;
      } while ( ch );                           /* until null char */
      fprintf ( lst, "\n     %-30s", name );         /* print name */
      fseek ( db, (fptr + 18), SEEK_SET );       /* skip to sales */
      fread ( &sales, sizeof(double), 1, db );          /* get data */
      fprintf ( lst, "    %12.2fl", sales );            /* print it */
    }
    fptr += reclen;                       /* advance to next record */
  }
}
```



# MODIFYING THE pulldown PREDICATE

## The Turbo Prolog Toolbox has two new features...the two you are about to add!

*Keith Weiskamp*

PROGRAMMER

*When making an axe handle*
*the pattern is not far off.*
*... We'll shape the handle*
*By checking the handle*
*Of the axe we cut with—*
*Lu Ji Wan Fu (4th century A.D.)*

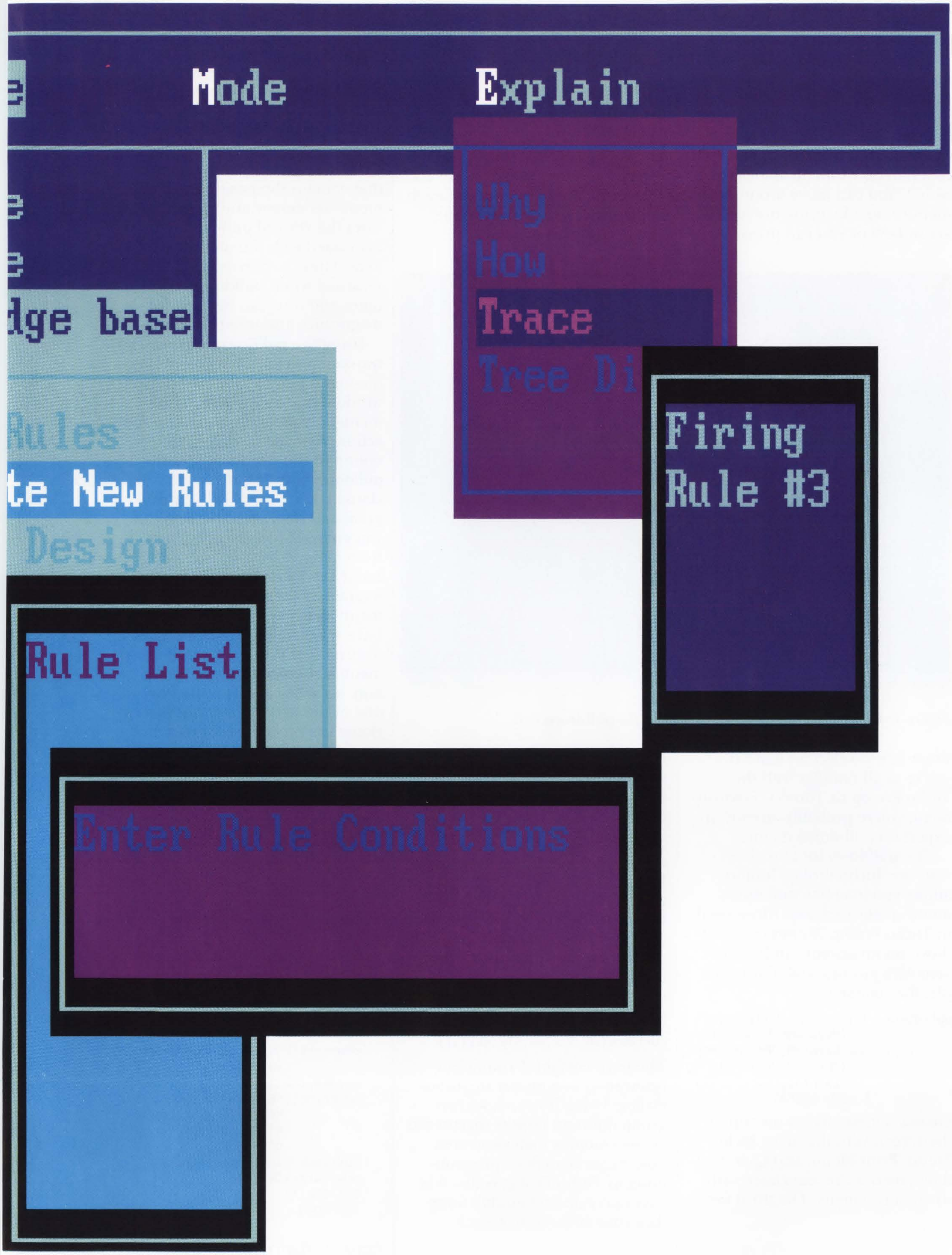
The Turbo Prolog Toolbox provides a treasure chest of tools for developing user interfaces. With the tools, you can add better user interface features, including pop-up, pull-down, and tree-type menus; status bars for messages; and context-sensitive help.

Particularly useful is **pulldown**, a predicate that allows you to create pull-down menus using a menu bar. In this article, I'll show you step-by-step how the **pulldown** tool works, and how you can modify it to add two enhancements: automatic update of status bar messages and a continuous scrolling feature for pull-down menus. The first enhancement allows you to display instructional messages in a reverse video status bar at the bottom of the screen as the user moves around the menu system. The other enhancement adds a continuous scrolling capability to pull-down menus. Thus, when you are at the beginning or end of a menu and attempt to move down or up, the reverse video menu selector wraps around instead of stopping. But, before we jump in and start dissecting and modifying the **pulldown** tool predicate, let's discuss the basic operations of pull-down menu systems and look at how the **pulldown** predicate is used.

### A TASTE OF PULL-DOWN MENUS

Pull-down menus consist of two components: a menu bar, which is usually displayed horizontally

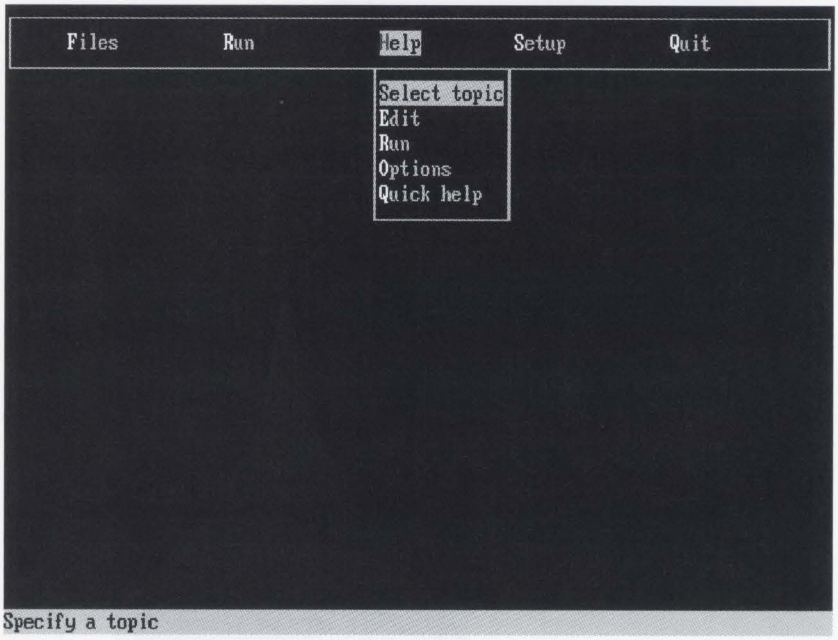
*continued on page 74*


e
Mode
Explain
e
e
dge base
Why
How
Trace
Tree Di
Rules
te New Rules
Design
Firing
Rule #3
Rule List
Enter Rule Conditions

## PULLDOWN PREDICATE

*continued from page 72*

across the top of the screen, and pull-down menus, which are displayed vertically under the menu bar (see Figure 1). Each entry in the menu bar has either an action or a pull-down menu associated with it. You can move around the menu system by using one of the arrow keys or you can press the Enter key to select an item. If you're at all familiar with the Turbo Prolog or Turbo C environment, you're probably already an expert on pull-down menus.



Specify a topic

*Figure 1. Sample pull-down menu created by the* **pulldown** *tool.*

The **pulldown** tool provided with the Turbo Prolog Toolbox allows you to create pull-down menus patterned after those used in Turbo Prolog. A complete pull-down menu system can be generated with just one call. For example, the statement

```
pulldown(7,[curtain(5,"Help Menu",
            ["System","Topic"]),
          curtain(20,"Print Menu",
            ["Draft","Bold"]),
          curtain(35,"Quit", [])
         ], CH, SUBCH).
```

creates a menu bar at the top of the screen with the items **H**elp Menu, **P**rint Menu, and **Q**uit. Pull-down menus are associated with the first two items. The third item, **Q**uit, does not have a pull-down menu associated with it since its menu list is empty.

The **pulldown** predicate takes four parameters:

```
pulldown(ATTRIBUTE,MENULIST,
CHOICE,SUBCHOICE)
```

The **ATTRIBUTE** defines the foreground and background colors that are to be used for each of the windows (menus) in the pull-down menu system. The **MENULIST** contains the list of strings for the pull-down menu bar and the strings for their associated pull-down menus. For example, the item

```
curtain(5,"Help Menu", ["System",
"Topic"])
```

creates an entry in the pull-down menu bar called **H**elp Menu at column 5, and defines its pull-down menu to contain the selections **S**ystem and **T**opic. Note that each entry in **MENULIST** is represented as a complex object:

```
curtain(COL,STRING,STRINGLIST)
```

The term **curtain** is commonly referred to as a *functor* in Turbo Prolog. Using functors, we can group different objects together to create complex data structures. One major benefit of programming in Turbo Prolog is that it is easy to create and modify these data structures. In fact, such structures can often be modified without forcing you to rewrite major sections of a program, which is often the case with programs written in procedural languages.

The last two arguments in **pulldown**, **CHOICE** and **SUBCHOICE**, are output parameters that contain the position of the menu-bar cursor and the selection from the vertical pull-down menu associated with the menu-bar item. These parameters are returned when **pulldown** terminates and thus can be used for diagnostics and other purposes.

Using the **pulldown** tool is a two-step process. First you define the menu bar and the pull-down windows for each entry in the menu bar. Second, you define the actions associated with each option in the windows. When **pulldown** is called, the main clause takes control of your program and interprets the keys that you enter. If you select an item from a menu by pressing the Enter key, one of two things may happen. If there is a pull-down menu associated with the item you have selected, then that menu is displayed. If there is no pull-down menu associated with the selection, an action, represented by one of the user-defined **pdwaction** clauses, is processed. Thus, when you use **pulldown**, you must utilize **pdwaction** to define an action for each possible menu selection.

### IMPLEMENTATION AND INTERNALS

Internally, the operation of the

```
pulldown(ATTR,LIST,SLIST,CH1,CH2):-
   makewindow(81,ATTR,ATTR,"",0,0,
              3,80),
   pdwlistlen(LIST,MAXCOL),
   writepdwlist(ATTR,LIST),
   pdwmovevert(0,0,ATTR,LIST),
   changepdwstate(pdwstate(0,0,up,
                              0,0)),
   repeat,
   pdwstate(ROW,COL,DOWN,MAXROW,
            LEN),
   readkey(KEY),
   pdwkeyact(KEY,ROW,COL,DOWN,
             MAXROW,MAXCOL,LEN,
             ATTR,LIST,SLIST,
             CONTINUE),
   CONTINUE=stop,removewindow,
   pdwstate(ROW1,COL1,_,_,_),!,
   CH1=COL1+1,
   CH2=ROW1.
```

*Figure 2. Main clause for* **pulldown**.

pull-down menu tool is similar to that of a **case** statement in a procedural language. Figure 2 shows the main **pulldown** clause.

The calls to **makewindow**, **pdwlistlen**, **writepdwlist**, and **pdmovevert** generate the pull-down menu bar or horizontal menu. This menu is created using Turbo Prolog's built-in **makewindow**, **field__attr**, and **scr__char** predicates. The value used for the window number is 81. Watch out! If you use **pulldown**, don't define any windows with this number.

In general, putting up menus takes some time: adjusting a lot of coordinates, writing strings to a window in different attributes, and creating a menu selection bar. Fortunately, to implement our status bar messages, we don't need to modify any of the code used for drawing and updating menus. If you decide you want to modify any of the internal menu attributes, now you know where to look.

The **changepdwstate** predicate updates the current state of the pull-down menu system. **changepdwstate** performs a **retract** and an **assert**, then stores the pull-down menu status in the dynamic database. The declaration of this **database** predicate is:

```
database
   pdwstate(ROW,COL,SYMBOL,ROW,COL)
```

Here the first two parameters contain the current cursor position (the row and column corresponding to one of the defined menus). The third parameter, **SYMBOL**, contains the current state of the menu system. Two different conditions are supported: "up" and "down." The "up" condition indicates that all of the pull-down menus are not displayed, and "down" indicates that a pull-down menu is currently displayed (active). The final two parameters contain the dimensions of a menu row and column to help control the movement through the actual pull-down menus.

The core of **pdwstate** is a short loop that gets the current state of the menu system, reads a key from the user, and performs

*continued on page 76*

LISTING 1: STATUS1.PRO

```
/* Listing1 */

/*  Pulldown window action corresponding to input key and
spulldown
    window  state */

  pdwkeyact(right,ROW,COL,up,MAXROW,MAXCOL,LEN,ATTR,LIST,SLIST,cont):-
        nextcol(COL,1,COL1,MAXCOL),
        pdwmovevert(COL,COL1,ATTR,LIST),
        setstatus(COL1,ROW,SLIST,up),
        changepdwstate(pdwstate(ROW,COL1,up,MAXROW,LEN)).

  pdwkeyact(right,ROW,COL,down,_,MAXCOL,_,ATTR,LIST,SLIST,cont):-
        nextcol(COL,1,COL1,MAXCOL),
        check_removewindow(ROW),
        pdwmovevert(COL,COL1,ATTR,LIST),
        makepdwwindow(COL1,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL1,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL1,down,MAXROW1,LEN1)).

  pdwkeyact(left,ROW,COL,up,MAXROW,MAXCOL,LEN,ATTR,LIST,SLIST,cont):-
        nextcol(COL,-1,COL1,MAXCOL),
        pdwmovevert(COL,COL1,ATTR,LIST),
        setstatus(COL1,ROW,SLIST,up),
        changepdwstate(pdwstate(ROW,COL1,up,MAXROW,LEN)).

  pdwkeyact(left,ROW,COL,down,_,MAXCOL,_,ATTR,LIST,SLIST,cont):-
        nextcol(COL,-1,COL1,MAXCOL),
        check_removewindow(ROW),
        pdwmovevert(COL,COL1,ATTR,LIST),
        makepdwwindow(COL1,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL1,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL1,down,MAXROW1,LEN1)).

  pdwkeyact(up,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,SLIST,cont):-
        ROW>1,!,
        ROW1=ROW-1,
        field_attr(ROW,1,LEN,ATTR),
        pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
        pdwindex(ROW1,LIST,WORD),
        intenseletter(ROW,1,ATTR,WORD),
        reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
        cursor(ROW1,1),
        R=ROW1-1,
        setstatus(COL,R,SLIST,down),
        changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)).

   pdwkeyact(down,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,SLIST,cont):-
        ROW<MAXROW,!,
        ROW1=ROW+1,
        field_attr(ROW,1,LEN,ATTR),
        pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
        INDX=ROW-1,pdwindex(INDX,LIST,WORD),
        intenseletter(ROW,1,ATTR,WORD),
        reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
        cursor(ROW1,1),
        setstatus(COL,ROW,SLIST,down),
        changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)).


  pdwkeyact(down,_,COL,up,_,_,_,ATTR,LIST,SLIST,cont):-
        makepdwwindow(COL,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL,down,MAXROW1,LEN1)).

  pdwkeyact(cr,_,COL,up,_,_,_,ATTR,LIST,SLIST,stop):-
        makepdwwindow(COL,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL,down,MAXROW1,LEN1)),
        FIRSTROW=0,
```

## PULLDOWN PREDICATE

*continued from page 75*

an action. In this respect, operating **pulldown** is similar to operating a simple state machine:

```
repeat,
pdwstate(ROW,COL,DOWN,MAXROW,LEN),
readkey(KEY),
pdwkeyact(KEY,ROW,COL,DOWN,MAXROW,
          MAXCOL,LEN,ATTR,LIST,
          SLIST,CONTINUE),
CONTINUE=stop, ...
```

The valid input tokens are the left, right, up, and down arrow keys; the Esc key; the Enter key, and the first highlighted letter of each menu option. Therefore, the pull-down menu tool contains a **pdwkeyact** clause to handle every combination of valid user input and the current state of the menu system. You can easily extend the input processing capabilities, for example adding a feature and linking it to a function key, by adding an additional **pdwkeyact** clause.

### ADDING STATUS BAR MESSAGES

A useful programming tool is a programmer's work of art. Thus, when modifying a tool, it is important to exercise some sensitivity and try to make changes that reflect the internal structure of the tool. In practice, most tools are created out of other tools. This is certainly true for the tools found in the Turbo Prolog Toolbox.

The **pulldown** predicate uses some internal tools for list manipulation that are useful for implementing our status bar update feature. In this section we'll take a close look at how we can enhance the status bar by making a minimum number of changes and using some of the internal tools provided with **pulldown**.

Our first goal is to redefine the call to the pull-down menu system. Here we make use of Turbo Prolog's compound object support. The new call is:

```
      CH=COL+1, SUBCH=0,
      not(pdwaction(CH,SUBCH)).

 pdwkeyact(cr,ROW,COL,down,_,_,_,_,_,_,stop):-
      CH=COL+1, SUBCH=ROW,
      not(pdwaction(CH,SUBCH)),
      check_removewindow(ROW).

 pdwkeyact(char(CHAR),ROW,COL,UP,_,_,_,ATTR,PDWLIST,SLIST,stop):-
      is_up(UP,ROW),!,
      pdwlist_strlist(PDWLIST,STRLIST),
      tryletter(CHAR,STRLIST,SEL),NEWCOL=SEL,
      pdwmovevert(COL,NEWCOL,ATTR,PDWLIST),
      makepdwwindow(NEWCOL,ATTR,PDWLIST,MAXROW1,LEN1,FIRSTROW),
      setstatus(NEWCOL,ROW,SLIST,up),
      setstatus(NEWCOL,0,SLIST,down),
      changepdwstate(pdwstate(FIRSTROW,NEWCOL,down,MAXROW1,LEN1)),
      FIRSTROW=0,
      CH=NEWCOL+1, SUBCH=0,
      not(pdwaction(CH,SUBCH)).

 pdwkeyact(char(CHAR),ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,
          SLIST, stop):-
      ROW><0,
      pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
      tryletter(CHAR,LIST,SEL),ROW1=SEL+1,
      field_attr(ROW,1,LEN,ATTR),
      R=ROW-1,
      pdwindex(R,LIST,OLDWORD),
      intenseletter(ROW,1,ATTR,OLDWORD),
      reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
      cursor(ROW1,1),
      CH=COL+1, SUBCH=ROW1,
      R2=ROW1-1,
      setstatus(COL,R2,SLIST,down),
      changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)),
      not(pdwaction(CH,SUBCH)),
      removewindow.

 pdwkeyact(esc,ROW,COL,down,_,_,_,_,_,SLIST,cont):-
      check_removewindow(ROW),
      setstatus(COL,ROW,SLIST,up),
      changepdwstate(pdwstate(0,COL,up,0,0)).
```

LISTING 2: STPMEX.PRO

```
/*****************************************************************

    Example using the pull-down menu tools with status
    bar update.

******************************************************************/

include "tdoms.pro"


DATABASE
  pdwstate(ROW,COL,SYMBOL,ROW,COL)

include "tpreds.pro"
include "status.pro"
include "spulldwn.pro"     /* modified pull-down menu package */

Predicates
  msg(ROW,COL,STRING)
```

```
spulldown(ATTRIBUTE,MENULIST,
STATLIST,CHOICE,SUBCHOICE)
```

and the definition of the new parameter **STATLIST** is:

```
domains
   STATITEM = stat(STRING,
               STRINGLIST)
   STATLIST = STATITEM*
```

Thus, an example for a status message structure is:

```
stat("Select for help options",
     ["Help about the system",
      "Help on files"])
```

The first string is the message displayed when the corresponding option from the menu bar (horizontal menu) is selected. The list of strings, on the other hand, includes the messages displayed when traversing the options from the accompanying pull-down menu.

To completely implement the status bar update feature we must enhance the definition of two existing predicates, add two new predicates, and modify some of the internal predicates used in PULLDOWN.PRO.

The **pulldown** tool contains two internal predicates for performing list processing operations. The first, **pdwlistlen**, determines the length of the list of arguments used in the **pulldown** call. The second, **pdwindex**, returns the list element at a specified position in the list. In the predicates section, **pdwlistlen** is defined as:

```
predicates
   pdwlistlen(MENULIST,COL)
```

To add the needed support for our status bar messages, we can define two other **pdwlistlen** predicates:

```
pdwlistlen(STATLIST,COL)
pdwlistlen(STRINGLIST,COL)
```

With these new definitions, we can use the same code from the **pdwlistlen** clause to determine the length of the status list arguments and also the length of a general list of strings. The following is the code for **pdwlistlen**:

```
pdwlistlen([],0).
pdwlistlen([_|T],N):-
   pdwlistlen(T,X),
   N=X+1.
```

We must also add a predicate definition for **pdwindex**:

*continued on page 78*

```
CLAUSES

/* After a menu item is selected, one of the corresponding actions
   is chosen.
*/
   /* The file pull-down menu options */

  pdwaction(1,1):-msg(3,10,"Load file selected").
  pdwaction(1,2):-msg(4,10,"Save file selected").
  pdwaction(1,3):-msg(5,10,"Directory selected").
  pdwaction(1,4):-msg(6,10,"Print selected").
  pdwaction(1,5):-msg(7,10,"Copy selected").
  pdwaction(1,6):-msg(8,10,"Rename selected").
  pdwaction(1,7):-msg(9,10,"Operating system selected").

   /* The Run menu */

  pdwaction(2,0):-msg(3,25,"Run selected").

   /* The Help pull-down menu options */

  pdwaction(3,1):-msg(3,40,"Topic selected").
  pdwaction(3,2):-msg(4,40,"Edit selected").
  pdwaction(3,3):-msg(5,40,"Run selected").
  pdwaction(3,4):-msg(6,40,"Options selected").
  pdwaction(3,5):-msg(7,40,"Quick help selected").

   /* The options pull-down menu options */

  pdwaction(4,1):-msg(3,44,"Screen selected").
  pdwaction(4,2):-msg(4,44,"Printer selected").
  pdwaction(4,3):-msg(5,44,"Mouse selected").
  pdwaction(4,4):-msg(6,44,"Options macros").

   /* The Quit menu */

  pdwaction(5,0):-exit.

  msg(R,C,S):-
        makestatus(112,"Press any key"),
        makewindow(1,7,7,"Message Window",R,C,5,30),
        window_str(S),
        readkey(_),
        removewindow,
        removestatus.

GOAL
/*
          1         2         3         4         5         6
0123456789012345678901234567890123456789012345678901234567890123456789
    Files         Run            Help         Setup          Quit
*/

  makewindow(1,7,0,"",0,0,24,80),
  makestatus(112," Select with arrows or use first upper
                 case letter"),
  spulldown(7,
          [ curtain(5,"Files",["Load","Save","Directory","Print",
                             "Copy","Rename","Operating System"]),
            curtain(20,"Run",[]),
            curtain(35,"Help"    ,["Select topic","Edit","Run",
                                "Options","Quick help"]),
            curtain(48,"Setup"     ,["Screen","Printer", "Mouse",
                                 "Macros"]),
            curtain(63,"Quit"      ,[])
          ],
      [ stat("Select for file options",
           ["Load a new file",
            "Save Current file to disk",
            "View current directory",
            "Print current file",
            "Make a copy of current file",
            "Rename file",
            "Execute DOS commands"]),
        stat("Execute a program",[]),
        stat("Select for help",["Specify a topic",
            "Get help about the editor",
            "Get help on running a program",
            "Get help on the systems options",
            "Get the quick guide"]),
        stat("Select to setup the system", ["Setup the screen",
            "Setup the printer", "Setup the mouse",
            "Setup macros"]),
        stat("Select to exit the program", [])
      ]
                ,CH,SUBCH ),
    write("\n    CH = ",CH),
    write("\n SUBCH = ",SUBCH),nl.
```

LISTING 3: SPULLDWN.PRO

```
/******************************************************************

     Turbo Prolog Toolbox
     (C) Copyright 1987 Borland International.
          modified by KJ Weiskamp to support:

           1) Automatic status bar update
           2) Continuos scroll inside pull-down menus

                PULL DOWN MENU

  The parameters are:
        spulldown(ATTRIBUTE,MENULIST,STATLIST,CHOICE,SUBCHOICE)

  where
        ATTRIBUTE is used in all the windows
        MENULIST is the text for the menus
        STATLIST is the text for the status strings
        CHOICE is the selection from the horizontal menu
        SUBCHOICE is the selection from the vertical menu
               (or zero if there is no vertical menu for
                the CHOICE horizontal item)
******************************************************************/

/* ----- Include this database in your program ----
DATABASE
        pdwstate(ROW,COL,SYMBOL,ROW,COL)

include tooldom and toolpred

And provide the clauses for the pdwaction predicate

*/


DOMAINS

 /* data structure for pull-down menu strings */
 MENUELEM=  curtain(COL,STRING,STRINGLIST)
 MENULIST=  MENUELEM*
```

*continued from page 77*

```
pdwindex(COL,STATLIST,STATITEM)
```

And the code is:

```
pdwindex(0,[H|_],H):-!.
pdwindex(N,[_|T],X):-
   N1=N-1,pdwindex(N1,T,X).
```

Again, we do not have to modify this code. But to understand how this predicate works, we might want to look at an example. **pdwindex** is a general tool to retrieve an element from a list of elements. For example, in the call

```
pdwindex(2,["one","two","three",
             "four"], Str).
```

**pdwindex** binds the string "three" with the variable **Str**. If you're confused, keep in mind that this tool assumes that the first element of the list is element 0.

## ADDING TWO PREDICATES

We have extended the definitions of the needed internal predicates, and we are now ready to implement the two new predicates, **setstatus** and **checkargs**. These are defined as:

```
predicates
   setstatus(COL,ROW,STATLIST,
             SYMBOL)
   checkargs(MENULIST,STATLIST)
```

**setstatus** is responsible for updating the status bar message. **checkargs** tests the arguments in the new **spulldown** call to make sure that the menu list arguments match the status message arguments. Let's look at **setstatus** first (see Figure 3).

```
setstatus(COL1,_,SLIST,up):-
   pdwindex(COL1,SLIST,
            stat(STR,_)),
   changestatus(STR).

setstatus(COL1,_,SLIST,down):-
   pdwindex(COL1,SLIST,
            stat(_,LIST)),
   listlen(LIST,LISTLEN),
   LISTLEN=0,
   pdwindex(COL1,SLIST,
            stat(STR,_)),
   changestatus(STR),!.

setstatus(COL1,ROW,SLIST,down):-
   pdwindex(COL1,SLIST,
            stat(_,LIST)),
   pdwindex(ROW,LIST,STR),
   changestatus(STR).
```

*Figure 3. Clauses for* **setstatus**.

**setstatus** takes four parameters. The first two, **COL1** and **ROW**, indicate the index position for the corresponding status string in the status list. Keep in mind that the functor **stat** has two objects, a string and a list of strings:

```
stat(STRING,STRINGLIST)
```

Also, the status argument itself is a list of **stat** objects:

```
[stat(...), stat(...), stat(...),
 ... ]
```

Therefore, the **COL1** argument refers to the position or member in the list of **stat** objects and the **ROW** argument refers to the position or member of a string in the list of strings.

The third argument, **SLIST**, contains the list of **stat** objects. The last argument indicates the current state of the menu system. Therefore, we have one clause that processes the pull-down menus in the "up" state, and two clauses to process the "down" state. As shown, the "up" state is simple to process. This action involves finding the corresponding message from the status list and displaying the message by using a call to **changestatus**—one of the status bar predicates provided with the Turbo Prolog Toolbox in STATUS.PRO.

Processing the "down" state is a little more difficult. In this case, we must first determine if there are any members in the status string list. A list of length equal to zero indicates that the corresponding pull-down menu bar option does not have a menu associated with it. Therefore, the message bar is updated with a status bar message and not a pull-down menu message. If the list has members, then the final clause is executed and the appropriate message corresponding to one of the options inside the pull-down menu is displayed.

## ERROR CHECKING

To guarantee that the status bar messages work in harmony with the menu system, we must verify that the number of menu list arguments is equivalent to the number

*continued on page 80*

```
 /* data structure for status bar strings */
 STATITEM= stat(STRING,STRINGLIST)
 STATLIST= STATITEM*

 STOP    =  stop(); cont()

PREDICATES

  /* the modified pulldown predicate */
  spulldown(ATTR,MENULIST,STATLIST,INTEGER,INTEGER)
  pdwaction(INTEGER,INTEGER)

  pdwkeyact(KEY,ROW,COL,SYMBOL,ROW,COL,COL,ATTR,MENULIST,
            STATLIST,STOP)
  pdwmovevert(COL,COL,ATTR,MENULIST)
  pdwindex(COL,MENULIST,MENUELEM)
  pdwindex(ROW,STRINGLIST,STRING)

  /* add this predicate to support status bar strings */
  pdwindex(COL,STATLIST,STATITEM)

  makepdwwindow1(ROW,COL,ROW,COL,ATTR,STRINGLIST,ROW)
  makepdwwindow(COL,ATTR,MENULIST,ROW,COL,ROW)
  writelistp(ROW,COL,ATTR,STRINGLIST)
  line_ver(ROW,ROW,COL)
  line_hor(COL,COL,ROW)
  lcorn(COL,CHAR)
  rcorn(COL,CHAR)
  pdwlistlen(MENULIST,COL)
  pdwlistlen(STATLIST,COL)      /* supports status strings */
  pdwlistlen(STRINGLIST,COL)    /* suuports general string lists */
  writepdwlist(ATTR,MENULIST)
  changepdwstate(DBASEDOM)
  check_removewindow(ROW)
  is_up(SYMBOL,ROW)
  nextcol(COL,COL,COL,COL)
  intense(ATTR,ATTR)
  intensefirstupper(ROW,COL,ATTR,STRING)
  intenseletter(ROW,COL,ATTR,STRING)
  pdwlist_strlist(MENULIST,STRINGLIST)
  setstatus(COL,ROW,STATLIST,SYMBOL)       /* update status  message*/
  checkargs(MENULIST, STATLIST)            /* test arguments */

CLAUSES

/* draw pulldown window */
  line_ver(R1,R2,C):-
        R2>R1,!, R=R1+1,
        scr_char(R1,C,'|'),
        line_ver(R,R2,C).
  line_ver(_,_,_).

  line_hor(C1,C2,R):-
        C2>C1,!, C=C1+1,
        scr_char(R,C1,'-'),
        line_hor(C,C2,R).
  line_hor(_,_,_).

/* Make the pulldown window */
  makepdwwindow(NO,ATTR,MENULIST,LISTLEN,MAXLEN,FIRSTROW):-
        pdwindex(NO,MENULIST,curtain(CCOL,_,LIST)),COL=CCOL,
        ROW=2,
        listlen(LIST,LISTLEN1),LISTLEN=LISTLEN1,
        maxlen(LIST,0,MAXLEN),
        makepdwwindow1(ROW,COL,LISTLEN,MAXLEN,ATTR,LIST,FIRSTROW).
```

```
/*  makepdwwindow1(_,_,_,_,_,_,0):-keypressed,!. */
  makepdwwindow1(_,_,0,_,_,_,0):-!.
  makepdwwindow1(ROW,COL,LISTLEN,MAXLEN,ATTR,LIST,1):-
        NOOFROWS=LISTLEN+2, NOOFCOLS=MAXLEN+2,
        adjustwindow(ROW,COL,NOOFROWS,NOOFCOLS,AROW,ACOL),
        makewindow(81,ATTR,0,"",AROW,ACOL,NOOFROWS,NOOFCOLS),
        writelistp(1,MAXLEN,ATTR,LIST),
        cursor(1,1),reverseattr(ATTR,REV), field_attr(1,1,MAXLEN,REV),
        ENDROW=NOOFROWS-1,
        ENDCOL=NOOFCOLS-1,
        line_hor(1,ENDCOL,0),
        line_hor(1,ENDCOL,ENDROW),
        line_ver(1,ENDROW,0),
        line_ver(1,ENDROW,ENDCOL),
        scr_char(ENDROW,0,'└'),
        scr_char(ENDROW,ENDCOL,'┘'),
        lcorn(COL,LCORN), scr_char(0,0,LCORN),
        RCOL=ACOL+ENDCOL,
        rcorn(RCOL,RCORN), scr_char(0,ENDCOL,RCORN).

/* draw pulldown window corners */
  lcorn(0,'├') :- !.
  lcorn(_,'┬').

  rcorn(79,'┤') :- !.
  rcorn(_,'┬').

  check_removewindow(0):-!.
  check_removewindow(_):-removewindow.

  is_up(up,_):-!.
  is_up(_,0).

  intense(ATTR,ATTR1):-
        bitxor(ATTR,$08,ATTR1).

  intensefirstupper(ROW,COL,ATTR,WORD):-
        frontchar(WORD,CH,_),
        CH>='A', CH<='Z',!,scr_attr(ROW,COL,ATTR).
  intensefirstupper(ROW,COL,ATTR,WORD):-
        frontchar(WORD,_,REST),COL1=COL+1,
        intensefirstupper(ROW,COL1,ATTR,REST).

  intenseletter(ROW,COL,ATTR,WORD):-
        intense(ATTR,INTENS),
        intensefirstupper(ROW,COL,INTENS,WORD),!.
  intenseletter(ROW,COL,ATTR,_):-
        intense(ATTR,INTENS),
        scr_attr(ROW,COL,INTENS).

  pdwlist_strlist([],[]).
  pdwlist_strlist([curtain(_,H,_)|RESTPDW],[H|RESTSTR]):-
        pdwlist_strlist(RESTPDW,RESTSTR).

  pdwmovevert(COL1,COL2,ATTR,LIST):-
        pdwindex(COL1,LIST,curtain(POS1,WORD1,_)),str_len(WORD1,LEN1),
        pdwindex(COL2,LIST,curtain(POS2,WORD2,_)),str_len(WORD2,LEN2),
        field_attr(0,POS1,LEN1,ATTR),
        intenseletter(0,POS1,ATTR,WORD1),
        reverseattr(ATTR,REV),
        field_attr(0,POS2,LEN2,REV),
        intenseletter(0,POS2,REV,WORD2),
        cursor(0,POS2).

  setstatus(COL1,_, SLIST,up):-
        pdwindex(COL1, SLIST, stat(STR,_)),
        changestatus(STR).
```

## PULLDOWN PREDICATE

*continued from page 79*

of status list arguments. We use a clause called **checkargs** to do this:

```
checkargs(LIST,SLIST):-
   pdwlistlen(LIST,SZ1),
   pdwlistlen(SLIST,SZ2),
   SZ1=SZ2,!.

checkargs(_,_):-
   makewindow(80,7,7,
           "Error Window",5,15,
           4,45),
   window_str("Menu list does not
           match with Status
           list"),
   readkey(_),
   removewindow,
   exit.
```

If both arguments are of the same length, everything proceeds nicely. On the other hand, if their lengths are not equal, we have a serious problem and the program stops. The second **checkargs** clause puts up an error window to display the error message. If we must abort the program, we might as well do it in style!

### MODIFYING THE PREDICATES

We're now ready for the last step: performing surgery on **pulldown** and **pdwkeyact**. First, let's modify **pulldown**. Figure 4 displays the modified clause.

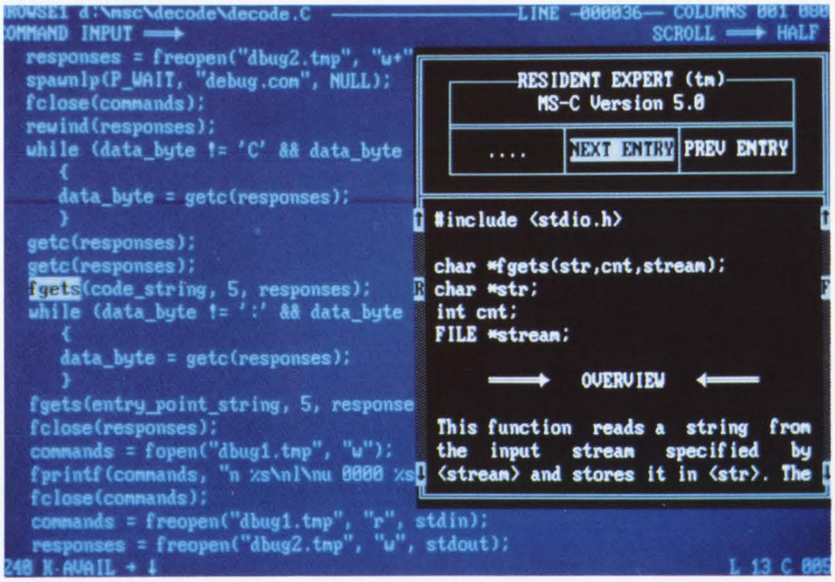
We change the name to **spulldown** and add the parameter **SLIST**. The second major change consists of adding a call to

```
spulldown(ATTR,LIST,SLIST,
        CH1,CH2):-
   checkargs(LIST,SLIST),
   makewindow(81,ATTR,ATTR,"",
           0,0,3,80),
   pdwlistlen(LIST,MAXCOL),
   writepdwlist(ATTR,LIST),
   pdwmovevert(0,0,ATTR,LIST),
   changepdwstate(pdwstate(0,0,
              up,0,0)),
   setstatus(0,0,SLIST,up),
   repeat,
   pdwstate(ROW,COL,DOWN,MAXROW,
          LEN),
   readkey(KEY),
   pdwkeyact(KEY,ROW,COL,DOWN,
           MAXROW,MAXCOL,LEN,
           ATTR,LIST,SLIST,
           CONTINUE),
   CONTINUE=stop,removewindow,
   pdwstate(ROW1,COL1,_,_,_),!,
   CH1=COL1+1,
   CH2=ROW1.
```

*Figure 4. Modified form of* **pulldown**.

*continued on page 82*

# RESIDENT EXPERT Pop-up Reference Guides...



```
ROWSE1 d:\msc\decode\decode.C  ————————————LINE -000036— COLUMNS 001 080
COMMAND INPUT ——→                                       SCROLL ——→ HALF
  responses = freopen("dbug2.tmp", "w+"
  spawnlp(P_WAIT, "debug.com", NULL);
  fclose(commands);
  rewind(responses);
  while (data_byte != 'C' && data_byte
     {
     data_byte = getc(responses);
     }
  getc(responses);
  getc(responses);
  fgets(code_string, 5, responses);
  while (data_byte != ':' && data_byte
     {
     data_byte = getc(responses);
     }
  fgets(entry_point_string, 5, response
  fclose(responses);
  commands = fopen("dbug1.tmp", "w");
  fprintf(commands, "n %s\nl\nu 0000 %s
  fclose(commands);
  commands = freopen("dbug1.tmp", "r", stdin);
  responses = freopen("dbug2.tmp", "w", stdout);
240 K AVAIL + ↓                                                L 13 C 005
```

```
RESIDENT EXPERT (tm)
MS-C Version 5.0

 ....    NEXT ENTRY  PREV ENTRY

#include <stdio.h>

char *fgets(str,cnt,stream);
char *str;
int cnt;
FILE *stream;

      ——→  OVERVIEW  ←——

This function  reads a  string  from
the  input   stream   specified   by
<stream> and stores it in <str>. The
```

# Use One of Ours or Build Your Own!

### THE POP-UP REFERENCE REVOLUTION BEGINS

How much development time could you save if you never had to open another PC language or technical reference manual again? What if you could just point at a compiler keyword, assembly instruction, or function name *on your screen* and with a keystroke have complete, authoritative information about language syntax, operands, parameters, examples, and much more?

### INTRODUCING THE RESIDENT EXPERT SYSTEM

A growing library of comprehensive, disk resident reference guides about the PC *and* your favorite PC languages. All available instantly through our unique memory resident pop-up access system.

### VIRTUALLY EVERYTHING YOU NEED TO KNOW

Each of our *Compiler Reference Guides* contains virtually everything you need to know to program with your *preferred implementation* of your favorite language. Language syntax, all library functions, compiler directives, and error codes are thoroughly documented.

Our *PC Programmer's Reference Guide* documents every PC (and AT) processor instruction and every BIOS and DOS service interrupt. You'll also find tables of keyboard codes, line drawing, ASCII, and IBM character sets, and much more.

### THE SPECIALIST'S LIBRARY

Your compiler is unique. That's why our reference guides are *specialized...*each one designed for a particular vendor's language implementation.

### *NEW!!*

RESIDENT EXPERT Compiler

Make your own Reference Guides

### QUICK DRAW ACCESS SYSTEM

*Point-and-shoot...*just place the cursor over any term on your screen. Chances are we've got it fully detailed in one of our data bases.

*Fully cross indexed...*if the instruction or library function you're using isn't quite right, our related topics cross index can help you find a better one.

*Multiple volumes on line...*you can have one or a dozen of our pop-up reference guides on line...a complete library available instantly.

### THE INFORMATION YOU NEED...WHERE YOU NEED IT

Our pop-up shell varies its size and shape dynamically, only taking as much space on your screen as it needs and it *never* covers your working area. You can see your work and our reference data at the *same* time.

| | |
|---|---|
| RESIDENT EXPERT Shell (*) | $19.95 |
| (with PC-DOS/MS-DOS Reference Guide) | |
| RESIDENT EXPERT Compiler | $39.95 |
| (create your own Reference Guides!) | |
| RESIDENT EXPERT Reference Guides | |
| Borland Turbo C (v1.0) | $19.95 |
| Borland Turbo Pascal (v4.0) | 19.95 |
| Borland Turbo Prolog (v1.1) | 19.95 |
| Lattice C (v3.2) | 39.95 |
| Mark Williams Let's C (v4.0) | 19.95 |
| Microsoft C (v5.0) | 39.95 |
| Microsoft Quick C (v1.0) | 19.95 |
| PC Programmer's Reference Guide | $39.95 |

*The RESIDENT EXPERT Shell is required to access and display all Reference Guides....

SantaRita

For the location of your nearest Santa Rita Software dealer, or to order direct, call us at 1-214-727-9217. We'd like to hear from you.

**Santa Rita Software**
**1000 E. 14th Street, Suite 365**
**Plano, Texas 75074**

# The RESIDENT EXPERT System

Resident Expert is a trademark of The Santa Rita Company. Borland, Turbo C, Turbo Pascal, and Turbo Prolog are trademarks of Borland International Inc. IBM and PC-DOS are trademarks of International Business Machines Corporation. Lattice C is a trademark of Lattice Inc. LetsC is a trademark of Mark Williams Company. Microsoft and MS-DOS are trademarks of Microsoft Corporation.

## PULLDOWN PREDICATE

*continued from page 80*

**checkargs**. Finally, we add a call to **setstatus**:

```
setstatus(0,0,SLIST,up)
```

This call initializes the menu system by displaying the first message from the status message list.

The final modifications consist of adding calls to **setstatus** from each of the **pdwkeyact** clauses. This task is fairly straightforward. Place the call immediately before the call to **changepdwstate** in each clause. Listing 1 shows the modified code.

### ADDING CONTINUOUS SCROLLING

In most respects, the **pulldown** tool allows you to create pull-down menu systems that are almost identical to those found in Turbo Prolog and Turbo C. Unfortunately, the designers of **pulldown** left out one feature: the menu bar does not continue to scroll around when it gets to the top or bottom of a pull-down (vertical) menu. In Turbo Prolog and Turbo C, the menu continues to scroll. If you get to the end of the menu and hit a down arrow key, the highlighted menu selection bar advances to the first item in the menu. Thus you can hold down either the up arrow or down arrow key and the menu selector loops around. If you don't believe me, try it. Right now!

Fortunately this feature can easily be added. To implement this, add two **pdwkeyact** clauses. The first **pdwkeyact** clause handles the case when a pull-down menu is "down," the menu selector bar (reverse video bar) is positioned at the first menu item, and the user input is the up arrow key (see Figure 5).

Note that the first **pdwkeyact** clause uses the statement **ROW = 1** to see if the menu bar is positioned at the first item. This clause then determines the length of the menu list using **pdwlistlen**. Once the length is determined, the last menu item is highlighted in reverse video. Also, **pdwkeyact** contains a call to **setstatus** to update the status message.

