TURBO TECHNIX™

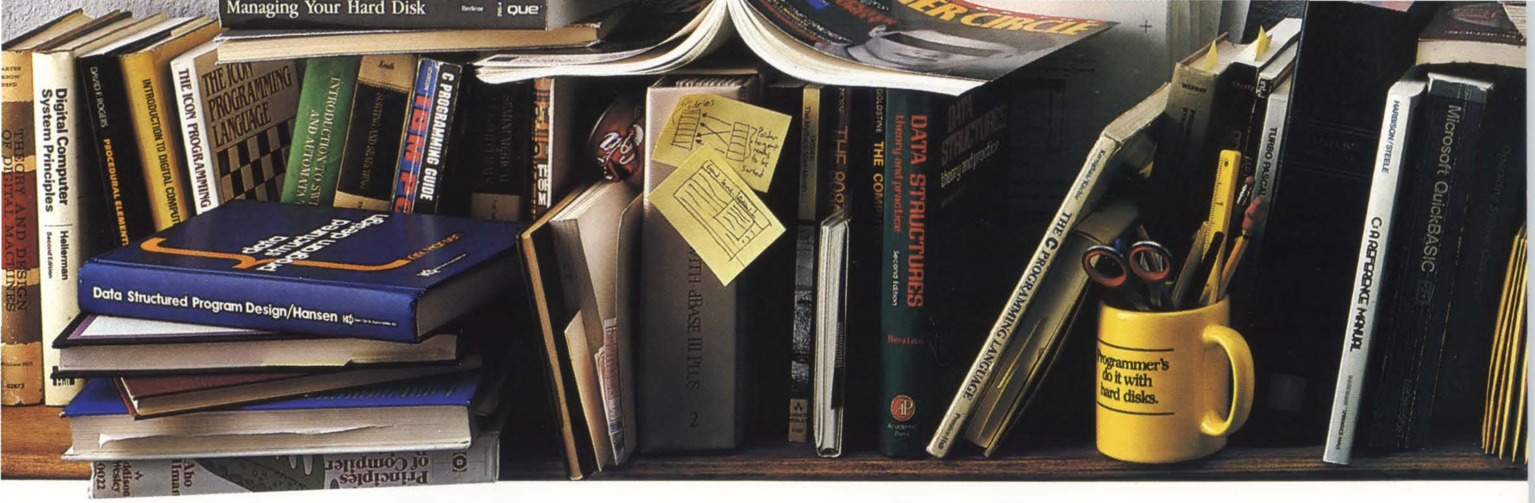
THE BORLAND LANGUAGE JOURNAL • NOVEMBER/DECEMBER 1987 • VOLUME ONE NUMBER ONE • $10.00

TURBO PASCAL 4.0 ARRIVES
Reflections on the crystalline language

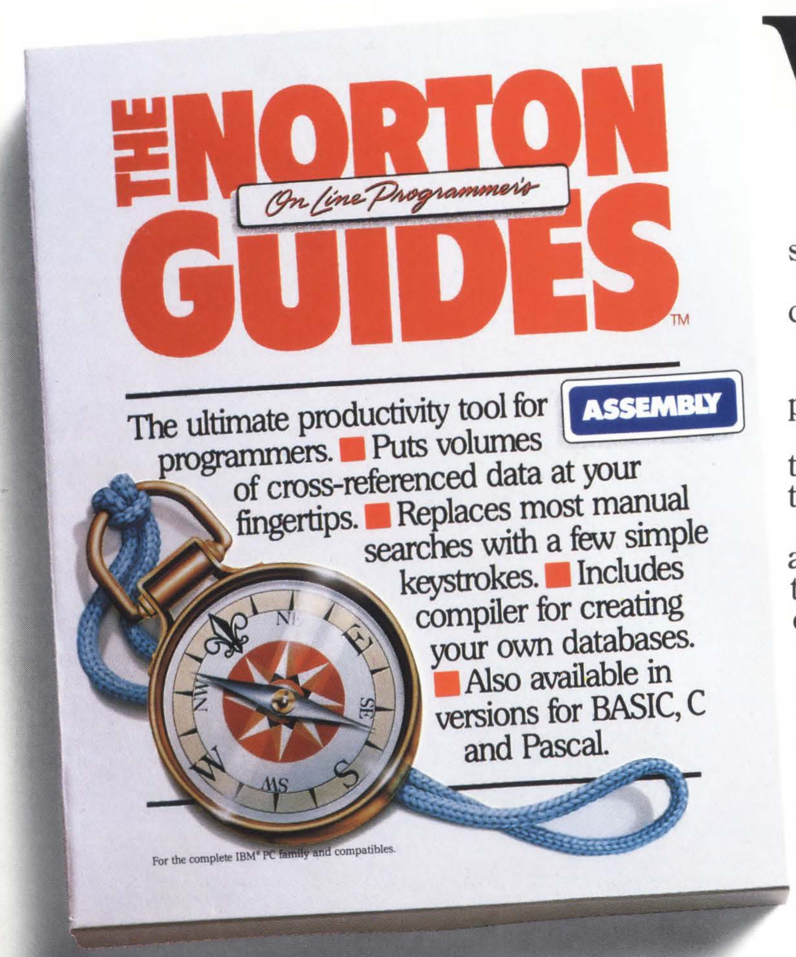
Using Recursive Types in Turbo Prolog

Taking Charge of DOS Volume Labels

Turbo Pascal EXEC


Managing Your Hard Disk
que
Digital Computer System Principles
Hellerman
THE ICON PROGRAMMING LANGUAGE
C PROGRAMMING GUIDE
Data Structured Program Design/Hansen
WITH dBASE III PLUS
DATA STRUCTURES theory and practice
THE C PROGRAMMING LANGUAGE
TURBO PASCAL
Programmer's do it with hard disks.
HARBISON/STEELE
C REFERENCE MANUAL
Microsoft QuickBASIC

Peter Norton. new programmi who hate


THE NORTON On-Line Programmer's GUIDES™
ASSEMBLY
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.
For the complete IBM® PC family and compatibles.

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 on the left.

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.

And when we say everything, we mean everything.

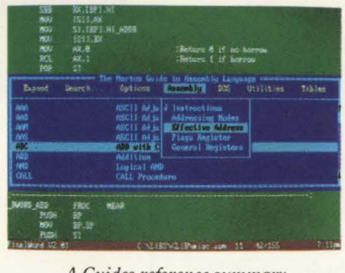
Everything from information about language

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

IBM
2.02
PROGRAMMER'S PROBLEM SOLVER
IBM
Macro Assembler
WRITING INTERACTIVE COMPILERS AND INTERPRETERS
DON'T PANIC!
VIEW EDIT
NUMERICAL RECIPES
IBM
3.0
COMPUTER SYSTEMS ARCHITECTURE
Coca-Cola

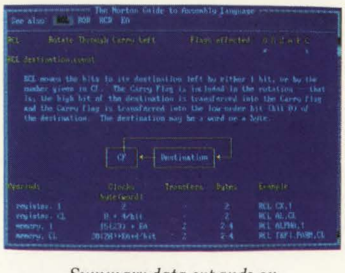
announces a ng tool for people manual labor.

syntax to a variety of tables, including ASCII characters, line drawing characters, error messages, memory usage maps, important data structures and more.



```
        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
 ABC                 ADD with C│ General Registers
 ADD                 Addition
 AND                 Logical AND
 CALL                CALL Procedure

_DWORD_ADD      PROC    NEAR
        PUSH    BP
        MOV     BP,SP
        PUSH    SI
```

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  ROL

RCL       Rotate Through Carry Left          Flags affected:  O D I 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.