```
setstatus(COL1,_, SLIST,down):-
      pdwindex(COL1, SLIST, stat(_,LIST)),
      listlen(LIST,LISTLEN),
      LISTLEN=0,
      pdwindex(COL1,SLIST, stat(STR,_)),
      changestatus(STR),!.

setstatus(COL1,ROW, SLIST,down):-
      pdwindex(COL1, SLIST, stat(_,LIST)),
      pdwindex(ROW,LIST,STR),
      changestatus(STR).

checkargs(LIST,SLIST):-
   pdwlistlen(LIST,SZ1),
   pdwlistlen(SLIST,SZ2),
   SZ1=SZ2,!.

checkargs(_,_):-
   makewindow(80,7,7,"Error Window",5,15,4,45),
   window_str("Menu list does not match with Status list"),
   readkey(_),
   removewindow,
   exit.

pdwlistlen([],0).
pdwlistlen([_|T],N):-
      pdwlistlen(T,X),
      N=X+1.

writepdwlist(_,[]).
writepdwlist(ATTR,[curtain(POS,WORD,_)|T]):-
      str_len(WORD,LEN),
      field_str(0,POS,LEN,WORD),
      intenseletter(0,POS,ATTR,WORD),
      writepdwlist(ATTR,T).

writelistp(_,_,_,[]).
writelistp(ROW,LEN,ATTR,[H|T]):-
      field_str(ROW,1,LEN,H),
      intenseletter(ROW,1,ATTR,H),
      ROW1=ROW+1,
      writelistp(ROW1,LEN,ATTR,T).

pdwindex(0,[H|_],H):-!.
pdwindex(N,[_|T],X):-N1=N-1,pdwindex(N1,T,X).

changepdwstate(_):-retract(pdwstate(_,_,_,_,_)),fail.
changepdwstate(T):-assert(T).

nextcol(0,-1,COL1,MAX):-COL1=MAX-1,!.
nextcol(COL,1,0,MAX):-COL=MAX-1,!.
nextcol(COL,DD,COL1,_):-COL1=COL+DD.

spulldown(ATTR,LIST,SLIST,CH1,CH2):-
      checkargs(LIST,SLIST),
      makewindow(81,ATTR,ATTR,"",0,0,3,80),
      pdwlistlen(LIST,MAXCOL),
      writepdwlist(ATTR,LIST),
      pdwmovevert(0,0,ATTR,LIST),
      changepdwstate(pdwstate(0,0,up,0,0)),
      setstatus(0,0,SLIST,up),
      repeat,
      pdwstate(ROW,COL,DOWN,MAXROW,LEN),
      readkey(KEY),
      pdwkeyact(KEY,ROW,COL,DOWN,MAXROW,MAXCOL,LEN,ATTR,LIST,
         SLIST,CONTINUE),
      CONTINUE=stop,removewindow,
      pdwstate(ROW1,COL1,_,_,_),!,
      CH1=COL1+1,
      CH2=ROW1.
```

```
pdwkeyact(up,ROW,COL,down,MAXROW,_,
        LEN,ATTR,PDWLIST,SLIST,
        cont):-
   ROW=1,!,
   ROW1=ROW-1,
   field_attr(ROW,1,LEN,ATTR),
   pdwindex(COL,PDWLIST,
           curtain(_,_,LIST)),
   pdwindex(ROW1,LIST,WORD),
   intenseletter(ROW,1,ATTR,WORD),
   pdwlistlen(LIST,LEN1),
   reverseattr(ATTR,REV),
   field_attr(LEN1,1,LEN,REV),
   cursor(LEN1,1),
   R=LEN1-1,
   ROW2=LEN1,
   setstatus(COL,R,SLIST,down),
   changepdwstate(pdwstate(ROW2,COL,
                    down,MAXROW,LEN)).
```

*Figure 5. Additional* **pdwkeyact** *clause to add continuous scrolling "up."*

The second **pdwkeyact** clause handles the menu scrolling when the menu selection bar is at the end of the menu, as shown in Figure 6.

```
pdwkeyact(down,ROW,COL,down,MAXROW,
        _,LEN,ATTR,PDWLIST,SLIST,
        cont):-
  ROW=MAXROW,!,
  ROW1=1,
  field_attr(ROW,1,LEN,ATTR),
  pdwindex(COL,PDWLIST,
          curtain(_,_,LIST)),
  INDX=ROW-1,
  pdwindex(INDX,LIST,WORD),
  intenseletter(ROW,1,ATTR,WORD),
  reverseattr(ATTR,REV),
  field_attr(ROW1,1,LEN,REV),
  cursor(ROW1,1),
  setstatus(COL,0,SLIST,down),
  changepdwstate(pdwstate(ROW1,COL,
                  down,MAXROW,LEN)).
```

*Figure 6. Additional* **pdwkeyact** *clause to add continuous scrolling "down."*

## USING THE NEW TOOL

A sample program, given in Listing 2, shows you how to use the new tool, **spulldown**. Listing 3 displays the complete **spulldown** tool. The sample program creates a pull-down menu system with five items: **F**iles, **R**un, **H**elp, **S**etup, and **Q**uit. In this program, some of these menu items are linked with pull-down menus and some are not. This allows you to see how the tool operates in both of these cases.

When using the new **spulldown**

*continued on page 84*

```
/*  Pulldown window action corresponding to input key and Pulldown
    window state */
  pdwkeyact(right,ROW,COL,up,MAXROW,MAXCOL,LEN,ATTR,LIST,SLIST,cont):-
        nextcol(COL,1,COL1,MAXCOL),
        pdwmovevert(COL,COL1,ATTR,LIST),
        setstatus(COL1,ROW,SLIST,up),
        changepdwstate(pdwstate(ROW,COL1,up,MAXROW,LEN)).

  pdwkeyact(right,ROW,COL,down,_,MAXCOL,_,ATTR,LIST,SLIST,cont):-
        nextcol(COL,1,COL1,MAXCOL),
        check_removewindow(ROW),
        pdwmovevert(COL,COL1,ATTR,LIST),
        makepdwwindow(COL1,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL1,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL1,down,MAXROW1,LEN1)).

  pdwkeyact(left,ROW,COL,up,MAXROW,MAXCOL,LEN,ATTR,LIST,SLIST,cont):-
        nextcol(COL,-1,COL1,MAXCOL),
        pdwmovevert(COL,COL1,ATTR,LIST),
        setstatus(COL1,ROW,SLIST,up),
        changepdwstate(pdwstate(ROW,COL1,up,MAXROW,LEN)).

  pdwkeyact(left,ROW,COL,down,_,MAXCOL,_,ATTR,LIST,SLIST,cont):-
        nextcol(COL,-1,COL1,MAXCOL),
        check_removewindow(ROW),
        pdwmovevert(COL,COL1,ATTR,LIST),
        makepdwwindow(COL1,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL1,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL1,down,MAXROW1,LEN1)).

  pdwkeyact(up,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,SLIST,cont):-
        ROW>1,!,
        ROW1=ROW-1,
        field_attr(ROW,1,LEN,ATTR),
        pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
        pdwindex(ROW1,LIST,WORD),
        intenseletter(ROW,1,ATTR,WORD),
        reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
        cursor(ROW1,1),
        R=ROW1-1,
        setstatus(COL,R,SLIST,down),
        changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)).

  pdwkeyact(up,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,SLIST,cont):-
        ROW=1,!,
        ROW1=ROW-1,
        field_attr(ROW,1,LEN,ATTR),
        pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
        pdwindex(ROW1,LIST,WORD),
        intenseletter(ROW,1,ATTR,WORD),
        pdwlistlen(LIST,LEN1),
        reverseattr(ATTR,REV),field_attr(LEN1,1,LEN,REV),
        cursor(LEN1,1),
        R=LEN1-1,
        ROW2=LEN1,
        setstatus(COL,R,SLIST,down),
        changepdwstate(pdwstate(ROW2,COL,down,MAXROW,LEN)).

  pdwkeyact(down,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,SLIST,cont):-
        ROW<MAXROW,!,
        ROW1=ROW+1,
        field_attr(ROW,1,LEN,ATTR),
        pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
        INDX=ROW-1,pdwindex(INDX,LIST,WORD),
        intenseletter(ROW,1,ATTR,WORD),
        reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
        cursor(ROW1,1),
        setstatus(COL,ROW,SLIST,down),
        changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)).
  pdwkeyact(down,ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,SLIST,cont):-
        ROW=MAXROW,!,
        ROW1=1,
        field_attr(ROW,1,LEN,ATTR),
        pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
        INDX=ROW-1,pdwindex(INDX,LIST,WORD),
        intenseletter(ROW,1,ATTR,WORD),
        reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
        cursor(ROW1,1),
        setstatus(COL,0,SLIST,down),
        changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)).

  pdwkeyact(down,_,COL,up,_,_,_,ATTR,LIST,SLIST,cont):-
        makepdwwindow(COL,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL,down,MAXROW1,LEN1)).

  pdwkeyact(cr,_,COL,up,_,_,_,ATTR,LIST,SLIST,stop):-
        makepdwwindow(COL,ATTR,LIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(COL,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,COL,down,MAXROW1,LEN1)),
        FIRSTROW=0,
        CH=COL+1, SUBCH=0,
        not(pdwaction(CH,SUBCH)).

  pdwkeyact(cr,ROW,COL,down,_,_,_,_,_,_,stop):-
        CH=COL+1, SUBCH=ROW,
        not(pdwaction(CH,SUBCH)),
        check_removewindow(ROW).

  pdwkeyact(char(CHAR),ROW,COL,UP,_,_,_,ATTR,PDWLIST,SLIST,stop):-
        is_up(UP,ROW),!,
        pdwlist_strlist(PDWLIST,STRLIST),
        tryletter(CHAR,STRLIST,SEL),NEWCOL=SEL,
        pdwmovevert(COL,NEWCOL,ATTR,PDWLIST),
        makepdwwindow(NEWCOL,ATTR,PDWLIST,MAXROW1,LEN1,FIRSTROW),
        setstatus(NEWCOL,ROW,SLIST,up),
        setstatus(NEWCOL,0,SLIST,down),
        changepdwstate(pdwstate(FIRSTROW,NEWCOL,down,MAXROW1,LEN1)),
        FIRSTROW=0,
        CH=NEWCOL+1, SUBCH=0,
        not(pdwaction(CH,SUBCH)).

  pdwkeyact(char(CHAR),ROW,COL,down,MAXROW,_,LEN,ATTR,PDWLIST,
            SLIST,stop):-
        ROW><0,
        pdwindex(COL,PDWLIST,curtain(_,_,LIST)),
        tryletter(CHAR,LIST,SEL),ROW1=SEL+1,
        field_attr(ROW,1,LEN,ATTR),
        R=ROW-1,
        pdwindex(R,LIST,OLDWORD),
        intenseletter(ROW,1,ATTR,OLDWORD),
        reverseattr(ATTR,REV),field_attr(ROW1,1,LEN,REV),
        cursor(ROW1,1),
        CH=COL+1, SUBCH=ROW1,
        R2=ROW1-1,
        setstatus(COL,R2,SLIST,down),
        changepdwstate(pdwstate(ROW1,COL,down,MAXROW,LEN)),
        not(pdwaction(CH,SUBCH)),
        removewindow.

  pdwkeyact(esc,ROW,COL,down,_,_,_,_,_,SLIST,cont):-
        check_removewindow(ROW),
        setstatus(COL,ROW,SLIST,up),
        changepdwstate(pdwstate(0,COL,up,0,0)).

/* pdwkeyact(fkey(1),_,_,_,_,_,_,_,_,cont):- help.
   If a help system is used*/
```

## PULLDOWN PREDICATE

*continued from page 83*

predicate, remember to include the status-bar file, STATUS.PRO, before including the file containing **spulldown**. If you don't, Turbo Prolog will scream at you because of the undeclared clause, **changestatus**. You also must include the file TPREDS.PRO. The **spulldown** tool is stored in the file SPULLDOWN.PRO.

### spulldown IN ACTION

Believe it or not, even with the modifications, the pull-down menu system is still fast. In fact, it's about as fast as the Turbo C environment according to my benchmarks. That's right. I've developed a set of benchmarks for testing user interfaces, especially pull-down menus. The major benchmark consists of holding down the right or left arrow key and counting the number of times the reverse video menu bar whips across the screen. In my test, I discovered that the Turbo C menu system can be traversed 47 times in 30 seconds; the test program developed here did 50 cycles. The test program, with status message update, actually beat Turbo C—but then again maybe the benchmark is unfair since Turbo C has seven entries in its pull-down menu bar and the test program only has five. Now, if you don't think this is a practical benchmark, ask yourself when was the last time you used the Sieve or Dhrystone in a program.

Nevertheless, the Turbo Prolog Toolbox is a great source of programming gems. After modifying **pulldown**, you might want to add personal features to some of the other tools. After all, there is great satisfaction in conquering a software tool and making it your own. ■

*Keith Weiskamp is the editor-in-chief of* PC AI Magazine, *and co-author of the forthcoming book,* Artificial Intelligence Programming with Turbo Prolog.

*Listings may be downloaded from CompuServe as PDOWN.ARC*

# THE TAIL RECURSION TIGER

## Given the right conditions, the Turbo Prolog compiler will optimize your code for speed and efficiency.

*Michael Covington*

Pascal, BASIC, or C programmers who start using Prolog are often dismayed to find that the language has no **FOR**, **WHILE**, or **REPEAT** statements. That is, there is no direct way to express iteration. Prolog allows only two kinds of repetition: *backtracking*, in which multiple solutions are sought for a single query, and *recursion*, in which a procedure calls itself.

As it turns out, this doesn't restrict the power of the language. In fact, Turbo Prolog recognizes a special case of recursion—called *tail recursion*—and compiles it into an interative loop in machine language. This means that although the program logic is expressed recursively, the compiled code is as efficient as it would be in Pascal or BASIC.

This article explores the art of coding repetitive processes in Prolog. As we'll see, recursion is in most cases clearer, more logical, and less error-prone than the loops that conventional languages use. But before we explore recursion, let's look at backtracking.

### BACKTRACKING

When a procedure backtracks, it looks for another solution to a query that has already been solved. A clause that is capable of generating multiple solutions is said to be *nondeterministic*. You can exploit backtracking as a way to perform repetitive processes.

Consider the program in Listing 1 (BAKTRAK.PRO). The predicate **country** simply lists the names of various countries, so that a goal such as

```
country(X)
```

has multiple solutions. The predicate **print_countries** then prints out all of these solutions. It is defined as follows:

```
print_countries:-
   country(X),
   write(X),
   nl,
   fail.
print_countries.
```

Look at the first clause. It says: "To print countries, find a solution to **country(X)**, then write **X** and start a new line, then fail." By *fail* we mean "assume that a solution to the original goal has not been reached, so back up and look for an alternative." The built-in predicate **fail** always fails, but we could equally well force backtracking by using any other goal that would always fail, such as $\mathbf{5 = 2 + 2}$ or **country(shangri__la)**.

The first time through, **X** is instantiated to **england**, which is printed. Then, when it hits **fail**, the program backs up. There are no alternative ways to satisfy **nl** or **write(X)**, so the program looks for a different solution to **country(X)**. The last time **country(X)** was executed, it gave a value to the previously uninstantiated variable **X**. So before retrying this step, Turbo Prolog de-instantiates **X**. Then it can look for an alternative solution for **country(X)** and instantiate **X** to a different value. If it succeeds, execution goes forward again and the name of another country is printed.

Eventually, the first clause runs out of alternatives. When this happens, execution falls through to the second clause, which succeeds without doing anything further. In this way the goal **print__countries** terminates with success. Its complete output is:

```
england
france
germany
denmark
True
```

If the second clause were not there, **print__countries** would terminate with failure, and the final message would be **False**. Apart from that, the output would be the same.

Backtracking is a good way to get all the alternative solutions to a goal. But even if your goal doesn't have multiple solutions, you can still use backtracking to introduce repetition. One way to do this is to simply define the two-clause predicate:

*continued on page 86*

## TAIL RECURSION

*continued from page 85*

```
repeat.
repeat :- repeat.
```

**repeat** is a nondeterministic clause; it tricks Prolog's control structure into thinking it has an infinite number of different solutions. (Never mind how—after we discuss tail recursion you'll know why it works.) The purpose of **repeat** is to allow backtracking ad infinitum.

Listing 2 (TYPEWRIT.PRO) shows how **repeat** works. The procedure **typewriter** accepts characters from the keyboard and prints them on the screen until the user types Enter (ASCII code 13):

```
typewriter:-
   repeat,
   readchar(C),
   write(C),
   char_int(C,13).
```

It works as follows: Execute **repeat** (which does nothing), then read a character into the variable **C**, write **C** and check whether the ASCII code of **C** is 13. If so, you're finished. If not, backtrack and look for alternatives. Neither **write** nor **readchar** generates alternative solutions, so backtrack all the way to **repeat**, which always has alternative solutions. Now execution can go forward again, reading another character, printing it, and checking whether it is ASCII 13.

Note that the character **C** gets de-instantiated when we backtrack past **readchar(C)**, which instantiated it. De-instantiation is vital when backtracking is used to obtain alternative solutions to a goal, but de-instantiation also makes it hard to use backtracking for any other purpose. The reason is that although a backtracking process can repeat operations any number of times, it can't "remember" anything from one repetition to the next. All variables lose their values when execution backtracks through the steps that instantiated them. Therefore, it is necessary to use the **database** to store intermediate results (such as the current value of the counter) within a **repeat** loop.

### RECURSION

This leads to the other way of expressing repetition—*recursion.* A recursive procedure is one that calls itself. Recursive procedures have no trouble keeping records of their progress because counters, totals, and intermediate results can be passed from each iteration to the next as arguments.

The logic of recursion is easy to follow if you allow yourself to forget, for the moment, how computers work. (Prolog is so different from machine language that ignorance of computers is often an asset to the Prolog programmer.) Forget that the program is trekking through memory addresses one by one, and imagine instead a machine that can follow recipes like this one:

> To find the factorial of a number N:
> If N is 1, the factorial is 1.
> Otherwise, find the factorial of N-1, then multiply it by N.

That is: To find the factorial of 3 you must find the factorial of 2, and to find the factorial of 2 you must find the factorial of 1. Fortunately, you can find the factorial of 1 without referring to any other factorials, so the repetition doesn't go on forever. When you have it, you multiply it by 2 to get the factorial of 2, then multiply that by 3 to get the factorial of 3, and you're done. The Prolog code for this is:

```
factorial(1,1).
factorial(X,FactX):-
   X > 1,
   Y = X-1,
   factorial(Y,FactY),
   FactX = X*FactY.
```

The second clause states **X > 1** as a condition, so that at most only one clause will apply for any given number. Listing 3 (FACT1.PRO) shows the complete program.

But wait a minute, you say. How does the program execute **factorial** while it's in the middle of executing **factorial**? If you call **factorial** with **X** equal to 3, **factorial** then calls itself with **X** equal to 2. Does **X** then have two values, or does the second one just wipe out the first, or what?

The answer is that the compiler creates a new copy of **factorial** so that **factorial** can call itself as if it were a completely separate procedure. The executable code doesn't have to be duplicated, of course, but the arguments and internal variables do. This information is stored in an area called a *stack frame*, which is created every time a procedure is called. When the procedure terminates, the memory occupied by its stack frame is returned to the heap, and execution continues in the stack frame that was previously being used.

Recursion has two advantages. First, it can express algorithms that can't conveniently be expressed any other way. Recursion is the natural way to describe any problem that contains within itself another problem of the same kind. Examples include tree searches (a tree is made up of smaller trees) and recursive sorting (to sort a list, partition it, sort the parts, and then put them together).

Second, recursion is logically simpler than iteration. Recursive algorithms have the structure of an inductive mathematical proof. Our recursive factorial algorithm above describes an infinite number of different computations by means of just two clauses. This makes it easy to see that the clauses are correct. Further, the correctness of each clause can be judged independently of the other.

But recursion has one big drawback: it eats memory. Whenever a procedure calls itself, the state of execution of the calling procedure has to be saved on the stack. This means that if a procedure calls itself 100 times, 100 different versions of its stack frame have to be stored at once. The memory of the IBM PC accommodates at most 300 or 400 stack frames (depending on the amount of available memory, the number of arguments being passed, whether there are nondeterministic calls within the recursive call, etc.) So what do you do if you want to repeat something more than 400 times?

### TAIL RECURSION

There's a special case in which a procedure can call itself without storing a stack frame. Recall that the stack frame enables the outer procedure to resume after the inner procedure finishes. What if the outer procedure isn't going to resume?

This isn't as crazy as it sounds. Suppose the outer procedure calls the inner procedure as its *very last step*. When the inner procedure terminates, the outer procedure won't have anything else to do. This means it doesn't need a stack frame. Control can go directly to wherever it would have gone when the outer procedure finished.

Suppose, for example, that procedure A calls procedure B, and B calls procedure C as its very last step. When B calls C, we know that B isn't going to do anything further. So instead of creating a new stack frame for C under B, we replace the stack frame of B with that of C, making appropriate changes in the addresses of parameters. When C finishes, it thinks it was called by A directly.

Now suppose that instead of calling C, B calls itself as its very last step. Our recipe says that when the call takes place, the old stack frame for B should be replaced by a new stack frame for B. This is a trivial operation; only the parameters need to be set to new values. It is much like updating the control variables in a loop.

This is called *tail recursion optimization* or *last call optimization*, and it was introduced into Prolog by David Warren at the University of Edinburgh. About half of the available Prolog implementations have it today. Last call optimization is also a prominent feature of Scheme—a LISP dialect developed at M.I.T.—and is available in some common LISP implementations.

## MAKING IT WORK IN PROLOG

What is meant when we say that one procedure calls another "as its very last step"? In Prolog, this means that:

1. The call is in the last clause of the predicate

*continued on page 88*

**LISTING 1: BAKTRAK.PRO**

```
/* BAKTRAK.PRO */

/* Uses backtracking to print
   all solutions to a query */

PREDICATES

  country(symbol)
  print_countries

CLAUSES

  country(england).
  country(france).
  country(germany).
  country(denmark).

  print_countries :- country(X),
                     write(X),
                     nl,
                     fail.

  print_countries.
```

**LISTING 2: TYPEWRIT.PRO**

```
/* TYPEWRIT.PRO */

/* Uses 'repeat' to keep accepting
   characters and printing them
   until Return is encountered. */

PREDICATES

  repeat
  typewriter

CLAUSES

  repeat.
  repeat :- repeat.

  typewriter :- repeat,
                readchar(C),
                write(C),
                char_int(C,13).
```

LISTING 3: FACT1.PRO

```
/* FACT1.PRO */

/* Recursive program to compute factorials.
   This is not tail recursive. */

PREDICATES

  factorial(integer,real)

CLAUSES

  factorial(1,1).

  factorial(X,FactX) :-
        X > 1,
        Y = X-1,
        factorial(Y,FactY),
        FactX = X*FactY.
```

LISTING 4: COUNT.PRO

```
/* COUNT.PRO */

/* Tail recursive program that
   never runs out of memory */

PREDICATES

   count(real)

     /* Reals can be much
        bigger than integers. */

CLAUSES

   count(N) :- write(N),nl,
               NewN = N+1,
               count(NewN).

GOAL

   count(1).
```

LISTING 5: BADCOUNT.PRO

```
/* BADCOUNT.PRO */

/* Three procedures that are like
   COUNT.PRO but not tail recursive;
   they run out of memory after
   a few hundred iterations. */

PREDICATES

  badcount1(real)
  badcount2(real)
  badcount3(real)
  check(real)
```

## TAIL RECURSION

*continued from page 87*

2. The call is the very last subgoal of the clause, and
3. There are no untried alternatives for earlier subgoals in the clause.

The following example satisfies all three conditions:

```
count(N):-
   write(N),nl,
   NewN = N+1,
   count(NewN).
```

This procedure is tail recursive; it calls itself without allocating a new stack frame and therefore never runs out of memory. As shown in Listing 4 (COUNT.PRO), given the goal

```
count(0).
```

the procedure **count** will print integers starting with 0 and never end. Eventually, rounding errors will make it print inaccurate numbers, but it will never stop.

### HOW NOT TO DO IT

Now that we've shown you how to do it right, Listing 5 (BADCOUNT.PRO) shows you three ways to do it wrong. First, if the recursive call isn't the very last step, the procedure isn't tail recursive. For example:

```
badcount1(X):-
   write(X),nl,
   NewX = X+1,
   badcount1(NewX),
   nl.
```

Every time **badcount1** calls itself, a stack frame has to be saved so that control can return to the calling procedure, which has yet to execute its final **nl**. So only a few hundred recursive calls can take place before the program runs out of memory.

Another way to lose tail recursion is to leave an alternative untried at the time the recursive call is made. Then a stack frame has to be saved so that if the recursive call fails, the calling procedure can go back and try the alternative. For example:

```
badcount2(X):-
   write(X),nl,
   NewX = X+1,
   badcount2(NewX).
badcount2(X):-
   X < 0,
   write("X is negative.").
```

Here the first clause of **badcount2** calls itself before the second clause has been tried. Again, the program runs out of memory after a few hundred calls.

The untried alternative need not be a separate clause for the recursive procedure itself. It can equally well be an alternative in some other clause that it calls. For example:

```
badcount3(X):-
   write(X),nl,
   NewX = X+1,
   check(NewX),
   badcount3(NewX).

check(Z):- Z >= 0.
check(Z):- Z < 0.
```

Suppose **X** is positive, as it normally is. Then when **badcount3** calls itself, the first clause of **check** has succeeded but the second clause of **check** has not yet been tried. So **badcount3** has to preserve a copy of its stack frame so that it can go back and try the other clause of **check** if the recursive call fails.

## CUTS TO THE RESCUE

This being so, you may think that it's impossible to guarantee that a procedure is tail recursive. After all, it's easy enough to put the recursive call in the last subgoal of the last clause, but how do you guarantee that there are no alternatives in any of the *other* procedures that it calls?

Fortunately, you don't have to. The built-in cut operator "!" allows you to discard whatever alternatives may exist. We can then fix up **badcount3** (leaving **check** as it was):

```
cutcount3(X):-
   write(X),nl,
   NewX = X+1,
   check(NewX),!,
   cutcount3(NewX).
```

The cut means "burn your bridges behind you" or, more precisely, "once you reach this point, disregard alternative clauses for this predicate and alternative solutions to earlier subgoals within this clause." That's precisely what we need. Because alternatives are ruled out, no stack frame is

*continued on page 90*

```
CLAUSES

/* badcount1:
   The recursive call is not the last step. */

  badcount1(X) :- write(X),nl,
                  NewX = X+1,
                  badcount1(NewX),
                  nl.

/* badcount2:
   There is a clause that has not been tried
   at the time the recursive call is made. */

  badcount2(X) :- write(X),nl,
                  NewX = X+1,
                  badcount2(NewX).

  badcount2(X) :- X < 0,
                  write("X is negative.").


/* badcount3:
   There is an untried alternative in a
   procedure called before the recursive call. */

  badcount3(X) :- write(X),nl,
                  NewX = X+1,
                  check(NewX),
                  badcount3(NewX).

  check(Z) :- Z >= 0.
  check(Z) :- Z < 0.
```

LISTING 6: CUTCOUNT.PRO

```
/* CUTCOUNT.PRO */

/* Shows how 'badcount2' and 'badcount3'
   can be fixed by adding cuts to
   rule out the untried clauses.
   These versions are tail recursive. */

PREDICATES

  cutcount2(real)
  cutcount3(real)
  check(real)

CLAUSES

/* cutcount2:
   There is a clause that has not been tried
   at the time the recursive call is made. */

  cutcount2(X) :- write(X),
                  nl,
                  NewX = X+1,
                  !,
                  cutcount2(NewX).
  cutcount2(X) :- X < 0,
                  write("X is negative.").


/* cutcount3:
   There is an untried alternative in a
   clause called before the recursive call. */

  cutcount3(X) :- write(X),
                  nl,
                  NewX = X+1,
                  check(NewX),
                  !,
                  cutcount3(NewX).

  check(Z) :- Z >= 0.
  check(Z) :- Z < 0.
```

### LISTING 7: FACT2.PRO

```
/* FACT2.PRO */

/* Tail recursive program
   to compute factorials */

PREDICATES

   factorial(integer,real)
   factorial_aux(integer,real,integer,real)

   /* Numbers likely to exceed
      32767 are declared as reals */

CLAUSES

   factorial(N,FactN) :-
        factorial_aux(N,FactN,1,1).

   factorial_aux(N,FactN,I,P) :-
        I <= N,
        NewP = P*I,
        NewI = I+1,
        !,
        factorial_aux(N,FactN,NewI,NewP).

   factorial_aux(N,FactN,I,FactN) :-
        I > N.
```

## TAIL RECURSION

*continued from page 89*

needed and the recursive call can go inexorably ahead.

A cut is equally effective in **badcount2**:

```
cutcount2(X):-
   write(X),nl,
   NewX = X+1,!,
   cutcount2(NewX).
cutcount2(X):-
   X < 0,
   write("X is negátive.").
```

If the cut is executed, the compiler assumes there are no untried alternatives and does not create a stack frame. See Listing 6 (CUTCOUNT.PRO) for the complete program.

Unfortunately, cuts won't help us with **badcount1**, whose need for stack frames has nothing to do with untried alternatives. The only way to improve **badcount1** would be to rearrange the computation so that the recursive call comes at the end of the clause.

### USING PARAMETERS AS LOOP VARIABLES

Now that we've mastered tail recursion, what do we do about loop variables and counters? To answer that question, let's do a bit of Pascal-to-Prolog translation.

We have already developed a recursive procedure to compute factorials; now let's develop an iterative one. In Pascal, this is:

```
P := 1;
for I := 1 to N do
  P := P*I;
  FactN := P;
```

There are four variables here. **N** is the number whose factorial we're finding; **FactN** is the result of doing so; **I** is the loop variable, counting from 1 to **N**; and **P** is the variable in which the product is accumulated. A more efficient Pascal programmer might combine **FactN** and **P**, but in Prolog it pays to be fastidious.

The first step in translating this into Prolog is to replace **for** with a simpler loop construct, making explicit what happens to **I** in each step. Let's recast the algorithm as a **while** loop:

```
P := 1;
I := 1;
while I <= N do
  begin
     P := P*I;
     I := I+1
  end;
FactN := P;
```

Now we're ready to construct the Prolog translation, shown in Listing 7 (FACT2.PRO). Our **factorial** procedure only has **N** and **FactN** as arguments. A second procedure,

```
factorial_aux(N,FactN,I,P)
```

actually performs the recursion; its four arguments are the four variables that need to be passed along from each step to the next. So **factorial** simply invokes **factorial__aux**, passing along **N** and **FactN** along with the initial values for **I** and **P**:

```
factorial(N,FactN):-
   factorial_aux(N,FactN,1,1).
```

That's how **I** and **P** get initialized.

Again you interrupt me with a puzzled look. How can we "pass along" **FactN** to another procedure? It doesn't even have a value yet! The answer is that all we're doing is unifying **FactN** in one clause with **FactN** in another. The same thing happens whenever **factorial__aux** passes **FactN** to itself as an argument in a recursive call. Eventually, the last **FactN** gets a value, and when this happens, all the other **FactN**'s, having been unified with it, get the same value.

Now for **factorial__aux**. Ordinarily, it checks that **I** is less than or equal to **N**—the condition for continuing the loop—and then calls itself recursively with new values for **I** and **P**. Another peculiarity of Prolog asserts itself here. In Prolog, as in arithmetic (but not in most programming languages), the statement

```
P = P + 1
```

is manifestly false. You can't change the value of a Prolog variable. Instead, you have to create a new variable and say:

```
NewP = P + 1.
```

So here is the first clause:

```
factorial_aux(N,FactN,I,P):-
   I <= N,
   NewP = P*I,
   NewI = I+1,!,
   factorial_aux(N,FactN,NewI,
                 NewP).
```

As in **cutcount2** above, the cut makes the clause tail recursive even though it isn't the last clause.

Eventually, the loop exceeds **N**. When it does, we simply unify the current value of **P** with **FactN** and stop the recursion. One way to do this is:

```
factorial_aux(N,FactN,I,P):-
   I > N,
   FactN = P.
```

But there is no need for **FactN** = **P** to be a separate step; the unification can be performed in the argument list by putting the same variable name in the positions occupied by **FactN** and **P**. This gives us the final clause:

```
factorial_aux(N,FactN,I,FactN):-
   I > N.
```

Remember that if **I** > **N** failed, the unification of the second and fourth arguments would have no effect, because execution would backtrack out of this clause immediately, returning all variables to their previous state. ■

## SUGGESTED READING

Tail recursion is discussed at greater length in Chapter 4 of *Prolog Programming in Depth*, by Michael Covington, Donald Nute, and Andre Vellino (Scott, Foresman and Company, to appear in January 1988). The most comprehensive—and elegant—study of tail recursive algorithms that I have seen is *Structure and Interpretation of Computer Programs*, by Abelson and Sussman (M.I.T. Press, 1984), which uses Scheme, a LISP dialect, but presumes no prior knowledge of it.

*Michael Covington does artificial intelligence research at the University of Georgia and has written over 100 magazine articles relating to microcomputers.*

*Listings may be downloaded from CompuServe as TAIL.ARC.*

# PARTNERS OF A SORT

## Turbo Prolog solves a sort puzzle by borrowing a piece from Turbo Pascal.

*Alex Lane*

WIZARD

Lists of objects can be elegantly sorted in Turbo Prolog using any of a number of well-known methods like Quicksort, bubble sort, and insertion sort. Unfortunately, once the number of items to be sorted gets above a few thousand, the Turbo Prolog programmer risks exhausting stack or heap (allocatable memory). The actual number of sortable items depends on several factors, including available memory, the size of the application itself, and the sorting method used. But eventually there comes a point where the data to be sorted is too large for any configuration.

I learned this the hard way while writing a custom Turbo Prolog application requiring the sorting of several thousand names. Try as I might, all my attempts crashed and burned due to a lack of available memory. What I needed was a way to call a time-out in the middle of my Turbo Prolog code, get the names sorted some other way, and then continue with my Turbo Prolog program.

### DATABASE TOOLBOX TO THE RESCUE!

I had everything I needed except a sorting program, and I really didn't relish the prospect of rolling one from scratch. The DOS SORT utility was out, because it can't handle files larger than 64K. As I scanned my library for my trusty reference on algorithms, I spied the *Turbo Pascal Database Toolbox Owner's Handbook* and picked it up on impulse. When I read the blurb on the back cover about Turbo Sort, I knew I'd hit paydirt.

The drawback to my discovery was that Turbo Prolog does not link directly with Turbo Pascal. Fortunately, breaking out to DOS to invoke a child process is simple in Turbo Prolog. Turbo Prolog provides the **system** predicate that takes a string as an argument and passes it to DOS as a command. For instance, if we issue the command

```
system("DIR B:")
```

the program will pause, display the directory of drive B:, and then resume execution of the Turbo Prolog code. In my program, the argument to the **system** predicate runs the sort program.

The last thing to consider is how to pass data between applications. Turbo Sort needs an input file to work on, so the Turbo Prolog program must create such a file containing the necessary information. After the sort program finishes writing the results to its output file, the newly resumed Turbo Prolog program must read that file to extract the results of the sort. A schematic diagram of this relationship is shown in Figure 1.

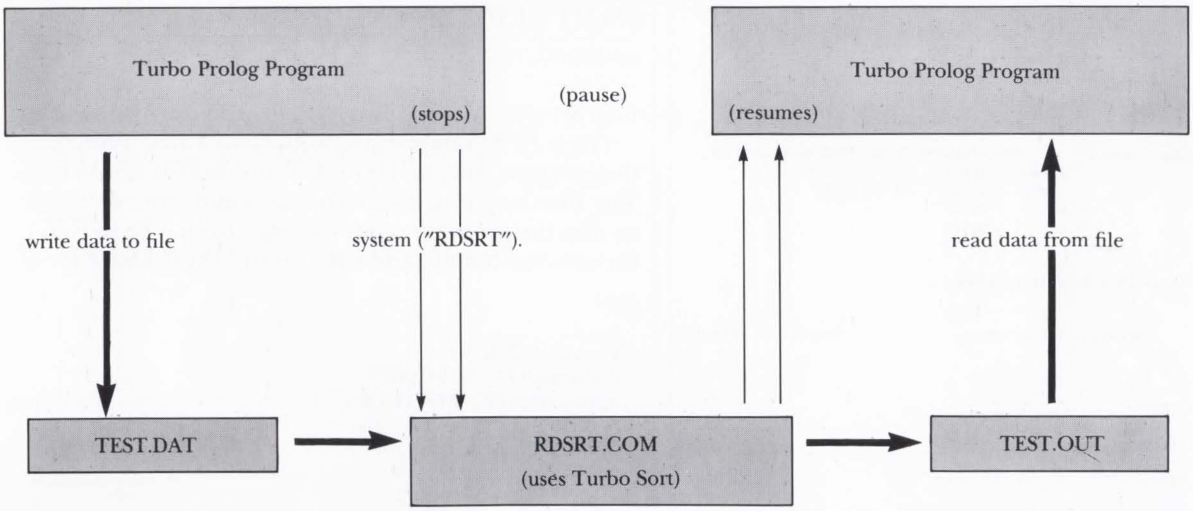
With the Turbo Pascal Database Toolbox manual in hand, I modified the sort routine to accommodate my requirements. In short order I had my sorting program, and the overall job was completed on time and on budget.

To give you a better idea of how this is done, let's use this technique to sort the filenames in all directories and subdirectories of a hard disk. The modified Turbo Pascal Turbo Sort program RDSRT.PAS is shown in Listing 1.

### ABOUT RDSRT.PAS

Turbo Sort, provided in the Turbo Pascal Database Toolbox, uses the Quicksort algorithm and virtual memory management to rapidly sort quantities of data that may exceed the memory available for sorting. Both features are important in this application, particularly memory management, because free memory may be limited while the sort program is running (don't forget the calling Turbo Prolog program is still resident, just suspended).

Turbo Sort requires the programmer to supply



Note: Programs execute from left to right in this schematic.

*Figure 1. Flow of program control between Turbo Prolog and Turbo Pascal routines.*

three procedures, **InP**, **OutP**, and **Less,** in order to create a sorting program. **Inp** passes items to be sorted to Turbo Sort, **OutP** fetches them back later, and **Less** lets Turbo Sort know the order in which you want things arranged.

To make things simple, we're going to arrange things such that the *parent* Turbo Prolog program always writes the information to be sorted in a file called TEST.DAT. Furthermore, each line in TEST.DAT is an 18-byte record where the first 12 characters are the filename, the next four characters are an identifying index, and the last two characters are a carriage return/line feed (0D 0A) combination. The Turbo Pascal procedure **Inp** then opens TEST.DAT, reads the data in one line at a time, and passes the records to Turbo Sort using the **SortRelease** procedure.

Once Turbo Sort is finished, the procedure **OutP** creates a file TEST.OUT (our Turbo Prolog program will expect to see it!). Then, using a **REPEAT..UNTIL** loop, **OutP** makes calls to the **SortReturn** procedure and writes the results to TEST.OUT. This continues until the **SortEOS** function returns the value **True**.

The **Less** function is used by Turbo Sort to decide where and how to compare items being sorted. In our case, we want to compare filenames and then proceed to sort in ascending alphabetical order. Declaring the variables **FirstObject** and **SecondObject** to be absolute at **X** and **Y**, respectively, is necessary because **Less** is called with two parameters (the addresses of the data items being compared).

Once all of these procedures are supplied, we set the sort in motion by calling the function **TurboSort** with the size (in bytes) of the items to be sorted as a parameter:

```
ErrorStatus := TurboSort(ItemSize);
```

The value returned in **ErrorStatus** may be used as a diagnostic to make sure everything worked smoothly.

## ON THE TURBO PROLOG SIDE

Listing 2 shows the source code for UNIVDIR.PRO, which is the Turbo Prolog program that does most of the mule work.

The predicate **univdir** is designed to retrieve all file information from all subdirectories. As mentioned before, each filename, along with a unique identifying number (generated by the program), is written to a work file called TEMP.DAT. We do this to limit the size of TEMP.DAT. Although we could, in principle, write all file information out to this file, the resulting file may be too large for our system. (Don't forget, we need at least as much space available on disk for the sorted copy of the file as for the original, unsorted file.)

The rest of the file information (size, date and time of creation, etc.) is saved in the dynamic database using **asserta**. Once all files have been found, the program calls the **system** predicate to execute RDSRT.COM (our Turbo Pascal sort module). RDSRT reads TEMP.DAT, sorts the filenames and outputs a list of identifying numbers that correspond to the order of the alphabetized files. To clarify: if the original file looks like

```
ZAPFNCS.C      1
FOO.BAR        2
DATABASE.DAT   3
PROLOG.EXE     4
TURBO.COM      5
```

then the output file TEST.OUT will look like:

```
3
2
4
5
1
```

This is done both to conserve space on the disk (remember, this technique was originally used to sort

*continued on page 94*

## PARTNERS OF A SORT

*continued from page 93*

nearly 5,000 names!) and to make the next step easy.

Once DOS hands the CPU back to Turbo Prolog, the program opens TEST.OUT and MYFILES.DAT. The next step is to assign the read and write devices so that the program reads the data from TEST.OUT, formats the output, and writes it to MYFILES.DAT:

```
goal
  ...
  system("RDSRT"),
  openread(f1,"TEST.OUT"),
  openwrite(f2,"MYFILES.DAT"),
  readdevice(f1),
  writedevice(f2),
  pretty_out(f1,f2),
  ...
```

The clause that does the formatting is **pretty__out**. First, the identifying tags in TEST.OUT are read one at a time. Then the associated file information is removed from the Prolog database using **retract**, and the same information is neatly output to a text file:

```
pretty_out(In,Out):-
   repeat,
   readint(A),
   retract(file(File,Ext,_,Size,
                Path,Hour,Min,Year,
                Mo,Day,A)),
   writef("%-8%-4 %6.0f %02:%02",
          "%02-%02-%4 %-24\n",
           File,Ext,Size,Hour,Min,
           Mo,Day,Year,Path),
   writedevice(screen),
   write("%"),
   writedevice(Out),
   eof(In).
```

When we've reached the end of TEST.OUT, no clauses remain in the database and control is passed back to the **goal**. The program can button up the output file and delete TEST.DAT and TEST.OUT. We must take care to reassign the read and write devices back to the **keyboard** and **screen,** respectively.

```
readdevice(keyboard),
writedevice(screen),
closefile(f1),
closefile(f2),
deletefile("TEST.DAT"),
deletefile("TEST.OUT"),
...
```

### FINDMATCH AND THE TURBO PROLOG TOOLBOX

The **univdir** predicate depends on the **findmatch** tool predicate from the Turbo Prolog Toolbox (located in BIOS.PRO on the Tools disk). However, there is one quirk in working with **findmatch** straight off the distribution disk.

The problem is that **findmatch** evaluates and returns file sizes as **integers**. Because the range for integer values is between -32768 and 32767, this limits the file sizes that **findmatch** can accurately report.

LISTING 1: RDSRT.PAS

```
program Read_Sort_Write_Sequential_File;

type
    FileName = record       (*  FileName is an 18-byte record *)
                     Filenm : array[1..12] of char;
                     Index  : array[1..4] of char;
                     CR     : byte;
                     LF     : byte;
                 end;

    DirectoryFile = file of FileName;

var
    Nom     : array[1..18] of char;             Results  : integer;

{$I SORT.BOX }

procedure Inp;
var
   rec : integer;

   f   : text;
begin
     ClrScr;
     writeln('Preparing to sort file names.'); writeln;
     writeln('Collecting file names...');     writeln;
     Assign(f,'TEST.DAT');
     {$I-}          Reset(f);      {$I+}

     if IOresult <> 0 then begin
                          writeln('Not there.');
                          end;
     rec := 0;
     repeat
           rec,:= rec + 1;
           write(#13, rec:6);
           readln(f,Nom);
           SortRelease(Nom);
     until EOF(f);

     writeln(' file names collected');    writeln;
     writeln('Now sorting...(this might take a few minutes).');
end;


function Less;
var
   FirstObject :  FileName absolute X;
   SecondObject:  FileName absolute Y;
begin
     Less := FirstObject.Filenm < SecondObject.Filenm;
end;
```

On the surface, it seems easy enough to fix this problem: just specify the file size to be **real** in both **findmatch** and the helping predicate **convert__match**. Having done this, however, you'll notice that the sizes of some files are reported correctly, the sizes of others are reduced, while some file sizes still have negative values.

The culprit is within the **DTA__word** predicate used in **convert__match**. **DTA__word** relies on the built-in predicate **memword**, which returns an **integer** corresponding to the specified segment and offset in memory. If the most significant bit of the word at that location in memory is set, the **integer** value is considered negative.

To solve this dilemma, I've created **adjust**, which guarantees that **convert__match** gets numbers in the correct format for the calculation of file size. Here's how it is defined:

```
predicates
   adjust(integer,real)

clauses
   adjust(I,R):-
      I < 0,
      R = I + 65536.0,
      !;
      R = I,!.
```

Recall that an integer is a 16-bit quantity. If the high-order bit is set, the integer is considered to be a negative number in the range -32768 to -1. To convert this value to a number from 32768 to 65535, we add 65536.0. By adding a **real** number (instead of just 65536) to the **integer** value, the result **R** is stored as a **real**. So, the conversion from **integer** to **real** is made automatically.

The corrected portion of the code has been highlighted in Listing 3 so that you can easily modify your toolbox code.

## TOOLBOX TO TOOLBOX

The technique of writing data to a file and then using the **system** predicate to efficiently process that data is simple, yet powerful. Although no single language can aspire to be all things to all programmers, the Borland family of language products and toolboxes offers an impressive arsenal of code for the programmer seeking results. The principle behind this method can easily be applied to situations where heavy number-crunching is required in a Prolog program, or perhaps, in situations where pattern matching or symbolic manipulation is needed inside a Pascal or C program. ■

---

*Alex Lane is a knowledge engineer living in Jacksonville, Florida. He is the moderator of the Prolog conference on the Byte Information Exchange (BIX). Direct your correspondence to him at 1873 Bartram Road, Jacksonville, FL 32207.*

---

*Listings may be downloaded from CompuServe as TSORT.ARC*

```
procedure OutP;
var
   i     : integer;
   Thing : FileName;
   g     : text;
begin
     writeln;
     writeln('Writing temporary disk files...');
     Assign(g,'TEST.OUT');
     Rewrite(g);
     repeat
           SortReturn(Thing);

           writeln(g,Thing.Index);
     until SortEOS;
     close(g);

end;

procedure DisplayResults(results : integer);                            }
begin
  Writeln;
  Writeln;
  case Results of                          { display sort results
     0 : Writeln('Returning to main program.');
     3 : Writeln('Error:  not enough memory to sort');
     8 : Writeln('Error:  illegal item length.');
     9 : Writeln('Error:  can only sort ', MaxInt, ' records.');
    10 : Writeln('Error:  disk full or disk write error.');
    11 : Writeln('Error:  disk error during read.');
    12 : Writeln('Error:  directory full or invalid path name');
  end; (* case *)
end; (* DisplayResults *)



begin
     Results := TurboSort(SizeOf(FileName));
     DisplayResults( Results );
end.
```

LISTING 2: UNIVDIR.PRO

```
/*   UNIVDIR.PRO

     This routine collects a database of all files in
     all subdirectories on a disk, sorts the filenames, and
     then uses the result to output a formatted, sorted list of
     files to the screen and an ASCII file.

     Copyright 1987, by Alex Lane

     This program uses modified routines from the Turbo Prolog
     Toolbox, which is Copyright 1987, Borland International.

*/


domains
   file = f1; f2
   charlist = char *
```

*continued on page 96*

```
database

   file(string,string,integer,real,string,integer,integer,real,
        integer,integer,integer)
   path(string) /* used to build a pseudo-stack of subdirectories */
   t(integer)  /* this functor provides a 'tag' identification */

predicates
   univdir(string)
   post_or_record(string,integer,integer,integer,real,integer,
                  integer,real,string)
   tag(integer)
   tokenize_filename(string,string,string)
   list_text(string,charlist)     /* (i,o) */

   list_text1(string,charlist,string)
   append(charlist,charlist,charlist)
   pretty_out(file,file)
   repeat

goal
   makewindow(1,7,7,"Hard disk directory list sorter",0,0,25,80),
   asserta(t(1)),  /* initialize the tag index to 1 */

/* The file TEST.DAT will contain 18-byte records consisting of
   a 12-byte file name, a 4-digit tag identification, and a
   carriage-return-line-feed comhbination
*/
   openwrite(f1,"TEST.DAT"),

/* the argument to univdir() specifies where to START the
   directory search.  Examples:
   "" - searches the default disk
   "C:" - searches the C: disk (typically the hard disk)
   "\\PROLOG" - searches the subdirectory \PROLOG on the
                default disk (recall that the '\' character
                must appear twice in a string since it is the
                escape character).

*/
   univdir("C:"),
   closefile(f1),
   retract(t(_)),  /* this fella's job is done */

/* we now make a system call to DOS, which will run the
   program RDSRT.COM for us.  RDSRT.COM already expects to find
   a file called TEST.DAT in the default directory, and after
   processing, will leave a file called TEST.OUT for us to use
   when control returns to this program.
*/
   system("RDSRT"),
```

```
   clearwindow,
   openread(f1,"TEST.OUT"),

/* we will write the sorted hard disk file information to the
   file MYFILES.TXT on the default disk.
*/
   openwrite(f2,"MYFILES.DAT"),
   readdevice(f1),
   writedevice(f2),
   pretty_out(f1,f2),
   readdevice(keyboard),
   writedevice(screen), /* restore the console devices for i/o */
   closefile(f1),
   closefile(f2),
   deletefile("TEST.DAT"),
   deletefile("TEST.OUT"), /* clean up a little before exiting */
   write("\n\nDone. The sorted file listing is in MYFILES.DAT").

/* FINDMATC.PRO is a modified excerpt of the Turbo Toolbox file
   BIOS.PRO Modifications to support file sizes greater 32767.
*/
include "findmatc.pro"

clauses

/* The univdir() predicate is the workhorse of the the program.  It
   calls the Toolbox predicate findmatch() to retrieve filenames. If
   the file name is the name of a subdirectory, the predicate
   post_or_record() asserts the subdirectory name using the path()
   functor. Otherwise, the filename is prepared for sorting and
   asserted through the file() functor.
*/
   univdir(Path) :-
      concat(Path,"\\*.*",SearchSpec),
      findmatch(SearchSpec,63,
               Filename,FileAttr,FileH,FileM,FileY,
               FileMo,FileD,FileSize),
      Filename <> ".",  /* ignore the DOS . and .. file entries */
      Filename <> "..",
      post_or_record(Filename,FileAttr,FileH,FileM,FileY,FileMo,
                     FileD,FileSize,Path),
      fail.  /* we use fail here to force backtracking to
               findmatch() */

/* Once we've exhausted the entries in any given directory, this call
   to univdir() sets us down another path to a new subdirectory by
   retracting a path name through the path() functor.
*/
   univdir(_):-
      retract(path(NewPath)),
      write("!"),
      univdir(NewPath).

/* if we've run out of entries and out of path names, we're done
   scanning the directory on the disk.
*/
   univdir(_):- !.

   post_or_record(Name,16,_,_,_,_,_,_,Path):-
      concat(Path,"\\",Halfway),
      concat(Halfway,Name,NewPath),
      asserta(path(NewPath)).
   post_or_record(Name,Attr,Hour,Min,Year,Mo,Day,Size,Path):-
      Attr <> 16,
      write("#"),
      tokenize_filename(Name,File,Ext),
      writedevice(f1),
      tag(A),
      writef("%-8%-4%4\n", File,Ext,A),
      writedevice(screen),
    assertz(file(File,Ext,Attr,Size,Path,Hour,Min,Year,Mo,Day,A)),
    !.

/*  I'd like to extract the file and extension parts of the name so
    as to be able to uniformly sort on an 8-char file and 3-char
    extension, with both the file and extension flush left. In other
    words, instead of having names like:
        C.BAT
        FOO.BAR
        DISKDOPE.C
        README
    I'd like to have:
        C       .BAT
        FOO     .BAR
        DISKDOPE.C
        README  .
*/
   tokenize_filename(Name,File,Ext) :-
      list_text(Name,Namelist),
      append(A,['.'|T],Namelist),
      B = [ '.' | T ],
      list_text(File,A),
      list_text(Ext,B).

   tokenize_filename(Name,Name,".").

   list_text("",[]) :- !.
   list_text(String,[H|T]) :-
      bound(String),
      frontchar(String,H,Rest),
      list_text(Rest,T).
   list_text(String,[H|T]) :-
      bound(H),
      list_text1("",[H|T],String).

   list_text1(A,[],A) :- !.
   list_text1(A,[H|T],Out) :-
      str_char(Hs,H),
      concat(A,Hs,AHs),
      list_text1(AHs,T,Out).

   tag(N) :-
      retract(t(A)),
      N = A + 1,
      asserta(t(N)), ! .

   append([],L,L).
   append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

/* The pretty_out() predicate lets me output a nicely formatted list
   of file information.
*/
   pretty_out(In,Out) :-
      repeat,
      readint(A),
      retract(file(File,Ext,_,Size,Path,Hour,Min,Year,Mo,Day,A)),
      writef("%-8%-4  %6.0f   %02:%02   %02-%02-%4   %-24\n",
            File,Ext,Size,Hour,Min,Mo,Day,Year,Path),
      writedevice(screen),
      write("%"),
      writedevice(Out),
      eof(In).

   repeat.
   repeat:- repeat.
```

LISTING 3: FINDMATC.PRO

```
/*******************************************************************

    Turbo Prolog Toolbox
    (C) Copyright 1987 Borland International.

(Corrections 10/87 by a.lane)


Returning all matching files to a file specified by backtracking.

Ex: FindMatch("c:\\prolog\\*.PRO",SearchAttribute,MatchFileName,
              FilesAttribute,Hour,Min,Year,Month,Day, FilesSize)


 Ex: FindMatch("c:\\*.*",63,MatchFileName,FilesAttribute,Hour,Min,
              Year,Month,Day,FilesSize)

Range for attributes remembering it is a bitmask
Attributes      =       0       Search for ordinary files
                =       1       File is read only
                =       2       Hidden file
                =       4       System file
                =       8       Volume label
                =       16      Subdirectory
                =       32      Archive file (used by backup & rest.)

Hour            =       0-23    Hour of day when the file was created
Min             =       0-59    Minutes
Year            =       1980-2099

Month           =       1-12
Day             =       1-31
FilesSize       =       0-30MB  Size of file

*******************************************************************/

PREDICATES
  nondeterm FindMatch(String,Integer,String,Integer,Integer,Integer,
                      Real,Integer,Integer,Real)
/* NOTE: Last parameter in FindMatch changed from Integer to Real
     10/87 a.lane
*/
  nondeterm findfiles(STRING,INTEGER)
  nondeterm findnext
  convert_Match(String,String,Integer,Integer,Integer,Real,Integer,
               Integer,Real)
/* NOTE: Last parameter in convert_Match changed from Integer to Real
        10/87 a.lane */
```

*continued on page 98*

*continued from page 97*

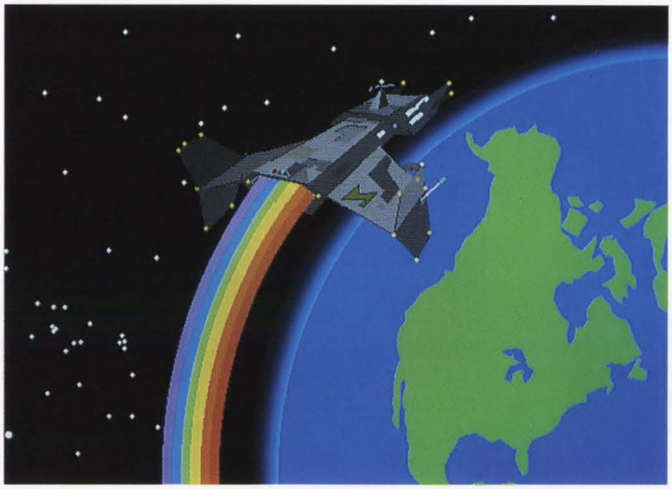
```
  DTA_word(String,Integer,Integer)
  FrontChar2(String,Char,String)
  isolate_bits(Integer,Integer,Integer,Integer)
  adjust( integer, real)

CLAUSES
  FindMatch(FileSpec,Attribute,
  FileName,FilesAttr,Hour,Min,Year,Month,Day,FilesSize):-
        /* Allocate Default Disk buffer area */
     str_len(DTA,128),
     ptr_dword(DTA,DTA_SEG,DTA_OFF),
     AX = $1A00, DS=DTA_SEG, DX=DTA_OFF,
     bios($21, reg(AX,0,0,DX,0,0,DS,0),_),
     findfiles(FileSpec,Attribute),
     convert_Match(DTA,FileName,FilesAttr,Hour,Min,Year,Month,Day,
                   FilesSize).

  findfiles(FileSpec,Attribute):-
     ptr_dword(FileSpec,FSPEC_SEG,FSPEC_OFF),
     bios($21, reg($4E00,0,Attribute,FSPEC_OFF,0,0,FSPEC_SEG,0),_),
     findnext.

  findnext.
  findnext:-
        bios($21, reg($4F00,0,0,0,0,0,0,0),reg(AX,_,_,_,_,_,_)),
        AX=0,
        findnext.


  convert_Match(DTA,FileName,FilesAttr,Hour,Min,Year,Month,Day,
              FilesSize):-
        DTA_word(DTA,21,FAttr), bitand(Fattr,255,FilesAttr),
        DTA_word(DTA,22,FilesTime), bitand(FilesTime,63,Min),
        isolate_bits(FilesTime,6,31, Hour),
        DTA_word(DTA,24,FilesDate), bitand(FilesDate,31,Day),
        isolate_bits(FilesDate,5,15,Month),
        isolate_bits(FilesDate,9,127,Year1),Year=Year1+1980,
        DTA_word(DTA,26,LowSize),
        DTA_word(DTA,28,HighSize),
/* LowSize and/or HighSize may be returned as negative numbers if
   they are larger than 32767 (they are, after all, integers). The
   adjust() predicate turns 'em into reals for us.   a.lane
*/
        adjust(LowSize,LS),
        adjust(HighSize,HS),
        FilesSize=LS+1024.0*64.0*HS,
        ptr_dword(DTA,DTA_SEG,DTA_OFF),
        NEW_OFF = DTA_OFF+30,
        ptr_dword(FileName1,DTA_SEG,NEW_OFF),
        concat(FileName1,"",FileName).  /* Create a copy */
```

```
/*****************************************************************
       Return a word from the DTA area
*****************************************************************/

  DTA_word(DTA,OFF,WORD):-
       ptr_dword(DTA,DTA_SEG,DTA_OFF),
       NEW_OFF = DTA_OFF+OFF,
       memword(DTA_SEG,NEW_OFF,WORD).


/*****************************************************************
       Special version of frontchar
*****************************************************************/

  FrontChar2(S,C,S2) :- FrontChar(S,C,S2),!.
  FrontChar2(S,'\000',S2) :-
       ptr_dword(S,S_SEG,S_OFF),
       S2_OFF=S_OFF+1,
       ptr_dword(S2,S_SEG,S2_OFF).

  isolate_bits(Word,ShiftFac,BitMask,V) :-

       bitright(Word,ShiftFac,V1),
       bitand(V1,BitMask,V).

/*****************************************************************

/*****************************************************************

  adjust(integer, real)

       (added 10/87 by a.lane )

Recall that an integer is a 16-bit quantity.  If the high-order bit
is set, the integer is  considered to be a negative number in the
range -32768 to -1. To convert this value to a number from 32768 to
65535, we add  65536.0 (the '.0' part makes sure we add a real
number, else we'd be effectively adding  zero!)
*****************************************************************/

adjust( I, R ) :-

     I < 0,
     R = I + 65536.0,
     !;
     R = I,!.
```

# New! Introducing Turbo C 1.5— the best optimizing compiler gets even better!

### *The professional optimizing compiler for less than $100*

Turbo C® is a technically superior production-quality compiler. (Borland's equation solver, Eureka™, is written in Turbo C.) And our Turbo C 1.5 offers a new library of the highest presentation-quality graphics in the industry—the kind you'll see in Quattro,™ our new professional spreadsheet.



Actual photograph of Turbo C graphics displayed on IBM 8514 screen.

*Turbo C 1.5 gets you into great pictures and adds dazzling new features*

And spectacular graphics are just part of the brand-new features. Turbo C 1.5 enhancements also include:

- 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

For professional-quality C at an affordable price, no one else comes close to Turbo C. Because no one can deliver technical superiority like Borland.

*60-Day Money-back Guarantee**

For the dealer nearest you or to order, call

**(800) 543-7543**

BORLAND
INTERNATIONAL

***Minimum system requirements:*** For the IBM PS/2™ and the IBM® and Compaq® families of personal computers and all 100% compatibles. PC-DOS (MS-DOS®) 2.0 or later. 384K.

*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 1165

## *It's easy to upgrade to Turbo C 1.5!*

Just complete this coupon and mail it with payment before June 30, 1988. Or, call us at (800) 543-7543 and be ready to give our operators your name, credit card number, and the serial number on your Turbo C master disk.

***Turbo C 1.5 Upgrade Price*** ***$ 33.50***

CA and MA residents add sales tax ______________

Shipping and handling
In US $5.00 (Outside US add $10) ______________

Total amount enclosed $ ______________

***Must include your Turbo C serial #*** ______________________________

Return this coupon and the Turbo C RTL source code registration form from your Turbo C manual along with your payment by March 31, 1988 and receive your Turbo C 1.5 upgrade for free! (No phone orders please.)

***Turbo C 1.5 Runtime Library Source Code*** ***$ 150.00***

CA & MA residents add sales tax ______________
Price includes shipping to all US cities.
(Outside US add $10) ______________

Total amount enclosed $ ______________

Please specify diskette size □ 5¼" □ 3½"

Method of Payment: □ VISA □ MC □ Check □ Bank Draft

Credit card expiration date: ________ / ________

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

Name ______________________________

Ship Address ______________________________

City ______________ State ______________

Zip ______________ Phone ( ____ ) ______________

***Mail coupon to:*** Turbo C 1.5 Upgrade Dept., Borland International
4585 Scotts Valley Drive, Scotts Valley, CA 95066

This offer is limited to one upgrade per valid product serial number. Not good with any other offer from Borland. Outside US make payments by bank draft payable in US dollars drawn on a US bank. CODs and purchase orders will not be accepted by Borland.

# USING RANDOM FILES IN TURBO BASIC

## Master Turbo Basic's random files and be outstanding in your FIELD.

*Ethan Winer*

PROGRAMMER

If you were to ask most BASIC programmers which area of programming was the most difficult for them to master, the answer would undoubtedly be creating and accessing random data files. While BASIC has always enjoyed a reputation for being the easiest of the high-level languages to learn, there is no disputing that the commands for manipulating database files are often less than obvious. Indeed, because part of my work is supporting a line of BASIC enhancement products, this is one of the topics I am most frequently asked to explain.

The focus of this article, therefore, is on the variety of techniques that are used to create, read, and write disk files with fixed-length records. Unlike simple sequential files that are accessed with **INPUT** and **PRINT** statements, the fixed-length files used in most databases require additional preparation.

### FILE BASICS

Before you can read or write any disk file in Turbo Basic, you must first use the **OPEN** command. There are actually two different forms of **OPEN**—one being an abbreviated version—but we will use the more formal syntax here because it's clearer. A valid DOS filename must be given, as well as a number that will be used for all subsequent references to the file. You may choose any number you'd like, as long as only one file with that number is open at one time.

The example shown in Figure 1 opens a file named TEST.DAT for sequential output, assigns it the number 1, and writes a 25-line test message. Notice that once the name and number have been originally specified, only the number is needed when printing to the file.

```
OPEN "test.dat" FOR OUTPUT AS #1       'open the file
FOR x = 1 TO 25                    'print the messages
  PRINT #1, "This is message number" x
NEXT
CLOSE #1                                'close the file
```

*Figure 1. Sequential files are written using the familiar* **PRINT** *command.*

Because this file is being written to with **PRINT** statements, a carriage return and line feed will be added at the end of each message line. This corresponds exactly to the way the **PRINT** command normally works when writing to the screen. As you might expect, sequential files are read using a disk form of the **INPUT** statement, and the carriage return/line feed pair that was originally added by **PRINT** tells Turbo Basic when the end of each line has been reached.

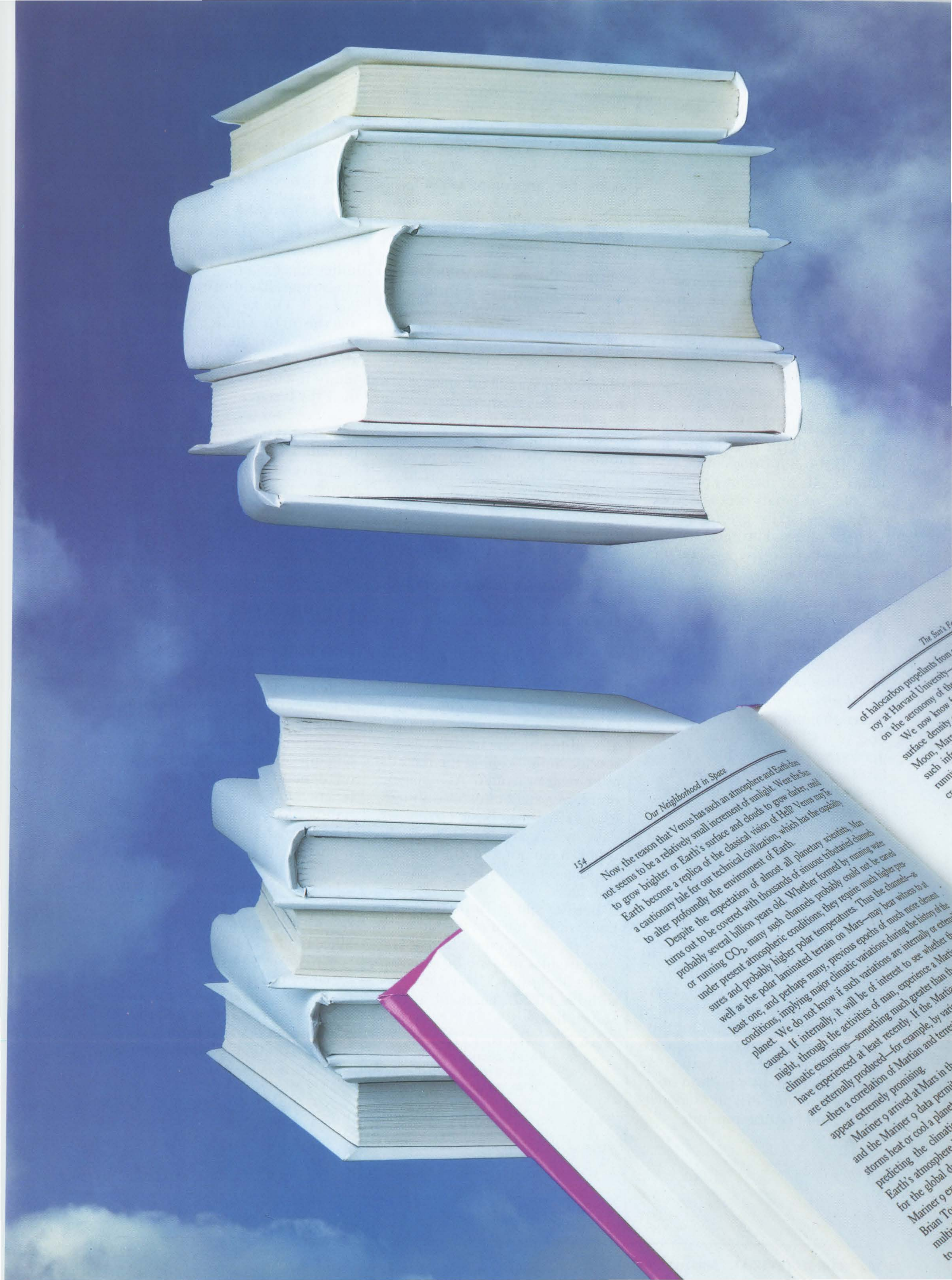
```
OPEN "test.dat" FOR INPUT AS #1    'open the file
FOR x = 1 TO 25
    LINE INPUT #1, test$           'read each line
    PRINT test$                    'show it on screen
NEXT
CLOSE #1                           'close the file
```

*Figure 2. Reading a sequential file is done with* **INPUT** *or* **LINE INPUT**.

The code in Figure 2 improves on Figure 1 by using **LINE INPUT** rather than **INPUT**, because **INPUT** cannot digest quotes, commas, or colons. In this example, we know that 25 lines are in the file, so a simple **FOR/NEXT** loop can be used to read each line. But what should we do if the number of lines to be read is unknown? Attempting to read beyond the end of a sequential file produces an error, and the best way around this problem is to use the **EOF** (End of File) statement just prior to reading each line, as shown in Figure 3.

*continued on page 102*

*Bradley Ream*



The Sun's F

of halocarbon propellants from
roy at Harvard University—
on the aeronomy of th
We now know
surface density
Moon, Mar
such inf
runni
cr

154 Our Neighborhood in Space

Now, the reason that Venus has such an atmosphere and Earth does
not seems to be a relatively small increment of sunlight. Were the Sun
to grow brighter or Earth's surface and clouds to grow darker, could
Earth become a replica of the classical vision of Hell? Venus may be
a cautionary tale for our technical civilization, which has the capability
to alter profoundly the environment of Earth.
Despite the expectation of almost all planetary scientists, Mars
turns out to be covered with thousands of sinuous tributaried channels
probably several billion years old. Whether formed by running water
or running $CO_2$, many such channels probably could not be carved
under present atmospheric conditions; they require much higher pres
sures and probably higher polar temperatures. Thus the channels—as
well as the polar laminated terrain on Mars—may bear witness to at
least one, and perhaps many, previous epochs of much more clement
conditions, implying major climatic variations during the history of the
planet. We do not know if such variations are internally or extern
caused. If internally, it will be of interest to see whether th
might, through the activities of man, experience a Marti
climatic excursions—something much greater than th
have experienced at least recently. If the Martia
are externally produced—for example, by vari
—then a correlation of Martian and terre
appear extremely promising.
Mariner 9 arrived at Mars in th
and the Mariner 9 data permi
storms heat or cool a plane
predicting the climati
Earth's atmosphere
for the global d
Mariner 9 ex
Brian To
multi
to

## RANDOM FILES

*continued from page 100*

```
OPEN "test.dat" FOR INPUT AS #1
WHILE NOT EOF(1)
    LINE INPUT #1, test$
    PRINT test$    'to the screen
WEND
CLOSE #1
```

*Figure 3. Using BASIC's* **EOF()** *function to determine the end of a file.*

This is the preferred approach for reading sequential files, since it can accommodate any number of lines without ever causing an error. But there are two fundamental problems with sequential files—reading them is relatively slow, and the only practical way to get to the last line is by grinding through all of the lines before it. True, Turbo Basic does offer a binary mode to read any portion of a file, but binary access is not really intended for getting individual lines of text. Further, how would you know where in the file the last line begins?

### ENTER RANDOM ACCESS

When a sequential file is being read with **INPUT** or **LINE INPUT**, Turbo Basic must examine every single byte as it comes from the disk, looking for either a carriage return that marks the end of the line, or a **CHR$(26)** (Ctrl-Z) that marks the end of the file. This takes a considerable amount of time when many lines are to be read. Worse, for each string that is being read, additional time is needed to locate a suitable place in memory to store it. Then, the string data must be placed there, and finally that portion of memory must be marked as being in use.

Random access files are instead read (and written) in chunks, which can quickly be placed into an area of memory that has previously been set aside just for this purpose. Instead of checking each byte as it is being read, Turbo Basic simply grabs a portion of the file from the disk in one operation. And because random access files are comprised of fixed-length records, it is easy to determine where in the file a given record begins.

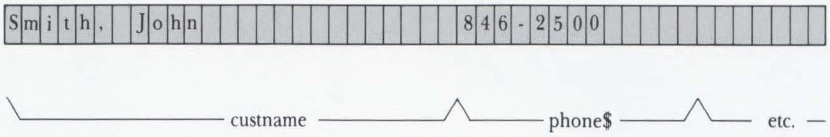


*Figure 4. A random field buffer is comprised of separate strings.*

Of course, you don't have to calculate the location of the bytes to be read or written—Turbo Basic does this automatically. By specifying the length of each record when the file is first opened, any record can be accessed by simply using its number. This is where the term "random" comes from, since *any* record can be accessed at random in the file, without having to start at the very beginning and read past every record before it. Another feature of random files is that once they have been opened, they can be both read and written at will.

The example below opens a file named JUNK.DAT for random access, and specifies that each record have a length of 56 bytes.

```
OPEN "junk.dat" FOR RANDOM _
   AS #1 LEN = 56
```

When this statement is executed in a running program, a 56-byte area of memory is set aside as a storage buffer to hold the records to be read or written. As a matter of interest, file buffers are always located in Turbo Basic's string data area. Again, the file buffer is simply a temporary holding area for data on its way to or from a disk file.

Besides telling Turbo Basic the name of the file and the length of each record, you must also define the variables that will be assigned to the file buffer. This is done with the **FIELD** statement, as shown below.

```
FIELD #1, 25 AS cust.name$,      _
          14 AS phone$,          _
           5 AS zip.code$,       _
           4 AS amount$,         _
           4 AS balance.due$,    _
           2 AS account$,        _
           2 AS price.code$
```

Once these field assignments have been made, the original buffer space is divided into separate strings, like those shown in Figure 4. The key point here is that the 56-byte buffer memory is set aside when the file is opened, and the positioning of the strings within the buffer is determined by the **FIELD** statement.

Notice how the underscore character is used to continue what is really a single **FIELD** statement over several program lines to increase readability. Also notice that even though some of the variables being defined are in fact numeric amounts, they must be represented here as strings. Let's take a closer look at this issue.

### VARIABLE STORAGE

Whenever you assign a string variable, Turbo Basic must locate a free area of memory sufficiently large to hold all of the string variable's characters. Of course, short strings occupy less memory than long strings; the point is that the amount of required memory varies, depending on the string's length. Numeric variables are handled very differently, though. They use a fixed number of bytes no matter what value the variable happens to contain. For example, all double-precision numeric variables occupy eight bytes of memory, while regular integer variables require only two.

Whenever you enter a numeric variable in response to an **INPUT** command, Turbo Basic must convert the digits you type into the appropriate internal format, which takes time. This contributes to the slowness of reading or writing sequential files, because in sequential files, numeric values are stored on disk as ASCII digits.

There is a problem inherent in writing ASCII digits out to disk to represent numeric values. There are three digits in the ASCII representation of the value 100, but only one digit in the ASCII representation of the value 6. Thus, the physical size of the representation changes depending on the value being represented.

This does not jive with Turbo Basic's requirement that each record of a random file be the same length as every other record. To keep all instances of any numeric type the same physical size, numeric values are instead represented on disk in Turbo Basic's internal numeric formats. In this format, the values 6 and 100 each occupy exactly two bytes—the same physical size as any other integer value throughout the integer range. Using a number's "native" internal format in a disk file also means that numbers read in from a random file can be processed very quickly, because the conversion from ASCII digits to a binary value has already been performed when the record was originally written to disk.

But remember that random file buffers are allocated and maintained in string space. In a sense, the buffer is a collection of strings. What we need to do, then, is to somehow convert a numeric value into a string containing the character equivalents of the numeric value's internal binary representation. This does *not* mean that we convert a binary integer value into a printable string like "100" or "6"! The string form of a binary numeric value may or may not be printable; in a sense, the string form is just the binary bytes of the numeric value with a string-descriptor "coating" to make them palatable to Turbo Basic's string-handling machinery.

Four different commands are provided to convert each of the numeric types to this string form, and another four convert the string form back to numeric values of specific types.

**MKI$** (Make an Integer into a String) takes the two bytes that represent an integer variable and converts them into string form. For example, the integer value 288 is stored internally by Turbo Basic as two bytes—32 and 1—as illustrated below.

```
x% = 288
address = VARPTR(x%)
DEF SEG = VARSEG(x%)
PRINT PEEK(address), _
  PEEK(address + 1)
PRINT PEEK(address) + _
  256 * PEEK(address + 1)
```

This results in the numbers 32 and 1 being displayed, followed by the total combined integer value 288. **MKI$** locates the integer variable **x%**, and creates a string from the two individual bytes comprising **x%**. Both of the Turbo Basic statements below do exactly the same thing, although as you can see, **MKI$** is much terser and simpler to use:

```
Value$ = MKI$(x%)
Value$ = CHR$(PEEK(address)) + _
  CHR$(PEEK(Address + 1))
```

The method used to store single- and double-precision numbers is more complicated; we won't bother with the exact formulas. It is only important to remember that the floating point conversion routines operate the same way. To create a string variable from a single-precision number requires the **MKS$** function, while **MKD$** does the same thing for a double-precision amount. The last converting function is **MKL$**, and it converts Turbo Basic's long integers to a string. Once the values have been converted to string form, they may then be assigned to the variables that were declared as part of the field statement.

In the example above, **Value$** was assigned directly using the **MKI$** function. Unfortunately, normal string assignments cannot be used when filling the variables in a field buffer, because of the way Turbo Basic allocates string memory. Each time a string is assigned—even if it was defined earlier—a new area of memory is set aside to hold it. But since the variables kept in a field buffer must stay in the same place, we can't use the normal assignment statements. Remember, one of the reasons random files are so fast is because their buffer memory location is fixed when the file is first opened.

### LSET, RSET, AND MID$

Turbo Basic provides three commands that let you assign a string that already exists without changing its location—**LSET**, **RSET**, and the statement form of **MID$**. All of these work by letting you replace characters within a string, as opposed to creating an entirely new string. Let's take a closer look at each of these statements.

If you are designing a database program to keep track of, perhaps, the names and addresses of your customers, at some point you must decide how many characters are to be allowed for each field. I usually limit address fields to 32 characters, because that's how many can comfortably fit on a standard 3 1/2-inch-wide mailing label.

Using that as an example, suppose an address occupies only 20 characters. What you'd really like is to place the name into the first 20 character positions in the string, and then pad the remaining 12 places with blanks. This is precisely what **LSET** does. It's important to understand that if the trailing positions in a field are *not* blanked out, then any remnants left over from a prior read or write will still be present. Again, the same area of memory is used repeatedly for all file reads and writes.

**RSET** is similar to **LSET**, except it *right* justifies the new string into the existing field variable. With either **LSET** or **RSET**, attempting to insert a string that is too large assigns only as many characters as will fit, without creating an error.

Notice that in addition to their intended use for assigning field variables, **LSET** and **RSET** can also be used to advantage in many other programming situations. Since new space isn't established each time the string is assigned, these commands work very quickly, while minimizing the clutter that normally occurs in the string data area.

*continued on page 104*

# RANDOM FILES

*continued from page 103*

Most programmers use **MID$** to extract a portion of a string, but it can also be used to insert characters, much like **LSET** and **RSET**. But where **LSET** and **RSET** fill any unused character positions with blanks, **MID$** instead leaves them undisturbed. The syntax for using **MID$** to assign characters is the same as when it is used to extract characters; you specify the starting position in the string, as well as the number of characters to include. Like **LSET** and **RSET**, if you attempt to replace too many characters in a string with **MID$**, those that don't fit will be omitted without causing an error. By the way, the **MID$** length parameter is optional, and if it's omitted, all of the characters through the end of the string will be included.

## PUTTING IT ALL TOGETHER

Now that we've seen the individual steps needed to prepare a field buffer, let's put it all together into a single program. The example in Figure 5 first opens the file and defines all of the string variables that comprise the field buffer. Next, each variable is assigned a value, and then the information is written to a disk record. The last step reads the *next* record from the file, and reassigns its contents to the original variables for display.