How much more?

Well, the databases for BASIC, C and Pascal give you detailed listings of all built-in and library functions.

While the Assembly database delivers a complete collection of DOS service calls, interrupts and ROM BIOS routines.

You can, of course, find most of this 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 full-screen or moveable half-screen mode.

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

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

And you're right.

But even Peter Norton can't think of everything.

Which is why 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.

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 ©1987 Peter Norton Computing

TURBO TECHNIX
The Borland Language Journal
November/December 1987
Volume I Number 1

TURBO PASCAL

8 Turbo Pascal Arrives!
Jeff Duntemann

10 Turbo Pascal at 4
Jeff Duntemann

20 Turbocharging: Moving from 3.0 to 4.0
Bruce Webster

24 Getting to Know Units
Tom Swan

32 Communicating with Child Processes
Neil Rubenking

36 Exploring the Borland Binary Editor
Jeff Duntemann

47 Sense and Semicolons
Jeff Duntemann

TURBO C

52 Taking Charge of DOS Volume Labels
Kent Porter

58 Pointers in Turbo C
Mike Floyd

62 The End of the Line
Jonathan Sachs

65 Implementing Binary Trees
Kent Porter

68 Julian Days and Dates
Marty Franz

74 A Programmer's Guide to the Parallel Port
Bruce Eckel

TURBO PROLOG

80 Recursive Data Types in Turbo Prolog
Michael Covington

87 Extracting Routines from the Turbo Prolog Toolbox
Dan Shafer

94 Thinking in Turbo Prolog
Alex Lane

99 Bit by Bit
Tom Castle and F. Barclay Shilliday

TURBO BASIC

104 Starting Out with the Turbo Basic Database Toolbox
Peter Aitken

111 DOS Calls from Turbo Basic
Ethan Winer

114 Turbo Basic Communications
Reid Collins

124 Exploring the CIRCLE Statement
Peter Aitken

BUSINESS LANGUAGES

128 Thinking in PAL
Todd Freter and Ken Einstein

DEPARTMENTS

4 From the Publisher
Marcia Blake

6 BEGIN: Hello, World
Jeff Duntemann

132 Binary Engineering: Divide and Rule
Bruce Webster

136 Language Connections: The Turbo Prolog—Turbo C Connection
Gary Entsminger

142 Archimedes' Notebook: Flexible Curve-fitting with Eureka
Namir Clement Shammas

147 Tales from the Runtime: Getting Started
Mark L. Van Name and Bill Catchings

152 Critique: QuickPak
Marty Franz

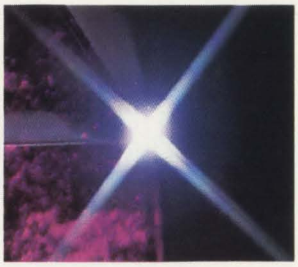
153 Critique: TECH Help!
Peter Aitken

155 Bookcase: Memory Resident Utilities
Reviewed By Rick Ryall

156 Bookcase: Advanced Turbo Prolog
Reviewed by Keith Weiskamp

158 Turbo Resources

160 Philippe's Turbo Talk
Philippe Kahn



8

Turbo Pascal 4.0 faithfully reflects the structure of your program designs in fast, compact code. The large code model, separate compilation, and a host of other enhancements are now at your command.

52

DOS volume labels can be more useful than you think, and manipulating them could be considerably less difficult than you may have feared.

80

Turbo Prolog allows types that contain themselves as fields. This powerful feature facilitates easy implementation of recursive data structures such as binary trees.

104

All the tools you need for building a custom database in Turbo Basic are contained in the Turbo Basic Database Toolbox.

Cover:
Turbo Pascal 4.0 will faithfully reflect your ideas as structured applications. Photograph by Bradley Ream.

Now, for Turbo Pascal 4.0!

Interlocking Pieces: Blaise and Turbo Pascal.

Whether you're a Turbo Pascal expert or a novice, you can benefit from using special tools to enhance your programs, make them reliable, and give them a professional look. With windows, menus, pop-up memory resident programs, and communications support, Blaise Computing offers you all the right pieces to solve your Turbo Pascal development puzzle.

◆**POWER TOOLS PLUS**™ is a general purpose library of over 140 functions and procedures, like fast direct video access, general screen handling including multiple monitors, and EGA 43-line text mode. Stackable and removable windows with optional borders and cursor memory provide complete windowing capabilities. Pop-up menus are easily incorporated into your programs using the menu management functions.

POWER TOOLS PLUS allows you to inspect, allocate, and free DOS memory. Execute any program, including internal DOS commands, from within your Turbo Pascal program.

Write general interrupt service routines in Turbo Pascal for either hardware or software interrupts using the POWER TOOLS PLUS ISR support functions. Blaise Computing's unique intervention code lets you develop memory resident applications that take full advantage of DOS capabilities. With simple procedure calls, "schedule" a Turbo Pascal procedure to execute either when a "hot key" is pressed or at a specified time.

◆**ASYNCH PLUS**™provides the crucial core of hardware interrupts needed to support asynchronous data communication. This package offers simultaneous buffered input and output to both COM ports at speeds up to 9600 baud, and supports the XON/XOFF protocol, XMODEM file transfer and modem control functions.

The underlying functions of ASYNCH PLUS are carefully crafted in assembler and drive the UART and programmable interrupt controller chips directly. These functions, installed as a memory resident system, require just 3.2K bytes. Complete Turbo Pascal interfaces to the assembler routines are provided.

The source code is included in all Blaise Computing products and is efficiently crafted, readable and easy to modify. The INCLUDE utility is provided to automatically determine which procedures and functions are required, and to compact them all into a single file to be included in your application. Each package is accompanied by an indexed manual describing each procedure and function in detail with example code fragments. The documentation reflects the attention to detail and commitment to technical support that have distinguished Blaise Computing over the years.

NOW, EVEN MORE POWER! Turbo ASYNCH PLUS and Turbo POWER TOOLS PLUS are now designed explicitly for Turbo Pascal Version 4.0. New high level functions provide reliable, fast, professional routines — the right combination of pieces to put your Turbo Pascal puzzle together. All source code is included for only **$129.00** each.

BLAISE COMPUTING INC.

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

THE BLAISE MENU

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.

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.

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

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

Turbo C TOOLS *$129.00*
Windows; ISRs; intervention code; screen handling and EGA 43-line text mode support; direct screen access; DOS file handling and more. For Turbo C.

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/5.0 *$129.00*
Intervention code, ISRs, Windows, direct screen access, DOS file handling and more. Designed specifically for Microsoft C 5.0 and QuickC.

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.

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.

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

TO ORDER CALL TOLL FREE
800-333-8087

TELEX NUMBER-338139

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

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

TURBO TECHNIX
The Borland Language Journal
November/December 1987
Volume I Number 1

Publisher
Marcia Blake

Editor in Chief
Jeff Duntemann

EDITORIAL

Managing Editor
Michael Tighe

Technical Editor
Michael A. Floyd

Copy Editor
Eileen Dempsey

TECHNICAL CONSULTANTS

Brad Silverberg
David Intersimone
Gary Whizin

DESIGN & PRODUCTION

Art Director
Karen Miner

Typesetting Manager
Walter Stauss

Typesetter/System Supervisor
Jeffrey Schwertley

Assistant Typesetter
Jeanie Maceri

Photographer
Bradley Ream

Typesetting Traffic
Charlene McCormick

ADMINISTRATION

Circulation
Pete Townsend

Purchasing
Brad Asmus

ADVERTISING

Assistant to the Publisher
Annette Fullerton
(408) 438-9321

Advertising Sales Manager
John Hemsath
(408) 438-9321

Western Region
Janet Zamucen
(714) 586-1517

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

South Region
Megan Patti
(813) 394-4963

FROM THE PUBLISHER

Marcia Blake

Welcome to the pages of Borland's new language journal!