```
cname$      = "Quux, Sam"               'make up some data
phon$       = "(408)-438-8400"
zip$        = "12345"
amt!        = 102.45
bal.due!    = 398.77
acct%       = 158
prc.code% = 32

record.number = 123

OPEN "stuff.dat" FOR RANDOM AS #1 LEN = 56   'open the file
FIELD #1, 25 AS cust.name$,    _             'set up fields
          14 AS phone$,        _
           5 AS zip.code$,     _
           4 AS amount$,       _
           4 AS balance.due$,  _
           2 AS account$,      _
           2 AS price.code$

LSET cust.name$   = cname$              'assign field variables
LSET phone$       = phon$
LSET zip.code$    = zip$
LSET amount$      = MKS$(amt!)
LSET balance.due$ = MKS$(bal.due!)
LSET account$     = MKI$(acct%)
LSET price.code$  = MKI$(prc.code%)

PUT #1, record.number                   'write to record #123
record.number = record.number + 1       'point to next record
GET #1, record.number                   'read it from disk

PRINT cust.name$
PRINT phone$
PRINT zip.code$
PRINT CVS(amount$)
PRINT CVS(balance.due$)
PRINT CVI(account$)
PRINT CVI(price.code$)

CLOSE
```

*Figure 5. Using* **GET** *and* **PUT** *to read and write random files.*

Two new commands have been introduced here: **GET** and **PUT**. Unlike their graphics counterparts (which have no relation to these file versions), **GET** and **PUT** are used to read and write disk records. Besides indicating which file number is to be read or written, you also specify which record number to operate on. In truth, the record number is an optional parameter, and if it's left out, Turbo Basic will default to the next one in sequence. Personally, I always include an explicit record number, just to eliminate any possibility of a mix-up.

Two other new commands being used are **CVI** and **CVS**, which complement **MKI$** and **MKS$**, respectively. Where **MKI$** obtains the two bytes that comprise an integer value and create a string from them, **CVI** does exactly the opposite. That is, it takes a two-character string and creates an integer value from the characters. **CVD** and **CVL** also convert strings to numbers, with the first intended for double-precision values, and the second for long integers. As in the **MKI$** example shown earlier, normal Turbo Basic commands can imitate the action of **CVI**. Of course, I'm not recommending that you program this way, but in the interest of completeness, here's what **CVI** really does:

```
x% = CVI(value$)
x% = ASC(value$) + 256 _
  * ASC(RIGHT$(value$, 1))
```

## RANDOM FILE TECHNIQUES

Now that we know the essential operations needed to manipulate random access files, let's look at a few real-life situations. After a file has been created and data placed into it, one of the first things you'll want to do is be able to report on that data. For example, you may need to identify all accounts that have had a balance due for more than 30 days. Or perhaps you want the ability to delete records from the file, or provide other reporting options. We're not going to pursue a lengthy discussion of indexing or sorting techniques here; however, a few practical examples come to mind.

First, you need to know how many records are in the file. This is easy to determine by dividing the file size by the record length:

```
OPEN "accounts.dat" FOR RANDOM AS #1 LEN = 125
num.recs = LOF(1) / 125

FIELD #1, 32 AS account.name$, _
          32 AS address$,      _
          25 AS city$,         _
           2 AS state$,        _
           5 AS zip$,          _
          14 AS phone$,        _
           6 AS date.due$,     _
           1 AS paid.yn$,      _
           4 AS last.paymnt$,  _
           4 AS bal.due$       _

today$ = RIGHT$(date$,2) + LEFT$(date$,2) + MID$(date$,4,2)

FOR x = 1 TO num.recs
    GET #1, x
    IF date.due$ <= today$ AND paid.yn$ <> "Y" THEN
       LPRINT account.name$, phone$, CVS(bal.due$)
    END IF
NEXT

CLOSE
```

*Figure 6. A database report that examines every record in the file.*

```
OPEN "my-stuff.dat" FOR RANDOM _
  AS #1 LEN = 87
number.of.records = LOF(1) / 87
```

Notice that **LOF** is a handy way to obtain the length of any file, including .COM or .EXE programs. But be careful to close the file as soon as you get its length, to avoid any possibility of altering it:

```
OPEN "anyfile.ext" FOR RANDOM _
  AS #1 LEN = 1
size! = LOF(1)
CLOSE
```

Most of the data you'll be storing in a random access file will consist of either strings or numeric values. However the best way to store certain information is not always obvious. For example, dates can be represented in a variety of ways. At the minimum, you should omit any separating hyphens or slashes, except when displaying them on the screen. That is, 01/15/88 would be kept on disk in a six-byte field as 011588. But this method does not allow a direct comparison; it is not obvious that 011588 is later than 123187, regardless of whether you use a string or numeric comparison.

A much better approach is to swap the digit pairs around so that the year comes first, followed by the month and day. Since one of the main objectives of a database report is to process the information as quickly as possible, using this technique provides a dramatic improvement. The example in Figure 6 opens a hypothetical disk data file, and then lists the name, phone number, and amount for all accounts that are due but not yet paid.

This example assumes that the field **paid.yn$** contains either a "Y" if the account has already been paid, or an "N" or a blank if it has not. We also assume that the dates in the file were swapped around into a YYMMDD format when each record was written. If the date the account was due is today or earlier, and it has not already been settled, then the name and other information are printed.

Since you'll undoubtedly be coding these date fields many times, this is a natural application for Turbo Basic's multiline user-defined functions. In fact, an even better approach is to pack all dates into only *three* bytes (instead of six) to save disk space, using:
**CHR$(year-1900)**
**CHR$(month)**
**CHR$(day)**.

## DELETING RECORDS

The last item is a method for deleting records from a database. Of course, there's no reasonable way to physically remove a record from a disk file, so our only recourse is to mark it in some way. Many commercial database programs reserve an extra byte in each record for exactly this purpose, however we can eliminate that wasted byte with some clever programming.

Since text fields such as a name or address don't need to accommodate the PC's extended graphics characters, the simplest approach is to convert one of the letters in a name to its corresponding graphic symbol in the "high" 128 bytes of the PC character set. This is accomplished by either adding 128 to the character's ASCII value, or by using the Turbo Basic **OR** function to do the same thing by explicitly setting bit 7 to a binary 1. The example below retrieves the record to be deleted from the file, adds 128 to the ASCII value of the first character in the last name field, and then writes the record back to disk:

```
GET #1, record.number
LSET l.name$ = _
  CHR$(ASC(l.name$) + 128) + _
  MID$(l.name$, 2)
PUT #1, record.number
```

Then, when you're reporting on the file and need to tell if a record was deleted and should not be included, all you have to do is check the ASCII value of the field:

```
GET #1, record.number
IF ASC(l.name$) => 128 _
  THEN ...  'record is deleted
```

## WRAPPING UP

We have looked at a variety of techniques for reading and writing random access disk files, as well as several tips and techniques you can apply in your own programs. While these examples are far from the final word on the subject, I hope they will encourage you to experiment on your own, and further explore one of Turbo Basic's most powerful capabilities. ■

*Ethan Winer owns Crescent Software, and is the author of the QuickPak utilities for Turbo Basic and QuickBASIC.*

# CONVERTING .COM FILES TO $INCLUDE FILES

## Managing Turbo Basic assembly language subroutines is easier if you turn them into text.

*Bruce Tonkin*

WIZARD

What can't be done in Turbo Basic directly can often be done from assembly language. Nearly anything done in assembly language will be faster than equivalent BASIC code. These two reasons form a compelling case for adding assembly language routines to your Turbo Basic programs.

Rather than import .OBJ files directly, Turbo Basic's interface to assembler routines requires the **$INLINE** metastatement. **$INLINE** works in either of two modes:

1. Place the hex values of the machine code instructions (in human-readable "&H" format) into your file after the **$INLINE** metastatement.
2. Specify the name of a .COM file, enclosed in quotes, after the **$INLINE** metastatement. The binary bytes contained in the .COM file are read into the native code program image being generated by Turbo Basic.

This can be handy, but if you use a lot of separate assembly language routines it is more convenient to put them all into a single library file. Your disk will be less cluttered and there will be fewer files to track. However, there's no way to combine .COM files, and Turbo Basic programs won't link with .OBJ files in the standard fashion. What to do?

The simplest way to create library files is to somehow put the routines into a text file, and either include them into your source code with the **$INCLUDE** metastatement, or simply read them into the editor and save them as part of your Turbo Basic source code file. Unfortunately, assembler programs as created according to the instructions in the *Turbo Basic Owner's Handbook* are .COM files, and manually converting a program from the .COM format to the **$INLINE** "&H" text format can be a real chore.

Fortunately, there are ways to avoid most of the effort. The method I provide in this article makes things just about as easy as the "linked module" approach to assembly language code used by QuickBASIC, and it allows you to make libraries of assembler routines with about the same effort you'd need for QuickBASIC.

I will assume for this discussion that you have referred to the *Turbo Basic Owner's Handbook* (see Appendix C, especially) and have created several .COM files from assembly language source code files according to the instructions given there. Let's suppose those two .COM files are called MY1.COM and MY2.COM. Each of them is to be inserted into your program at a specific location.

### FROM BINARY TO TEXT

First, we need to create text files MY1.INC and MY2.INC from their equivalent .COM files. I have written a short program called COM2INC to do that, as shown in Listing 1.

COM2INC.BAS can be compiled to an .EXE file with Turbo Basic and run from the command line as you need it. It will convert the input file to an output file with the .INC file extension. An input filename entered without any file extension is assumed to have a .COM extension. If you forget COM2INC's syntax, or need to use input and output files with different names, you can invoke COM2INC without any command-line parameters and it will prompt you for the input and output filenames.

For example, to convert MY1.COM to MY1.INC, enter the command:

```
C>COM2INC MY1
```

That is sufficient to complete the conversion. Or, if you want to convert MY2.BIN to MY2.HEX, enter the command:

```
C>COM2INC
```

Answer the questions, filling in the complete filenames when asked.

The output file is a DOS text file with a maximum of five hexadecimal values per line. If you'd like more or less than that number, change the **MOD 5** in the **FOR** loop to **MOD** *n*, where *n* is whatever value you'd like. If you want only one value per line, the IF test can be eliminated entirely and the **FOR** loop can run from 1 to **LASTBYTE**; in that case, you can also eliminate the lines:

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

Because of the way the program operates, the first line in the output file will always be blank. The blank line does no harm.

COM2INC has no error checking. If the input file doesn't exist, it will create a zero-length file with the name you specified. If the output file already exists, it will be overwritten. If there's no room on the disk for the output file, an error message will be displayed.

Place the Turbo BASIC metastatement

```
$INCLUDE MY1.INC
```

at the point where you would like to insert the code from MY1.COM. Likewise, insert the line

```
$INCLUDE MY2.INC
```

where you want to insert MY2.COM. Keep in mind that such assembly code include files must be framed within a **SUB..INLINE** and **END SUB** framework.

Since libraries of .INC files are simply text files, they are easy to maintain: to add a routine to a library, just read the .INC file into the library from disk and place it within the required framework:

```
SUB <MYCODE> INLINE
<include the .INC file here>
END SUB
```

and you're done. (You can pick whatever name you like in place of **<MYCODE>**.) To get rid of a routine, just mark it and delete it from Turbo Basic's editor. It's simple!

## A REAL EXAMPLE

Elsewhere in this issue, Juan Jimenez presents an assembly language routine for detecting the CPU type from within a program. Using **GETCPU** from Turbo Basic is not difficult, and provides a good example of the use of **COM2INC**.

From within a BASIC program, **GETCPU** is called this way

```
CALL GETCPU(X%)
```

where **X%** is an integer parameter that returns a value of 86, 186, 286, or 386, depending on which processor is detected.

The assembly language source code for GETCPU.ASM needs some slight modification to work properly with Turbo Basic. The modified assembly language source code file is given in Listing 2, GETCPUTB.ASM. The changes only involve the entry and exit code. I have added the first three instructions:

```
PUSH BP
MOV BP,SP
LES DI,[BP+6]
```

*continued on page 108*

LISTING 1: COM2INC.BAS

```
'  COM2INC.BAS
'  Written in Turbo Basic by Bruce Tonkin on 5/11/87
'
'  This program converts COM files to $INCLUDE text files
'  with the Turbo Basic $INLINE meta-command.  The output
'  files may be inserted or easily $INCLUDEd into
'  Turbo Basic programs.
'
DEFINT A-Z            'All variables will be integers
F$=COMMAND$           'Check to see if there's a command line
WHILE F$=""
    PRINT"This program will convert COM files to $INCLUDE files"
    PRINT"for use with Turbo BASIC.  The default file type of"
    PRINT"the source file is COM.  The default file type of the"
    PRINT"output file is INC.  You may override either default"
    PRINT"by entering a spacific file type specification."
    PRINT"If you enter no name for the output file, it will be"
    PRINT"named the same as the input file, but will have a file"
    PRINT"type specification of INC."
    LINE INPUT"Enter the name of the file to convert: ";F$
WEND

IF COMMAND$="" THEN
    LINE INPUT"Enter the name of the desired output file: ";O$
    END IF

IF INSTR(F$,".")<2 THEN F$=F$+".COM"        'fix input spec
IF O$="" THEN
    O$=LEFT$(F$,INSTR(F$,"."))+"INC"        'fix output spec,
    ELSE
         IF INSTR(O$,".")<2 THEN O$=O$+".INC"    'both ways.
    END IF

OPEN"R",1,F$,1          'input file will be read one byte
FIELD #1,1 AS A$        'at a time into A$
LASTBYTE=LOF(1)         'end of file position
OPEN"O",2,O$            'output file is opened
FOR I=1 TO LASTBYTE-1
    GET 1,I
    X=ASC(A$)
    IF ((I-1) MOD 5=0) THEN PRINT #2,"":PRINT #2,"$INLINE ";
    PRINT #2,"&H";HEX$(X);
    IF ((I-1) MOD 5<>4) THEN PRINT #2,",";
NEXT I
GET 1,LASTBYTE
PRINT #2,"&H";HEX$(ASC(A$))
CLOSE
PRINT"Conversion is complete. ";LASTBYTE;" bytes read."
END
```

LISTING 2: GETCPUTB.ASM

```
   name GETCPU
   page 55,132
   title     'GETCPU.BIN --- Determines which INTEL CPU is installed'
;-------------------------------------------------------------------
; By Juan Jiminez -- Modified for Turbo Basic by Bruce Tonkin
;   Last modified 10/15/87
;
; This program determines which one of the Intel CPU's is being used
; in the machine, whether it is an 8088/86, 80188/186, 80286 or 80386.
; It uses the differences in flag register bit configurations to
; determine whether the CPU is an 80286 or 80386, and the differences
; in shifting using CL to determine if it is an 8088/86 or
; 80188/186.  It returns an integer result in the form of the last
; three digits of the processor type, as depicted in the table below.
;
; If the processor is    The routine returns
; -------------------    -------------------
;      80386                     386
;      80286                     286
;      80188/186                 186
;      8088/86                    86
;
;-------------------------------------------------------------------
; Use of the routine in Turbo BASIC is:
;   CALL GETCPU(X%)
;
; Where GETCPU is an inline subprogram of form:
;
;   SUB GETCPU INLINE
;   $INLINE "GETCPU.BIN"
;   END SUB
;
; Alternatively, GETCPU may be placed inline by means of byte values
; generated with Bruce Tonkin's COM2INC utility.  See text for this
; listing.
;
;-------------------------------------------------------------------
; To assemble:
;
; MASM GETCPU,,,;
; LINK GETCPU,,,;
; EXE2BIN GETCPU.EXE GETCPU.COM
;
;-------------------------------------------------------------------
; Code segment begins here
;-------------------------------------------------------------------
cseg     segment   para public 'CODE'
    assume    cs:cseg,ds:cseg,es:cseg,ss:cseg
    org  100h
;-------------------------------------------------------------------
; Actual id routine begins here
;-------------------------------------------------------------------
getcpu   proc near
    push bp          ; Turbo Basic requires you save the base pointer.
    mov bp,sp        ; Move the stack pointer to bp.
    les di,[bp+6]    ; Offset address of the integer parameter.
```

## CONVERTING .COM

*continued from page 107*

These instructions became necessary because Turbo Basic expects the return value to be passed on the stack rather than in a register, as with Turbo Pascal and Turbo C. **PUSH BP** is required because we will be using BP. The original GETCPU.ASM neither uses nor modifies BP. **MOV BP,SP** loads the value of the stack pointer into BP. **LES DI, [BP+6]** puts the address of Turbo Basic's **X%** parameter into ES : DI. Using this address, we will later write into **X%** the code value indicating the type of CPU.

The exit code required a little enhancing as well. These instructions were added after the **POPF** instruction that restores the flags register from the stack:

```
CLD
STOSW
POP BP
```

The first two instructions store the return value into the integer parameter **X%**. **POP BP** restores Turbo Basic's BP value, which the entry code had pushed onto the stack.

Finally, the **RET** instruction that Turbo Pascal and Turbo C require when returning from machine-code subroutines is commented out. Turbo Basic's **CALL** statement takes care of returning control—the machine-code subroutine merely has to end.

Assemble and link GETCPUTB.ASM, and convert the .EXE file to a .COM file:

```
MASM GETCPU,,,;
LINK GETCPU,,,;
EXE2BIN GETCPU.EXE GETCPU.COM
```

This done, convert GETCPU.COM to an include file with COM2INC:

```
COM2INC GETCPU.COM
```

Listing 3 shows the text file produced by COM2INC for GETCPU.COM. Note that line 1 is blank.

Building the converted include file into a program is easy, as shown in Listing 4, WHATCPU.BAS. The file GETCPU.INC is read into the file and bracketed by the **SUB GETCPU INLINE** and **END SUB** statements.

The 5-up "&H" format used here is fairly compact, if not especially readable. (Readability could be improved by making the listing 1-up and loading the assembly language mnemonics to the right of the opcode bytes, in comments.) No linking or other special processing is necessary. Once your suite of assembly language utilities is proven reliable, you can tuck it into your main source code files without a second thought. ■

*Bruce Tonkin is an independent program developer who insists on using BASIC in preference to other languages. He is the author of the My Word! word processor.*

*Listings may be downloaded from CompuServe as COMINC.ARC.*

```
; These first three instructions have changed from Juan's original
; code.  See the Turbo BASIC manual, page 401, for a small example
; program and explanations of the logic.  Fortunately, the original
; routine didn't use di or the direction flag.
; Turbo BASIC doesn't demand that you save and restore the flags,
; but I will leave that part of the program logic alone--the changes
; I've made will be easier to see that way.
    pushf           ; Save the flag registers, we use them here...
    xor  ax,ax      ; Clear AX and push it onto the stack
    push ax
    popf            ; Pop 0 into flag registers (all bits to 0),
    pushf           ; attempting to set bits 12-15 of flags to 0's
    pop  ax         ; Recover the saved flags
    and  ax,08000h  ; If bits 12-15 of flags are set to zero then
    cmp  ax,08000h  ; cpu is 8088/86 or 80188/86
    jz   _8x_18x
;---------------------------------------------------------------
; It is either an 80286 or an 80386, let's find out which...
;---------------------------------------------------------------
    mov  ax,07000h ; Try to set flag bits 12-14 to 1's
    push ax        ; Push the test value onto the stack
    popf           ; Pop it into the flag register
    pushf          ; Push it back onto the stack
    pop  ax        ; Pop it into AX for check
    and  ax,07000h ; If bits 12-14 are cleared then the chip is
    jz   _286      ; an 80286
;---------------------------------------------------------------
; Ok, we know it's an 80386 now, tell the user about it!
;---------------------------------------------------------------
    mov  ax,386         ; It's not a 286, so it must be a 386
    jmp  DONE
;---------------------------------------------------------------
; Tell the user it is an 80286
;---------------------------------------------------------------
_286:     mov  ax,286         ; Get the msg ready
    jmp  DONE
;---------------------------------------------------------------
; We know it is either an 8088/86 or 80188/86, but which one is it?
;---------------------------------------------------------------
_8x_18x:
    mov  ax,0ffffh ; Set AX to all 1's
    mov  cl,33          ; Now we try to shift left 33 times. If it's
    shl  ax,cl          ; an 808x it will shift it 33 times, if it's
                   ; an 8018x it will only shift one time.
    jnz  _18x      ; Shifting 33 times would have left all 0's.
                   ; If any 1's are left it's an 80188/186
    mov  ax,86          ; No 1's, it's an 8088/86
    jmp  DONE
;---------------------------------------------------------------
; It's an 80188 or 80186...
;---------------------------------------------------------------
_18x:     mov  ax,186     ; Found a 1 in there somewhere, it's an 80188
                          ; or an 80186
;---------------------------------------------------------------
; All done, let's go back...
;---------------------------------------------------------------
DONE:     popf            ; Restore the flag registers
    cld                   ; clear direction flag
    stosw                 ; store ax in location for numeric variable
    pop bp                ; restore base pointer
;   ret                    for Turbo BASIC, the ret must *not* be used.
;---------------------------------------------------------------
; End of code and segment
;---------------------------------------------------------------
getcpu    endp
cseg      ends
    end   getcpu
```

### LISTING 3: GETCPUTB.INC

```
$INLINE &H55,&H8B,&HEC,&HC4,&H7E
$INLINE &H6,&H9C,&H33,&HC0,&H50
$INLINE &H9D,&H9C,&H58,&H25,&H0
$INLINE &H80,&H3D,&H0,&H80,&H74
$INLINE &H18,&HB8,&H0,&H70,&H50
$INLINE &H9D,&H9C,&H58,&H25,&H0
$INLINE &H70,&H74,&H6,&HB8,&H82
$INLINE &H1,&HEB,&H19,&H90,&HB8
$INLINE &H1E,&H1,&HEB,&H13,&H90
$INLINE &HB8,&HFF,&HFF,&HB1,&H21
$INLINE &HD3,&HE0,&H75,&H6,&HB8
$INLINE &H56,&H0,&HEB,&H4,&H90
$INLINE &HB8,&HBA,&H0,&H9D,&HFC
$INLINE &HAB,&H5D
```

### LISTING 4: WHATCPU.BAS

```
' WHATCPU.BAS   --  By Bruce Tonkin
'
' Determines and displays the Intel CPU type on the host machine
' This version includes the machine code as INLINE statements.
'
' For TURBO TECHNIX V1#2
' Last modified 10/15/87
'
' Requires Juan Jiminez's GETCPU assembly language
'   routine modified to run with Turbo Basic.
'
X%=0
CALL GETCPU(X%)
X$=MID$(STR$(X%),2)
PRINT"Your CPU is an [80";X$;
        IF X%=86 THEN PRINT"/88";
PRINT"]"
END

SUB GETCPU INLINE
$INLINE &H55,&H8B,&HEC,&HC4,&H7E
$INLINE &H6,&H9C,&H33,&HC0,&H50
$INLINE &H9D,&H9C,&H58,&H25,&H0
$INLINE &H80,&H3D,&H0,&H80,&H74
$INLINE &H18,&HB8,&H0,&H70,&H50
$INLINE &H9D,&H9C,&H58,&H25,&H0
$INLINE &H70,&H74,&H6,&HB8,&H82
$INLINE &H1,&HEB,&H19,&H90,&HB8
$INLINE &H1E,&H1,&HEB,&H13,&H90
$INLINE &HB8,&HFF,&HFF,&HB1,&H21
$INLINE &HD3,&HE0,&H75,&H6,&HB8
$INLINE &H56,&H0,&HEB,&H4,&H90
$INLINE &HB8,&HBA,&H0,&H9D,&HFC
$INLINE &HAB,&H5D
END SUB
```

# DRAWING AHEAD

## Draw the line at building complex graphics images pixel by pixel. Turbo Basic provides an easier way.

*Peter G. Aitken*

SQUARE ONE

One of the highlights of Turbo Basic is its powerful set of built-in graphics statements, particularly the flexible graphics capabilities found in the **DRAW** statement. **DRAW** operates somewhat like a miniature graphics programming language embedded within Turbo Basic. First, one or more commands are put into a string expression; the string expression is passed to the **DRAW** statement; and the **DRAW** statement executes the commands to produce screen images. What can be accomplished with **DRAW** is limited only by your creativity.

**DRAW** operates only in graphics mode, and can be used only if you have a Color Graphics Adapter (CGA), an Enhanced Graphics Adapter (EGA), a Hercules graphics card, or a Video Graphics Array (VGA) installed in your PC. Before executing a **DRAW** statement, you must switch the display mode to a graphics mode (using the **SCREEN** statement). In order to understand the details of **DRAW**, you'll need some familiarity with the nature of graphics displays and the way they work.

### GRAPHICS BASICS

The smallest display unit on a graphics screen is called a *pixel*. One pixel is the smallest dot that can be displayed, and the smallest meaningful measure of distance or movement on a particular screen is the distance between two adjacent pixels. The overall resolution of a screen is expressed as the number of pixels available in the horizontal and vertical directions. Thus, the CGA has a horizontal-by-vertical resolution of 640 by 200 in high-resolution mode (**SCREEN** 2) and 320 by 200 in medium-resolution mode (**SCREEN** 1). In contrast, the EGA has a maximum resolution of 640 by 350 (**SCREEN** 9), and the new VGA standard found in IBM's Personal System 2 has a resolution of 640 by 480. As you might expect, higher resolution—more pixels per unit area—gives a clearer and more detailed image.

The location of any given pixel on the screen is expressed in terms of x,y coordinates, with *x* representing distance (in pixels) from the left edge of the screen, and *y* the distance from the top edge. By convention, the pixel in the upper left corner of the screen has coordinates of 0,0, which results in the pixel at the lower right corner of the screen having coordinates (X−1), (Y−1), where X and Y are the maximum resolution for your screen.

The relationship between distances in the X and Y directions determines the *aspect ratio* of a particular display. If the physical distance between adjacent pixels is the same in the X direction as it is in the Y direction, the aspect ratio is 1/1, or 1. To facilitate the use of inexpensive TV-derived display hardware, however, the folks at IBM had to design almost all PC graphics standards so that the aspect ratio is less than one. In **SCREEN** 1 mode the aspect ratio is 5/6; the ratio is 5/12 in **SCREEN** 2 mode, and 8.76/12 in **SCREEN** 7/8/9 modes. The only exception is the new VGA graphics standard, which has an aspect ratio of 1. (Desktop graphics has finally come into its own and is no longer beholden to TV receiver designs.) Thus, with most graphics displays, a line in the X direction will always be shorter than a line of the same number of pixels in the Y direction. When using the **DRAW** statement, take into account the aspect ratio of your graphics display.

### THE CONCEPT OF DRAW

Using the **DRAW** statement is conceptually similar to drawing on a piece of paper with a pencil. When drawing with a pencil, you control the direction and distance the pencil moves, whether or not it makes a line (by lifting it from the paper), and the color of the line (by changing pencils). With the **DRAW** statement, you have control over the same parameters of drawing, keeping in mind that color, of course, is available only if you have a color display.

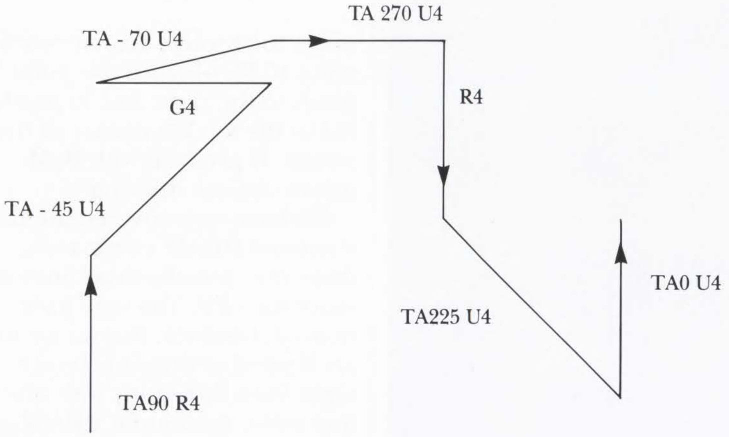


*Figure 1. This drawing illustrates how the direction of drawing can be changed with the "*TA*" command. Each line segment is paired with the* **DRAW** *command used to create it. The length of each segment, four units, is arbitrary—you should pay attention only to the* TA *commands and the direction commands (*U, R, *etc.). The arrows are not part of the screen output these commands produce; they were added to show the direction in which the lines are drawn.*

The fundamental **DRAW** commands control movement in the four major directions and the four diagonal directions. These commands are:

| | |
|---|---|
| **U***n* | Move up |
| **D***n* | Move down |
| **L***n* | Move left |
| **R***n* | Move right |
| **E***n* | Move up and right |
| **F***n* | Move down and right |
| **G***n* | Move down and left |
| **H***n* | Move up and left |

(In all examples here and later, *n* is the number of pixels to move).

These commands specify the direction and distance to draw, but how does the computer know where to start drawing? Unless otherwise specified (with commands to be discussed below), drawing always begins at the *last point referenced*, or LPR. The LPR is the screen position most recently referenced by a graphics statement. For example, after the execution of a **CIRCLE** statement, the LPR is at the center of the circle. When you first enter graphics mode with a **SCREEN** statement, Turbo Basic sets the LPR to the center of the screen. The Turbo Basic manual has additional information on the LPR.

Now that you know the basic **DRAW** commands, let's type a simple Turbo Basic program:

```
CLS
SCREEN 9  'SCREEN 2 for CGA
DRAW "U50 R50 D50 L50"
```

***When you first enter graphics mode with a* SCREEN *statement, Turbo Basic sets the Last Point Referenced (LPR) to the center of the screen.***

This draws a square with sides 50 pixels in length. Now change the **DRAW** argument to **"E50 F50 G50 H50"** and run the program again. This time you get a diamond shape, which is nothing more than a square tilted by 45 degrees. Experiment with these drawing commands until you have a good feel for them.

### MODIFYING MOVEMENT

There are two modifying prefixes that can precede any movement command. The prefix **B** causes the movement to be made without drawing—in effect, it lifts your pencil point from the paper. To see its effect, modify the sample program given above by prefixing one of the drawing commands with **B**, for example:

```
DRAW "U50 BR50 D50 L50"
```

The second modifier is **N**, which does not change what is drawn, but causes the LPR to be reset to its original position after the command is executed rather than being left at the end of the just-drawn line. Its effects can be seen by modifying the drawing command in the sample program to read:

```
DRAW "NU50 NR50 ND50 NL50"
```

Executing the modified program results in a cross being drawn. Because the **N** prefix resets the LPR after each line is drawn, each line starts from the same central point rather than from the end of the last line.

Another command is **M**, or *move*. This is the only drawing command that takes two arguments. Thus, **"M** *x,y***"** draws from the LPR to point *x,y*. If *x* has a leading plus or minus sign, the move is relative to the LPR. If not, the move is made to absolute screen coordinates. Thus

```
DRAW "M +10,10"
```

draws to a point 10 pixels to the right and 10 pixels below the LPR, while

```
DRAW "M 10,10"
```

*continued on page 112*

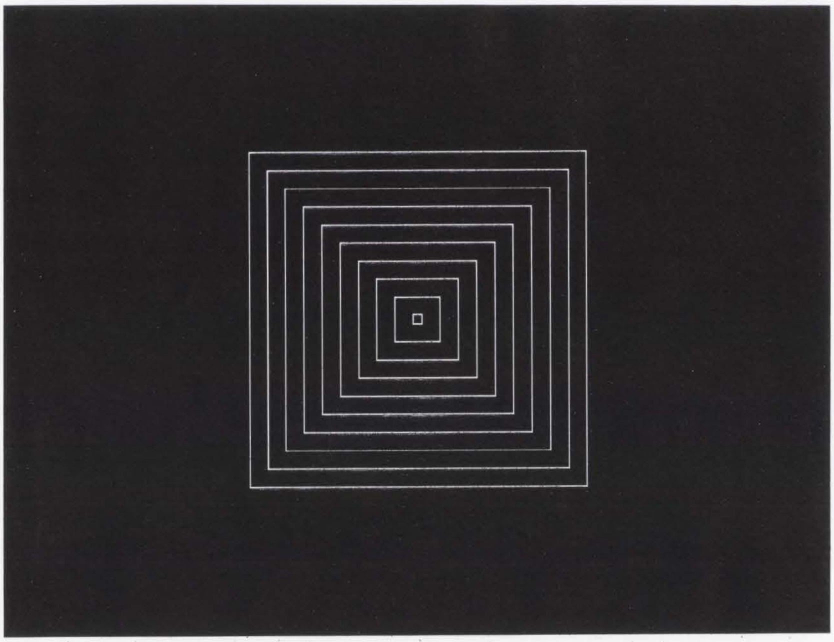
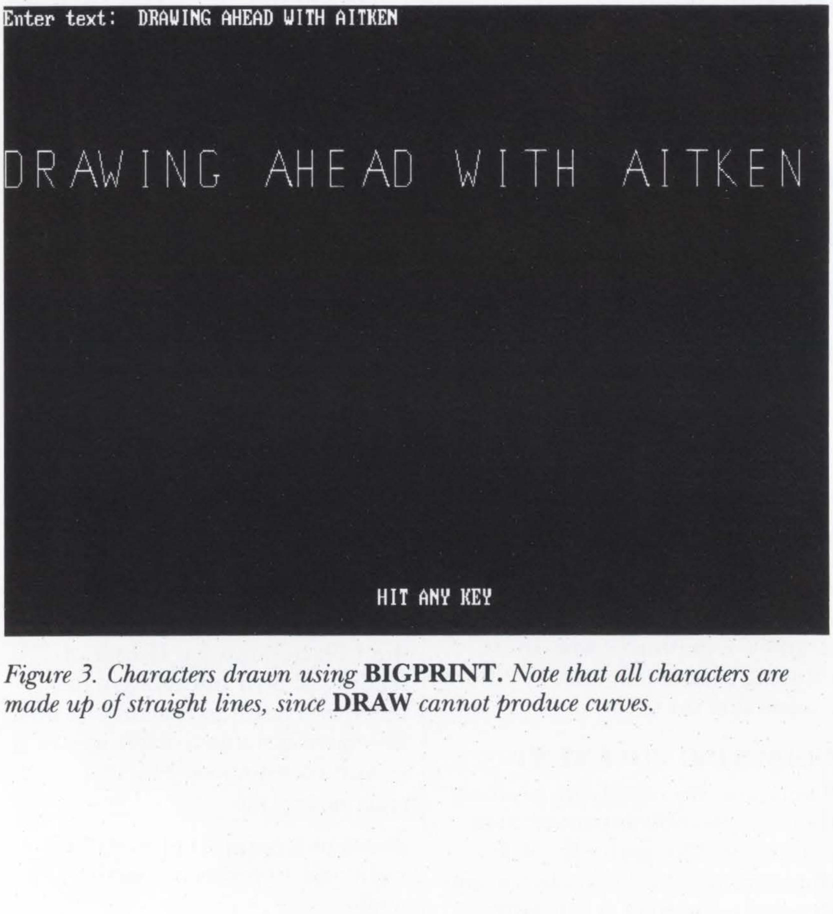


*Figure 2. Each square was drawn with the same* **DRAW** *commands, (given in Listing 1) with only the scale factor changed between squares. The LPR returns to the center of the squares after each one is drawn.*



```
Enter text:  DRAWING AHEAD WITH AITKEN

DRAWING AHEAD WITH AITKEN

HIT ANY KEY
```

*Figure 3. Characters drawn using* **BIGPRINT.** *Note that all characters are made up of straight lines, since* **DRAW** *cannot produce curves.*

## DRAWING AHEAD

*continued from page 111*

draws to absolute screen coordinates 10,10, which is the point 10 pixels to the right and 10 pixels below the top left corner of the screen. If prefixed with **B**, **M** moves without drawing.

We have now covered the fundamental **DRAW** commands, those that actually draw lines and move the LPR. You may have noticed, however, that so far we are limited to drawing in only eight fixed directions with one line color. Additional **DRAW** commands permit greater control.

There are two commands that change the angle of movement: **"A***n***"** and **"TA***n***"**. In both cases, *n* specifies the new angle of movement. For the **"A"** command, *n* can take only the values 0, 1, 2, or 3, specifying respectively 0, 90, 180, and 270 degrees (measured counterclockwise from the vertical). For the **"TA"** command, *n* specifies the turn in degrees and can take values between -360 and 360. Positive values cause a counterclockwise turn, negative values a clockwise turn.

Both of these commands function by rotating the reference axes used by the **DRAW** command. To illustrate what this means, imagine that the drawing angle was turned counterclockwise by 90 degrees. This could be done by issuing one of two commands: **DRAW "A1"** or **DRAW "TA90"**. After either of these commands, the reference axes will be shifted so that "up" is toward the left, "left" is down, "down" is toward the right, and so on. This is shown in the following; both of these commands will draw a box 100 pixels on a side:

```
DRAW "U100 L100 D100 R100"
DRAW "U100 TA90 U100 TA180"+_
     "U100 TA270 U100 TA0"
```

Note that the second line ends with**"TA0"**, which resets the drawing axes to their normal position. It's a good idea to reset the axes to their default orientation at the end of any command string that rotates them; otherwise, subsequent **DRAW** commands may give very strange results! The generation of lines and movement of the LPR using various **"TA"** commands is shown in Figure 1.

Before moving on to the next **DRAW** command, we need to look at ways in which commands can be presented to the **DRAW** statement. The **DRAW** command can accept either a literal string or a string variable as its argument. Thus, the statement

```
DRAW "U10 R20 G30"
```

has the same effect as the statements:

```
A$ = "U10 R20 G30"
DRAW A$
```

### INCORPORATING VARIABLES

While **DRAW** can accept a string literal *or* a string variable, it cannot accept a combination of literals *and* variables. However, there are ways to incorporate variables into command strings; to illustrate, let's look at a programming example.

You have designed a small pattern whose commands are contained in the string **DESIGN$**. The pattern is 40 by 40 pixels, and you want to repeat the design across the top of the screen. One method is to use repetitive statements:

```
DRAW "BM 1,40"     'go to x=1, y=40
DRAW DESIGN$       'draw the pattern
DRAW "BM 41,40"    'go to x=41, y=40
DRAW DESIGN$       ' etc.
                   .
                   .
                   .
DRAW "BM 601,40"
DRAW DESIGN$
```

Of course, Turbo Basic has several ways, such as **FOR-NEXT** loops, to simplify the programming of repetitive operations. Realizing this, you try the following:

```
FOR I% = 1 to 601 STEP 40
  DRAW "BM ",I%,",40"
  DRAW DESIGN$
NEXT I%
```

The result? An error message! Although the idea is logical, Turbo Basic will not accept the combined string/variable argument passed to the first **DRAW** statement. There are two ways around this. The first is to use Turbo Basic's number-to-string and string concatenation commands to incorporate the variable **I%** in the command string, as follows:

```
FOR I% = 1 to 601 STEP 40
  MOVE$ = "BM "+STR$(I%)+",40"
  DRAW MOVE$
  DRAW DESIGN$
NEXT I%
```

***By setting the drawing color the same as the background color, you can draw invisible lines that can later be made visible by changing the background color.***

The first time through the loop, **MOVE$** will equal **"BM1,40"**; the second time through, it will equal **"BM41,40"**, and so on.

The second method is to use the **VARPTR$()** function, which returns a pointer to a variable in string form. This permits the **DRAW** statement to access the contents of variables, as follows:

```
FOR I% = 1 to 601 STEP 40
  DRAW "BM ="+VARPTR$(I%)+",40"
  DRAW DESIGN$
NEXT I%
```

Note the equal sign following the **DRAW** command letter(s) that the variable is to be associated with.

### SIZING IMAGES

Now that you understand how to incorporate numeric variables in **DRAW** command strings, we can look at the **DRAW** command for changing the size of the drawn images. The **DRAW** statement uses a scaling factor to determine final image size. The arguments to the movement commands (U, D, L, R, E, F, G, H, and relative M) are multiplied by the scaling factor, which is 1 by default, to give the final dimension in pixels. The command to modify the scaling factor is **"S*n*"**. *n* can range between 1 and 255, and is divided by 4 to give the actual scaling factor, ranging from 0.25 to almost 64. With the **"S*n*"** commands the actual size of objects can be changed over a range from one-quarter to 64 times the size specified by the numerical arguments themselves. For example, the command **DRAW "S1 R8"** would produce a line two pixels long, and **DRAW "S255 R8"** would produce a line 510 pixels long. A good demonstration of the scaling factor is provided by the short program **SCALER**, Listing 1. The screen output of this program is shown in Figure 2.

Finally, there are a few miscellaneous **DRAW** commands. First, the **"C*n*"** command causes subsequent lines to be drawn in color *n*. The range of colors available depends on the type of graphics adapter and monitor, and on the display mode being used. When using an EGA or VGA, the **PALETTE** and **PALETTE USING** statements also play a role. Generally, the background color and available palette are set using the **COLOR** statement, and the foreground color is set with the **"C*n*"** argument to the **DRAW** command. The default color is the highest legal color attribute. By setting the drawing color the same as the background color, you can draw invisible lines that can later be made visible by changing the background color and/or palette. (See the sections of the Turbo Basic manual on the **COLOR**, **SCREEN**, and **PALETTE** statements for more details about control of color.)

The final **DRAW** command is **"P** *color, boundary***"**, which is very

*continued on page 114*

### LISTING 1: SCALER.BAS

```
SCREEN 9                           'SCREEN 2 for CGA
FOR I% = 2 TO 74 STEP 8
     DRAW "BM 320,175"             'move to screen center
                                   'change 175 to 100 for CGA
     DRAW "S =" + VARPTR$(I%)      'set scale factor
     DRAW "BM +7,5"                'move to corner where next
                                   'square starts
     DRAW "U10 L14 D10 R14"        'draw square
     DELAY 0.5                     'wait a bit
NEXT I%
```

### LISTING 2: DRAWDEMO.BAS

```
'************************************************************
'program DRAWDEMO.BAS
'demonstration of Turbo Basic DRAW command
'requires EGA or CGA adapter
'************************************************************

CLS : SCREEN 9          'change to SCREEN 2 for CGA

'Loop accepts messages and passes them to subroutine BIGPRINT

DO
  CLS : LOCATE 1,1
  INPUT "Enter text: ", MESSAGE$
  IF MESSAGE$ = "" THEN EXIT LOOP     'exit loop if null entry
  CALL BIGPRINT(1, 100, MESSAGE$)
  LOCATE 24,37
  PRINT "HIT ANY KEY";
  WHILE NOT INSTAT : WEND
LOOP

SCREEN 0
END

'************************************************************

SUB BIGPRINT(XLOC%, YLOC%, MESSAGE$)

LOCAL MESSAGE.LENGTH%

'Declare and initialize array.  The array is dimensioned 26
'elements long with subscripts 65 thru 90, so that the array
'subscript will match the ASCII code of the corresponding letter.
'Thus, ASC("A") = 65, and the commands to draw an "A" are in
'LETTER$(65)

DIM LETTER$(65:90)

'DRAW commands for letters A thru Z in order
```

## DRAWING AHEAD

*continued from page 113*

similar to the **PAINT** statement. The **"P"** command starts at the current x,y coordinate and fills pixels with the color *color* until it reaches the boundaries indicated by *boundary*. This command is useful for filling shapes that you have created with other **DRAW** commands. Two points about the **"P"** command need mentioning. First, if the current x,y pixel is already the boundary color, **"P"** will have no effect. Thus, if you draw a figure and want to fill it, you'll have to move the current x,y position into the interior of the figure before issuing the **"P"** command. Second, the **"P"** command is very good at finding "leaks" in boundaries. Even one missing pixel in the boundary will cause the fill color to leak out and possibly cover the entire screen.

### DRAWING ON WHAT YOU'VE LEARNED

Now that you know all about the **DRAW** command, what can you do with it? To illustrate its flexibility, I have written a subroutine that produces letters on the graphics screen. Of course, you can write normal-sized text to the graphics screen with Turbo Basic's **PRINT** statement, but there may be times when you want larger letters. The Turbo Basic subroutine **BIGPRINT** (given as part of the DRAWDEMO.BAS program in Listing 2) uses the **DRAW** command to produce letters that are approximately four times as large as normal screen text. The letters were designed on graph paper by roughly tracing each letter, using straight lines only, within a 25 by 25 grid.

Once I was satisfied with the designs, the lines that made up each letter were then translated into **DRAW** commands. The letters were designed for optimum reproduction on a 640 by 350 EGA screen (**SCREEN 9** mode); the subroutine will work on a CGA in **SCREEN 2** mode, but the different aspect ratio will result in rather tall and thin letters. An EGA screen display of **BIGPRINT**'s output is given in Figure 3.

**BIGPRINT** requires three parameters: **XLOC%** and **YLOC%** are integers giving the x and y position of the lower left corner of the first letter; **MESSAGE$** is the text to be printed. As written, the only characters that **BIGPRINT** accepts are letters and the space character. Lowercase letters are converted to uppercase, and bounds checking is not performed on the x and y coordinates. If you find **BIGPRINT** useful, you can design your own numbers, punctuation marks, and lowercase letters. You could also modify **BIGPRINT** to use the color, scaling factor, and rotation commands to produce text in different colors, sizes, and angles. ■

***The "P" command is very good at finding "leaks" in boundaries. Even one missing pixel in the boundary will cause the fill color to leak out and cover the entire screen.***

*Peter Aitken is an assistant professor at Duke University Medical Center, and is the author of DigScope, a scientific software package. He writes and consults in the microcomputer field.*

*Listings may be downloaded from CompuServe as DRAW.ARC.*

```
LETTER$(65) = "TA-20 U21 TA-160 U21 D10 TA0 L11"
LETTER$(66) = "U20 R10 F2 D6 G2 L10 R10 F2 D7 G2 L10"
LETTER$(67) = "BU2 F2 R8 E2 BL12 U16 E2 R8 F2"
LETTER$(68) = "U20 R9 F3 D14 G3 L9
LETTER$(69) = "U20 NR10 D10 NR10 D10 R10"
LETTER$(70) = "U20 NR10 D10 R9"
LETTER$(71) = "BU2 F2 R8 E2 U4 NL2 NR2 BD4 BL12 U16 E2 R8 F2"
LETTER$(72) = "U20 D10 R11 U10 D20"
LETTER$(73) = "BR6 R4 BL2 U20 L2 R4"
LETTER$(74) = "BU4 D2 F2 R6 E2 U18 L4 R8"
LETTER$(75) = "U20 D9 TA-35 U11 BD11 TA-137 U15 TA0"
LETTER$(76) = "NU20 R14"
LETTER$(77) = "U20 TA-142 U10 TA-36 U10 TA0 D20"
LETTER$(78) = "U20 TA -155 U22 TA0 U20"
LETTER$(79) = "BU3 U14 E3 R8 F3 D14 G3 L8 H3"
LETTER$(80) = "U20 R10 F2 D6 G2 L10"
LETTER$(81) = "BU3 U14 E3 R8 F3 D14 G1 H2 F4 BH2 G2 L8 H3"
LETTER$(82) = "U20 R10 F2 D6 G2 L10 R4 TA-150 U11 TA0"
LETTER$(83) = "BU3 F3 R6 E3 U4 H3 L6 H3 U4 E3 R6 F3"
LETTER$(84) = "BR8 U20 NL8 R8"
LETTER$(85) = "BU20 D18 F2 R10 E2 U18"
LETTER$(86) = "BU20 TA-160 U22 TA-20 U22 TA0"
LETTER$(87) = "BU20 BL3 TA-170 U20 TA-20 U9 TA-160 U9 TA-10 U20 TA0"
LETTER$(88) = "TA-30 U22 TA0 BL14 TA-150 U22 TA0"
LETTER$(89) = "BR8 U10 NH10 E10"
LETTER$(90) = "BU20 R16 TA150 U23 TA0 R16"

'code begins

MESSAGE.LENGTH% = LEN(MESSAGE$)          '# characters in MESSAGE$

DO UNTIL MESSAGE.LENGTH% = 0

'get the ASCII value of leftmost character in MESSAGE$

        CODE% = ASC(MESSAGE$)

'convert lower case codes to corresponding upper case codes

        IF CODE% > 96 AND CODE% < 123 THEN CODE% = CODE% - 32

'decrement length and strip off leftmost character

        DECR MESSAGE.LENGTH%
        MESSAGE$ = RIGHT$(MESSAGE$,MESSAGE.LENGTH%)

'if not uppercase or space, loop

        IF CODE% < 65 OR CODE% > 90 THEN_
                IF CODE% <> 32 THEN GOTO ENDLOOP

'move to start point for next character

        DRAW "BM= "+ VARPTR$(XLOC%) + ", =" + VARPTR$(YLOC%)

'if not a space, draw the letter

        IF CODE% <> 32 THEN DRAW LETTER$(CODE%)

'next letter will be 25 pixels to the right

        XLOC% = XLOC% + 25

ENDLOOP:
        LOOP

END SUB
```

# BUILDING ON QUATTRO: INTRODUCTION

## Borland is opening the door wide to third-party add-in products for Quattro: The Professional Spreadsheet.

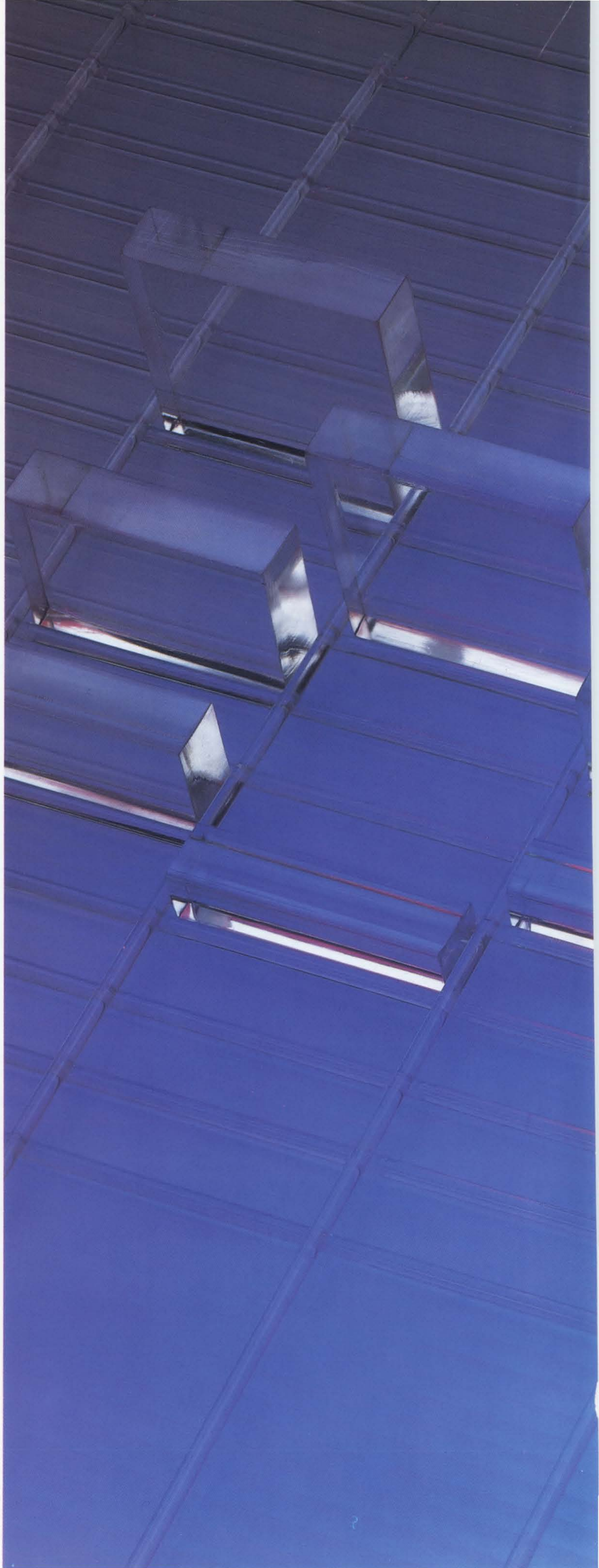
*Jeff Duntemann*

In a sense, a spreadsheet is a nonprocedural language displayed in two dimensions. In the spreadsheet paradigm, relationships between cells are defined, and the recalculation command invokes the machinery that propagates new values into the cells according to their relationships. How the spreadsheet accomplishes recalc is hidden from the user, and only the results are seen.

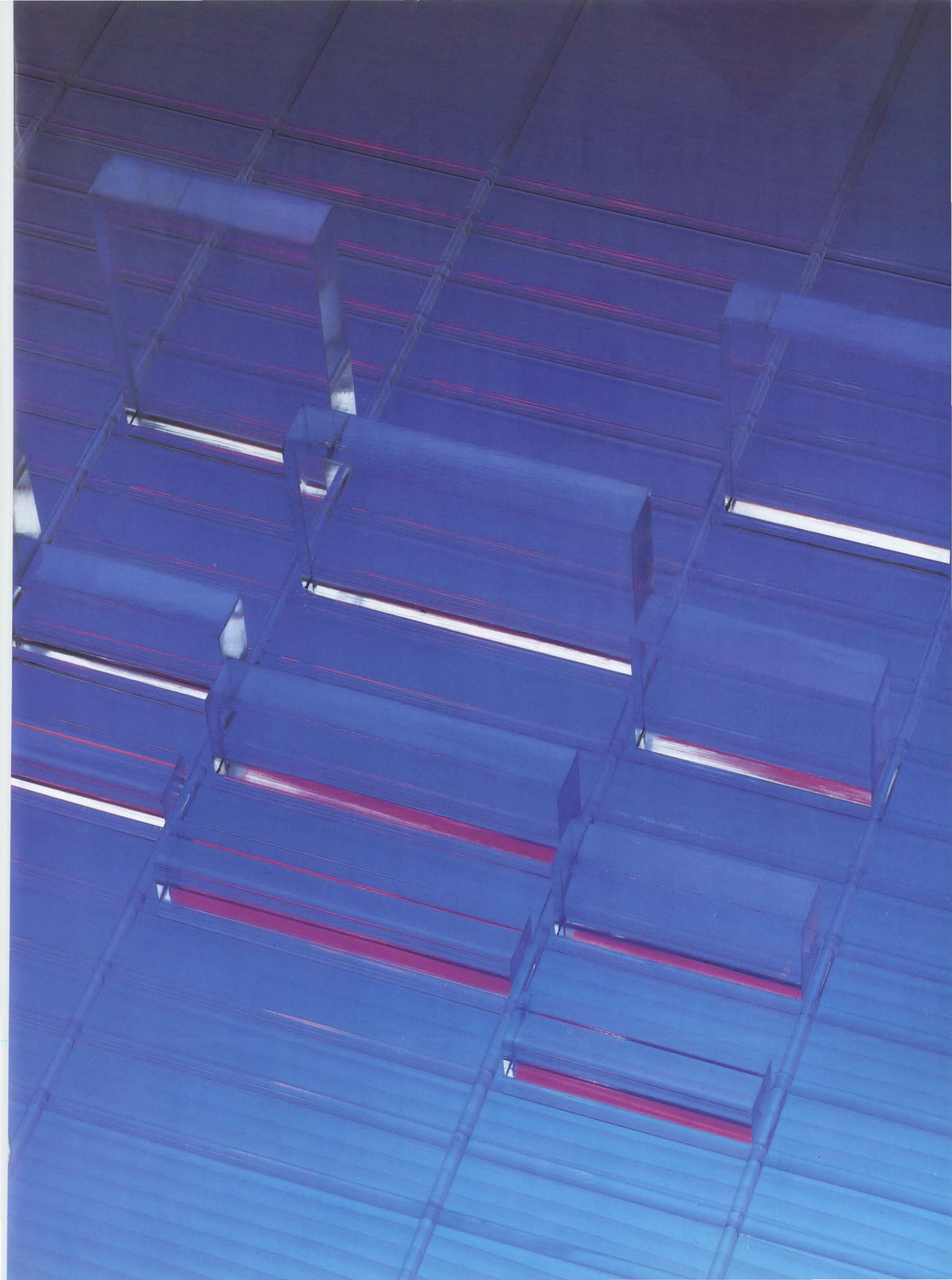
This makes it difficult to build on a spreadsheet as a platform for vertical-market applications. To go further than canned models and collections of macros requires both more resources and more control. The resources are generally there if the spreadsheet is good, but the conceptual difficulties of opening up a closed calculating engine to outside tinkering have held back any movement toward a truly programmable spreadsheet. To be programmable, a spreadsheet must be *designed* for programmability—it cannot be tacked on as an afterthought.

By developing the Quattro API (Application Program Interface), Borland is doing for the spreadsheet paradigm what PAL and the dBase languages have done for the database paradigm. The Quattro API allows developers to generate vertical-market applications based on a powerful general-purpose "engine." The Quattro spreadsheet has been designed so that its central engine facilities are accessible as callable routines. At any given time, the many aspects of Quattro's state may be read as though they were global variables. By providing documentation of all these entry points and status items, and bindings for the Borland high-level languages, the Quattro Add-In Toolkit becomes the foundation for a family of extensions that will work correctly and without unsuspected side effects.

*continued on page 119*



*Bradley Ream*



NEW!

# Quattro: Twice the speed. Twice the power. Half the price!

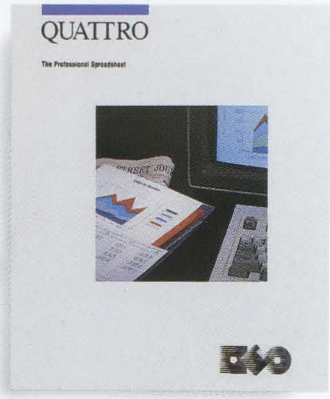
Quattro™, our new generation professional spreadsheet proves there are better and faster ways to do everything. To do graphics. To recalculate. To do macros. To save and retrieve. To search, sort, load. To do anything and everything that state-of-the-art spreadsheets should do.

### *Our technical superiority means product superiority*

Lotus® Development, makers of 1-2-3,® is bigger by factors than Borland. Bigger, not better. Technical superiority is a Borland trademark, and Quattro is fresh proof that it produces a better product.

### *Quattro gives you presentation-quality graphics*

Quattro brings new highs in quality graphics to your spreadsheet. It also brings 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—*directly* from the spreadsheet without leaving the spreadsheet. And Quattro also gives you PostScript™ support.



### *Quattro recalculates a lot faster than you-know-who*

The smartest and fastest way to recalculate a spreadsheet is to do what Quattro does, which is something called "intelligent recalc," which in English means you only re-count the formulas that count. Unlike 1-2-3, Quattro recalculates just the formulas that matter.

### *If you know how to use spreadsheets, 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. Technically superior to 1-2-3? Yes. Half the price of 1-2-3? Yes!

### *Get Quattro, the professional spreadsheet for only $195.00*

Quattro is so advanced it's easy to use and it's less than half the price of 1-2-3. It's not copy protected, and 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 it makes your work go faster.

*60-day money-back guarantee**

### *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.* |
| | *Price* | *$195* | *$495* |

*Benchmark details available upon request.*

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

*Customer satisfaction is our main concern; if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Lotus and 1-2-3 are registered trademarks of Lotus Development Corp. Other brand and product names are trademarks or registered trademarks of their respective holders. Copyright ©1987 Borland International. BI-1144E

BORLAND
INTERNATIONAL

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

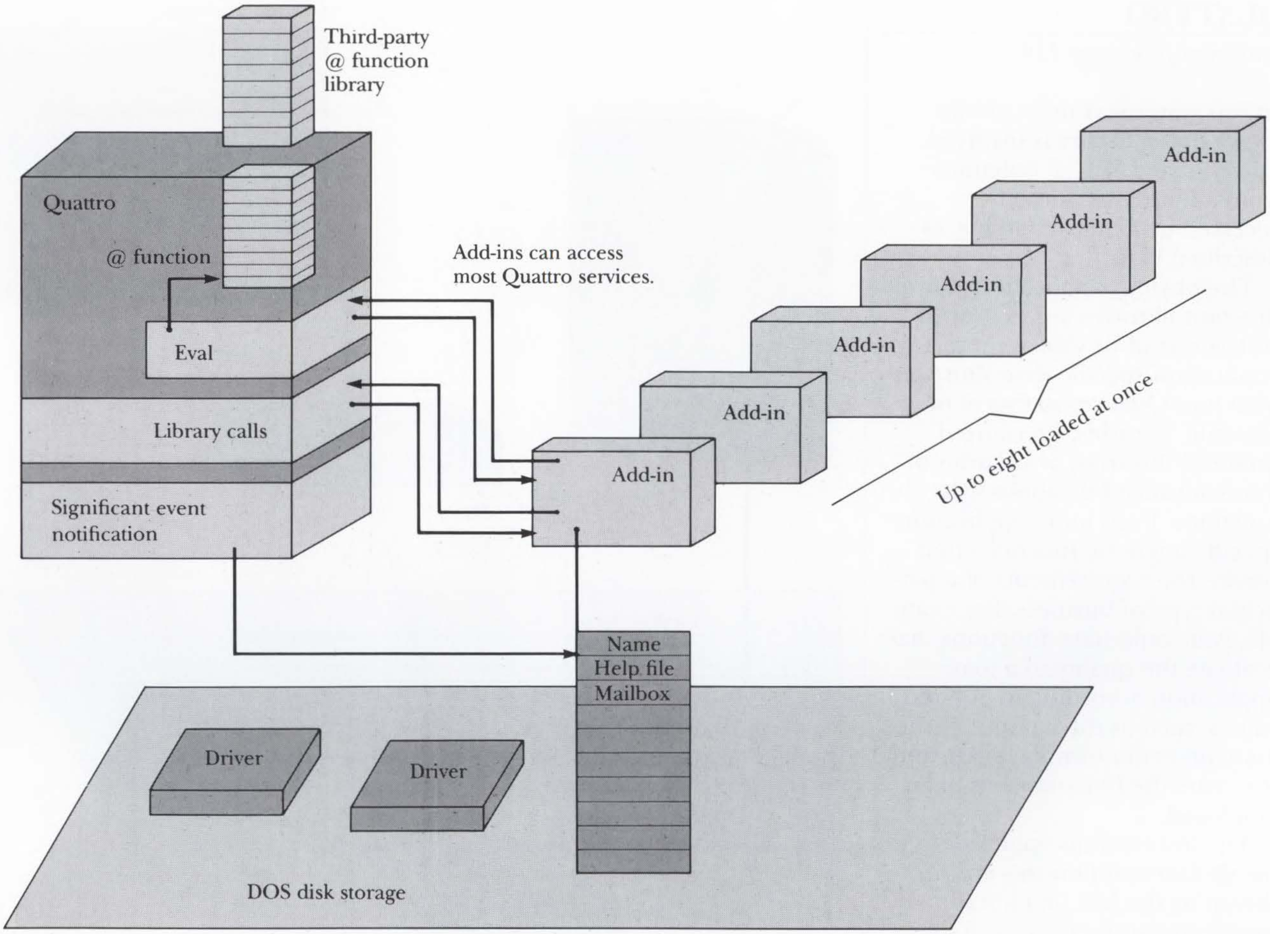


*Figure 1. The Quattro add-in architecture*

## QUATTRO

*continued from page 116*

In this and future articles in *TURBO TECHNIX,* we will present the architecture of Quattro extensions and the methods involved in writing them.

### EXTENDING QUATTRO THREE WAYS

Quattro extensions fall into three general categories:

- @ function libraries
- drivers
- add-ins

The relationship of these extensions to Quattro and to one another is illustrated in Figure 1.

**@ Function libraries.** Much of the power of the spreadsheet paradigm proceeds from the notion of relating one cell or group of cells to another cell or group of cells. Built-in spreadsheet functions constitute one important element of the cell-to-cell relationship. These relationships are defined by formulas that may be attached to a given cell. Most spreadsheets, and many databases (including Reflex), support a repertoire of @ *functions,* so called because their names begin with an "@" character. One simple example is **@SUM(<range>)**, which calculates the sum of the values in a range of cells. For example, the function **@SUM(B1..B12)** returns the sum of all the values stored in cells B1 through B12. Functions can also be written to return text strings, dates, times, and other types of data.

Quattro contains about 100 different built-in @ functions, covering most general requirements for business spreadsheet use. They include the logical, string, and counting functions common to most spreadsheets, plus transcendental functions, database aggregation functions, and financial functions that compute compound interest, annuities, and depreciation.

These are powerful but very horizontal functions. To some extent, application-specific functions can be constructed by combining the @ functions into formulas, but there are computational and performance limits when using formulas built only from basic operators and @ functions. Quattro addresses these limitations by supporting loadable @ function libraries produced by third-party vendors.

@ function libraries are collections of routines written to allow their interpretation by Quattro's function evaluator. Once loaded, @ functions from a third-party library are indistinguishable from Quattro's built-in functions. Library @ functions have the same limitations—typically, they can only return a single value when they are called—but they are potentially more powerful. Loading the library may be made automatic by specifying the library name in Quattro's Startup menu.

*continued on page 120*

## QUATTRO

*continued from page 119*

In this way, users need not be aware that a library is involved. Once loaded, the @ functions from a library may also be accessed by Quattro add-ins, as described below.

The obvious applications for @ function libraries are to provide collections of functions more application-specific than those built into Quattro: statistical or scientific libraries, specialized financial libraries, or libraries of more advanced database-style functions. Even more application-specific might be functions that service the requirements of a particular type of business. For example, you could have functions that evaluate the quality of a loan application according to entered values, such as the amount of the loan, percent down payment, and the years the borrower has been employed.

The less obvious applications for @ function libraries are driven by the fact that library functions can access any of the Quattro services documented in the Add-in Toolkit. They can create and write to their own DOS files, examine cells and ranges of cells, query Quattro attributes, and many other things. A special-purpose @ function could be written to act as a recalculation "demon"—it would "awaken" when its cell was reached during recalculation, and take some action based on Quattro's current state. This demon function could act as a recalculation "breakpoint" for spreadsheet debugging and auditing. Another use might be a real-time securities trading system in which, during recalculation, the demon function actually dials an online service, checks the current price of a given security, and then inserts the price into the cell that is associated with the demon function. Add-ins and drivers do not allow that fine a level of control during recalculation.

@ function libraries are loaded into one of Quattro's eight "slots" for resident extensions. The total number of loaded libraries and add-ins cannot be greater than eight, because libraries must share these slots with the add-ins described below.

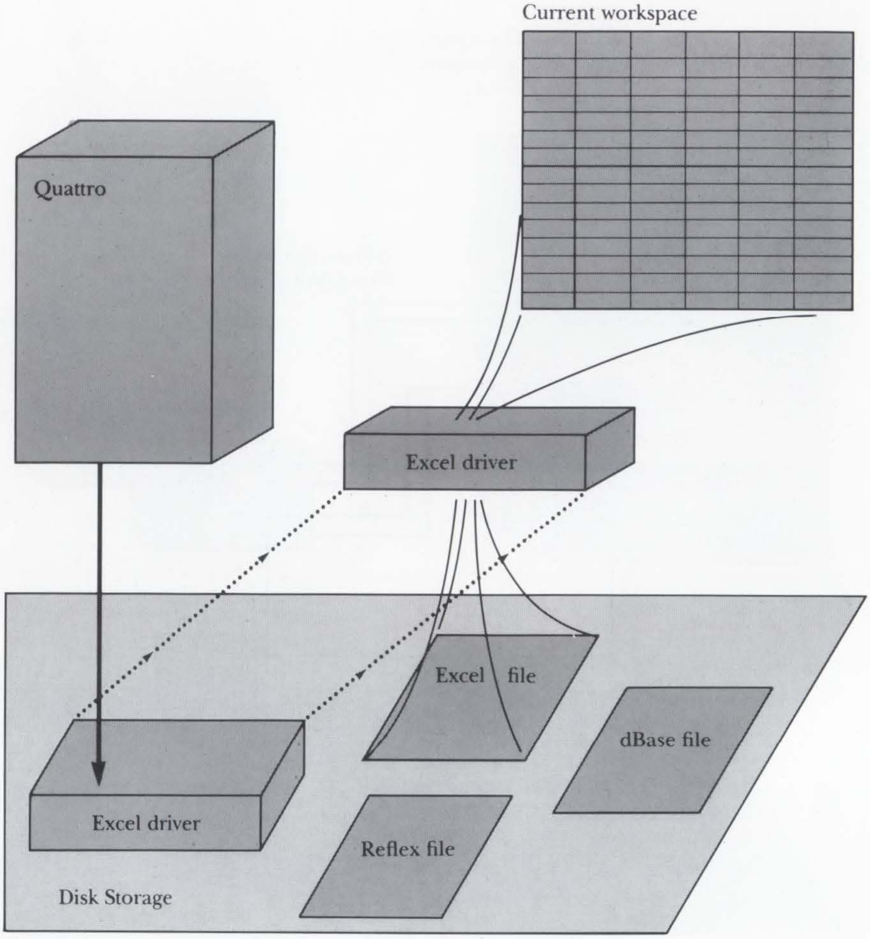


*Figure 2. A Quattro driver.*

**Drivers.** The job of a Quattro driver is quite specific since drivers either convert Quattro spreadsheets into other program file formats, or import program data files into a Quattro spreadsheet. A separate module is required for both file import and file export.

The conversion process is handled transparently from a user's standpoint. From Quattro's perspective, the type of data file is determined by its file extension. Quattro's own spreadsheet files are given a .WKQ extension. When a user asks to load a .WKQ file, Quattro loads the file directly under the assumption that the file is in its native format. A request to load a file with any extension other than .WKQ prompts Quattro to look on disk for a driver module associated with that file extension.

Driver module names are keyed to the file types they support, according to the following pattern:

FR???.TRN—Imports foreign file formats to Quattro

FS???.TRN—Exports Quattro spreadsheets to foreign formats

For example, Paradox database files have a .DB extension. The import driver for Paradox files would be named FRDB.TRN. The FR stands for "File Retrieve," and the .TRN is a common extension for all drivers. The export module would be named FSDB.TRN. The FS stands for "File Save."

If a user requests the loading of a file ending in .DB, Quattro searches the disk for a driver file named FRDB.TRN. If the driver is not found, an error message is displayed. Note that the user simply requests a filename and does not have to notify Quattro that the file is in a format other than Quattro's own. If the required driver is located on disk, Quattro loads the driver into memory and executes it. (Figure 2 illustrates this situation. ) The driver reads the foreign data file and translates the data into Quattro spreadsheet form.

Once a given translation task is completed, the memory occupied by the driver is released. The driver is reloaded each time a file must be translated.

Drivers are more compact than add-ins. They are not notified of significant events (explained below). Drivers can take advantage of Quattro services such as displaying a menu of choices. For example, the dBase driver allows the user to change a certain field's type and size by selecting from a list of fields.

**Add-ins.** These are likely to be the most numerous Quattro extensions, as well as the most general and versatile. An add-in is a program module that can be loaded by the user through the Load menu, or can be automatically loaded at Quattro start-up time. The add-in has access to nearly all Quattro services, and appears to the user as another Quattro feature invoked from the menu tree.

Typical add-ins might include the following:

- spreadsheet auditor programs
- annotator programs that attach small note-windows to specific spreadsheet cells
- drivers for WORM optical storage devices that create an audit trail by archiving every revision of a spreadsheet
- special-purpose text editor or telecommunications windows

These are all extensions that might otherwise be implemented as external TSR programs that have little or no knowledge of the Quattro spreadsheet internals. Writing them as Quattro add-ins not only provides close integration with the Quattro product itself, but also saves considerable work for the developer. The tremendous resources of the Quattro program are placed at the disposal of the add-in via Turbo C or Turbo Pascal interface.

Add-ins are .EXE or .COM files containing interface routines linked from the library distributed with the Quattro Add-in Toolkit. The extension is changed to .QAI from .EXE or .COM so that a user cannot accidentally execute an add-in from DOS.

Quattro can have up to eight extensions loaded at any one time. These may be either @ function libraries or add-ins, but the total number of resident extensions is limited to eight. Extensions can be unloaded when no longer needed to allow other extensions to be loaded in their places.

## RESPONDING TO SIGNIFICANT EVENTS

Add-ins exercise control in two ways:

1. The user may explicitly invoke an add-in from the Run menu. Add-ins may also be spliced into any point in the Quattro menu tree by a "menu-builder" utility, making the add-in indistinguishable (to the user) from any standard Quattro menu feature. When invoked, the add-in does its job and then returns control to Quattro. A spreadsheet auditor or report-generator might work in this fashion. The add-in might communicate with the user through a window overlaying the Quattro spreadsheet, or it might take over the screen completely in the manner of an entirely separate application. The type of face the add-in shows to the user is entirely up to the developer.
2. The add-in also has an opportunity to run whenever certain things happen in the Quattro program itself. These triggering occurrences are called *significant events.* Examples of these events are saving or retrieving a spreadsheet file; erasing a spreadsheet; moving a block of cells; inserting or deleting rows or columns of cells; and repainting the screen. The add-in is also notified when the user chooses to unload it, allowing the add-in to close any open files and put its house in order before being overwritten. Add-ins are notified when Quattro is about to save a .WKQ file, and they can save their own data, regardless of format, in the .WKQ file. Similarly, add-ins are notified when Quattro intends to read a .WKQ file, so that they can prepare to receive any data in the file tagged as belonging to the add-in. This allows a "sticky-notes" type application, where the add-in keeps its note data in the .WKQ file, thus making the added complication of separate disk files unnecessary.

These two methods (user invocation and event-triggered invocation) are not exclusive. A given add-in may run in both modes. During those times when the add-in is not in the foreground running, it may have to keep track of what is happening to the current spreadsheet in memory. Being able to run periodically allows the add-in to update its own status tables or files as the spreadsheet changes in the foreground.

Figure 3 details the logical structure of an add-in program. The program contains a structure called the *add-in information table.* This table contains the add-in's name, a pointer to a help file, 24 status bytes, and 32 addresses representing entry points to the add-in itself. Each significant event has a corresponding entry point in the add-in information table. Quattro notifies the add-in of a significant event by calling the add-in through the appropriate address in the table. The address points to a service routine within the add-in that takes necessary action and then returns control to Quattro.

The add-in's developer need not implement a service routine for every significant event. For example, an add-in might only need to respond to the saving or loading of a spreadsheet. The addresses corresponding to all other significant events would be set to zero.

Up to eight add-ins may be resident in memory at once, and all of them are notified of every significant event. Each add-in has the opportunity to take control in sequence if it has a service routine corresponding to the event.

*continued on page 122*

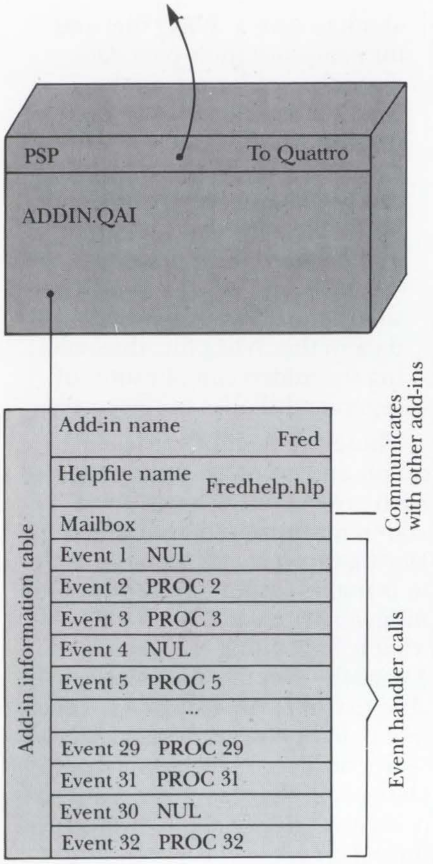


*Figure 3. A Quattro add-in.*

### ADDING IN FUNCTIONALITY

Add-ins are much more powerful and general than either @ function libraries or drivers. The following are the most important things an add-in can do in cooperation with Quattro:

- *Get status information.* The add-in can interrogate anything that can be set in a menu, including screen colors, default directories, column widths, and whatever options the user can set.
- *Feed keystrokes to Quattro.* The add-in can play "automated user" by passing sequences of keystrokes to Quattro. Therefore, an add-in can generate any sequence of keys that the user can type.
- *Execute Quattro commands.* An add-in can execute Quattro menu commands by passing menu-equivalent codes to Quattro.
- *Evaluate formulas.* The add-in can pass an @ function to Quattro for evaluation and keep the returned value for its own purposes, rather than storing it in a cell. Both built-in and library functions are accessible.
- *Use the Quattro UI.* The add-in can use the same screen handling routines Quattro uses for prompts and dialogue boxes, windowing, screen repainting, and so on, in the course of communicating with its own users.
- *Trace cell-to-cell dependencies.* The add-in can determine the cells on which a given cell depends for its own values, as well as those cells that depend on a given cell for their values.
- *Obtain memory through Quattro.* Quattro manages both real and expanded memory for data storage, and an add-in can request memory through Quattro for its own dynamic data structures. This allows the add-in to make use of expanded memory without having to determine whether or not expanded memory is available.
- *Use Quattro's help system.* At an add-in's request, Quattro switches to the add-in's help file. The add-in's help support can use features of Quattro's help facility including context-sensitive screen display. The Quattro Add-in Toolkit provides a help linker that automatically builds custom help files.

### CREATING QUATTRO EXTENSIONS

The preferred form for Quattro extensions is an .EXE file created with Turbo C or Turbo Pascal 4.0. Bindings specific to each language must be linked with the .EXE file. These bindings include function prototypes for all the numerous Quattro services, and also startup code specific to the type of extension. For example, a replacement **c0** routine must be linked into every extension created with Turbo C, and the **c0** for a driver differs from the **c0** for an add-in. In a future release, a custom .TPU unit file will be provided that contains interface procedures for Turbo Pascal.

Certain rules and restrictions on what an extension may do must be respected, or the add-in is considered "ill-behaved" and may not function correctly with Quattro. Particularly, keystroke capture and display output must be done through Quattro services to avoid conflict with Quattro's keyboard and screen-repaint management.

From a height, the Quattro API is relatively simple. The details grow more subtle, because extending a spreadsheet is fundamentally different from extending a database manager with a procedural command language. As with any language, you need to think in its terms to gain the most benefit from its power. Future issues of *TURBO TECHNIX* will provide detailed tutorials on the Quattro extension development process, along with examples in Turbo C and Turbo Pascal. ■

## QUATTRO DEVELOPER CONFERENCE '88

On January 20–23, 1988, Borland will present the Quattro Developer Conference '88, a four-day technical conference on developing Quattro extensions.

The coverage will be advanced and is targeted for serious programmers and 1-2-3 add-in developers who are already fluent in Turbo C and/or Turbo Pascal 4.0 at a system level. The training will be hands-on and computers will be provided. Since the Turbo C and Turbo Pascal sessions will be running in parallel, it will not be possible to attend both.

The $395 fee includes the Quattro Add-in Developers' Toolkit, lodging, and all meals. In addition to the purely technical sessions, attendees will be able to meet with Borland executives and technical staff to discuss ideas, wishlists, and other matters. The conference will be held at the Chaminade Conference Center in Santa Cruz, California. Attendance will be limited; to reserve your place, call the Borland Corporate Communications Group at (408) 438-8400. ■

# PAL PROCEDURES AND PROCEDURE LIBRARIES

## PAL procedures are smaller and run faster when you place them in libraries.

*Todd Freter*

PROGRAMMER

Procedures? Libraries? In a database manager's application language?

Why not? PAL accommodates the expectations of serious programmers who have come to rely on efficient tools like those provided by the C language. PAL procedures and libraries represent just such tools.

### PAL PROCEDURES

As in other programming languages, a procedure in PAL is a named set of program statements. When taken together, the statements constitute a discrete module. In PAL, a procedure is defined within a script and may be stored in procedure libraries. A procedure can take multiple arguments and return a single value.

A PAL procedure is considered defined once Paradox has read the source code and parsed it into binary form in memory. After that, the procedure can repeatedly perform its task without being re-parsed or interpreted on a step-by-step basis each time the procedure is invoked. Because of this, when a procedure is called in a PAL application, it executes more quickly than the equivalent sequence of PAL statements on which the procedure is based. PAL statements executed *outside* of a procedure must be parsed and interpreted one-by-one, with the expected performance overhead.

In addition, PAL procedures can be stored in and accessed from libraries. A procedure stored in a library is already defined and parsed into its low-level operations. This can further improve application performance by eliminating the need for Paradox to define procedures prior to running the application.

### SCRIPTS AND PROCEDURES IN PAL

The connection between a script and a procedure in PAL is quite close. In fact, a procedure usually begins as a script written in PAL, with almost no special or distinctive use of the language. If you know how to write PAL scripts, it is easy to develop procedures.

To use a PAL example that will be familiar to C programmers, you can start with a script called **hello**, shown in Listing 1, HELLO.SC. Listing 1 corresponds closely to the canonical program for introducing the C language to new programmers, shown in Listing 2, HELLO.C. The **?** command in PAL, like **printf** in C, displays a string of characters, and the **SLEEP 5000** command delays execution of the next command for five seconds (5000 milliseconds) so that the program user has time to read the string "Hello, world!" on the screen before it disappears. This is because Paradox would immediately clear the screen after displaying "Hello, world!" to the user. Then control returns to whatever called **hello**.

To implement the script in Listing 1 as part of a PAL procedure, simply surround the statements with two other PAL commands. The **PROC procedurename()** command names the procedure, which must be followed by a pair of parentheses, and the **ENDPROC** command terminates the procedure. All statements between the **PROC procedurename()** and **ENDPROC** commands constitute the body of the procedure. For instance, you could write a script called **greet** containing a procedure based on **hello**, shown in Listing 3, GREET.SC. This script defines a procedure called **hello()** with the same PAL commands as the **hello** script. After this new script, **greet**, defines the **hello()** procedure, it calls and executes **hello()** as a custom command. To continue the comparison with C, refer to Listing 4 (GREET.C), which is a C program that performs the same tasks as the PAL procedure in Listing 3.

Although our example presents a procedure that is trivial in its simplicity, a PAL procedure can be of any practical length. In addition, procedures can be nested to any level.

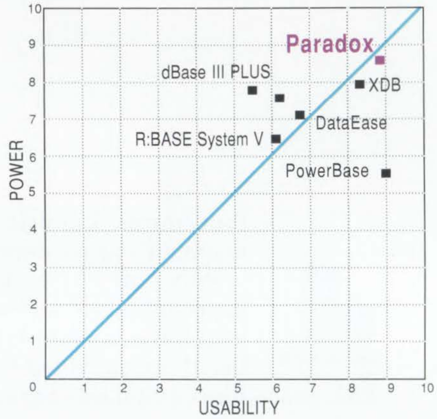
*continued on page 126*

# Paradox: the top-rated relational database manager in the world

"Paradox® is once again the top-rated program, with the latest version scoring even higher than last year's top score." (Software Digest's 1987 Ratings Report—an independent comparative ratings report for selecting IBM PC business software. All tests for the Ratings Report were done by the prestigious National Software Testing Laboratory, Philadelphia, PA.) The Ratings Report message is crystal clear: there is no better relational database manager than Paradox. NSTL tested 12 different programs and amongst other results, discovered that Paradox is 3 times faster than dBASE® and 6 times faster than R:BASE® on a two-file join with subtotals test.†

## *Paradox does the impossible: Combines ease of use with Power and Sophistication*

Even if you're a beginner, Paradox is the only relational database manager that you can take out of the box and begin using right away. Because Paradox employs state-of-the-art artificial intelligence technology, it does almost everything for you—except take itself out of the box. (If you've ever used 1-2-3® or dBASE,® you already know how to use Paradox. It has Lotus-like menus, and Paradox documentation includes "A Quick Guide to Paradox for Lotus users" and "A Quick Guide to Paradox for dBASE users.")



*Source: Software Digest**

*Ideal programs have high levels of both power and usability. Programs plotted in the upper righthand portion of the diagram above come closest to achieving that ideal.*

## *Paradox responds instantly to "Query-by-Example"*

The method you use to ask questions is called Query-by-Example. Instead of spending time figuring out *how* to do the query, you simply give Paradox an example of the results you're looking for. Paradox picks up the example and automatically seeks the fastest way of getting the answer. Paradox, unlike other databases, makes it just as easy to query multiple tables simultaneously as it is to query one.

*Source: Software Digest**

| Software Digest Rating | Overall Evaluation | Program Name | Version Tested | Ease of Learning | Ease of Use | Error Handling | Performance | Versatility | Memory Requirement | Price |
|---|---|---|---|---|---|---|---|---|---|---|
| ☆☆☆☆ | 8.7 | Paradox | 1.1 | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 | 512K | $495 |
| ☆☆☆☆ | 8.2 | XDB | 1.10 | 🟦 | 🟦 | 🟦 | 🟦 | 🟦 | 320K | $750 |
| ☆☆☆ | 7.6 | PowerBase | 2.3 | 🟦 | 🟦 | 🟦 | 🟨 | 🟪 | 384K | $349 |
| ☆☆☆ | 7.0 | Open Access II | 2.0 | 🟨 | 🟨 | 🟦 | 🟦 | 🟦 | 256K | $395 |
| ☆☆☆ | 7.0 | DataEase | 2.5/2 | 🟦 | 🟦 | 🟪 | 🟦 | 🟦 | 384K | $600 |
| ☆☆ | 6.6 | dBASE III PLUS | 1.1 | 🟨 | 🟨 | 🟦 | 🟦 | 🟨 | 384K | $695 |
| ☆☆ | 6.4 | R:BASE System V | 1.1 | 🟨 | 🟨 | 🟦 | 🟪 | 🟦 | 512K | $700 |

RATINGS KEY
(On a scale of 0 to 10)
Overall Evaluation

| | |
|---|---|
| ☆☆☆☆☆ | 9.0 or higher |
| ☆☆☆☆ | 8.0 - 8.9 |
| ☆☆☆ | 7.0 - 7.9 |
| ☆☆ | 6.0 - 6.9 |
| ☆ | 5.0 - 5.9 |

All Other Ratings

| | |
|---|---|
| 🟦 | 7.0 - 9.9 |
| 🟨 | 5.0 - 6.9 |
| 🟪 | UNDER 5.0 |

***How to make your network network***

To run Paradox 2.0 or the Paradox Network Pack on a network you need:

- Novell with Novell Advanced Netware version 2.0A or higher
- 3Com 3Plus with 3Com 3+ operating system version 1.0, 1.1 or higher
- IBM Token Ring or PC Network with IBM PC Local Area Network Program version 1.12 or higher
- Torus Tapestry version 1.4 or higher
- AT&T Starlan Network with AT&T PC 6300 Network Program version
- Other network configurations that are 100% compatible with DOS 3.1 and one of the listed networks

***System Requirements for Single User:***

- DOS 2.0 or higher
- IBM® PS/2 and PC, Compaq® PC families and other 100% compatibles
- 512K RAM
- Two disk drives, 3½-inch and 5¼-inch supported
- Compatible monochrome, color, or EGA monitor with adapter

***System Requirements for the Network Workstation:***

- DOS 3.1 or higher
- 640K RAM
- Any combination of hard, floppy, or no disk drives
- Compatible monochrome, color, or EGA monitor with adapter

*Reprinted with permission by Software Digest from its July 1987 Report covering 12 relational database programs.

**Rebate request must be received by Borland no later than February 15, 1988.
Mail to: Paradox Rebate Department, Borland International,
4585 Scotts Valley Drive, Scotts Valley, CA 95066

†Test was designed and executed by NSTL. A 1,000-record and a 10,000-record file were joined. A short text field from the 1,000-record file and a numeric field from the 10,000-record file were selected (using the 1,000-record file indexes). The short text field was grouped and sorted in ascending order, the numeric field was subtotaled for each group, and the results output to a null printer. Test times from the last keystroke on the command sequence until return of program control were recorded and averaged.

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-1158B

*continued from page 123*

### PARAMETERS AND VALUES IN PROCEDURES

A key capability of procedures is that they can take parameters and return values. Consider a procedure that calculates the length of the hypotenuse of a right triangle. The procedure then displays the hypotenuse and lengths of the other sides by means of a ? command, as shown in Listing 5, HYPO.SC.

The **Hypotenuse()** procedure is called from within the ? command. Then the body of the procedure executes, assigning the product of **x** squared to **w** and the product of **y** squared to **z**. The procedure returns the square root of the sum of **w** and **z**. The ? command displays the returned hypotenuse and lengths of the other sides, supplied by the program.

Listing 6, HYPO.C, shows the equivalent of the **Hypotenuse()** procedure in C. The difference in program size alone illustrates some efficiencies of PAL over C.

### VARIABLES IN PROCEDURES

PAL procedures can make use of PAL variables as arguments. To allow you maximum flexibility in writing scripts and procedures, PAL automatically designates any variables that function as arguments as private to that procedure. This allows you to duplicate variable names in and out of a procedure without confusion. In the **Hypotenuse()** procedure example above, variables **x** and **y** are named as arguments. These variables are not disturbed by variables with the same names outside the procedure.

Variables declared inside the procedure that are *not* formal arguments (like **w** and **z** in HYPO.SC) are *global to the script in which the procedure is called.* If you declare a variable inside the body of a procedure that has the same name as a variable outside the procedure, you may not get the result you expect. Variables that are not formal arguments might carry in a value from outside of the procedure.

If you wish to declare variables to be used for "scratch" storage inside of a PAL procedure, precede them with the **PRIVATE** keyword. In this way, the variable inside the procedure body exists separately from a variable of the same name outside the procedure.

***Variables declared inside the procedure that are* not *formal arguments are global to the script in which the procedure is called.***

### THE DETAILS OF PROCEDURE DEFINITION

In the examples above, we have seen how a procedure is defined. You simply surround a set of PAL commands with **PROC procedurename()** and **ENDPROC**. But procedure definition in PAL is more than merely adding two statements before and after a selection of commands. The actual sequence of events when you execute a procedure-defining script is significant.

When you play a script that defines a procedure, Paradox first parses all the PAL statements in the body of the procedure and stores the parsed statements in a section of memory specially reserved for them. Then, whenever the script invokes the procedure, Paradox directly accesses the preparsed statements from memory and executes them. This technique avoids Paradox having to individually interpret the procedure's statements during each invocation and results in significant performance improvements.

Once defined this way within a script, a procedure remains in memory until one of two things happens: all script play ends, or the **RELEASE PROCS** command releases all procedures from memory. Paradox reserves a portion of main memory for procedure definitions. If you define all of your procedures dynamically within script play, you may use up all of that memory. Once you reach that limit, other new procedures cannot be defined. If you wish to define more procedures than can fit in the memory reserved for them, then you must use the **RELEASE PROCS** command to free up memory for them. For example, the **greet** script described earlier might be modified to manage its procedure as shown in Listing 7, GREET2.SC.

This example defines the **hello()** procedure, executes it and then releases the memory it had occupied. It provides explicit control, but at the cost of requiring explicit programming. In a script with several procedures, such explicit control can require more attention to details than you may care to devote.

If you store procedure definitions in libraries, however, you can rely on Paradox 2.x automatic virtual memory management (VMM), described later in this article.

In contrast to PAL, C does not include its own automatic VMM system. Nor does C include such high-level constructs as the **RELEASE PROCS** command that dynamically releases memory occupied by procedures. Memory management in C is considerably different and lower-level than memory management in PAL and Paradox.

### PAL PROCEDURE LIBRARIES

PAL emulates C with its libraries much as it does with its procedures, but PAL offers libraries with a different philosophy.

A PAL library corresponds both to a C library and also to a C header file that is accessed with the **#include** directive. Both PAL libraries and C libraries or header files contain procedures and routines. Unlike C, however, PAL enables you to access a library's procedures as specifically or as generally as you wish. In C, access to routines that are not part of the main program occurs by way of linking separately compiled libraries together into a single executable file, or else by including header files into the program code.

A PAL library can contain up to 300 preparsed procedures. You create libraries in the context of a script, just as you define procedures in a script. PAL includes four commands to manage procedure libraries:

- **CREATELIB** creates a procedure library.
- **READLIB** reads procedures into memory from a library.
- **WRITELIB** writes procedures into a library.
- **INFOLIB** provides information about procedures in a library.

Listing 8, SALUTE.SC, illustrates how libraries are created. This example creates the **salute** procedure library to reside in a DOS file called SALUTE.LIB. (Paradox recognizes DOS files with the .LIB extension as procedure libraries.) The **hello()** procedure is defined, written to the salute library, and then read from the salute library. Finally, **hello()** is invoked.

To accomplish the same thing as SALUTE.SC in a C program, you would have two options. One is to put the **hello()** routine into a header file like SALUTE.H in the INCLUDE directory and include it at the beginning of the C program. Listing 9, HELLOINC.C, illustrates this first option.

The other option is to put the **hello()** procedure definition into a library in the LIB directory and to link that library into the final executable program file.

Procedure libraries developed with the **CREATELIB** command normally contain up to 50 PAL procedures, but if you use the **SIZE** number option, you can create a library with up to 300 procedures. In Listing 8 above, the **salute** library could contain up to 50 procedures, but if the statement read