We've had a great time assembling the first issue of *TURBO TECHNIX*, especially since we've done it "the Borland way." Unlike other new magazines, we never printed a "dummy" with free space given to advertisers in an attempt to fill the book. We didn't spend a fortune promoting the magazine to readers (they were already there, asking for copies long before the magazine's existence was anything more than a rumor). We simply collected a small-but-mighty staff of a few of the best people in the business, gathered material from some of the most talented writer/programmers around, and the rest, as they say, "is history."

And you're lookin' at it.

Like Borland software, the magazine is a jet fighter—powerful and compact—and, also like the software, the magazine will continue to provide full value long after it is first published. Advertisers and readers alike will benefit repeatedly from the structure and the editorial charter of *TURBO TECHNIX*, and those benefits are both immediate and lasting.

Readers of *TURBO TECHNIX* know they can find a broad spectrum of Borland language expertise in these pages, and they'll also find all the "latest and greatest" offerings from a variety of manufacturers and developers. And *TURBO TECHNIX* advertisers can be confident that their advertising dollars are reaching a valuable, qualified market of power users: registered Borland language owners...early *adapters* and early *adopters*...true power users *and power buyers* on the cutting edge of new technology.

Powerful and compact ...that's the magazine, and that's the team that put it together. And, with our readers and advertisers, that's the team that will go on to build ever-better programs for the ultimate benefit of all. So, climb aboard—and enjoy the ride as this jet fighter takes off! ■

Bradley Ream

The TURBO TECHNIX launch team, from left: copy editor Eileen Dempsey, managing editor Michael Tighe, publisher Marcia Blake, assistant to the publisher Annette Fullerton, editor in chief Jeff Duntemann, art director Karen Miner and technical editor Mike Floyd.

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. U.S. subscriptions are $49.95 per year; Canada US $60.00 per year. Single copy price is $10.00. Third-class postage pending at Santa Cruz, CA, and at additional mailing offices. Phone (408) 438-9321. Postmaster: send address changes to Subscription Dept., 4585 Scotts Valley Drive, Scotts Valley, CA 95066. For a statement of our permission policy for use of listings appearing in the magazine, please send a self-addressed stamped envelope to TURBO TECHNIX, Permissions, Borland Communications, 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.

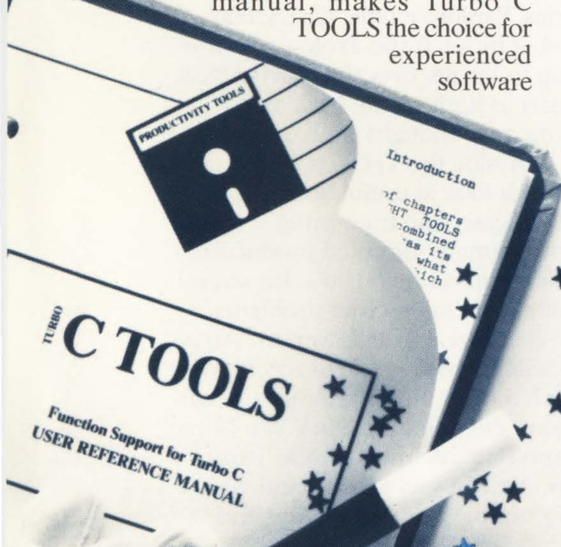
More Magic from Blaise.

Turbo C TOOLS™

```
#include <bwindow.h>
WIN_NODE *wncover(pnr
WIN_NODE *pnode;
LOC      *pcorn
DIM      *pd
{
    WIN_N
    BWIN
    int
    int
    if
    {
```

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

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!

BLAISE COMPUTING INC.

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

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.

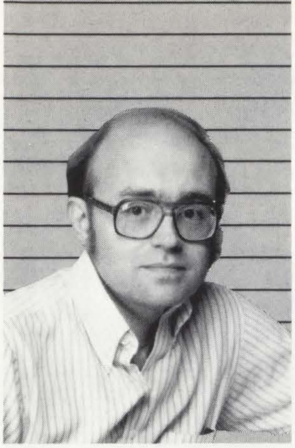
RUNOFF *$49.95*
Text formatter for all programmers; flexible printer control; user-defined variables; index generation; general macro facility. Crafted in Turbo Pascal.

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: ______

Turbo C is a trademark of Borland International.



BEGIN

Hello, world.

Jeff Duntemann

You can write a grocery list. Or you can write *The Great Gatsby.* One will get you through a shopping trip, and the other is bought by millions and remembered forever as a classic. Both are works executed in a language called English, and the fact that few of us can aspire to writing a timeless classic doesn't keep a great many grocery lists from being written.

The great truth that not everyone needs to write a 200,000-line application was part of the force that propelled Borland International to prominence a few years ago. The right ten lines of Turbo Pascal can get you out of some nasty corners. Industry pundits have tried to discredit the notion of general computer *language* literacy (as distinct from simple computer literacy, i.e., not shrinking in horror from the C> prompt) by stating that the need for professional programmers will always be minuscule compared to the number of small computers in use. That's true, but entirely beside the point.

Again, the comparison to English is strong and valid: The number of people who make their living writing English prose is vanishingly small compared to the number of people who read and write well. You can live in America as an illiterate, just as you can use your computer without knowing a thing about programming, but in both cases you are very much at the mercy of others. To take control of your own life or your own PC, you need to understand the language of the realm.

This magazine is in your hands because your recent purchase and registration of a Borland language or toolbox entitles you to a 12-month free subscription—a $49.95 value. If you're reading a friend's copy, you can purchase a subscription through the card attached to the binding. It's as simple as sending in one card or another—except that the "other" card has to have money along with it.

Our mission in creating *TURBO TECHNIX* is to help you become more literate in your chosen Borland language. Our method can be summed up as the process of answering two questions: *How does it work?* and *How do I do it?* In each article we'll explain some programming principle at work or some facet of the machine beneath it—then cement the knowledge by offering some useful code that illustrates the principle under discussion.

The heart of every issue is what we call the Technix Core. The Core consists of five departments, one each for Turbo Pascal, Turbo C, Turbo Prolog, and Turbo Basic, plus a separate department for Borland's new line of programmable business products: Quattro, Paradox/PAL, Sprint and Eureka.

Within each language department are at least three articles, one at each of three broad experience tracks.

Square One articles are for computer-literate newcomers to a given language. These articles explain the structures and features of the language without assuming a great deal of background.

Programmer articles are targeted at competent practitioners of the language. We assume that you know your way around and are looking for a little more depth and breadth. By and large, Programmer articles are less about language features than about how to apply them toward getting your work done.

Finally, *Wizard* articles explore the language areas that may have lain just beyond your reach: assembly language interface, floating point arcana, interlanguage calls, interrupt handlers, and other items that (unlike your editor) are said to have considerable hair.

All this structure exists to keep confusion to a minimum, but we don't intend our tracks to become ruts. Explore a little. Check out what those Turbo Prolog guys are up to...or discover that Turbo Basic isn't as basic as the basic BASIC you once thought so...base.

Outside the Technix Core, we take a look at useful books in the Bookcase, and interesting programmer-oriented products in Critique. We help you solve scientific and engineering problems with Eureka: The Solver, in "Archimedes' Notebook." Plus, we have columns by Bruce Webster on software engineering, Mark Van Name and Bill Catchings on the Turbo C runtime library source, and Gary Entsminger on interfacing Turbo Prolog to Turbo C.

We plan to balance it all, from issue to issue, so that no one goes away from the feast hungry, and we intend to do it with the sort of light heart that keeps work from becoming drudgery.

In the Borland tradition, no magazine has been done quite this way before. Tell us what you think. Certainly, tell us what you, as Turbo programmers, really *need.* In helping you work with our languages, please let us know how well we're speaking yours. ■

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.

TURBO PASCAL 4.0 ARRIVES!

Jeff Duntemann

Pascal is the crystalline language. The image of every component mirrors the structure of the whole, and in every glint is the brilliance of Niklaus Wirth's seminal design. As a vehicle for teaching the creation of structured programs it has no equal, but Pascal came only slowly to acceptance as a worthy tool for commercial software development.

This was largely due to the fact that early implementations were as lame as the original design was brilliant. The state of the Pascal art improved with time, but one common thread ran through even the best implementations: a single-minded insistence on portability over speed. Writing a Pascal compiler in Pascal makes for a ponderously slow (if easily portable) compiler. Generating P-code instead of native code allows an application to run badly on as many different machines as possible.

Time and again, a lesson too easily forgotten in our industry must be relearned: The single most important attribute of any software tool is *speed.* Time is only money in most industries. In software development, time may be money, but speed is *survival.* Early Pascal implementors continued to shout about portability and wondered why no one seemed to be listening.

Then came the autumn and winter of 1983-84. All around the country, thousands of skeptical hands were snapping a single disk into diskette drives, and typing TURBO at the A> prompt. The reaction, passing from mouth to mouth as thousands mounted to tens and finally hundreds of thousands, could be characterized in only one word: *Wow!*

Turbo Pascal improved markedly through its first three major versions. Greater speed in both compilation and generated code, overlays, better graphics, and improved control over the underlying system all contributed to its quick dominance of the Pascal scene. There comes a time, however, when evolutionary change is not enough, so building on four years of experience, Borland's development team has rewritten Turbo Pascal from scratch. In this premiere issue of *TURBO TECHNIX,* we've devoted the greater part of the Pascal section of the Technix Core to introducing you to the new features and power of Turbo Pascal 4.0.

I'll provide an overview of the new compiler, and its completely new user interface, in "Turbo Pascal at 4." The matter of converting source code from Turbo Pascal 3.0 to 4.0 is well-covered by Bruce Webster, in "Turbocharging Your Code for 4.0." Tom Swan eases the way toward separate compilation under Turbo Pascal 4.0 in "Getting to Know Units." In "Communicating with Child Processes," Neil Rubenking explains how to

EXEC to child processes and pass data back and forth between parent and child—surely one of the thorniest problems of the Turbo Pascal 3.0 era. Finally, "Exploring The Borland Binary Editor" describes an intriguing new feature of the updated and completely rewritten Turbo Pascal Editor Toolbox.

I've always thought of computer programming as the reflection of the visions of the programmer in the power of the language. There is a necessary tension between the blazing if unformed light of an idea and its final coalescence into gritty reality as a program. The measure of a computer language is how faithfully it can reflect that blaze of an idea within the jumble of compromises we call the PC architecture. A language of limited expressibility like COBOL bends the light always toward its own narrow shape, while a language of unlimited amorphous complexity like PL/1 scatters the light in every direction, making every application a uniform shade of gray. Somewhere in the middle are the crystalline facets of Turbo Pascal 4.0, faithfully reflecting your idea as the application it will (with planning and care) become. ■

Bradley Ream

TURBO PASCAL AT 4

With Release 4.0, Turbo Pascal celebrates its fourth birthday—but the presents are for you.

Jeff Duntemann

SQUARE ONE

In four years there have been three Turbo Pascals, and now, entering its fifth year, there will be a fourth. In a larger sense, the first three releases were one product, each a proper superset of its predecessor with only minor differences between them. Turbo Pascal 4.0, by contrast, is a radical change. The compiler has been completely redesigned for a new memory model, separate compilation, and a very sophisticated user interface. Changes like these do not sum up well in a paragraph or two. Follow along on a quick tour of the new facets Turbo Pascal has turned toward the light on its fourth birthday.

IN UNITS THERE IS STRENGTH

Separate compilation has never been part of the Pascal language definition, but time has shown that it's very hard to manage large projects without it. Turbo Pascal 4.0 adopts the units paradigm for separate compilation pioneered by UCSD Pascal years ago and used by Turbo Pascal for the Macintosh since its release.

Elsewhere in this issue, Tom Swan discusses units and their use in "Getting To Know Units." In brief, a unit is a collection of data definitions and subprogram declarations compiled together and stored as a .TPU (Turbo Pascal Unit) file on disk. In Modula 2 fashion, a unit has two parts: An *interface* part containing the definition of constants, data types and variables, and the headers of procedures and functions; and the *implementation* part, which contains the procedure and function local variables as well as the source code. This allows the exhaustive interface to a group of routines to be published without revealing the exact details of their inner workings. Quite apart from protecting trade secrets in third party libraries, this scheme facilitates the division of large software products into components that may be implemented by separate groups of programmers without the possibility of "sneak paths" and other bug generators that happen when one module is allowed to "peek" at another's innards.

A program makes use of units through a new statement type—the **USES** statement:

```
PROGRAM JiveTalk;

USES DOS,CRT,CircBuff,
      PullDowns,XMODEM;
```

The Turbo Pascal linker automatically links the code for each unit into the memory image of the program being compiled *without* recompiling any of the units.

Units have another interesting property: Each unit contains an *initialization section* and an *exit procedure pointer*. The initialization section is a special procedure which is executed before the main program starts running, allowing a unit to set up hardware, save and modify interrupt vectors, etc. The exit procedure pointer allows the definition of a procedure that will be executed after the main program terminates. Exit procedures are not likely to be needed as frequently as the initialization section, but they do allow for tasks such as resetting hardware to a default state and restoring saved interrupt vectors. (The main program, while not considered a unit, also has its own initialization section and exit pointer.) This allows a unit (or the main program) to contain system-level constructs that need to install themselves before execution begins and then remove themselves before control returns to DOS.

The standard units CRT and DOS, the runtime library, and the 3.0 compatibility unit TURBO3 are present in a special file called TURBO.TPL (the extension means Turbo Pascal Library) which is read into memory when the compiler is invoked. Because the resident library contains the routines most frequently linked into compiled programs, its presence speeds linking considerably. TPLIB can be used to add frequently referenced programmer-created units to the resident library, although at the cost of some free RAM. For example, when developing a program that makes heavy use of graphics, the memory lost to making GRAPH.TPU resident will be more than paid for by the increase in compile/link speed. Similarly, if