```
CREATELIB salute SIZE 100
```

then the **salute** library could contain up to 100 procedures.

***Multiple procedures can be written to or read from a library in a single* WRITELIB *or* READLIB *statement.***

Procedures can be written to a library at any time after the library is created. Also, multiple procedures can be written to or read from a library in a single **WRITELIB** or **READLIB** statement. The only requirement is that the library have space for the new procedures. For example, if you create a library that allows 100 procedures, and you write the 100th procedure to that library, you will have no problem. If you attempt to write a 101st procedure to that library, the write attempt will fail.

In C, by contrast, there is no practical limit on the number of routines that can be placed in a header file or library. In addition, defining C procedures does not include preparsing, nor does it occur dynamically in a script. Instead, a C routine is defined simply by being written and stored in a header file or a library, which are simply DOS files. C routines, whether part of a library, a header file, or the main program code, must be compiled before running the programs into which they are linked. By comparison, PAL is an interpreted language with no traditional compilation process.

### USING PROCEDURE LIBRARIES

As shown in Listing 8, you can create a PAL procedure library, define procedures, write the procedures into the library, and read procedures from the library into memory; all within a single script. In small applications that perform a limited number of tasks, this can constitute thoroughness to the point of overkill. PAL procedure libraries are more appropriate in larger, more ambitious applications.

As your application takes shape, use a separate script to create a procedure library, define the procedures, and write the procedures into the library. Then you can pare down the application itself by replacing the parts that were defined as procedures with procedure calls into the library instead. You can either use the explicit **READLIB** command, specifying the library and procedure name or names as needed, or you can load the procedures automatically.

### AUTOLOADING PROCEDURES

In addition to explicitly reading (loading) procedures into memory with the **READLIB** command, you can load them automatically with an *autoload library* that contains all the procedures you need. An autoload library is one that Paradox automatically consults for procedure definitions whenever you play a script that invokes the procedures. By default, the autoload library is a library named PARADOX2.LIB in the current working directory; procedures that are stored in this library need not be loaded with **READLIB**.

You can use another library instead of PARADOX2.LIB as the autoload library. Simply assign a string to the **autolib** system variable corresponding to the library name, minus the .LIB extension. For instance, to designate SALUTE.LIB as the autoload library, simply include

```
autolib = "salute"
```

*continued on page 128*

*continued from page 127*

in your application. Then you can invoke procedures in SALUTE.LIB without explicitly reading them into memory with the **READLIB** command.

Autoloading is a convenient way to provide maximal access to procedures in your application with a minimum of code. You can reassign the **autolib** system variable as often as you wish and invoke procedures contained in the currently specified autoload library.

However, the convenience of autoloading must be weighed against the benefits of using **READLIB** and **RELEASE PROCS** for efficient application performance. **READLIB** supports the optional **IMMEDIATE** keyword, which allows you to specify when the procedure definitions are read from disk into memory. This can have a key effect on performance, since using autoloaded procedures may require a large number of disk accesses.

Further, scripts that use explicit **READLIB** and **RELEASE PROCS** commands are easier to debug than scripts that modify the **autolib** variable to cause procedures to load automatically. It's up to you to determine which convenience you prefer in developing applications.

PAL's autoloading corresponds more closely to C's use of header files loaded with the **#include** directive. However, C does not allow you to substitute one header file for another as easily and dynamically as PAL does, by merely reassigning a name to the **autolib** system variable.

## PROCEDURES AND MEMORY MANAGEMENT

As mentioned above, procedures defined within the body of a script are not subject to automatic memory management. Procedures defined in a script remain in memory until the script play ends or until they are cleared with the **RELEASE PROCS** command.

PAL procedures stored in libraries, however, are subject to Paradox's VMM system. This automatic virtual memory management system, introduced with Paradox 2.0, contains the following tools:

- automatic loading and swapping of procedures into and out of memory
- the use of expanded memory (vs. extended memory) for procedure storage
- the **SETSWAP** command, to control the point at which Paradox swaps procedures out of memory
- the **MEMLEFT** function, for additional control in memory management.

> *The convenience of autoloading must be weighed against the benefits of using* READLIB *and* RELEASE PROCS *for efficient application performance.*

As explained above, C has no equivalent to Paradox's VMM system and thus does not provide high-level tools such as those listed here for control in memory management.

## AUTOMATIC PROCEDURE SWAPPING

Paradox reserves a portion of main memory for procedure definitions. This portion of memory is not infinite, and it may not be able to accommodate all the procedure definitions that your application uses.

For this reason, Paradox automatically swaps procedure definitions into and out of memory on a least-recently-used (LRU) basis. This swapping takes place strictly as needed for minimal impact on the application's performance. This automatic memory management can save much work for application developers, because Paradox's LRU strategy has no effect on program logic or flow control.

When Paradox swaps procedures automatically, it looks for the procedure in the order of precedence shown below:

1. In main memory (including extended memory, if your computer supports it).
2. In the temporary storage area of expanded memory, if the computer has it.
3. On disk in the library from which the procedure was originally read.
4. On disk in the current autoload library.

The net effect of extended memory on computers that support it is more main memory for all operations. Thus, extended memory provides the quickest swapping of all. On computers that cannot support extended memory, expanded memory can make procedure swapping barely noticeable to application users.

However, the swapping mechanism itself is independent of how your computer is equipped, so no special coding is required to take advantage of extended or expanded memory. Any provisions that eliminate disk accesses, such as extended or expanded memory, will improve performance.

## RESTRICTIONS ON AUTOLOADING AND AUTOSWAPPING

Some important restrictions apply to automatic memory management of procedures:

- Autoloading and autoswapping in Paradox are limited to procedures that are read in from libraries with **READLIB** or accessed from the autoload library. Procedures defined within the script itself are unaffected by automatic memory management. (This can actually benefit performance when

*continued on page 130*

# WATCH YOUR LANGUAGE

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.

# WATCH *TURBO TECHNIX*

MAY/JUNE 1988

ISSUE CLOSING DATE: FEBRUARY 10

Device-independent graphics means that one source code file can support many different graphics boards, and the Borland Graphics Interface does it all for Turbo Pascal and Turbo C . . . check printer readiness with a simple PAL procedure . . . build mouse support into your Turbo C and Turbo Pascal graphics programs . . . experiment with hypertext using Turbo Prolog . . . understand how multiple indexing works with the Turbo Pascal Database Toolbox . . . perform fast text pattern searches in telecommunications programs . . . and lots more!

JULY/AUGUST 1988

ISSUE CLOSING DATE: APRIL 24

Learn how to design and write TSR's that coexist in memory with Sidekick Plus . . . create custom text devices in Turbo Pascal . . . build a meta interpreter in Turbo Prolog . . . add your own pull-down menus to MicroStar in the Turbo Basic Editor Toolbox . . . drive stepper motors through the parallel port with a TSR utility in Turbo Pascal . . . draw Bezier curves quickly . . . generate custom reports with a PAL program . . . all the TECHNIX you've come to expect, and a whole lot 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

LISTING 1: HELLO.SC

```
? "Hello, world!"
SLEEP 5000
```

LISTING 2: HELLO.C

```
#include <stdio.h>

main()
{
  printf("Hello, world!\n");
}
```

LISTING 3: GREET.SC

```
PROC hello()
        ? "Hello, world!"
        SLEEP 5000
ENDPROC

hello()
```

LISTING 4: GREET.C

```
#include <stdio.h>

hello()
{
        printf("Hello, world!\n");
}

main()
{
        hello();
}
```

LISTING 5: HYPO.SC

```
PROC Hypotenuse(x,y)
        w=x*x
        z=y*y
        RETURN (SQRT(w+z))
ENDPROC

x = 4
y = 5
? "If a right triangle has sides ",x," and ",y,","
? "then its hypotenuse is ",Hypotenuse(x,y),"."
SLEEP 4000
```

*continued from page 128*

you must rely on immediate access to a procedure whenever it is invoked.)

- The size of the largest procedure that can be loaded is a function of how many images (tables, queries, etc.) and variables are used in your application. The largest procedure cannot exceed the difference between total available memory and the sum of memory for images and memory for variables.
- Some procedures that are higher on the call chain than the currently active procedure cannot logically be swapped out. Refer to Chapter 15 (Performance and Resource Tuning) in the *PAL User's Guide* for more information.

## DEBUGGING PROCEDURES

You can define procedures at any point in the process of developing your application. However, in the debugging phase, you must observe some simple precautions so that normal debugging can take place.

Remember that procedures are sets of preparsed instructions based on PAL statements. The PAL Debugger can only debug PAL statements, not lower-level interpretations of PAL statements. However, if the PAL Debugger has access to the PAL statements from which a procedure is defined, then debugging can take place normally.

When you define a procedure and write it to a library, the full pathname of the script is stored in the library along with the procedure itself. The PAL Debugger relies on this information to locate the script containing PAL statements on which the procedure is based. Therefore, the script that defined the procedure must be in the same drive and directory that it was in when it was originally written to the procedure library, because that is where the PAL Debugger will look for the PAL code. If the script that defines a procedure is moved to a different directory, the PAL Debugger will not find it and will issue an error message.

When you debug the script on which a procedure is based, it's your responsibility to write the corrected version into the procedure's library. The PAL Debugger does not do this for you.

## USING PAL PROCEDURES IN APPLICATIONS

As a programmer you have many options on how to use PAL procedures and libraries, and the best options will be a matter of strategy. Your decisions will reflect the use that your program makes of procedures, the computer resources available to your program, and other key factors. In general, procedures are easy to define and use; how effectively they augment your application depends upon how well you fine-tune them. Here, too, PAL is like C, where a variety of techniques are available to optimize your program code, but they exist at a considerably higher level in PAL. We'll explore some of PAL's fine-tuning techniques in a future article. ■

> ***If the script that defines a procedure is moved to a different directory, the Debugger will not find it and will issue an error message.***

*Todd Freter is Senior Writer/Editor at Ansa Software.*

*Listings may be downloaded from CompuServe as PALPRO.ARC.*

### LISTING 6: HYPO.C

```
#include <stdio.h>
#include <math.h>

float Hypotenuse(x,y)
int x,y;
{
        float w,z;

        w = x*x;
        z = y*y;
        return sqrt(w+z);
}

main()
{
        int x,y;

        x = 4;
        y = 5;

        printf("If a right triangle has sides %d and %d,\n",x,y);
        printf("then its hypotenuse is %f.\n",Hypotenuse(x,y));
}
```

### LISTING 7: GREET2.SC

```
PROC hello()
        ? "Hello, world!"
        SLEEP 5000
        ENDPROC

hello()

RELEASE PROCS hello
```

### LISTING 8: SALUTE.SC

```
CREATELIB "salute"
PROC hello()
        ? "Hello, world!"
        SLEEP 5000
ENDPROC

WRITELIB "salute" hello

READLIB "salute" hello

hello()
```

### LISTING 9: HELLOINC.C

```
#include <stdio.h>
#include <salute.h>

main()
{
        hello();
}
```

# BINARY ENGINEERING

## "Go to, go to."—*Troilus and Cressida*

*Bruce F. Webster*

Consider the following opening paragraph of a letter published in a computer magazine:

> Editor:
> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of the **GOTO** statements in the programs they produce. More recently, I discovered why the use of the **GOTO** statement has such disastrous effects, and I became convinced that the **GOTO** statement should be abolished from all "higher level" programming languages (i.e., everything except, perhaps, plain machine code).

This letter could come from any one of a number of publications during the last 10 years, where arguments about the pros and cons of **GOTO** statements have raged from time to time. What makes this letter interesting, though, is that it was published 20 years ago, in the March 1968 issue of *Communications of the ACM.* It was written by Edsger Dijkstra, an internationally respected computer scientist known for his blunt and controversial statements. The letter was titled "Go To Statement Considered Harmful," and was the initial volley in a conflict that continues to this day over the proper role (if any) of **GOTO** statements in computer programs.

To a large extent, the battle is over. The anti-**GOTO** forces are the clear victors, with a two-decade emphasis on "structured programming" and the introduction of constructs such as **WHILE** loops and true **IF..THEN..ELSE** statements into formerly unrepentant languages, such as BASIC and FORTRAN. However, the pro-**GOTO** forces have not been completely vanquished. They still occupy a niche, and point out the algorithms and circumstances where use of a **GOTO** statement produces better, cleaner code than the alternative. Between the territories lies a demilitarized zone, across which both sides face off. Insults and the occasional potshot are not uncommon, but more common still are secret incursions across the border to use privately the techniques disavowed publicly. Proponents of **GOTO**-less programming usually have a few guilty **GOTO**s in their closets, while the most unrepentant **GOTO** fanatics may find themselves drifting into **WHILE** loops.

So, what's all the fighting about anyway? Why did Dijkstra (and others) argue so strenuously against the **GOTO** statement, while Knuth (and others) argued so eloquently for it? Why, 20 years after Dijkstra's letter, can you still start a heated argument in any group of two or more programmers with either of the following statements:

> I think that the **GOTO** statement should be banished from all high-level languages.
>
> I think that the **GOTO** statement is a useful and vital part of any programming language and should be used freely.

### A LOOK AT THE GOTO STATEMENT

First, we'd better clarify the subject for those unfamiliar with it. After all, there are programmers out there—good ones, too—who have never used a **GOTO** statement.

Most programs are written in programming languages that follow a sequential/procedural model, such as FORTRAN, COBOL, BASIC, Pascal, and C. Simply put, execution starts with the first statement in the program, then continues to the next, then the next, and so on until encountering the last statement (or a statement explicitly directing the program to halt). A statement might reference a procedure, also known as a subroutine, subprogram, or function. In that case, the statements in the procedure are executed in a similar fashion, after which execution continues with the statement following the procedure reference.

This is all very nice but also very boring: a program limited to executing each statement exactly once and in strict sequential order is limited, indeed. A conditional statement—**IF** some condition is true, **THEN** execute this statement—provides variation, but does little to divert the inexorable march from start to finish. Like a ski run, you can choose an alternative course down the slope, but you can't suddenly jump back up the slope to repeat a section or try several alternatives all on the same run.

Ah, but you can perform such magic in a program with a **GOTO** statement. A **GOTO** statement usually takes the form

```
GOTO <label>
```

where <label> is an identifier or line number telling which statement to go to next. This lets you go back to a preceding (in sequential order) statement or skip over some following statements.

Note that the **GOTO** statement isn't very useful without the **IF** statement (or its local equivalent).

### CONTROL STRUCTURES

With both the **GOTO** and **IF** statements, you can build all the major control structures used in procedural languages: **FOR**, **WHILE**, **REPEAT**, **IF..THEN..ELSE**, and **CASE**. Consider the parallel constructs shown in Table 1.

```
FOR X := A TO B DO                          X := A;
  <statement>;                      Again: if X > B GOTO Done;
                                           <statement>;
                                           X := X + 1;
                                           GOTO Again;
                                    Done:
-----------------------------------------------------------------
WHILE <condition> DO                Again: if not <condition>
  <statement>;                                then GOTO Done;
                                           <statement>;
                                           GOTO Again;
                                    Done:
-----------------------------------------------------------------
REPEAT                              Again: <statement>;
  <statement>                              if not <condition)
UNTIL <condition>;                           then GOTO Again;
-----------------------------------------------------------------
IF <condition>                             if not <condition>
  THEN <statement1>                         then GOTO Else;
  ELSE <statement2>;                              <statement1>;
                                           GOTO Done;
                                    Else:  <statement2>;
                                    Done:
-----------------------------------------------------------------
CASE V OF                                  if (V = C1) then GOTO Do1;
  C1 : <statement1>;                       if (V = C2) then GOTO Do2;
  C2 : <statement2>;                       if (V = C3) then GOTO Do3;
  C3 : <statement3>;                       <statement4>;
  ELSE <statement4>                        GOTO Done;
END;                                Do1: <statement1); GOTO Done;
                                    Do2: <statement2); GOTO Done;
                                    Do3: <statement3);
                                    Done:
```

*Table 1. The classic structured statements implemented with GOTO.*

As you can see, all the fancy statements on the left are just disguised combinations of **IF** and **GOTO**, though easier to read and understand. However, they don't provide any functionality that **IF** and **GOTO** don't, and there are certainly some things that you can do with **GOTO** that you cannot with these statements. Why use them, and why avoid using **GOTO**?

### PROBLEMS WITH THE GOTO STATEMENT

Dijkstra's letter focused on one major concern: program verification and debugging. He noted that the ability to clearly trace the sequence of execution is a tremendous help in getting your program to work. As procedures and control structures are introduced (such as those in Table 1), tracing becomes more complex, but is still well defined, and the intent of the program remains clear.

Unbridled use of the **GOTO** statement, though, can quickly complicate a program beyond reasonable human comprehension. When Dijkstra wrote his letter, the predominant computer languages—COBOL and FORTRAN—had little in the way of control structures beyond the simple **IF** and the **GOTO**, so programmers used them as shown in Table 1. Unfortunately, not all of them understood how they were using them, nor did they limit themselves to well-defined control structures. The results were programs that (to use Martin Hopkin's phrase), "look like a bowl of spaghetti"; hence the derogatory phrase, "spaghetti code." (Hopkins 1972).

What's wrong with "spaghetti code"? Simply put, it's a pain to debug and modify. Effective debugging of a program usually involves looking at a set of statements and being able to clearly define all possible conditions under which that code is executed. If there is a clearly defined flow path to those statements, then tracing back along that path lets you see everything that's happened up to that point. That, in turn, helps you to define the set of conditions for that code's execution.

But what if you have **GOTO** statements in different locations in your program, all branching to one chunk of code? Defining the set of conditions under which that code is executed now becomes complicated, since you have multiple flow paths to it. And it is even more complicated when those flow paths themselves are entangled with **GOTO** statements. The "bowl of spaghetti" imagery quickly replaces that of a single, complex-but-traceable path.

### ARGUMENTS FOR ELIMINATING THE GOTO STATEMENT

Dijkstra has never been known for pulling punches; this is, after all, the man who said, "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense." (Dijkstra 1982). So when he saw the problems caused by the **GOTO** statement, he simply proposed dropping it from all high-level languages. Others jumped on the bandwagon, and the elimination of **GOTO** statements, both from languages and from existing programs, became something of a crusade.

The key to this movement was a set of papers showing how all uses of **GOTO** statements could be replaced by program reorganization and appropriate use of con-

*continued on page 134*

*continued from page 133*

trol structures. Most often cited was a paper with the inviting title of "Flow Diagrams, Turing Machines, and Languages with Formation Rules." (Böhm and Jacopini 1986) This highly theoretical paper proved, in essence, that you can turn any flowchart (hence, any piece of spaghetti code) into a structured program given the proper control structures. Since it had been "proven" that you didn't need **GOTO** statements, and since **GOTO** statements were so heavily abused, therefore they should be eliminated altogether.

Perhaps the best summary of the arguments against the **GOTO** statement are presented in a paper by W. A. Wulf. (1972) His initial comments focus on the problems resulting from abusing the **GOTO** statement and the need for clearer program structure, especially for large programs. He reviews the theoretical proofs showing that you can eliminate the **GOTO** statement, then he addresses the practical issues of convenience and efficiency. He concludes that in order to "produce large programs of predictable reliability . . . unrestricted branching between components cannot be allowed." His solution is to replace all uses of the **GOTO** statement with the corresponding control structures.

## ARGUMENTS FOR KEEPING THE GOTO STATEMENT

The theory is all well and good, but the first problem you encounter is that not all languages have the control structures necessary to replace all uses of the **GOTO** statement. When Dijkstra wrote his letter to *CACM*, FORTRAN really only offered the **FOR** loop, the logical and arithmetic **IF** , and the computed **GOTO** (a simplistic **CASE** statement). Today, many versions of BASIC still only offer the **FOR** statement and an **IF..GOTO** statement.

In these situations, use of the **GOTO** is unavoidable and, in fact, desirable—as long as it builds well-formed control structures, such as those shown in Table 1. Unfortunately, it seldom stops there, and the usual problems of unwise **GOTO**s arise. It's better for the language to implement the control structures directly; for example, the **IF..THEN..ELSEIF ..ELSE..ENDIF** and **WHILE..WEND** structures found in some versions of BASIC. But lacking that, **GOTO** statements are often necessary and wise.

A second problem is that certain algorithms or sections of code are more understandable and/or more efficient with **GOTO** statements than without. Donald Knuth, perhaps one of the best known and most iconoclastic computer scientists, took the anti-**GOTO** movement as a challenge. He jointly published several papers demonstrating that code fragments containing **GOTO** statements could not be made **GOTO**-less without introducing additional calculations or variables. He also showed that the readability of the code suffered as well. His provocatively entitled paper, "Structured Programming with go to Statements," lists several such examples. (Knuth)

Interestingly enough, Knuth contradicts himself a bit in that same paper. After stating the famous dictum that "premature optimization is the root of all evil," he goes on to justify the use of **GOTO** statements in certain instances in order to avoid an extra test in an inner loop.

Another look at keeping the **GOTO** statement is found in a paper by Martin Hopkins (1972). He puts forth five basic arguments. First, as mentioned earlier, the alternatives to **GOTO** statements aren't always there. Second, the **GOTO** statement can be used to create new control structures; after all, someone had to invent the **CASE** structure (in this case, Tony Hoare back in 1966). Third, given the pressures and demands of certain programming tasks, "sometimes a **GOTO** is a useful, if ugly, tool to handle an awkward situation." Fourth, the poor code generated by compilers structures often leads programmers to use **GOTO**s in the name of efficiency. Fifth, Hopkins fears a monolithic programming style will stifle creativity. He needn't have worried.

## GUIDELINES

So, when should you use **GOTO** statements, or should you use them at all? Even among the proponents of the **GOTO** statement, the message is clear: Use it only when it is a superior alternative. What are those circumstances?

- When the language doesn't contain the control structure desired. For example, if you're writing code that needs a **WHILE** loop and the language doesn't provide one, you don't have much choice. In such cases, be familiar with the constructs in Table 1 and drop them in as "boilerplate" code, plugging in the necessary substitutions.
- When it makes an algorithm more clear. Certain algorithms that are concise and straightforward when using **GOTO**s can become expanded and muddled when the **GOTO**s are dropped. Don't sacrifice clarity for ideology, but first be sure that the algorithm is indeed clearer with the **GOTO**s than without.
- When it makes an algorithm significantly more efficient. If using a **GOTO** allows you to drop a statement or comparison from a time-critical inner loop that is executed thousands or millions of times, do it. However, beware of making your code obscure for marginal improvements in performance.
- When it's late, you're tired, you know the **GOTO** will do the trick, and you're only going to use the program once anyway. Hopkins' comment is, "I tend to sympathize with the programmer who fixes up a one-time program at 3:00 a.m. with a **GOTO**." Of course, Hopkins also says, "A programmer should be able to justify each use of **GOTO**."
- When none of the above conditions is true, but you really want to irritate some acquaintance who is fanatic about

avoiding **GOTO** statements and who will eventually read your code.

One of the few necessary uses of **GOTO** in Pascal is breaking out of a **FOR** loop before the control variable has stepped through its range. In most Pascal implementations, modifying the control variable to terminate the loop early is either forbidden during the compile pass or else generates unpredictable code. Two keywords associated with the **FOR** loop, **Break** and **Cycle**, have been implemented in only a few Pascal compilers, *not* including Turbo Pascal. **Cycle** short-circuits the loop, increments the control variable, and begins at the top of the loop again. **Break** ends the loop and passes control to the first statement following the **FOR** statement.

Without **Break**, Turbo Pascal must resort to **GOTO**:

```
FOR I := 1 TO 50 DO
  BEGIN
    .  .  .

    IF PanicButton THEN GOTO 100

    .  .  .

  END
100: Writeln('Loop finished!');
```

Granted, this structure might have been written as a **WHILE** loop, but it would not have been as obvious. Situations like this do not come up often, but when they do, they can be devilishly difficult to massage into a different but equivalent form.

Given the right control structures, it is possible to write vast amounts of code without ever using a **GOTO** statement. I know; I've written vast amounts of Pascal and can count the number of **GOTO** statements I've used on one hand. Once I copied a shellsort algorithm out of a book. The other two or three times I was a few levels deep in control structures, wanted to exit the procedure I was in, and the Pascal compiler I was using didn't have the **Exit** statement. Even there, I could have rewritten the procedure to avoid the **GOTO**, but I felt the code was cleaner than the rewritten version would have been.

By the way, this avoidance hasn't been out of ideological fervor. I programmed heavily in FORTRAN for several years before using Pascal. However, I was so tickled at all the new tools I had to use that I never got around to making the **GOTO** statement part of my Pascal programming style.

That is probably the best approach for you as well: try not to make the **GOTO** statement part of your programming style. It can be a useful tool, but it is used best when used sparingly. ■

## REFERENCES

The papers cited in this article are reprinted in *Classics of Software Engineering* (edited by Edward Nash Yourdon, New York: Yourdon Press, 1979).

1. Dijkstra, Edsger. "Go To Statement Considered." *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp. 147-148. Reprinted in *Classics*, pp. 29-33.

2. Hopkins, Martin E. "A Case for the GOTO." *Proceedings of the 25th National ACM Conference*, August 1972, pp. 787-90. Reprinted in *Classics*, pp. 101-109.

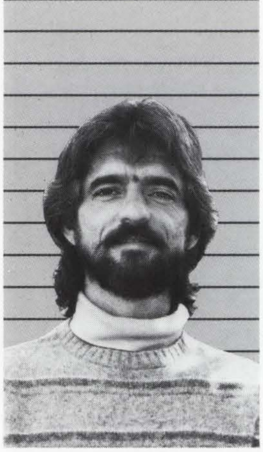
3. Dijkstra, Edsger. "How Do We Tell Truths That Might Hurt." Found in *Selected Writings on Computing: a Personal Perspective*, Springer-Verlag, 1982, pp. 129-131.

4. Böhm, C. and Jacopini, G. "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules." *Communications of the ACM*, Vol. 9, No. 5 (May 1966), pp. 366-71. Reprinted in *Classics*, pp. 13-25.

5. Wulf, W. A. "A Case Against the GOTO." *Proceedings of the 25th National ACM Conference*, August 1972, pp. 791-97. Reprinted in *Classics*, pp. 85-98.

6. Knuth. D. E. "Structured Programming with go to Statements." Taken from *Current Trends in Programming Methodology*, Volume 1, Raymond T. Yeh, ed. Reprinted in *Classics*, pp. 259-321.

*Bruce Webster is a computer mercenary living in the Rockies. He can be reached at Jadawin Enterprises, P.O. Box 1910, Orem, UT 84057; via MCI MAIL (as Bruce Webster) or on BIX (as bwebster.)*



# LANGUAGE CONNECTIONS

## Monochrome graphics in two languages.

*Gary Entsminger*

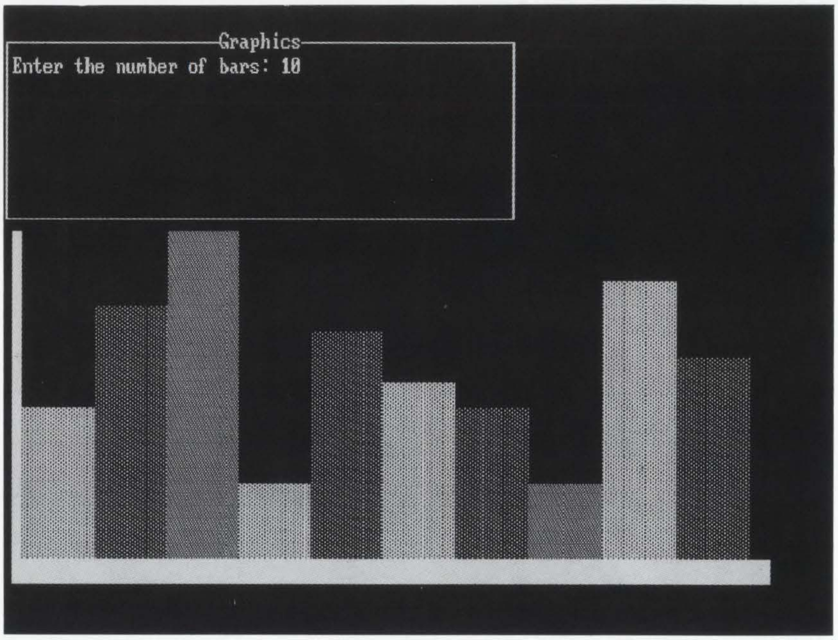
You can spruce up almost any application by adding a window interface and/or a graphics display. Turbo Prolog's built-in windowing commands, **makewindow**, **shiftwindow**, **removewindow**, **window__attr**, and **window__str**, make this especially easy.

Creating graphics displays is only slightly more difficult. If you have an EGA or CGA video card, you can use the built-in **graphics** predicate and its support predicates, **dot** and **line**, to create delightful colors and shapes.

But owners of Turbo Prolog 1.1 will find that monochrome graphics modes are not supported. This means that if you need graphics with a monochrome graphics card (such as the Hercules card), you must develop your own graphics interface. This isn't as hard as it sounds. You can create surprisingly good images on Hercules and monochrome cards, and on any system in general, by using character graphics (in text mode).

In fact, writing to a text mode display is generally faster than writing to a graphics mode display, since we're writing blocks instead of individual pixels. The tradeoffs are obvious—no color, no grand details, and no diagonal lines—but we still have a lot of power.

For example, we can draw a pretty fair bar graph by writing characters directly to screen memory. All it takes is a little understanding of how the display works, and some programming tools.