Bradley Ream

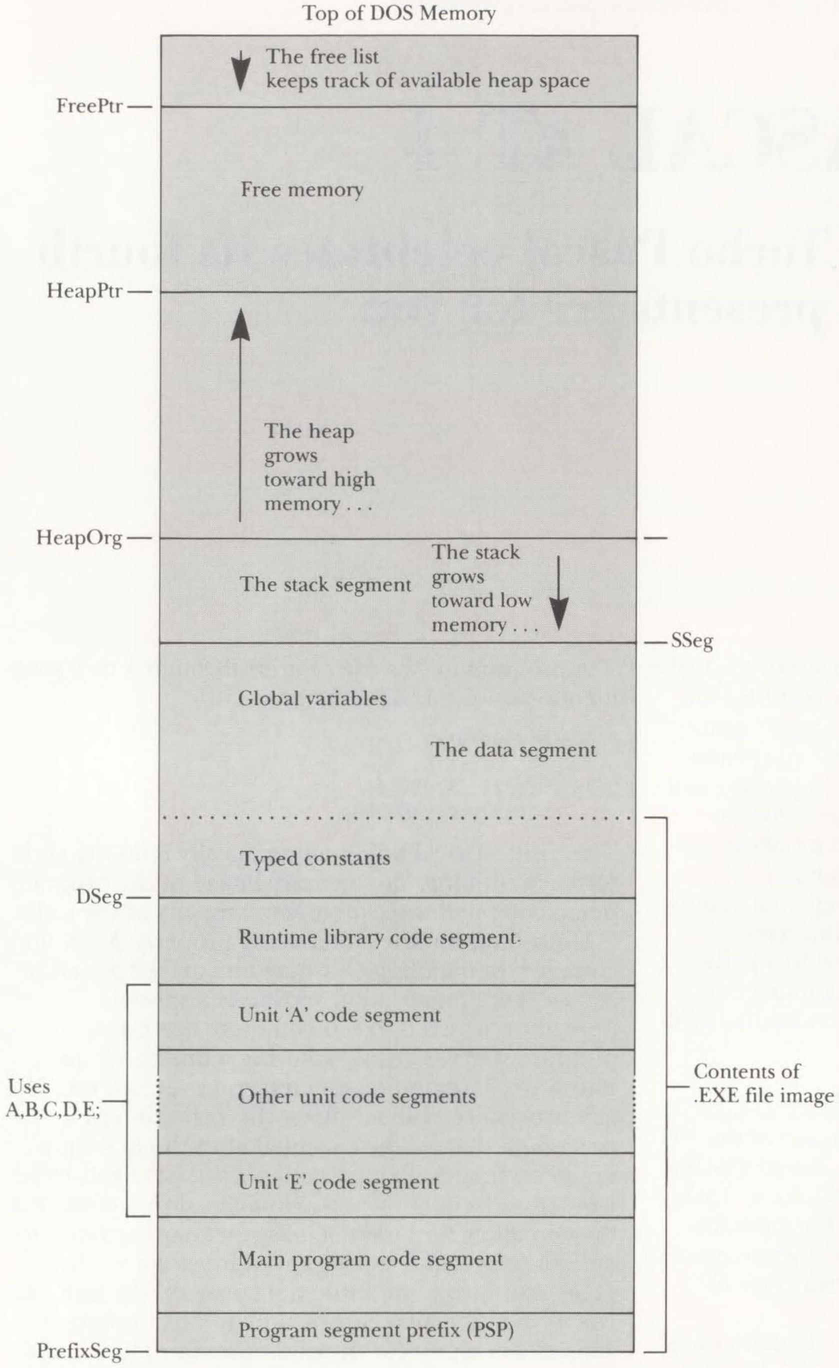

Top of DOS Memory
The free list keeps track of available heap space
FreePtr
Free memory
HeapPtr
The heap grows toward high memory . . .
HeapOrg
The stack segment
The stack grows toward low memory . . .
SSeg
Global variables
The data segment
Typed constants
DSeg
Runtime library code segment
Unit 'A' code segment
Uses A,B,C,D,E;
Other unit code segments
Unit 'E' code segment
Main program code segment
Program segment prefix (PSP)
PrefixSeg
Contents of .EXE file image

Figure 1. The Turbo Pascal 4.0 memory map.

TURBO PASCAL AT 4

memory use becomes a problem, unneeded resident units can be removed from memory with TPLIB.

R.I.P. SMALL MODEL

Almost certainly, the single most clamored-for feature in the new compiler is the move to the large code model. The .COM files are gone, and with them that 64K ceiling that we all came to bump against entirely too soon. Now, beyond certain minimum requirements for data, stack, and heap, your programs may contain as much code as will fit under the DOS 640K memory limit. Each separately compiled unit has its own code segment, which may contain up to 64K of code.

The rest of the runtime memory map closely resembles that of Turbo Pascal 3.0. Data and stack are each given a segment; all memory not occupied by code, data, or stack is allocated to the heap (see Figure 1). Note that typed constants now reside in the data segment rather than in the code segment. Thus it's no longer possible or even necessary to store the program's DS value in a typed constant during execution of Turbo Pascal interrupt service routines—the new interrupt directive for interrupt procedures handles the saving and storing of DS automatically, as I'll explain.

Version 3.0's use of the 8086 small code model masks much of the complication of the 86-family architecture from the programmer. In moving to a large code model some of this complication must be taken into account. Procedures and functions may now be declared as "near" or "far" using the $F compiler directive. A near procedure may *only* be called from within its unit. A far procedure may be called from anywhere in the program. When the $F directive is passive ({**$F−**}, the default) the compiler decides whether to compile a subprogram as near or far. Subprograms declared in the interface section of a unit are compiled as far, while subprograms declared wholly in the implementation part of a unit (and hence are private to that unit) are declared near.

When $F is active ({**$F+**}), however, all subprograms are compiled as far subprograms. Bracketing an otherwise near subprogram between $F+ and $F− directives will force it to be compiled as far:

```
{$F+}
FUNCTION PanicButton(SystemStatus)
         : Boolean;
{$F-}
```

Far subprograms require 32-bit calls and returns, which can add noticeable overhead when executed from within a tight loop.

A WEALTH OF NUMBERS

For a strongly typed language, Pascal was not originally defined with a great many numeric types. The familiar 16-bit **Integer** fits neatly in an 8086 register and thus meshes well with optimized code, but these days, **MaxInt** (32,767) is small change. Counting things like bytes on a disk, free bytes on the heap, or records in a DOS disk file has, until now, been done with type **Real**, which is a "measured quantity" type and shouldn't

continued on page 14

```
PROCEDURE MyISR(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP : Word);
INTERRUPT;

BEGIN
  .
  .
  .

END;
```

Figure 2. An interrupt procedure declaration.
The pseudo-parameters do not accept actual parameters, but rather exist to allocate stack space to store machine register values. Because the registers are saved on the stack rather than in a static buffer, the interrupt procedure is re-entrant. The pseudo-parameters may be modified within the procedure, and the modified values will be loaded into the machine registers before the procedure returns control to the interrupted code.

{$define SYMBOL}	define symbol
{$undef SYMBOL}	undefine symbol
{$ifdef SYMBOL}	if SYMBOL is defined
{$ifndef SYMBOL}	if SYMBOL is not defined
{$else}	
{$endif}	
{$ifopt X+ }	if X compiler directive ON (for ON/OFF switches only)
{$ifopt X–}	if X compiler directive OFF
PREDEFINED SYMBOLS:	
{$define TP40}	defined if this is V4.0
{$define INTEL}	defined if this is DOS Turbo Pascal
{$define INTEL87}	defined if coprocessor present

Table 1. Conditional Compilation symbols and directives.

TURBO PASCAL AT 4

continued from page 12

be used for counting things.

Turbo Pascal 4.0 puts things right with a whole passel of new numeric types that should serve us well long into the gigabyte era. The Most Valuable Player in this new lineup is almost certainly **LongInt**, implemented as a 32-bit quantity with the high bit acting as a sign bit. Values of type **LongInt** range from −2,147,483,648 to 2,147,483,647. The familiar file-handling procedures **Seek**, **FilePos**, and **FileSize** now take **LongInt** parameters instead of **Real** parameters. **LongInts** can't index arrays, since arrays must be definable as static data; i.e., an array must fit in a single 64K 8086 segment. Type **Word** (see below) can index any array that will fit in a single segment, so there is no need for **LongInt** array indices. Otherwise, **LongInt** is assignment-compatible and expression-compatible with other numeric types.

Type **Word** is an "unsigned integer," similar to Modula 2's type **Cardinal**. **Word** is implemented as 16 bits but without any sign bit, allowing it to express values from 0 to 65,535. It may index arrays and otherwise do the same work as **Integer** for positive quantities. **Word** will be especially useful for counting in-segment memory locations beyond 32,767 when using **Mem** and **MemW**.

At the other end of things, type **ShortInt** provides a "signed byte" expressing values from -128 to 127. **ShortInt** can do anything that the other integer types can do within its somewhat limited range.

The default type **Real** is identical to the fast six-byte real type used in earlier versions of the compiler. In machines with a numeric coprocessor, however, several new real-number types become available, all of them part of the IEEE floating point spec.

Type **Double** represents the IEEE eight-byte double-precision real format. This is the same as type **Real** as implemented in Turbo-87 Pascal through version 3.0. Type **Single** is the IEEE single-precision real format, which is expressed in four bytes.

Type **Extended** is the IEEE 10-byte temporary real, which provides unprecedented range in the Pascal world, extending from 3.4×10^{-4932} to 1.2×10^{4932}. This type must be used with the caution that there is no "larger" numeric type to "graduate" an intermediate result to if it overflows **Extended**'s range.

Finally, **Comp** provides an eight-byte integer type that correponds to the 8087 "long integer" type. (What Turbo Pascal implements as a "long integer" is actually the 8087's "short integer," while an ordinary Pascal integer is called a "word integer" in the 8087 lexicon.)

Types **Single**, **Double**, **Extended**, and **Comp** are not emulated but are implemented using inline coprocessor opcodes and the 8087 stack. A new compiler directive, **{$N+}**, instructs the compiler to generate coprocessor-specific code. The compiler is capable of detecting an installed coprocessor, and will refuse to compile source code containing the coprocessor-based real number types in a machine lacking a coprocessor.

NEW HEAP SOPHISTICATION

Heap management and pointers have undergone a little evolution between Turbo Pascal 3.0 and 4.0. A terser synonym for the **Addr** function has been provided: @. The @ operator returns a new type, the *generic pointer*. The generic pointer is nothing more than an 8086 machine address, and has no type of its own. It cannot be dereferenced in an expression, but can be cast onto any other pointer type.

Pointer types may now be included in record constants by using the predefined pointer constant **Nil**.

While the syntax of heap access changes little from Turbo Pascal 3.0, the heap itself is implemented differently. It is still a "long" heap accessed by 32-bit pointers, and is allocated all of memory not used for code, data, and stack. Unlike the 3.0 heap the 4.0 heap begins above the stack, and grows toward high memory (see Figure 1). The start of heap memory is given by a programmer-accessible pointer

continued on page 16

Turbo Pascal Programmers

sophisticated software

Presents

30 Day Money Back Guarantee!

PROFESSIONALS LIKE IT!

"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*, Feb. 24, 1987

"turboMAGIC is the Cadillac of prototyping tools for Turbo Pascal. ... Unlike the others, turboMAGIC is extremely flexible ... On the whole, turboMAGIC does what it claims it will do."
Jim Powell, *Computer Language*, June 1987

" I was impressed with the turboMAGIC package ... the end product is slick and professional ... the procedures created by turboMAGIC are well commented and easy to add to your own code. ... turboMAGIC is well worth its price. I only wish I had had it for the last menu-driven software I wrote!"
Kathleen Williams, *Turbo Tech Report*, June 1987

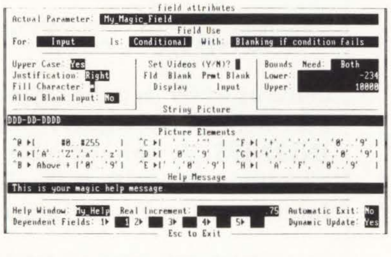
The World's Best Code Generator!

turboMAGIC™ saves you time

MAGIC will generate all of the Turbo Pascal code you need for your input forms, help windows, pop up menus, and pull-down menu systems.

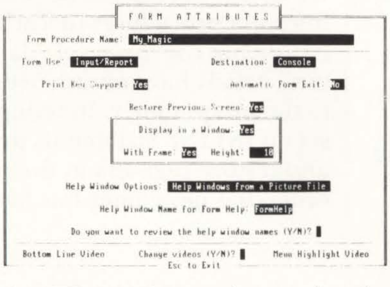
turboMAGIC™ is easy to use

1. Paint the form you want with the powerful editor.
2. Select any special attributes that you want for fields or the form.
3. Let *turboMAGIC*™ do the rest.



```
                          field attributes
Actual Parameter: My_Magic_Field
                             Field Use
For:   Input      Is: Conditional  With: Blanking if condition fails

Upper Case: Yes          Set Videos (Y/N)? ▌    Bounds  Need:  Both
Justification: Right    Fld  Blank  Prmt Blank  Lower:          -234
Fill Character: ▪         Display      Input    Upper:         10000
Allow Blank Input: No
                           String Picture
DDD-DD-DDDD
                          Picture Elements
^@ ▸[     #0..#255    ]  ^C ▸[  ' '..'~'  ]  ^F ▸['+',' ',' ','0'..'9' ]
^A ▸['A'..'Z','a'..'z']  ^D ▸[  '0'..'9'  ]  ^G ▸['+','-',' ',' ','0'..'9']
^B ▸ Above + ['0'..'9']  ^E ▸[' ','0'..'9']  ^H ▸[  'A'..'F',  '0'..'9'  ]
                            Help Message
This is your magic help message

Help Window: My_Help  Real Increment:            .75  Automatic Exit: No
Dependent Fields: 1▸  1 2▸    3▸    4▸    5▸         Dynamic Update: Yes
                             Esc to Exit
```

The FIELD ATTRIBUTES screen lets you describe a field.

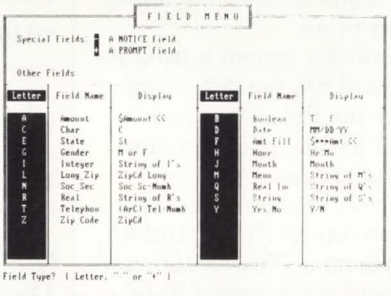


```
                        F O R M   A T T R I B U T E S
  Form Procedure Name: My_Magic

 Form Use: Input/Report                       Destination: Console

    Print Key Support: Yes                 Automatic Form Exit: No

                     Restore Previous Screen: Yes

                      Display in a Window: Yes

                     With Frame: Yes   Height:   10

          Help Window Options: Help Windows from a Picture File

             Help Window Name for Form Help: FormHelp

        Do you want to review the help window names (Y/N)? ▌

 Bottom Line Video       Change videos (Y/N)? ▌      Menu Highlight Video
                               Esc to Exit
```

The FORM ATTRIBUTES screen lets you describe a form.

turboMAGIC™ provides the ordinary... and a bit of the extraordinary!