```
Graphics
Enter the number of bars: 10
```

*Figure 1. Bar graph produced by the program in Listings 1 and 2.*

### SETTING UP THE BAR

Let's describe a program for drawing a bar graph on a monochrome system. The program combines the high-level power of Turbo Prolog and the low-level power of Turbo C. Our minimum requirements are to:

1. Set up a simple interface for communicating between program and user
2. Get information (scale, etc.) about the bar graph from the user
3. Set up the graph (calculate axis values, bar widths, and heights)
4. Clear the screen
5. Remove the cursor
6. Draw the graph
7. Interact again with the user (do we continue or not?)
8. Clear the screen
9. Restore the cursor.

The program should also consider the ways a user might try to blow up the program (by entering bar values outside a range, etc.). Listing 1 shows the Turbo Prolog code for this sequence of events.

Prolog's built-in **makewindow** predicate sets up the interface, and uses read and write statements to interact with the user. Control is then passed to Turbo C for the low-level acrobatics.

## THE VIDEO ADAPTER

The EGA, CGA, and monochrome display adapters use different addresses, so if we write to the wrong address, nothing (visible) happens on the CRT. The base address for the monochrome and Hercules video cards is b000H, defined as **SCREEN__BASE** in **defs.h** (see Listing 3). Change this address for other adapters. The **SCREEN__WIDTH** is set to 80 characters (also shown in Listing 3). If you have another screen size, change this definition as well.

We can write a character and its attribute directly to the screen with Turbo C's built-in function **pokeb**, which requires three parameters—**Segment**, **Offset**, and **Value**. For example, to write the letter A to the screen at (10,20) in normal video (i.e., white characters on a black background) you would make the following calls:

```
_pokeb(SCREEN_BASE,
      ((10 * SCREEN_WIDTH)
      + 20) * 2,0x41);
_pokeb(SCREEN_BASE,
      (((10 * SCREEN_WIDTH)
      + 20 ) * 2) + 1, 7);
```

**SCREEN__BASE** is equal to the address of the video adapter card

```
(((y * SCREEN_WIDTH) + X) * 2)
```

which is equal to the offset address (we have to multiply by 2 to coordinate the 8088's and 6845's view of memory: the 6845 doesn't take attribute bytes into consideration). 0x41 is the ASCII hexadecimal code for a capital A, and 7 is the normal attribute code.

## THE 6845 CRT CONTROLLER

The 6845 CRT Controller is the brain of the video adapter, selecting each character code to be displayed from the adapter memory and controlling the horizontal and vertical characteristics of the display, as well as the cursor image. We'll use it to turn the cursor off and restore it after we've finished drawing our bar graph. To program the controller, all we need do is write to its registers (at 3b4H and 3b5H). We must write to 3b4H first in order to select any of the 6845's 16 other registers that begin at I/O port address 3b5H.

*continued on page 138*

LISTING 1: P-BAR.PRO

```
/* Listing 1 -- PROLOG MAIN MODULE */

global predicates
   draw_bar(integer,integer,integer,integer) -
                              (i,i,i,i) language c
   clear_screen language c
   remove_cursor language c
   draw_axis language c
   restore_cursor language c

predicates
   main
   process
   set_up_bar(integer,integer,integer,integer,real)

goal                            /* PROLOG module must contain */
   main.                        /* a GOAL in order to compile */
                                /* to .OBJ. */

clauses
   main:-                       /* Set up user interface. */
      makewindow(1,7,7,"Graphics",0,0,25,80),
      process.                  /* Process the problem. */
   main:-                       /* Continue? */
      write("Do you want to continue? y or n"),
      readchar(Answer), Answer = 'y',
      main.                     /* Call main recursively. */
   main:-                       /* We're done. */
      nl,write("Press any key to end."),
      readchar(_),
      clear_screen(),           /* Clean up for next applic.*/
      restore_cursor().

   process:-                    /* Process the problem. */
      write("Enter the number of bars: "),
      readint(NumOfBars),
      clearwindow,
      NumOfBars <= 10,          /* The system is currently */
                                /* set up to handle 10 bars.*/
      write("Enter max value on Y scale: "),
      readint(Max),
      Max > 0,                  /* Avoid division by zero. */
      clearwindow,
      YScale = 18/Max,          /* Our axis is currently set */
                                  /*   up as 18 by 60. */
      Width = 60/NumOfBars,
      Xstart = 2,                 /* Start the first bar next */
                                  /*   to the y axis. */
      clear_screen(),           /* Call C functions. */
      remove_cursor(),          /* Who needs it? */
      draw_axis(),
      Num = 1,                  /* Set Num for first bar. */
                                  /* And begin. */
   set_up_bar(Num,NumOfBars, Width, Xstart, Yscale),
   fail.                     /* Fail when we're finished */
                               /*  and reclaim memory. */

                               /* Error report. */
process:-
   write("System is currently set up to draw 10 bars"),
   write(" & Y > 0, <= 30,000 max."),
   nl, fail.

set_up_bar(0,_,_,_,_):-!.         /* There are no bars to */
                               /* process, so we're done. */
set_up_bar(Num,NumOfBars,Width,Xstart,Yscale):-
   Num <= NumOfBars,         /* Are we done? */
   write("Enter the value for bar ",Num ,": "),
   readint(Y),
   Height=Y*YScale,          /* Calculate height of bar. */
                               /* Call C function to draw. */
   Height <= 18,             /* Value <= Y max? */
   draw_bar(Num,Xstart,Height,Width),
                               /* Get next x position. */
   XNewPos= Xstart + Width + 1,
   XNuNew=Num + 1,           /* Keep count of bars. */
                               /* Continue getting bar */
                               /* values until we're done. */
   !,set_up_bar(XNuNew,NumOfBars,Width,XNewPos,Yscale).
set_up_bar(Num,NumOfBars,_,_,_):-
   Num >= NumOfBars,!.       /* We're done. */
set_up_bar(Num,NumOfBars,_,_,_):-
   Num <= NumOfBars,         /* There's an input error. */
   write("Bar value exceeds Y max."),
   !.
```

LISTING 2: C-BAR.C

```
#include "defs.h"  /* Hercules #defines */

/* Put a character and its attribute anywhere on the screen. */

void putc_at_location(char ch, int x, int y, unsigned char attrib)
{
_pokeb(SCREEN_BASE,((y * SCREEN_WIDTH) + x) * 2,ch);

          /* SCREEN_BASE = Address of the video adapter card;
              ((y * SCREEN_WIDTH) + X) * 2 = offset address */

_pokeb (SCREEN_BASE,(((y * SCREEN_WIDTH) + x) * 2) + 1, attrib);

}
```

## LANGUAGE CONNECTION

*continued from page 137*

For example, to change the shape of the cursor on the monochrome adapter, we write a 10H to I/O port address 3b4H (the cursor start register). Then we set the value we want for the cursor (e.g., 12 for a two-line underline 12-10) at 3b5H. To turn the cursor off, we write a 10H to 3b4H and a 20 (the code to turn the cursor off) to 3b5H. (Note: the Hercules and monochrome adapters produce a character image of 14 scan lines. If you're working with a CGA adapter, you only have eight scan lines, so you need to write a smaller value (for instance, 6) to 3b5H.)

Also, note that in order to address the 6845's registers, we must write to both addresses (3b4H and 3b5H).

We can write to the 6845 register address by using the built-in C function **outportb**, which requires two parameters (the address, and an integer value):

```
_outportb(0x3b4,10);
_outportb(0x3b5,20);
```

The complete remove and restore cursor functions (in Turbo C) are in Listing 2.

### THE BAR GRAPH

The easiest way to draw a bar in character mode is to fill a block (or bar) of locations with a character. In other words, each bar is equal to a rectangle without a border. In order to decide how big to make each bar we need to know the scale we're using to draw the bars, as well as each bar's height and width. We'll calculate these values from the user's input.

In my example (Listings 1 and 2), I've set up an X/Y axis 18 (Y axis) by 72 characters (X axis). After the user supplies the maximum value for the Y axis, we calculate the value of one unit on the Y axis by dividing 18 by the chosen maximum value. The height of each bar is then the user-supplied value for each bar multiplied by the value of one unit.

To get the width of each bar we divide the total width (72 characters) by the number of bars. This distributes the bars evenly across the graph.

The other order of business is to decide which graphics characters will fill our bars. I've chosen the three characters with hex values b0, b1, and b2. By setting up a **switch** statement in C (see Listing 2), we can alternate the characters, making our bars more readable. The resulting graph of 10 bars is in Figure 1.

## THE CONNECTION

Looking over the Turbo Prolog code, you should first note that the main module of our mixed-language program is written in Turbo Prolog, so the Turbo Prolog compiler handles all the memory management and function calling (see "Language Connections" in the November/December issue for details on memory management).

Our Turbo C functions are subroutines of the Turbo Prolog main program, so we must declare them as **global predicates**, and we must specify the flow patterns of the functions. We omit parameters and flow patterns for the few functions that neither receive nor pass variables or values.

The user interface implements **makewindow** within a recursive loop to allow for successive runs (see **set_up_bar** in Listing 1). By using a **fail** at the end of **process**, the program backtracks, thus freeing any memory previously allocated. I've also included some error checking (see the predicates **main**, **process**, and **set_up_bar** in Listing 1) to protect the program from errant user input. If a value is out of range, the program explains why it's not executing, and asks the user if he/she wants to continue.

Note that we can include files in the Turbo C code, but if an include file (or library) contains a built-in function, we must prefix the function with an underscore. The built-in functions **pokeb** and **outportb** must be written as **_pokeb** and **_outportb**, respectively. This requirement holds in spite of the fact that the Turbo C compiler option **G**enerate underbars has been turned off.

*continued on page 142*

```
/* Draw a bar. If height = 0, don't draw the bar,
   just go to the next. Use different characters for
   different bars up to a maximum of 10 bars. Add more
   bars by adding more case statements. */

void draw_bar_0(int no,int start_x, int height, int width)
{
        int i,j,end_x,end_y,start_y;

        switch (height){
          case 0 :
            break;
          default:
          end_x = start_x + width;
          end_y = 22 - height;
          start_y = 22;
            for (j = start_x; j <= end_x; ++j)
              for (i = end_y; i <= start_y; ++i)
                switch (no){
                case 0 :
                  putc_at_location(0xB1, j,i, 7);
                  break;
                case 1 :
                  putc_at_location(0xB2, j,i, 7);
                  break;
                case 2 :
                  putc_at_location(0xB0, j,i, 7);
                  break;
                case 3 :
                  putc_at_location(0xB1, j,i, 7);
                  break;
                case 4 :
                  putc_at_location(0xB2, j,i, 7);
                  break;
                case 5 :
                  putc_at_location(0xB0, j,i, 7);
                  break;
                case 6 :
                  putc_at_location(0xB2, j,i, 7);
                  break;
                case 7 :
                  putc_at_location(0xB0, j,i, 7);
                  break;
                case 8 :
                  putc_at_location(0xB1, j,i, 7);
                  break;
                case 9 :
                  putc_at_location(0xB2, j,i, 7);
                  break;
                case 10 :
                  putc_at_location(0xB0, j,i, 7);
                  break;
                }
```

# Identifying the CPU from Turbo Prolog

Those Turbo Prolog programmers who read Juan Jimenez's article about identifying the system CPU with an assembly language subroutine called from Turbo Pascal (elsewhere this issue) might want to add a similar little routine to their Turbo Prolog repertoire. It's an easy connection, but one requiring a few alterations in Mr. Jimenez's assembly language code.

First, let's write the Turbo Prolog part—a little program that calls the assembly language subroutine, then writes it to the screen after getting the CPU value. See Listing 1a.

Note the global predicate declaration and the flow pattern of the subroutine. Since we're not going to pass anything to the subroutine, but we do want to get a value back, our flow pattern is **(o)**.

Next, we'll write the assembly language interface. Since the internal part of the subroutine is identical whether you use Turbo Pascal or Turbo Prolog, refer to the Jimenez article for the code. I will, however, detail the Turbo Prolog-to-assembler interface.

First, we rename our external procedure by appending a **__0**, which enables Turbo Prolog to recognize it as an external procedure. Also, Turbo Prolog requires all external procedures to be **FAR**, so we call it accordingly. Then, we preserve the Base Pointer (**BP**), and load the Stack Pointer (**SP**) so that we can use **BP** to address parameters. See Listing 2a.

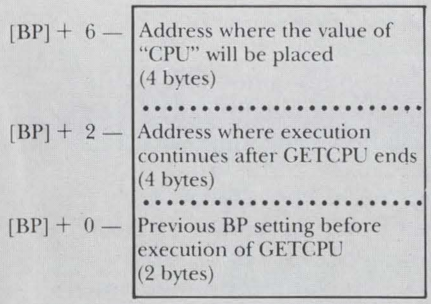
Each call to an external subroutine has an *activation record*, which goes onto the stack. This record includes the previous **BP** setting (before we called the external subroutine), the address from which execution is to continue after the subroutine is executed (**IP**), and the values and addresses of input or output parameters. See Figure 1a.

LISTING 1a: PROCPU.PRO

```
/* Listing 1a -- PROCPU.PRO which calls the assembly language
                 subroutine and displays the CPU's name

TO LINK:  Execute the following link line from the DOS prompt:

          LINK init PROCPU ASMCPU PROCPU.SYM,GETCPU,,PROLOG

*/

code = 2500

global predicates
   getcpu(integer) - (o) language asm

goal
     getcpu(CPU),                        /* Call ASM subroutine. */
     makewindow(1,77,7,"",10,20,10,40), /* Write result */
     write("Processor is 80",CPU),
     nl,
     write("<RET>"),
     readln(_),
     removewindow.
```

LISTING 2a: ASMCPU.ASM

```
; Set up for call from Turbo PROLOG

CSEG      SEGMENT
          ASSUME CS:CSEG,DS:CSEG,ES:CSEG,SS:CSEG

PUBLIC    getcpu_0            ; Turbo PROLOG expects external procedures
                              ; to end with "_0"
getcpu_0  PROC FAR            ; Turbo PROLOG requires all ext procedures
                              ; to be "FAR"
          PUSH BP             ; Save old Base Pointer, and load Stack
                              ; Pointer, so that BP can be used to
          MOV BP,SP           ; address parameters

; MAIN subroutine (code contributed by Juan Jimenez)

          PUSHF               ; Save the flag registers
          XOR AX,AX           ; Clear AX and push it onto the stack
                              ; etc..............
          PUSH AX
          POPF                ; Pop 0 into flag registers (all bits to
                              ; 0),
          PUSHF               ; attempting to set bits 12-15 of flags to
                              ; 0's
          POP AX              ; Recover the saved flags
          AND AX,08000h       ; If bits 12-15 of flags are set to zero
          CMP AX,08000h       ; then cpu is 8088/86 or 80188/86
          JZ  _8x_18x
```



[BP] + 6 — Address where the value of "CPU" will be placed (4 bytes)

[BP] + 2 — Address where execution continues after GETCPU ends (4 bytes)

[BP] + 0 — Previous BP setting before execution of GETCPU (2 bytes)

*Figure 1a. Activation record for the call to* **getcpu.**

When a program calls an external subroutine, the **CALL** instruction (generated in this case by Turbo Prolog) stores (or **PUSH**es) the instruction pointer (**IP**) onto the stack. This is essential since the assembly language subroutine uses the **IP** itself. In order to return to the correct instruction in our Turbo Prolog program we need to retrieve the original **IP**.

Since all external calls in Turbo Prolog are **FAR**, this **PUSH** requires four bytes—two for the contents of the **CS** register and two for the contents of the **IP**. When our subroutine finishes execution, it **POP**s the original value of the **IP** off the stack.

The results of the CPU check (handled by the subroutine) are in the **AX** register. In order to get these results back to our Turbo Prolog program, we put them onto the stack (the next space = **[BP] + 6**), where Turbo Prolog expects to find them.

Then, we restore the Base Pointer and return to Turbo Prolog, deallocating the space we used for parameters on the stack (four bytes in this case for the output parameter).

And that's it. By making slight variations (depending on the number of parameters and the flow pattern) you can use this method to call most external assembly language subroutines from Turbo Prolog. ■

*—Gary Entsminger*

*Listings may be downloaded from CompuServe as PROCPU.ARC*

```
; Decide whether CPU is 80286 or 80386

         MOV  AX,07000h    ; Try to set flag bits 12-14 to 1's
         PUSH AX           ; Push the test value onto the stack
         POPF              ; Pop it into the flag register
         PUSHF             ; Push it back onto the stack
         POP  AX           ; Pop it into AX for check
         AND  AX,07000h    ; If bits 12-14 are cleared then the chip
         JZ   _286         ; is an 80286

; It's an 80386

         MOV  AX,386       ; It's not a 286, so it must be a 386
         JMP  DONE

; It's an 80286

_286:    MOV  AX,286       ; Get the msg ready
         JMP  DONE

; It's an 8088/86 or 80188/86

_8x_18x:
         MOV  AX,0ffffh    ; Set AX to all 1's
         MOV  CL,33        ; Now we try to shift left 33 times. If
         SHL  AX,CL        ; it's an 808x it will shift it 33 times,
                           ; id it's an 8018x it will only shift one
                           ; time.
         JNZ _18x          ; Shifting 33 times would have left all
                           ; 0's. If any 1's are left it's an
                           ; 80188/186
         MOV  AX,86        ; No 1's, it's an 8088/86
         JMP  DONE

; It's an 80188 or 80186

_18x:    MOV  AX,186       ; Found a 1 in there somewhere, it's an
                           ; 80188 or an 80186

; New "DONE" gets us back to Turbo PROLOG

DONE:
         LDS SI,DWORD PTR [BP]+6  ; Move returned value's address
                                  ; into SI (another register which
                                  ; allows indirect addressing can
                                  ; be used
         MOV [SI],AX              ; Move the value in AX (i.e. the
                                  ; processor # to the address in SI
         POPF                     ; Restore the flag registers
         POP BP                   ; Restore Base Pointer
         RET 4                    ; Return to next execution address
                                  ; IP and de-allocate parameters (4
                                  ; bytes for returned variable)


getcpu_0  ENDP
CSEG      ENDS
END
```

```
        }
}

/* Draw an inverse video x/y axis. */

void draw_axis_0()
{
        int x,y;
        y = 23;
        for (x = 1; x <= 73; ++x)       /* from 1,23 to 1,73 */
           putc_at_location(0xdb,x,y,7);
        x = 1;
        for (y = 3; y <= 23; ++y)       /* from 3,1, to 23,1 */
           putc_at_location(0xdb,x,y,7);
}

/* Clear the screen by writing blanks to every
   screen location. */

void clear_screen_0()

{
unsigned int x,y;
  for (y = 0; y <=25; ++y)
    for (x = 0; x <=80; ++x)
      putc_at_location(' ',x,y,7);
}

/* Remove or restore the cursor by writing to the
        6845 CRT Controller. */

void remove_cursor_0()

{
  _outportb(0x3b4,10);          /* 0x3b4 = PC I/O address register
                                   of the 6845;
                                   10 = 6845 register which
                                   defines cursor start */
  _outportb(0x3b5,20);          /* 20 = code to turn cursor off */
}

void restore_cursor_0()
{
  _outportb(0x3b4,10);          /* 0x3b4 = PC I/O address register
                                   of the 6845;
                                   10 = 6845 register which
                                   defines cursor start */
  _outportb(0x3b5,(CURSOR)); /* CURSOR = cursor end */
}
```

LISTING 3: DEFS.H

```
/* Listing 3 - C Support module */

#define SCREEN_BASE 0xb000    /* base address of hercules card */
#define CURSOR 12
#define SCREEN_WIDTH 80
```

*continued from page 139*

Our programmer-defined Turbo C functions that are called directly by Turbo Prolog must have a **_0** suffix, since Turbo Prolog requires it. This applies to **clear_screen**, **remove_cursor**, **draw_axis**, and **draw_bar**. **putc_at_location**, on the other hand, is only called by another Turbo C function (not by Turbo Prolog), so it needs no underscore cosmetics.

To make the executable program, link the .OBJ code (generated by the source code in Listings 1 and 2) with INIT, CPINIT, the symbol table, and the PROLOG and CL libraries. Your command line (for linking) should look like:

```
Tlink init cpinit p-bar c-bar
      p-bar.sym,bar,,prolog+cl
```

As always, when you compile the Turbo C code, remember to use the large-memory model; select **J**ump optimization (optional); and turn off **U**se register variables and **G**enerate underbars.

You can, of course, make a number of enhancements. For instance, you might experiment by reading the information needed by the bar graph from a file, or you might push on into graphics mode if you have a Hercules, CGA, or EGA adapter. Initializing the Hercules adapter for graphics mode (as opposed to text mode) is a bit tricky, so I'll save it and the EGA for another time. ■

## REFERENCES

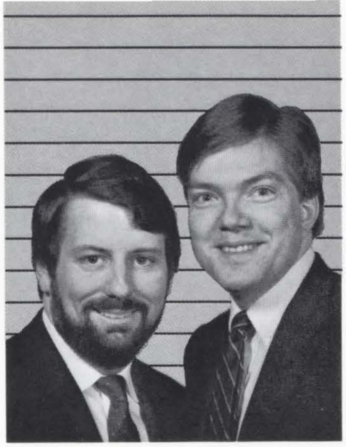
Eckel, Bruce. "Tools For Quick System Construction." *Micro Cornucopia* #38, 1987.

Kernighan, B.W. & D.M. Ritchie. *The C Programming Language*, Prentice-Hall, Inc., 1978.

Willen, D.C. & J.I. Krantz. *8088 Assembler Language Programming: The IBM PC*, Howard Sams & Co., 1983.

*Listings may be downloaded from CompuServe as LCV1N2.ARC*

*Gary Entsminger is a freelance writer and is an associate editor of* Micro Cornucopia *magazine.*



# TALES FROM THE RUNTIME

## Expanding wildcard support

*Bill Catchings and Mark L. Van Name*

In our last column we discussed a directory structure that is helpful when you begin to modify parts of the Turbo C Runtime Library source code. This time we make some changes to that source code, and we assume that you are using this directory structure.

### EXPANDING WILDCARD SPECIFICATION

While Turbo C offers many useful functions, it lacks one feature that is offered by C under the UNIX operating system: the automatic expansion of wildcard file specifications. In DOS you can type commands such as "Dir *.Bat" and see a list of all files that match the given specification. UNIX carries this ability further. The command processor, or shell, expands any wildcard file specifications it encounters on a command line before the filenames are passed to the program being executed.

For example, if you have a word counting program named **wc** and type the command "wc *.bat" from the shell, the shell replaces "*.bat" with a list of files that match that template. Each filename is an individual null-terminated string. If three files matched this specification, **wc** would see its **argc** as four (these three plus one for the program name) and its **argv** would contain the three filenames. The programmer who wrote **wc** would not have to do anything to get this feature.

You can mimic this ability by calling a routine from within your **main** routine to make a wildcard expansion pass over **argv**. It then adjusts **argc** and **argv** appropriately, so that the rest of the code never knows the difference. This approach works, but it requires you to modify every program, and it is not as elegant as the UNIX approach. Because we miss this feature, and because we prefer elegance, we went into the Runtime source and changed it so that wildcard expansions occur before your **main** routine is ever called. Once you have installed the changes, you'll never worry about wildcard expansion again!

We limited our changes to two files, one old and one new. Both are shown in the accompanying listings. The new one, WILD.C (Listing 1), does the actual work of expanding wildcard file specifications. We modified the existing file, SETARGV.ASM (Listing 2) in order to force a call to our routine prior to calling the **main** routine. When you're ready to add them to the library, use UPDC.BAT (Listing 3) to update WILD.C, and UPDASM.BAT (Listing 4) for the new SETARGV.ASM. Don't be alarmed if both of these procedures take a little while to execute, because each one has to update five libraries, one for each of the memory models. One final reminder: Make a copy of the original SETARGV.ASM and stash it somewhere safe in case you need it later.

As we noted last time, much of the Runtime source is written in assembler. We try whenever possible to do our work in Turbo C, but some assembly language programming is almost always required. This means that you need a copy of MASM to complete the modifications.

The changes we made this time are a typical example. We started by altering an assembly language routine, **__setargv**. We made as few changes as possible, however, and called our C routine, **expwild**, to do the bulk of the work.

We chose to start the wildcard expansion process in **__setargv** because it is the natural place to begin. One of the last things the startup procedure **c0.asm** does is to call **__setargv** before calling your **main** routine. Because all startup functions other than the command line processing have already been done, we are able to call a Turbo C routine for most of the work. As you look over the changes we made to **__setargv**, remember that the C calling conventions require that the first argument be on the top of the stack. This means that our assembly language code must push the arguments to any Turbo C routine onto the stack in reverse order of their appearance in the routine's declaration. Also, we skirt the entire problem of the underscores that prefix Turbo C routine names called from assembly language. We use

*continued on page 144*

## FROM THE RUNTIME

*continued from page 142*

the **ExtProc** macro to declare our external Turbo C procedure. This macro is defined in the file RULES.ASI and handles such underscore problems automatically.

### THE JOB OF __setargv

The job of **__setargv** is to turn the command line that COMMAND.COM provides, which is one string, into a set of null-terminated strings usable by your Turbo C program. For ease of reference we call these strings *words*. Turbo C passes these words to your main program by pushing pointers to them and pushing the words themselves onto the stack. **__setargv** is written in assembler primarily because it does so much stack manipulation. It has to place these arguments on the stack and make sure that they are still in the right place when your **main** routine starts. When called, **__setargv** returns the address of its caller in a variable, and then jumps to that address when it finishes. This avoids having an argument on the top of the stack popped as if it were the return address.

**__setargv** performs one other major function; it handles quoted strings. Normally, two command line words that are separated by spaces or tabs are pushed as two separate arguments. All characters within a quoted string, however, are treated as one argument.

Since **__setargv** already reads through the command line once in order to process quoted strings, the most efficient way to handle wildcard expansion is to do it during this pass. But that would have forced us to change a great deal of **__setargv**, which would have meant even more assembly code. Because this is startup code that is executed only once, and because we wanted most of the work to be in Turbo C, we opted for a simpler approach that runs slightly slower. We process the command line twice. The first time we run through it in Turbo C, expanding any wildcard file specifications we find. We then give this expanded string to the rest of the code in **__setargv**, which handles any quoted strings.

Because we are creating an intermediate form of the command line, we need some memory in which to store this string. We chose to get that memory from the lowest level Turbo C memory allocation routine available, **sbrk**. **sbrk** costs the least to use, both in terms of the amount of computation it performs and in the number of routines that it causes to be linked into our final code. Higher level Turbo C memory allocation functions, such as **malloc**, ultimately use **sbrk** and additional control information.

We wanted to allow as large an expanded command line as possible, but we did not want to stop anyone's program from executing because we consumed tons of additional memory. We decided to allocate as much as half the size of the stack. This choice should be fine for most programs. Also, we give back the memory before your main routine ever starts. To free the memory we only have to call **sbrk** with the negative of the amount of memory we allocated previously. We get the stack size from the variable **__stklen**.

### expwild TAKES OVER

Once we have the needed memory, we get the initial command line and terminate it with a null. Then we pass it, and the destination command line buffer we just allocated, as well as the maximum size of that buffer, to our Turbo C routine **expwild**. We pass the initial command line as a **far** pointer so that **expwild** works regardless of the memory model with which you compile your code.

**expwild** is yet another of those string processing procedures that has to be careful to keep its offsets right while remaining familiar and straightforward. It plows through the command line, copying character after character from the initial string to the destination buffer. It treats either a space or a tab as an end-of-word marker, remembering word boundaries as it goes. When it finally encounters a word that contains a wildcard character ("*" or "?"), it backs up the pointer in the source string to the start of that word. It then copies that word into a local variable buffer (**wildspec**) and passes that to the routine **wild** for expansion. We use the local variable rather than the source string itself so that **wild** does not have to deal with the **far** pointer that **expwild** receives. This makes **wild** a general purpose routine that you can call at any time.

**expwild** has some limitations. If there is not enough space available to hold the expanded command line, we truncate the line. Your program never knows that the truncation occurred. On the bright side, this choice allows your program to work on all of the files that did fit.

Another limitation of **expwild** is that it doesn't handle quoted strings. This creates two problems. First, it expands all wildcards it finds, even if they are in quoted strings. This may go counter to the very reason you quoted the string in the first place. More importantly, you cannot pass an asterisk or a question mark as an argument to your program. The correct method is to modify **expwild** so that it handles these two cases. It should not expand quoted strings—simple enough. As to passing either wildcard character as an argument, we recommend the UNIX convention: precede the character with a backslash ("\"). Because of space limitations we have left these improvements to you.

**wild** receives a wildcard file specification from **expwild**; a buffer to hold the resulting filenames; and the maximum size of this expansion buffer. It separates the filenames in the expansion buffer with spaces. If no files match the wildcard, it just copies the input string to the output string and returns. If you would like to treat such cases as errors, you could easily modify it to return a null string any time no files match a given wildcard specification.

## A WALK ON THE wild SIDE

**wild** works by calling three Turbo C library routines. Because we want your main program to receive complete filenames, **wild** first calls **fnsplit** with the wildcard specification. This routine extracts the drive and directory names. We then make these the first part of the name of every matching file. If at any time we run out of space in the expansion buffer, we go back to the last name we managed to fit into the buffer, add a null terminator, and return. **wild** always returns the size of the resulting string of filenames. After **fnsplit**, **wild** calls **findfirst** to find the first matching wildcard specification, and **findnext** to find all subsequent matches. These two routines, however, return the simple filename and extension for each matching file.

Once you have installed **wild** in your Runtime library, you can call it any time you need to expand a wildcard file specification. This ability is useful in any program that prompts the user for a filename.

That's it. The code contains further comments that should help clear up any confusion you might still have. Add these two routines to your library and any programs you compile will automatically be able to handle wildcard file specifications. Of course, if a program is not designed to accept more than one filename, you will have to change it to take full advantage of this ability.

As you might expect, this new ability does cost something other than the tiny bit of extra time it takes to execute. It adds about 1K to the size of your final executable file, most of which comes from other routines that are linked with it. We don't mind this additional amount, but you might if you have any programs that are really pushing the edge of the memory envelope. If you run into such a program, just go back to your original copy of the libraries.

Next issue we will delve deeper into the depths of the Runtime. Let us know if you have any special requests for future topics. Until next time, happy wildcarding! ■

*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 EXPWLD.ARC.*

LISTING 1: WILD.C

```
#include <stdio.h>
#include <dir.h>              /* The stuff for fnsplit and ffblk */

/* We use a far pointer for the source so that no matter what
   memory model you use the DOS command line can be passed to
   expwild.  Because the command line resides outside of Turbo
   C's data segment, it must be passed this way.  We do not want
   the routine wild() to have to deal with far pointers, however.
   To avoid this problem we copy any file specification we are
   about to pass to wild() into a local variable.  */


expwild (source, dest, maxsize)    /* called from setargv to */
                                   /* handle any wildcard */
                                   /* specifications in the */
                                   /* command line */


char far *source;
char *dest;
int maxsize;

{
  int count = 1, savecount = 1, wildcount;
  char *dwrdptr,                /* destination string pointer */
       wildspec[128],           /* 128 is max command line size */
       *wildptr;                /* used to extract wildcard spec */
  char far *swrdptr;            /* Far pointer to match source */

  swrdptr = source;             /* Initialize the source and */
  dwrdptr = dest;               /* destination word pointers */

/* Move characters from the source to the destination until
   we hit the end of the null-terminated string. */

  while ((*dest = *source++) != '\0')
  {

/* If the string is too big, we're done.  Exit the loop. */

    if (++count >= maxsize) break;

/* Possibly take some action based on what the character was */

    switch (*dest++)
    {
      case '*':
      case '?':

/* We just moved a wildcard character.  Move back in the source to
   the start of the last word and copy into wildspec (our local
   variable buffer for the call to wild) that word.  In this way
   wild need not deal with a far pointer.  We copy until we hit a
   space, tab or the end of the string.  Terminate the wildcard
   specification string with a null.  Reset count to what it was
   at the start of the word. */

        wildptr = wildspec;
        source = swrdptr;
        while ( *source != ' ' && *source != '\t' &&
                *source != '\0' ) *wildptr++ = *source++;
        *wildptr = '\0';
        count = savecount;
```

*continued on page 146*

*continued from page 145*

```
/* Call wild to expand the wild spec.  Put the answer where the
   last destination word pointer was.  Update the count and the
   destination pointer. */

        wildcount = wild (wildspec, dwrdptr, maxsize - count);
        count += wildcount;
        dest = dwrdptr + wildcount;
        break;

/* dest now contains any previous arguments plus this one that
   has been expanded by wild().  We're done with this case. */

      case ' ':
      case '\t':

/* We hit a word terminator.  We update the source and
   destination word pointers to point to the next character.
   We also save the count as of the start of this word, in case
   we have to roll back to this position later. */

        swrdptr = source;
        dwrdptr = dest;
        savecount = count;
        break;

/* Don't do anything for any other character.  Just move it from
   the source string to the destination string.  */

      default:
        break;

    }  /* switch */

  }  /* while loop */

  return (count);               /* Return the length of the new */
                                /* command line */
}  /* expwild */


int wild (wildspec, expand, maxsize)   /* general purpose routine */
                                       /* that accepts an input */
                                       /* wildcard spec wildspec */
                                       /* and puts the resulting */
                                       /* full file names, if any, */
                                       /* in expand */

char *wildspec, *expand;
int maxsize;

{
  struct ffblk ffblk;           /* Special file block */
  char drive[MAXDRIVE], dir[MAXDIR], name[MAXFILE], ext[MAXEXT];
                                /* these constants are in dir.h */
                                /* and reflect DOS size limits */
  char *bufptr;
  int size = 0;

  bufptr = expand;              /* Save our original pointer */

/* Split the file name into its fundamental components. We need
   the drive and directory information because the findfirst and
   findnext functions return only simple file names, and we want
   to return to our caller complete, unambiguous file pathnames. */

  fnsplit (wildspec, drive, dir, name, ext);
```

```
/* Find the first file that matches the input wildcard specification.
   If none match, then copy the input string to the output output
   string and return the length of the resulting string. Note that
   another option is to return a null if the caller passes in a
   wildcard spec that matches no file.  We elected to let the caller
   handle the results in this case.  Both findfirst and findnext
   require a special file information block structure of type
   ffblk. */

  if (findfirst (wildspec, &ffblk, 0) != 0)
```

LISTING 2: SETARGV.ASM

```
        NAME    setargv
        PAGE    60,132
;[]------------------------------------------------------------[]
;|      SETARGV.ASM -- Parse Command Line                       |
;|                                                              |
;|      Turbo-C Run Time Library        version 1.0             |
;|                                                              |
;|      Copyright (c) 1987 by Borland International Inc.        |
;|      All Rights Reserved.                                    |
;[]------------------------------------------------------------[]

        INCLUDE RULES.ASI

;       Segment and Group declarations

Header@

;       External references

ExtSym@         __argc, WORD, __CDECL__
dPtrExt@        __argv, __CDECL__
ExtSym@         _psp, WORD, __CDECL__
ExtSym@         _envseg, WORD, __CDECL__
ExtSym@         _envLng, WORD, __CDECL__
ExtSym@         _osmajor, BYTE, __CDECL__
ExtProc@        abort, __CDECL__

;*** Begin addition
; Here we declare our new C routine, expwild, as well as the fact
; that we will use the library routine sbrk (for memory allocation).
; We also need to know the size of the stack, and so access it.
; The macros ExtProc and ExtSym are defined in RULES.ASI.
;
ExtProc@        expwild, __CDECL__
ExtProc@        sbrk, __CDECL__
ExtSym@         _stklen, WORD, __CDECL__
;
;*** End addition


        SUBTTL  Parse Command Line
        PAGE
;/*                                                     */
;/*-----------------------------------------------------*/
;/*                                                     */
;/*     Parse Command Line                              */
;/*     ------------------                              */
;/*                                                     */
;/*-----------------------------------------------------*/
;/*                                                     */
PSPCmd          equ     00080h
```

```
CSeg@

IF      LPROG
SavedReturn     dd      ?
ELSE
SavedReturn     dw      ?
ENDIF
SavedDS         dw      ?
SavedBP         dw      ?

;*** Begin addition
; Here we declare a double word pointer to hold the address of our
; command line string and a word to hold the string's size.  We
; do not initialize either one.
;
NewCmdLn        dd      ?
NewCmdSz        dw      ?
  {
    strncpy (expand, wildspec, maxsize);
    return (strlen (expand));
  } /* if no files match this wildcard specification */

  while (1)  /* loop until no more files match the wildcard spec */
  {

/* Build up the complete file name from the optional drive name
   and path name of the original wild file spec and the file name
   found by either findfirst or findnext.  maxsize-size is always
   the total space remaining for the string.  */

    strncpy (bufptr, drive, maxsize - size);
    strncat (bufptr, dir, maxsize - size);
    strncat (bufptr, ffblk.ff_name, maxsize - size);

    size += strlen (bufptr);      /* Increase the string size to */
                                  /* reflect the added file name */

/* If the string is now too long, roll back to the previous
   file name, deposit a null and return the string size.  bufptr
   still points to the start of this new file name, so rolling
   back is simple.  */

    if (size >= maxsize - 1)
    {
      *(bufptr - 1) = '\0';
      return (strlen (expand));
    } /* if the string is too long */

/* Update the pointer to move past the newly added file name and
   put a space after that file name.  Because we compared size to
   maxsize-1 above, we know there is room for the blank. */

    bufptr = expand + size++;
    *bufptr++ = ' ';
```

```
/* Get the next file.  If there are no more, back up and put
   a null over the space after the last name and return the size. */

    if (findnext (&ffblk) != 0)
    {
      *(bufptr - 1) = '\0';
      return (size);
    }  /* if there are no more files that match this spec */

  }  /* loop through all matching file names */
}  /* wild */

;
;*** End addition

PubProc@        _setargv, __CDECL__

;       First, save caller context and Return Address

        pop     word ptr SavedReturn
IF      LPROG
        pop     word ptr SavedReturn+2
ENDIF
        mov     SavedDS, ds

;*** Begin deletion
; This block of code got the address of the command line, zeroed
; the registers, got the command line's length, appended a null to
; the command line, and set up the registers for later.  cx ended
; up containing the size of the command line string, including the
; null, and other registers were set to zero.  The code we give
; below to replace this code will end up in the same state, but will
; first call our routine expwild to preprocess the command line.
;
;       cld
;       mov     es, _psp@
;       mov     si, PSPCmd        ; ES: SI = Command Line address
;       xor     ax, ax
;       mov     bx, ax
;       mov     dx, ax            ; AX = BX = CX = DX = 0
;       mov     cx, ax            ; AX = BX = CX = DX = 0
;       lods    byte ptr es:[si]
;       mov     di, si
;       xchg    ax, bx
;       mov     es:[di+bx], al    ; Append a \0 at the end
;       inc     bx
;       xchg    bx, cx            ; CX = Command Line size including \0
;
;*** End deletion

;*** Begin addition
; Preprocess the command line for wild card expansion
;
; First, we get the stack size to see how large the expanded text
; can be.  We limit ourselves to one-half of the available stack.
```

*continued on page 148*

*continued from page 147*

```
IFDEF   __HUGE__
        mov     ax, seg _stklen@   ; if we are using the huge model,
        mov     es, ax             ; we need the segment that holds
        mov     ax, es:_stklen@    ; the stack data
ELSE
        mov     ax, _stklen@       ; here we need only the stack's
                                   ; length, as all data in one seg
ENDIF
        sar     ax, 1           ; Divide by two to leave room

        mov     NewCmdSz, ax    ; Save the maximum command size

; Now we call sbrk to allocate the desired memory.  We push the
; amount we need, which was in ax, onto the stack and do the call.

        push    ax
        call    sbrk@           ; Get some memory

; After a call we must restore the stack to its previous state.
; Rather than popping the argument we just adjust the stack pointer.
; Also, because we called a C routine, which could have messed up
; our data segment, we restore it to the segment value saved earlier
; by the existing code.  Finally, we save the address of the new
; command line buffer we allocated, which was returned in ax.

        add     sp, 2           ; Clean up the stack
        mov     ds, SavedDS     ; Necessary?
        mov     word ptr NewCmdLn, ax   ; Save the pointer offset

; sbrk returns the segment in dx for large memory model code.
; So, if we're not using that model, set dx to our segment so that
; the following code will work.

IFE     LDATA
        mov     dx, ds          ; Get our own segment if not returned
ENDIF
        mov     word ptr NewCmdLn+2, dx ; Save the pointer offset
                                        ; segment just after the
                                        ; pointer offset
        cld                     ; tell it to increment pointers
                                ; after string operations

; Now we do basically what the earlier deleted code did.  We get
; the command line address and append a null to the command line.

        mov     es, _psp@       ; Get the DOS command line
        mov     si, PSPCmd      ;  address
        xor     ax, ax          ; Zero AX and BX
        mov     bx, ax
        lods    byte ptr es:[si]
        xchg    ax, bx
        mov     es:[si+bx], al  ; Append a \0 at the end

; Here we push the arguments for expwild and then make the call.
; The third arg, and so the first to be pushed, is the maximum
; command line size.

        mov     ax, NewCmdSz    ; Get the maximum command size
        push    ax

; The second arg is the destination string.  We push first its
; segment if we are in large model.  In any case, we then push its
; offset.
```

```
IF LDATA
        mov     ax, word ptr NewCmdLn+2 ; Segment for large data
        push    ax
ENDIF
        mov     ax, word ptr NewCmdLn   ; Destination offset
        push    ax

; The first arg is the source command line string.  We push both its
; segment and offset in all cases because expwild expects a far ptr
; for this arg.

        mov     ax, _psp@       ; Segment of DOS command line
        push    ax
        mov     ax, PSPCmd+1    ; Offset (+ 1 to pass the count byte)
        push    ax
        call    expwild@        ; Finally, call expwild to expand
                                ; the command line

; Clean up the stack after the call.  We adjust by 10 if in large
; model because of the extra segment id we pushed.