- Forms that scroll in framed windows
- Forms up to 66 lines long using any colors and graphics characters
- Automatic updating of dependent fields
- Conditional-use fields (field and prompt can disappear if not needed)
- A help window per field or menu item
- Full-featured editing in fields
- Bomb-proof input handlers
- Support for printing forms, with video to font translation
- 19 Built-in field types available and you can add your own!



```
                          F I E L D   M E N U
  Special Fields: ▪ A NOTICE field
                  ▪ A PROMPT field

  Other Fields

 Letter | Field Name | Display          Letter | Field Name | Display

   A    | Amount     | $Amount CC         B   | Boolean    | T  F
   C    | Char       | C                  D   | Date       | MM/DD/YY
   E    | State      | St                 F   | Amt Fill   | $***Amt CC
   G    | Gender     | M or F             H   | Hour       | Hr:Mn
   I    | Integer    | String of I's      J   | Month      | Month
   L    | Long_Zip   | ZipCd Long         M   | Menu       | String of M's
   N    | Soc_Sec    | Soc Sc-Numb        Q   | Real Im    | String of Q's
   R    | Real       | String of R's      S   | String     | String of S's
   T    | Telephon   | (ArC) Tel-Numb     Y   | Yes No     | Y/N
   Z    | Zip Code   | ZipCd

Field Type? ( Letter, " " or "+" )
```

The FIELD MENU — an extensive set of types

ORDER TODAY. $99

CALL 800-225-3165 or 205-342-7026

Requires 256K IBM and Turbo Pascal 3.X

Sophisticated Software, Inc.
6586 Old Shell Road
Mobile, Alabama 36608

TURBO PASCAL VERSION:	V4.0	V3.0
MICROSTAR PROGRAM SIZE:	20,965 LINES	20,433 LINES
4.77MHz IBM PC, 80ms hard disk	3:04 (6.8K)	18:03 (1.1K)
6MHz AT-compatible 286, 29ms hard disk	1:07 (18.7K)	5:30 (3.7K)
10MHz AT-compatible 286, 29ms hard disk	0:54 (23.3K)	4:05 (5.0K)
16MHz AT-compatible 386, 29ms hard disk	0:43 (29.3K)	2:28 (8.3K)
16MHz AT-compatible 386, DOS 3.1 VDISK	0:21 (59.9K)	2:15 (9.0K)

Figures are compile time in minutes:seconds, and (lines per minute)

Table 2. Turbo Pascal 4.0 compilation speed benchmarks.The times given are for compiling the MicroStar application from the Turbo Pascal Editor Toolbox 2.0.

TURBO PASCAL AT 4

continued from page 14

called **HeapOrg**, and the address of the last allocated byte of the heap (and thus the start of free heap space) is given by pointer **HeapPtr**.

Space can be allocated on the heap in single-byte blocks, in contrast to the minimum eight-byte block of Turbo Pascal 3.0.

Free blocks of memory on the heap are tracked by a stack-like list of descriptors called the "free list." This list grows downward from the top of memory, and the end of the list is pointed to by a pointer called **FreePtr**. New descriptors are added to the free list as blocks of memory are freed up in the heap by **FreeMem** or **Dispose**. The free list is limited to one 64K segment, allowing about 8000 descriptors.

A very similar system is used in Turbo Pascal 3.0, the difference being that the details are now explained in the documentation and the heap manager's components, including the free list, are much more under the control of the programmer. The most important of these components from the programmer's perspective is a pointer to an optional heap error function. Once installed, the heap error function will take control if a call to **GetMem** or **New** requires more memory than is present in any single block on the free list. The heap error function makes possible true virtual storage systems, since you can "back out" of any attempted allocation if it's not successful, and swap occupied storage out to disk.

CASTING ABOUT

Turbo Pascal 3.0 has always allowed free conversion (or "casting") of scalar types from one to another with its retype facility. Turbo Pascal 4.0 greatly extends retyping by allowing the casting of any two types to one another providing both types are *exactly* the same size. This provides the same function as the free union variant record, but without the free union's feel of being a "dirty hack."

A simple example provides independent access to the two halves of a long integer. Given this record definition

```
Halfer = RECORD
           LoWord,HiWord : Word
         END;
```

and a variable named **WordVals** of this type, it is possible to cast a **LongInt** variable **LL** to **WordVals** by writing:

```
WordVals := Halfer(LL);
```

This provides word-access to **LL**'s internal storage without running afoul of Pascal's strong type checking. The free union variant record is still available for type casting between types of differing sizes, if that's ever necessary.

INLINE MACROS

An enhancement to Turbo Pascal's **INLINE** facility provides for **INLINE** procedures and functions through a new **INLINE** directive used in a subprogram header. An **INLINE** subprogram is different from an ordinary subprogram in that its code isn't called by way of the 8086 **CALL** and **RETURN** family of instructions, but rather is inserted into the code stream emitted by the code generator. **INLINE** subprograms are therefore actually machine-code macros. One copy of the subprogram is inserted into the code file at each invocation.

An **INLINE** subprogram may have parameters. The parameters are pushed onto the stack immediately before the code in the body of the subprogram takes control. As with ordinary subprograms, VAR parameters are passed as addresses to the corresponding actual parameters. Parameters are used by directly popping them off the stack into registers, and are not available symbolically. Using BP-relative addressing isn't necessary because no return address is pushed onto the stack after the parameters. The parameters are waiting at the top of the stack when the subprogram code begins execution.

Quite apart from the machinery of passing values to a macro, INLINE subprograms can add to a program's clarity by hiding the messy details of an INLINE code sequence behind a symbolic identifier. The two definitions

```
PROCEDURE EnableInterrupts;
  INLINE($FB);    { STI }
```

and

```
PROCEDURE DisableInterrupts;
  INLINE($FA);   { CLI }
```

allow us to replace the cryptic statements **INLINE($FA)** and **INLINE($FB)** with the more meaningful identifiers **DisableInterrupts** and **EnableInterrupts**.

INTERRUPT PROCEDURES

It's always been possible to write interrupt procedures in Turbo Pascal, but the low-level "dangerous" details have always been left to the programmer. In trying to set up INLINE statements to save and restore registers in the proper order, the newcomer has had to face some predictable havoc.

Turbo Pascal 4.0 pours some oil on the interrupt waters with a special procedure type designed to handle register protocol during execution as an Interrupt Service Routine (ISR). The declaration includes a new reserved word, **INTERRUPT**, and pseudo-parameters allocated on the stack for each of the 8086 registers, as shown in Figure 2.

The parameters are "pseudo" because of their special relationships to the 8086 machine registers. The Turbo Pascal runtime code copies the machine register contents into the pseudo-parameters just before execution begins, and will copy them back into the various machine registers just before execution ends. The interrupt procedure can therefore alter the pseudo-parameters, and the altered values will take the place of the register values saved when execution of the interrupt procedure began. This allows an ISR written in Turbo Pascal 4.0 to communicate with its callers through register values, as most BIOS ISRs do.

Pointing an interrupt vector to the interrupt procedure (and saving the previous vector, if necessary) are not handled automatically, and must be done explicitly by the programmer during the initialization of the application that wishes to use the interrupt procedure.

TEXT DEVICE DRIVERS

Turbo Pascal's predefined and preopened device files **Con**, **Trm**, **Kbd**, **Lst**, **Aux**, and **Usr** are no longer present, nor are their associated device names. Instead, a new mechanism has been provided for writing custom drivers for text file devices.

The mechanism involves writing four interface routines and assigning their addresses to four interface routine pointers in a special Turbo Pascal 4.0 text file interface block. This is done within a custom assign routine specific to a given device. The custom assign routine takes the place of the standard **Assign** procedure used to assign ordinary files to physical file names. Once the custom assign routine associates an ordinary text file variable with the desired device hardware, the file may be used exactly as any other text device.