IF LDATA
        add     sp, 10          ; Clean up the stack
ELSE
        add     sp, 8           ; Clean up the stack
ENDIF
        mov     ds, SavedDS     ; Restore our data segment

; Then, process the command line.  We put the size in cx and the
; string itself in es:[si], just as their code did.  Finally, we
; zero the same registers as the original (now deleted) code.

        mov     cx, ax          ; Put size in CX
        mov     es, word ptr NewCmdLn+2
        mov     si, word ptr NewCmdLn   ; Get the address of the
                                        ; expanded string
        mov     di, si          ; Set up the registers correctly
        xor     ax, ax          ; AX = BX = CX = DX = 0
        mov     bx, ax
        mov     dx, ax
;
;*** End addition

Processing      label   near
        call    NextChar
        ja      NotQuote        ; Not a quote and there are more
InString        label   near
        jb      GetArg0Lgth     ; Command line is empty now
        call    NextChar
        ja      InString        ; Not a quote and there are more
NotQuote        label   near
        cmp     al, ' '
        je      EndArgument     ; Space is an argument separator
        cmp     al, 9
        jne     Processing      ; TAB  is an argument separator
EndArgument     label   near
        xor     al, al          ; Space and TAB are argument
                                ; separators
        jmp     short Processing

;       Character test function used in SetArgs
;               On entry AL holds the previous character
;               On exit  AL holds the next character
;                        ZF on if the next character is quote (")
;                           and AL = 0
;                        CF on if end of command line and AL = 0
```

```
NextChar        PROC    NEAR
        or      ax, ax
        jz      NextChar0
        inc     dx              ; DX = Actual length of CmdLine
        stosb
        or      al, al
        jnz     NextChar0
        inc     bx              ; BX = Number of parameters
NextChar0       label   near
        xchg    ah, al
        xor     al, al
        stc
        jcxz    NextChar2       ; End of command line --> CF ON
        lods    byte ptr es:[si]
        dec     cx
        sub     al, '"'
        jz      NextChar2       ; Quote found --> AL = 0 and ZF ON
        add     al, '"'
        cmp     al,'\'
        jne     NextChar1       ; It is not a \
        cmp     byte ptr es:[si], '"'
        jne     NextChar1       ; Only " is transparent after \
        lods    byte ptr es:[si]
        dec     cx
NextChar1       label   near
        or      si, si          ; Be sure both CF & ZF are OFF
NextChar2       label   near
        ret
NextChar        ENDP

;       Invalid program name

BadProgName     label   near
        jmp     abort@

;       Now, Compute Argv[0] length

GetArg0Lgth     label   near
        mov     bp, es          ; BP = Program Segment Prefix address
        mov     si, _envLng@
        add     si, 2           ; SI = Program name offset
        mov     cx, 1           ; CX = Filename size (includes \0)
        cmp     _osmajor@, 3
        jb      NoProgramName
        mov     es, _envseg@
        mov     di, si          ; SI = argv[0] address
        mov     cl, 07fh
        repnz   scasb
        jcxz    BadProgName
        xor     cl, 07fh        ; CX = Filename size (includes \0)
NoProgramName   label   near

;       Now, reserve space for the arguments

ReserveSpace    label   near
        inc     bx              ; argv[0] = PgmName
        mov     __argc@, bx
        inc     bx              ; argv ends with NULL
        mov     ax, cx          ; Size = PgmNameLgth +
        add     ax, dx          ;        CmdLineLgth +
        add     bx, bx          ;        argc * 2       (LDATA = 0)
IF      LDATA
        add     bx, bx          ;        argc * 4       (LDATA = 1)
```

```
ENDIF
        add     ax, 1
        and     ax, not 1       ; Keep stack word aligned
        add     bx, ax
        mov     di, sp
        sub     di, ax          ; SS:DI = DestAddr for PgmName
        sub     sp, bx          ; SS:SP = &argv[0]
        xchg    bx, bp          ; BX = Program Segment Prefix address
        mov     bp, sp          ; BP = &argv[0]
        mov     word ptr __argv@, sp
IF      LDATA
        mov     word ptr __argv@+2, ss
ENDIF
        mov     ax, ss
        mov     es, ax          ; ES:DI = Argument's area

;       Copy program name

CopyArg0        label   near
        mov     [bp], di        ; Set argv[n]
IF      LDATA
        mov     [bp+2], es
        add     bp, 4
ELSE
        add     bp, 2
ENDIF
        mov     ds, _envseg@
        dec     cx
        rep     movsb
        xor     al, al
        stosb

;       Copy the command line

;*** Begin deletion
; This code used to ready the command line size and address.
; Because we have changed these two, however, we must delete this
; code and replace it with our own.
;
;       mov     ds, bx
;       xchg    cx, dx          ; CX     = Command Line size
;       mov     si, PSPCmd + 1  ; DS: SI = Command Line address
;
;*** End deletion

;*** Begin addition
; We ready the size and address of our newly expanded command line.
; The code below ours puts all of the arguments on the stack.  It
; uses the stack pointer that previous code has already adjusted
; to be in the correct position to hold all of the arguments.
```

*continued on page 150*

*continued from page 149*

```
        xchg    cx, dx          ; CX     = Command Line size
        push    ax              ; Save AX
        mov     ax, word ptr NewCmdLn+2 ; Use expanded command line
        mov     ds, ax
        mov     si, word ptr NewCmdLn   ; DS:SI = Command Line
                                        ; address
        pop     ax
;
;*** End addition

CopyArgs        label   near
        jcxz    SetLastArg
        mov     [bp], di        ; Set argv[n]
IF      LDATA
        mov     [bp+2], es
        add     bp, 4
ELSE
        add     bp, 2
ENDIF
CopyArg         label   near
        lodsb
        or      al, al
        stosb
        loopnz  CopyArg
        jz      CopyArgs
SetLastArg      label   near
        xor     ax, ax
        mov     [bp], ax
IF      LDATA
        mov     [bp+2], ax
ENDIF

;       Restore caller context and exit

;*** Begin addition
; We are finished.  We call sbrk with the negative of the amount
; of memory it gave us, which causes that memory to be freed.

        mov     ax, NewCmdSz    ; Get the command size
        neg     ax              ; Negate
        push    ax
        call    sbrk@           ; Free up any storage we use
        add     sp, 2           ; Clean up the stack after the call
;
;*** End addition

        mov     ds, SavedDS
IF      LPROG
        jmp     dword ptr SavedReturn
ELSE
        jmp     word ptr SavedReturn
ENDIF
EndProc@        _setargv, __CDECL__
CSegEnd@
        END
```

LISTING 3: UPDC.BAT

```
ECHO OFF
ECHO ***
ECHO ***  Updating C Module %1 In All Memory Model CLIB Libraries
ECHO ***
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -mt %1.c*
lib \TURBOC\RUNTIME\LIB\CS -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -mm %1.c*
lib \TURBOC\RUNTIME\LIB\CM -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -mc %1.c*
lib \TURBOC\RUNTIME\LIB\CC -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -ml %1.c*
lib \TURBOC\RUNTIME\LIB\CL -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -mh %1.c*
lib \TURBOC\RUNTIME\LIB\CH -+%1;
ECHO ***
ECHO ***  Finished Updating Module %1 In All CLIB Libraries
ECHO ***
```

LISTING 4: UPDASM.BAT

```
ECHO OFF
ECHO ***
ECHO *** Updating Assembler Module %1 In All CLIB Libraries
ECHO ***
masm %1 /D__SMALL__ /MX;
lib \TURBOC\RUNTIME\LIB\CS -+%1;
masm %1 /D__MEDIUM__ /MX;
lib \TURBOC\RUNTIME\LIB\CM -+%1;
masm %1 /D__COMPACT__ /MX;
lib \TURBOC\RUNTIME\LIB\CC -+%1;
masm %1 /D__LARGE__ /MX;
lib \TURBOC\RUNTIME\LIB\CL -+%1;
masm %1 /D__HUGE__ /MX;
lib \TURBOC\RUNTIME\LIB\CH -+%1;
ECHO ***
ECHO *** Finished Updating Assembler Module %1 In All Memory Model
ECHO *** CLIB Libraries
ECHO ***
```

# ARCHIMEDES' NOTEBOOK

## Solving equations of state for ideal gases.

*Namir Clement Shammas*

The equation of state for ideal gases represents a particular class of problem common in science and engineering. But it's a problem well suited to Eureka's capabilities.

The equation of state for ideal gases, also known as the ideal gas law, describes the relationship between the pressure, volume, temperature, and quantity (moles) of a gas. The ideal gas model is stated in the following simple equation

$$Pv = nRT \qquad (1)$$

where $P$ is the pressure in atmospheres, $v$ is the volume in liters, $n$ is the number of moles (weight per molecular weight), $R$ is the universal gas constant, and $T$ is the absolute temperature in Kelvin. Equation (1) is also often written in the following form:

$$PV = RT$$

where $V$ is the molar volume:

$$V = \frac{v}{n}$$

The ideal gas model succeeds in approximating the characteristics of real gases under relatively low pressures and temperatures. The deviation from the ideal state depends on the chemical nature of the gas. To better correlate the pressure, the molar volume, and the temperature, numerous researchers have devised empirical and semi-empirical equations. We consider two popular equations and present their Eureka solutions.

### SOLVING THE VAN DER WAALS EQUATION

The first equation is the van der Waals equation

$$\left(P + \frac{a}{V^2}\right)(V\text{-}b) = RT \qquad (2)$$

where $a$ and $b$ are coefficients calculated using:

$$\left(\frac{27}{64}\right)\left(\frac{R^2T_c^{\,2}}{P_c}\right) \qquad (3)$$

$$b = \frac{RT_c}{8P_c} \qquad (4)$$

$P_c$ and $T_c$ are the critical pressure and temperature, respectively. Since each gas has a particular critical pressure and temperature value, the corresponding values for $a$ and $b$ are also unique to each gas. Consequently, $a$ and $b$ express the unique deviation of each gas from the ideal state.

The second equation used in tackling real gases is the Redlich-Kwong equation

$$P = \frac{RT}{V\text{-}b} - \frac{a}{T^{0.5}V(V+b)}$$

where coefficients $a$ and $b$ are evaluated using:

$$a = 0.427480\,\frac{R^2T_c^{2.5}}{P_c}$$

$$b = 0.086640\,\frac{RT_c}{P_c}$$

The overall effect of deviating from the ideal gas law is expressed in the following equation

$$PV = ZRT \qquad (5)$$

where $Z$ is the compressibility factor. When $Z$ approaches unity, the behavior of the gas also approaches the ideal state. Many charts and graphs list values for the compressibility factor as functions of the reduced temperature, $T_r$, and reduced pressure, $P_r$. They are calculated using:

$$T_r = \frac{T}{T_c} \qquad (6)$$

$$P_r = \frac{P}{P_c} \qquad (7)$$

The next step is to build the Eureka equation files that state the above equations, and then allow Eureka to solve them. Listing 1 contains a Eureka equation file that solves the van der Waals equation of state, and Table 1 shows its solution. The equation file of Listing 1 contains the following:

*SOLUTION:*

| VARIABLES | | VALUES |
|---|---|---|
| a | = | 4.2090807 |
| b | = | .037225067 |
| MW | = | 17.0000000 |
| n | = | 43.099625 |
| p | = | 10.0000000 |
| Pc | = | 112.500000 |
| Pr | = | .888888889 |
| R | = | .082600000 |
| T | = | 298.000000 |
| Tc | = | 405.600000 |
| Tr | = | .73471400 |
| V | = | 2.3203058 |
| Volume | = | 100.000000 |
| W | = | 732.69363 |
| Z | = | .94260598 |

*Maximum error is 3.5527137e—15*

*Table 1. Solution of the van der Waals equation of state.*

*continued on page 152*

LISTING 1: VDERWAAL.EKA

```
; VDERWAAL.EKA
; ---------
;
; version 1.0
;
; date 7/24/1987
;
; Copyright (c) 1987 Namir Clement Shammas
;
; This Eureka program provides for the algebraic manipulation of the
; van der Waals equation of state.
;
; P is the pressure of the gas
; Pc is the critical pressure
; Pr is the reduced pressure (= P/Pc)
; V is the molar volume of the gas
; W is the weight of the gas
; MW is the molecular weight of the gas
; n is the number of moles (= W/MW)
; R is the universal gas constant, expressed in corresponding units
; T is the absolute temperature
; Tc is the critical temperature
; Tr is the reduced temperature (= T/Tc)

; van der Waals equation of state
(P + a / V^2) * (V - b) = R * T

; compressibility factor
Z = P * V / (R * T)

; calculate the number of moles
n = W / MW

; molar volume
V = Volume / n

; calculate coefficient 'a'
a = 27 * (R * Tc)^2 / (64 * Pc)

; calculate the coefficient 'b'
b =  R * Tc / (8 * Pc)

; reduced temperature
Tr = T / Tc

; reduced pressure
Pr = P / Pc

; for ammonia: Tc = 405.6 K, Pc= 112.5 atm and MW = 17
Tc = 405.6
Pc = 112.5
MW = 17

; the universal gas constant, in atm-l/gmole/degK is:
R = 0.0826

;---------------------------------------------------
; Normally, the above variables provide given data. The variables
; below lend themselves to the algebraic manipulation process.  Either
; comment out the sought variable or use the ':=' instead of the '='
; to assign a 'guess' value.

; the pressure (in atm) is:
P = 10

; the mass of the gas (in gm) is:
; W = 10

; the temperature (in degrees Kelvin) is:
T = 273 + 25

; the gas volume (in liter) is:
Volume = 100.0
```

*continued from page 151*

1. Comments that describe the problem solved and list the variables involved;
2. The van der Waals equation of state, copied verbatim from equation (2) (notice that Eureka does not require the left hand side of the equation to be a simple variable).
3. The equation to evaluate the compressibility factor, using equation (5);
4. The number of moles of the gas, given its weight and molecular weight; (I have added this simple equation because in many engineering calculations you are dealing with weights of gases, as opposed to moles of gases.)
5. The molar volume, which is calculated by dividing the actual volume with the number of moles;
6. The equation to evaluate coefficients *a* and *b* using equations (3) and (4);
7. The reduced temperature and pressure values, using equations (6) and (7);
8. The critical temperature, critical pressure, and molecular weight for the particular gas being considered; (The values I have included are for ammonia [$NH_3$])
9. The value for the universal gas constant that corresponds to the units being used;
10. The input variables for the equation of state; in this example, the weight of the ammonia gas that occupies 100 liters, under 10 atmospheres of pressure and at 25 degrees Celsius.

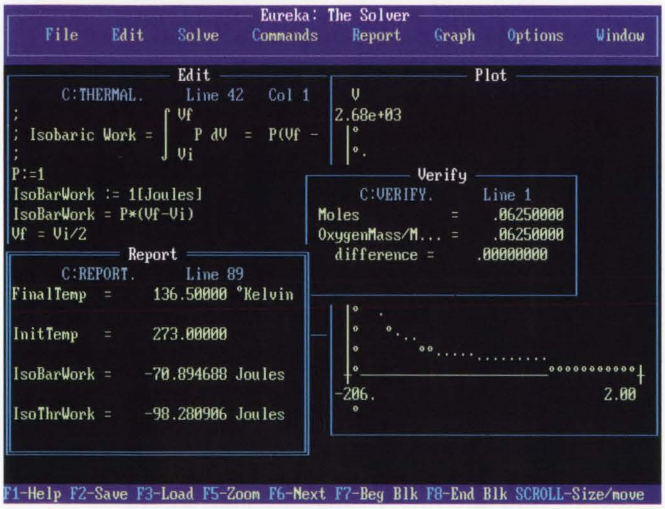
The solution indicates that there are 732.7 grams of ammonia. The compressibility factor is estimated at 0.94260598. The solution section displays alphabetically the input values, the intermediate solutions, and the results.

To solve for other physical properties such as the pressure, volume, or temperature, you can comment out the sought variable (and uncomment the previous one). In principle, you may even solve for the critical pressure, critical temperature, and the molecu-

*continued on page 154*

# How Eureka: The Solver instantly solves equations that used to keep you up all night

The state-of-the-art answer to any of your scientific, engineering, financial, algebraic, trigonometric, or calculus equations = Eureka: The Solver™


Eureka: The Solver
File Edit Solve Commands Report Graph Options Window
F1-Help F2-Save F3-Load F5-Zoom F6-Next F7-Beg Blk F8-End Blk SCROLL-Size/move

```
Edit
    C:THERMAL.      Line 42    Col 1
;                 ⌠ Vf
; Isobaric Work = │   P dV  =  P(Vf -
;                 ⌡ Vi
P:=1
IsoBarWork := 1[Joules]
IsoBarWork = P*(Vf-Vi)
Vf = Vi/2

Report
    C:REPORT.       Line 89
FinalTemp   =     136.50000 °Kelvin

InitTemp    =     273.00000

IsoBarWork =    -70.894688 Joules

IsoThrWork =    -98.280906 Joules

Plot
 V
2.68e+03

-206.                          2.00

Verify
       C:VERIFY.      Line 1
Moles                 =    .06250000
OxygenMass/M... =    .06250000
  difference =     .00000000
```

*Eureka instantly solved this Physics equation by immediately calculating how much work is required to compress isobarically 2 grams of Oxygen initially at STP to ½ its original volume. In Science, Engineering, Finance and any application involving equations, Eureka gives you the right answer, right now!*

Eureka can solve most equations that you're likely to meet. So you can take a mathematical sabbatical.

Most problems that can be expressed as linear or nonlinear equations can be solved with Eureka. Eureka also handles maximization and minimization, plots functions, generates reports, and saves you an enormous amount of time.

Eureka instantly solves equations that would've made the ancient Greek mathematicians tear their hair out by the square roots—and it's all yours for only $167.00.

### *It's easy to use Eureka: The Solver*

1. Enter your equation into the full-screen editor
2. Select the "Solve" command
3. Look at the answer
4. You're done

You can then tell Eureka to

- Evaluate your solution
- Plot a graph
- Generate a report, then send the output to your printer, disk file or screen
- Or all of the above

### *You can key in:*

- ☑ A formula or formulas
- ☑ A series of equations—and solve for all variables
- ☑ Constraints (like X has to be < or = 2)
- ☑ A function to plot
- ☑ Unit conversions
- ☑ Maximization and minimization problems
- ☑ Interest Rate/Present Value calculations
- ☑ Variables we call "What happens?," like "What happens if I change this variable to 21 and that variable to 27?"

> "Merely difficult problems Eureka solved virtually instantaneously; the almost impossible took a few seconds.
>
> *Stephen Randy Davis, PC Magazine*"

### *Eureka: The Solver includes*

- ☑ A full-screen editor
- ☑ Pull-down menus
- ☑ Context-sensitive Help
- ☑ On-screen calculator
- ☑ Automatic 8087 math co-processor chip support
- ☑ Powerful financial functions
- ☑ Built-in and user-defined math and financial functions
- ☑ Ability to generate reports complete with plots and lists
- ☑ Polynomial finder
- ☑ Inequality solutions

> "Get Eureka. You won't regret it. Highly recommend it.
>
> *Jerry Pournelle, Byte*"

**Minimum system requirements:** For the IBM PS/2™ and the IBM® and Compaq® families of personal computers and all 100% compatibles. PC-DOS (MS-DOS®) 2.0 and later. 384K.

Eureka: The Solver is a trademark of Borland International, Inc.
BI-1145B

BORLAND
INTERNATIONAL

For the dealer nearest you or to order by phone

**Call (800) 255-8008**

In CA: (800) 742-1133;
In Canada: (800) 237-1136

LISTING 2: KWONG.EKA

```
; KWONG.EKA
; ---------
;
; version 1.0
;
; date 7/24/1987
;
; Copyright (c) 1987 Namir Clement Shammas
;
; This Eureka program provides for the algebraic manipulation of the
; Redlich-Kwong equation of state.
;
; P is the pressure of the gas
; Pc is the critical pressure
; Pr is the reduced pressure (= P/Pc)
; V is the molar volume of the gas
; W is the weight of the gas
; MW is the molecular weight of the gas
; n is the number of moles (= W/MW)
; R is the universal gas constant
; T is the absolute temperature
; Tc is the critical temperature
; Tr is the reduced temperature (= T/Tc)

; Redlich-Kwong equation of state
P = R * T /(V - b)  -   a /(sqrt(T) * V * (V + b))

; Compressibility factor
Z = P * V / (R * T)

; calculate the number of moles
n = W / MW

; calculate molar volume
V = Volume / n

; calculate coefficient 'a'
a = 0.42748 * R^2 * Tc^2.5 / Pc

; calculate the coefficient 'b'
b = 0.086640 * R * Tc / Pc

; reduced temperature
Tr = T / Tc

; reduced pressure
Pr = P / Pc


; for ammonia: Tc = 405.6 K, Pc= 112.5 atm and MW = 17
Tc = 405.6
Pc = 112.5
MW = 17

; the universal gas constant is:
R = 0.0826

;----------------------------------------------------
;
; Normally, the above variables provide given data. The variables
; below lend themselves to the algebraic manipulation process.  Either
; comment out the sought variable or use the ':=' instead of the '='
; to assign a 'guess' value.

; the pressure (in atm.) is:
P = 10

; the mass of the gas (in grams) is:
; W = 10

; the temperature (in degrees Kelvin) is:
T = 273 + 25

; the gas volume (in liter) is:
Volume = 100.0
```

## NOTEBOOK

*continued from page 152*

lar weight. However, such solutions rarely lead to meaningful results.

### SOLVING THE REDLICH-KWONG EQUATION

Listing 2 shows a Eureka equation file for solving the Redlich-Kwong equation and Table 2 shows its solution. To build the equation file, you follow steps that are very similar to those for the van der Waals equation. The differences are the equation of state used and the formulas for calculating coefficients *a* and *b*.

```
SOLUTION:
VARIABLES              VALUES
-----------------------------------
    a          =      85.895070
    b          =      .025801438
    MW         =      17.0000000
    n          =      43.988585
    p          =      10.0000000
    Pc         =      112.500000
    Pr         =      .888888889
    R          =      .082600000
    T          =      298.000000
    Tc         =      405.600000
    Tr         =      .73471400
    V          =      2.2733171
    Volume     =      100.000000
    W          =      747.80594
    Z          =      .92355699
Maximum error is 1.7763568e-14
```

*Table 2. Solution of the Redlich-Kwong equation of state.*

The equation file of Listing 2 solve the same problem stated above (i.e., the physical properties of ammonia gas). The results are that the calculated mass of ammonia is 747.8 grams and the compressibility factor is 0.92355699. The Redlich-Kwong equation is more reliable than the van der Waals form. The estimated gas weight is about 15 grams higher (corresponding to a 2 percent increase). The compressibility factor obtained by the Redlich-Kwong equation is about 2 percent less than that for the van der Waals equation. ■

*Namir Clement Shammas is the editor of* Turbo Report *newsletter, and a columnist for* Dr. Dobb's Journal *and* PC AI.

*Listings may be downloaded from CompuServe as GASLAW.ARC.*

# BOOKCASE

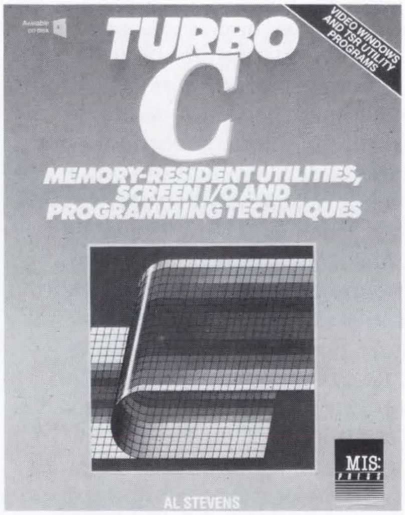
## TURBO C: MEMORY-RESIDENT UTILITIES, SCREEN I/O AND PROGRAMMING TECHNIQUES

*Al Stevens, Management Information Source, Inc., Portland, OR: 1987, ISBN 0-943518-35-0, 400 pages, soft cover, $24.95, disk $20.00.*

At first I wasn't sure whether to review this as a book of source code, or to review it as a C function library that has an unusually detailed manual. Either would have been appropriate. Stevens' book contains a complete library of routines for programming screen windows and memory-resident utilities in Turbo C, plus clear explanations of how the routines work and how to use them.

The book devotes short chapters to the C language and to the Turbo C implementation, and then presents a small collection of low-level, general purpose routines that manage the screen display and keyboard. These routines, which deal directly with the computer's hardware and BIOS, support the higher-level functions developed later in the book.

Next, Stevens discusses the concept of video windows in relation to the PC's video architecture, and then presents source code for a windows function library. This library is quite complete, allowing you to establish single or multiple windows, write to them, change their colors and position, promote and demote windows, and perform just about any other window function you might want.


TURBO C
MEMORY-RESIDENT UTILITIES, SCREEN I/O AND PROGRAMMING TECHNIQUES
VIDEO WINDOWS AND TSR UTILITY PROGRAMS
MIS: PRESS
AL STEVENS

Two types of windows are supported: stacked, in which modifications are made only to windows that are in full view on the screen, and layered, in which modifications can be made to partially or totally hidden windows. Stacked windows are less flexible than layered ones, but are much faster. A program can use either stacked or layered windows, but not both.

After Stevens details how easily you can include windows in your programs, he then tells you what you can do with them. Four of the most popular uses for windows are context-sensitive help, data entry, free-form text editing, and pop-up menus. Stevens devotes a chapter to each of these topics, and provides a library of functions for each. The functions are well designed and very useful, and greatly reduce the effort needed to include sophisticated windowing in your programs.

The remainder of the book deals with techniques for writing programs that terminate and stay resident. TSR programs are difficult to write because they must be able to pop up and function properly no matter what the state of the machine, and then must completely restore that state when exited. Stevens gives a detailed discussion of the various hardware and software factors that must be taken into account when writing TSR programs, some routines for use in TSRs, and a TSR program that shows the routines in use.

You will need the Microsoft Macro Assembler to use the TSR routines, because Turbo C's startup code—supplied by Borland in assembly source code—must be modified and reassembled to enable your TSR program to restore the divide-by-zero interrupt vector. The macro assembler is also needed if you plan to use window routines on an IBM Color Graphics Adapter, because two routines specific to the CGA include inline assembly language. The book's optional program disk ought to contain assembled object code for these files, but it doesn't.

This book is not a primer on C programming. The author has assumed that readers have a reasonable knowledge of the C language and of DOS. Familiarity with assembly language and the PC's register and address architecture will help in getting the most out of the book, but this is not necessary to make use of the software routines that are provided.

The book is 400 pages, with a good chunk of that devoted to program listings. It is worth the cost for either the function libraries or the programming information alone. Since it provides both, and is clearly written to boot, I recommend it highly. ■

*—Peter G. Aitken*

# CRITIQUE

## TurboWINDOW/C

*TurboWINDOW/C*
*Metagraphics*
*269 Mt. Hermon Road*
*Scotts Valley, CA 95066*
*(408) 438-1550*
*$95.00*

While watching me create a graphics demonstration for one of my classes, my wife insightfully commented, "A single picture is worth 1000 lines of code." Drawing pictures on computers is hard work. This goes triple if you need to support many different and incompatible graphics devices. So if a third party vendor markets a library that supports the functions necessary for drawing, so much the better—the vendor gets the work, and you get the fun.

Metagraphics Software Corporation provides such a library for Turbo C. TurboWINDOW/C costs $95.00, isn't copy protected, and requires no royalties for linking with commercial applications. Included in the package is an excellent 248-page manual, a memory-resident graphics driver written in assembly language, and the necessary libraries to allow Turbo C to communicate with the memory-resident driver. Finally, the company includes a rich selection of example programs to help you understand how to use the 195 procedures provided in the library.

When writing graphics programs to sell to the public, the first problem you face is supporting the bewildering array of graphics boards, graphics printers, and (somewhat) PC-compatible computers in use. To solve this problem, TurboWINDOW/C understands more than 50 of these printers and graphics boards, and automatically configures itself to the graphics board it detects at runtime. (Autodetection does not apply to printers and plotters.) You must still handle some of the work, such as scaling drawings and correcting for aspect ratios, but enough information is available in the documentation to make the task, if not simple, at least comprehensible. Therefore, you can write a program that will run on an IBM clone with a CGA color card, or on an IBM AT with an EGA color card using the *same* source code. The runtime configuration is handled automatically with a TurboWINDOW/C function **grquery**. If you wish, you can override the automatic selections by naming devices on the command line.

You can use two different coordinate systems to draw: local or virtual. A third coordinate system (world) is used internally by the TurboWINDOW/C software to transfer and scale bit images. Although local coordinates are usually tied to the display width and height, both local and virtual coordinates can be set to any value as long as the difference between minimum and maximum doesn't exceed 64K.

All drawing in TurboWINDOW/C is done through a viewport, which stores the text and drawing defaults, local and virtual coordinates, and clipping limits. The actual displayed pixels are stored in a bitmap, which is managed transparently by the TurboWINDOW/C software. Bitmaps are quite handy when converting plotter-style vector scan output to printer-style raster scan output. Such a conversion only requires creating a bitmap with the proper resolution, drawing on the bitmap with TurboWINDOW/C's vector drawing commands, and finally dumping the bitmap's pixels to the printer. You can have several viewports referencing different sections of a bitmap, each having different local and virtual coordinates. The only limit to the size and number of bitmaps in use is the amount of memory you have available. All images are clipped outside of local or virtual coordinate bounds. In addition, a smaller area can be clipped by setting the necessary coordinates within the port. All memory management tasks are handled automatically, so you needn't worry about segments or 64K limits.

Eight different graphics cursors come standard with TurboWINDOW/C, but if one of them won't work in your application, you can create your own. Cursors can be visible or hidden, and will automatically track your mouse if you wish. Mouse support is provided for Mouse Systems, Logitech, Microsoft, and other Microsoft-compatible mice. In most cases (the new Microsoft bus mouse being a notable exception), the necessary mouse driver is incorporated within the Metagraphics driver.

Lines can be set to any odd pixel width (1, 3, 5, etc.), one of three end styles (round, flat, or square) and join types (round, bevel, or miter). Also standard are 32 different fill patterns and eight line-dash styles. All of these are stored in the current viewport, and can be changed along with color and drawing mode as often as you like.

Drawing ovals and rectangles is handled by passing the coordinates of the lower left and upper right corners of an enclosing box to the corresponding procedure, which will draw a rectangle or oval that fits inside that box. Both ovals and rectangles can be filled with any of 32 different patterns, or framed with any available line width and style. Arcs are drawn by again specifying a box, with a start and end angle of the arc. Rectangles may be drawn with rounded corners. Polygons can be drawn, filled, inverted, or erased. Entire sections of the screen can be saved in an image buffer for later use, repositioned on the bitmap, and scaled up or down.

Metagraphics provides a number of bitmapped and outline fonts for use in TurboWINDOW/C. They can be loaded on demand for the current graphics card. Using Metafonts, another Metagraphics product, virtually any type of font can be created. Bitmapped and outline fonts can be drawn in all four directions, using standard, bold, italic, underline, or strikeout typefaces. Printing can be aligned left, right, or center; drawn in strings, or as characters. In addition, outline fonts can be slanted, and may be scaled to fit a given space.

The current keystroke, position of the cursor, and state of the mouse buttons can be queried with one function call. Other system utilities are provided to help with graphics file operations, transformations between local and virtual coordinates, command-line processing, and hard-copy output to a printer or plotter.

Customer support is excellent. I've called Metagraphics several times, and have gotten prompt, accurate help each time. Upgrades to the product will cost you the price of a phone call to the Metagraphics BBS.

TurboWINDOW/C is a powerful programming tool, and even though the folks at Metagraphics try to make your exposure to the package as painless as possible, some head scratching and wall kicking may happen while you learn to use some of its more complex features, such as event handling. The line-up of supported graphics boards is impressive, but there should be more support for input and output devices like the newer laser printers, plotters, graphics tablets, and trackballs.

Metagraphics has a winner with TurboWINDOW/C. I have developed and sold a major circuit board design program over the past year and a half, and I used their companion product Turbo-WINDOW/Pascal (which uses the same memory-resident driver and command set) to do it. I've had no customer complaints traceable to the Metagraphics portions of the program. After a month of testing the Turbo C version, I didn't find any reason to fault the Turbo C-specific interface. If you do graphics programming in Turbo C, try Metagraphics, TurboWINDOW/C—you'll like it. ■

*—Don Fletcher*

# TURBO RESOURCES

### YOUR SUBSCRIPTION

You're looking for Borland language information. Where to go? Well, for starters, right here. 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—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, just write, "I would like to subscribe to TURBO TECHNIX" in the bottom margin of the registration card.

### COMPUSERVE

The best online information about the Borland languages can be found on CompuServe. Subscribing to CompuServe can be done through the coupon enclosed with every Borland product (which also includes $15 worth of online time for your first month) or by calling CompuServe at (800) 848-8199. You'll need a modem and some sort of communications software that supports the XMODEM file transfer protocol.

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 automate sessions and help you minimize connect time. It is available for downloading on CompuServe from DL 12 of the Word Perfect 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 DL1 (Data Library 1) of the BPROGA Borland programming forum (**GO BPROGA**). Turbo C and Turbo Prolog listings are stored in DL1 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.

***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 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 in one operation.

The all-issue files follow a naming convention such that NVDC87.ARC contains all listing archives from the November/December 1987 issue, 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 each article.

To download an archive file, type

```
DOW<filename>/PROTO: XMO
```

at the DL 1 prompt. 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 this way:

```
C>ARC-E <filename>
```

### NATIONAL USER GROUPS

**TUG**

The national user group for Turbo languages is TUG, the Turbo User Group. TUG publishes a bimonthly newsletter 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. 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

**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 download 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 | Washington, DC |
| (703) 525-4066 | 24Hr $ |
| Illinois BBS | Chicago, IL |
| (312) 885-2303 | 24Hr $ |
| PC-TECH BBS | Santa Clara, CA |
| (408) 435-5006 | 24Hr |

$ = membership fee required

## C:>CLASS.ADS

This is our Display Classified section, brand new to *TURBO TECHNIX*. We congratulate our Charter Advertiser and extend a welcome to vendors of all sizes who would like to take advantage of the special rates for this section: $300 per column inch, with a 2-inch minimum (a minimum ad measures 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, Scotts Valley, CA 95066. For additional information, call (408) 438-9321.

# Coming Up

## Expert System Design with Turbo Prolog...

*Learn how to create your own computer expert from top to bottom in this theme issue as we explore the major components of expert system design. Starting with a bird's-eye view of expert systems, Mike Floyd discusses what's important and why. Next, you'll learn how to organize and represent knowledge with a popular technique known as* frames. *Keith Weiskamp explains the heart of an expert system—the inference engine; Safaa Hashim shows how metalogical features can be added to an expert system. Finally, Carl Townsend will show you how an English language interface can be implemented to give your expert system a true air of intelligence.*

## Join the Dialog...

The *TURBO TECHNIX* mailbag is bulging, and we'll be reaching in for the best our readers have to offer. Starting with our next issue, "Dialog" will be joined by the editorial staff and *TURBO TECHNIX* authors for lively give-and-take on issues that start at Turbo programming and go where they will. Gripes? Requests? Favorite hax? Give us your best shot, and enjoy the ricochets.

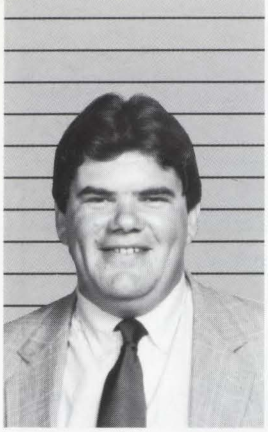
## Understand custom exit procedures...

*Turbo Pascal 4.0 units can execute special procedures before a program runs and after it terminates. This advanced technique allows a unit to create a special machine setup for programs that incorporate it, and then to restore the prior machine setup before the program returns to DOS. It's subtle, but Tom Swan will show you how it's done.*

## and the TECHNIX keep coming...

*Learn how to deal with important events when they happen in Turbo Basic, under the able tutelage of Ralph Roberts. Choosing one from many is a technique that comes under the lights in three separate articles, for Turbo Pascal on* **CASE**, *Turbo C on* **switch**, *and Turbo Basic on* **SELECT CASE**.

*We critique the Norton Guides and a pair of new books on Turbo Prolog and Turbo C programming, and return with more expert insights from our honored columnists.*



# PHILIPPE'S TURBO TALK

## If we don't take care of the future...

*Philippe Kahn*

I always say that if we don't take care of the future, the future will take care of us. We'll all be living in the future for the rest of our lives, so we'd better pay it some attention. What is ignored will be decided by default—by people who do not have our best interests in mind.

The future should be open. I don't mean *Glasnost*, (the Russian version of "openness," which somehow includes the Berlin Wall), but real openness within the software industry. The trend has been moving the other way in the past year or two: Look-and-Feel, audiovisual copyrights, name-calling, and endless lawsuits. Sometimes you can almost feel that the SP in Software Publisher stands for "Software Protectionist," "Software Paranoid," "Software Padlock," "Software Patent," "Software Predator" or perhaps "Software Psychoneurotic"! That kind of future we don't need. In reality, we all have to collaborate on industry-wide issues. It's not a zero-sum game. Everyone can play, and we'll all win.

### IDEAS AND STANDARDS ARE MEANT TO BE SHARED

As the old wisdom goes, "When you steal ideas from one author, it's plagiarism. If you steal from many, it's research." And we all know that nothing is more dangerous than an idea that's the only one we have. Then again, sharing ideas and information isn't always completely open-ended. Sometimes it's better to keep your mouth shut. Maybe Gary Hart shouldn't have challenged the press to follow him around. And ripping off ideas is no good either; Joe Biden found that out. He should have read what Einstein said: "The secret to creativity is knowing how to hide your sources." Guess about mine...

As software people, we're in the idea business. We know the way and should lead the way. We should never forget to be pioneers and we should never spend more time with lawyers than with developers.

### MEMO: WHY WE EXIST

We can never forget why we exist, as NASA did. Instead of responding to Kennedy's pioneering call to "go to the moon and back within a decade," and keeping the momentum, NASA has allowed smaller minds to intervene. The accountants are now in charge. The spirit of adventure and exploration, and the thirst for new knowledge, are lost. Instead of installing a space station and man on Mars, we're asked to be excited about accountants in space. Kennedy never asked if getting an American on the moon would make a profit—and we have to be just as adventurous and open-minded.

### DON'T WAIT FOR ALPHONSE

People have talked and argued about building a tunnel between England and France for more than 100 years—and no one's dug the first hole. Nothing gets accomplished when Alphonse waits for Gaston and Gaston waits for Alphonse. We can't sit around waiting for someone else to take the risks and invest the money for the next technological breakthrough. If someone makes the first move, we'll get started. If everyone makes the first move, watch out!

### INNOVATION CAN ALSO BE MISCALCULATION

However we approach the business of innovation, we can't forget that we're also in the business of not being dumb. Innovation can turn into miscalculation. Sometimes the cure is worse than the disease. And sometimes there is no disease.

In trying to honor Susan B. Anthony with a dollar coin, we did it cheap and made it the same size as a quarter. People were confused and ended up paying a dollar for a 25-cent newspaper; nor was there any place to put Susan B. dollars in most American cash registers. Susan is now history—twice. And we don't miss the coin, mostly because we didn't need it.

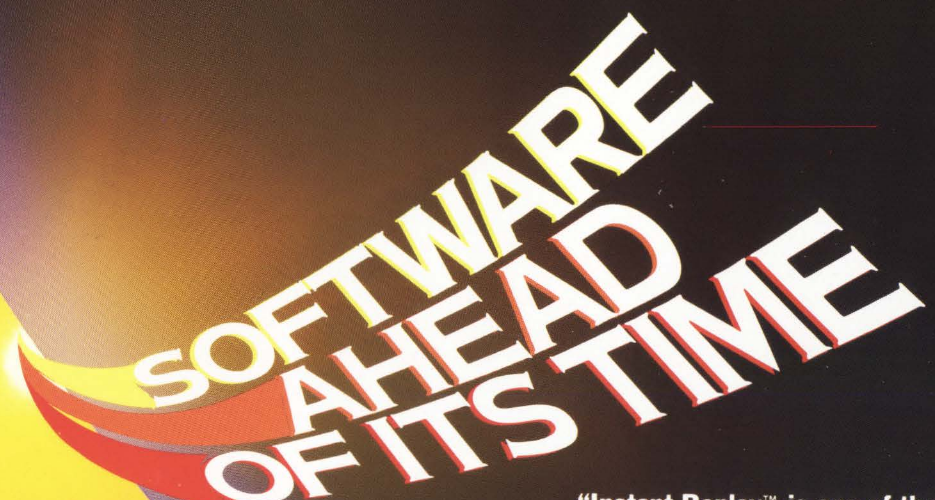
In the same vein, you don't build programs that eat up far more memory than most people have in their computers. You don't build hardware that is "largely IBM compatible." You don't sell software that makes the job more difffficult than using pencil and paper. Just because you thought of it doesn't mean the world needs it.

### RIDING THE TECHNOLOGICAL DRAGON

By riding and taming the technological dragon, we will build tomorrow's future. Not just a simple extension of today's world, but new opportunities, quantum leaps beyond what we know today. It's innovation time. This is the time to forge ahead, to establish and share standards, to keep moving on. Sharing standards and ideas will go a long way towards making this vision happen. Let's all build the future together. Today. ■

# Instant Replay

## Version III



"Instant Replay™ is one of those products with the potential to go from unknown to indispensable in your software library." *PC Magazine*

"Incredible . . . We built our entire Comdex Presentation with Instant Replay.™" *Panasonic*

"Instant Replay™ brings new flexibility to prototypes, tutorials, & their eventual implementation." *Electronic Design*

"I highly recommend Instant Replay.™" *Computer Language*

"You need Instant Replay™!" *Washington Post*

***Instant Replay*™ is a unique authoring system for creating:**

**Demonstrations · Tutorials · Music
Presentations · Prototypes · Menus**

When active, *Instant Replay™* is a DOS Shell that runs, memorizes, and replays programs. It has the unusual ability to insert prompts, pop-up windows, prototypes, user involvement, music, and branching menus into replays.

Building a Demo or Tutorial is easy. Just run any program with *Instant Replay™* and insert explanation pop-up windows, prompts, user involvement, music and so on. *Instant Replay™* remembers everything; it builds a demo that will re-run the actual program or a screens only prototype version.

*Instant Replay™* includes a *Screen Maker* for building pop-up windows, prototype windows, and menu windows. Other useful tools include a *Prototyper, Keystroke Editor, Music Maker, Menu Maker, Presentation Text Editor, Control Guide, and Insertion Guide.*

The screen editor *Screen Genie™* is designed to be memorized by *Instant Replay.™* Creating animations is easy and fun; just run *Screen Genie™* and memorize your activity.

***Instant Replay*™** for IBM and True Compatibles, requires DOS 2.0 or greater. *Instant Replay™* is not copy protected. **There are no royalties required for distribution of Demos.**

**Instant Replay™** at $149.95 is an exciting new product. Because of the quality of this product, *Nostradamus®* provides a 60-day satisfaction money back guarantee. Call or write, we accept VISA, AmEx, C.O.D., Check or P.O. with orders. Demo diskettes and free product brochure available.

**Nostradamus, Inc.** 3191 S. Valley Street,
(ste. 252) Salt Lake City, Utah 84109
**voice (801) 487-9662**
Data/BBS 801-487-9715 1200/2400,n,8,1

**Instant Replay™ Features:**

- *Illustrated manual with index*
- *Music Maker*
- *Text Editor*
- *Online Help*
- *Screen Painter*
- *Magic Demo Animator*
- *Dynamic Menu Maker*
- *Memorize and Replay actual programs*
- *Memorize and Replay screens only*
- *Insert: prompts, pop-ups, prototypes, music, and user involvement into replays*
- *Make insertions while creating or reviewing*
- *Generate Vapor Ware from actual programs*
- *Resident Screen Painter for creating and grabbing windows on the fly*
- *Prototyper that includes slide shows, menus, and nesting*
- *Keystroke/time editor, inserter, and merger*
- *Replay chaining and linking*
- *Modular demo making facilities*
- *Fast forward and single step modes*
- *Self Made Tutorial included*
- *Timed Keyboard Macros*
- *Numerous and powerful operator input options for Tutorials*
- *Text File Presentation facility*
- *Transparent Windows*
- *Change Defaults*
- *Foreground or Background Music*
- *Canned special sound effects*
- *Unlimited replay branching*
- *Compressed screens*
- *Object oriented programming*
- *Tracking editor*
- *Plus much more . . .*

**Instant Replay™**

**Instant Assistant™**

**Screen Genie™**

**Word Genie™**

**NoBlink Accelerator™**

**Assembler Genie™**

**DOS Assistant™**

**Programming Libraries**
*Supports Turbo Pascal 4.0*

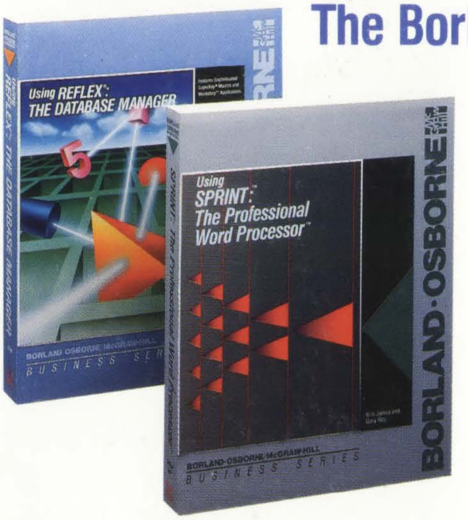
**HardRunner™**

**. . . and more**

**Nostradamus®**

*Announcing*

# Two Dynamic New Imprints


Using REFLEX: THE DATABASE MANAGER
Using SPRINT: The Professional Word Processor
BORLAND·OSBORNE
BORLAND-OSBORNE/McGRAW-HILL BUSINESS SERIES

## The Borland-Osborne/McGraw-Hill Business Series

### Using REFLEX®: THE DATABASE MANAGER

*by Stephen Cobb*

Features sophisticated SuperKey® macros and REFLEX Workshop™ applications.

**$21.95** paperback, *ISBN 0-07-881287-9*

### Using SPRINT™: The Professional Word Processor

*by Kris Jamsa and Gary Boy*

Take advantage of this fabulous new word processing system that is powerful, fast, and includes many desktop publishing features.

**$21.95** paperback, *ISBN 0-07-881291-7*

Advanced TURBO PASCAL
Using TURBO PASCAL
TURBO PASCAL Programmer's LIBRARY
Advanced TURBO PROLOG Version 1.1
Advanced TURBO C
Using TURBO C
BORLAND-OSBORNE/McGRAW-HILL PROGRAMMING SERIES

## The Borland-Osborne/McGraw-Hill Programming Series

### Using Turbo C®

*by Herbert Schildt*

Here's the official book on Borland's tremendous new language development system for C programmers.

**$19.95** paperback, *ISBN 0-07-881279-8*

### Advanced Turbo C®

*by Herbert Schildt*

For power programmers. Puts the amazing compilation speed of Turbo C® into action.

**$22.95** paperback, *ISBN 0-07-881280-1*

### Advanced Turbo Prolog® Version 1.1

*by Herbert Schildt*

Now Includes the Turbo Prolog Toolbox™ with examples

**$21.95** paperback, *ISBN 0-07-881285-2*

### Turbo Pascal® Programmer's Library

*by Kris Jamsa and Steven Nameroff*

Revised to cover Borland's Turbo Numerical Methods Toolbox™

**$21.95** paperback, *ISBN 0-07-881286-0*

### Using Turbo Pascal®

*by Steve Wood*

Featuring MS–DOS programs, memory resident applications, in-line code, interrupts, and DOS functions

**$19.95** paperback, *ISBN 0-07-881284-4*

### Advanced Turbo Pascal®

*by Herbert Schildt*

Expanded to include Borland's Turbo Pascal Database Toolbox® and Turbo Pascal Graphix Toolbox®

**$21.95** paperback, *ISBN 0-07-881283-6*

Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710

***Available at Book Stores and Computer Stores.***
***OR CALL TOLL-FREE 800-227-0900***
***800-772-2531 (In California)***

*In Canada, contact McGraw-Hill Ryerson, Ltd. Phone 416-293-1911*