By opening up the text file device machinery to the programmer, Turbo Pascal 4.0 allows the creation of drivers for custom hardware, or logical ports like COM3 and COM4 that are available as commercial hardware products but not supported by DOS. Equivalents to Turbo Pascal 3.0's very commonly used **Lst** and **Kbd** devices are provided in the **TURBO3** unit for compatibility's sake.

STANDARD UNIT GOODIES

Separate compilation makes a lot of things easier, foremost among them the distribution of subprogram libraries in compact, precompiled form. System-specific features that once had to be built into the compiler parser itself can now be left in a unit. Several such standard units are delivered with Turbo Pascal 4.0, including:

DOS. The **DOS** unit implements the familiar **MSDOS and Intr** procedures from Turbo Pascal 3.0, plus a number of new procedures giving high-level access to many useful DOS function calls. These include "find first" and "find next" file search routines; routines to set and query the system clock; to return the capacity and remaining space of a given disk drive; to set and query the attribute bits and time stamp of a given file; and to set and query interrupt vectors. Two *very* difficult Turbo Pascal 3.0 problems are solved in the **DOS** unit: Procedure **Keep** retains a correctly written Pascal program in memory as a Terminate and Stay Resident (TSR) utility; and **Execute** uses the DOS **EXEC** function to spawn a child process or invoke COMMAND.COM.

TURBO3. This is a compatibility unit containing functional equivalents to certain Turbo Pascal 3.0 built-ins that 4.0 either alters significantly or makes obsolete, including the **LongFileSize**, **LongFilePos**, **LongSeek**, **MemAvail**, and **Maxavail** routines, and the **KBD** and **LST** predefined device files.

CRT. All of Turbo Pascal's text-based CRT handlers (**GotoXY**, **ClrScr**, etc.) have been reimplemented as a unit written entirely in assembly language. This provides the fastest possible screen I/O for the standard IBM PC display modes while still allowing a programmer to implement a functionally compatible suite of CRT routines for non-standard display hardware.

GRAPH3. As with **CRT**, all the pixel-oriented graphics routines either built into Turbo Pascal 3.0 or residing in the GRAPH.BIN external file have been gathered together into a single assembly language unit. This unit supports the CGA and the EGA, as well as the new VGA.

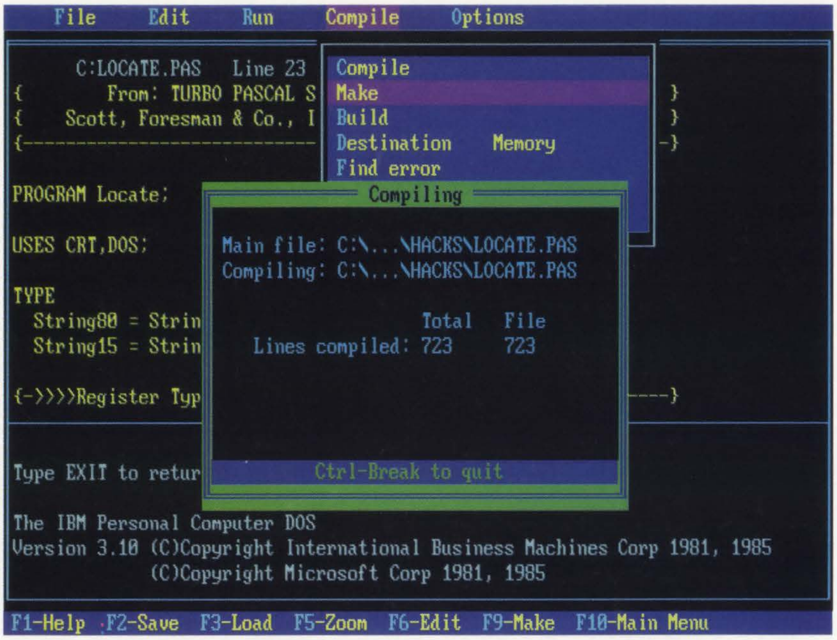
METALINGUISTIC MATTERS

New compiler directives add a great deal of richness to the Turbo Pascal language definition. First of all, include files may now be nested as many as eight levels deep. This allows include files to share other include files containing standard type or constant definitions.

Perhaps the most powerful of the new compiler directives implement conditional compilation. A summary of the supported conditional metacommands is given in Table 1.

The most obvious example of conditional compilation involves the real number dichotomy hinging on the presence or absence of a math coprocessor. Because a real number must be defined in one of two ways at compile time, conditional compilation allows a single source code file to compile to either six-byte software-only real numbers or the IEEE real numbers requiring the math coprocessor.

To illustrate, suppose an application uses a suite of real-number variables named **R**, **S**, and **T**. As software-only reals, they would have to be defined as **Real**; for use in a coprocessor-equipped system they would be have to be defined as **Double**. Conditional compilation predicated on the predefined symbol **INTEL87** handles it nicely:



```
     File      Edit      Run      Compile      Options

      C:LOCATE.PAS   Line 23  Compile
{         From: TURBO PASCAL S Make                              }
{    Scott, Foresman & Co., I  Build                             }
{----------------------------  Destination    Memory            -}
                               Find error
PROGRAM Locate;          ======= Compiling ========

USES CRT,DOS;            Main file: C:\...\HACKS\LOCATE.PAS
                         Compiling: C:\...\HACKS\LOCATE.PAS
TYPE
  String80 = Strin                          Total   File
  String15 = Strin          Lines compiled: 723     723

{->>>>Register Typ                                           ----}


Type EXIT to retur                 Ctrl-Break to quit

The IBM Personal Computer DOS
Version 3.10 (C)Copyright International Business Machines Corp 1981, 1985
             (C)Copyright Microsoft Corp 1981, 1985

F1-Help F2-Save F3-Load F5-Zoom F6-Edit F9-Make F10-Main Menu
```

Figure 3. The Turbo Pascal 4.0 Development Environment.

TURBO PASCAL AT 4

```
{$ifdef INTEL87}

TYPE
  Float = Double;

{$else}

TYPE
  Float = Real;

{$endif}

VAR
  R,S,T : Float;
```

The machinery for defining the symbol **INTEL87** is built into the compiler: If the compiler detects an 80X87 math coprocessor in the system it defines **INTEL87**, otherwise the symbol remains undefined. The **#ifdef INTEL87** directive *only* checks for the presence or absence of a defined symbol named **INTEL87**; it is *not* a Boolean matter of truth or falsehood.

A PLACE TO STAND

A programming environment is simply a place to stand, with access to tools. DOS qualifies, after a fashion: the C> prompt is a place to stand, but the tools are not well-integrated and you have to reach for them. Over a period of several years and through the development of three other Turbo languages, the Borland R&D team has zeroed in on a potent arrangement of tools. With 4.0, the place to stand is solid indeed.

Turbo Pascal 4.0's programming environment strongly resembles that of Turbo C. Also like Turbo C, a command-line invocable version of the compiler is shipped with the environment-based compiler. This will make it possible to convert large, batch-file based Pascal projects to Turbo Pascal 4.0 from other command-line based Pascal compilers.

The environment screen is divided into an edit window and an output window, with a pull-down menu bar across the top of the screen, and a function-key prompt bar along the bottom. Dialog boxes appear as necessary to report on the progress of a compile, or to prompt for further input. If the environment senses an EGA-type display adapter, a 43-line screen mode becomes available as a menu option. If a VGA or VGA-compatible is detected, a 50- line text screen may be selected instead. Figure 3 shows the Turbo Pascal 4.0 environment with both windows visible and the **Compile** menu active.

The edit window contains the familiar Borland editor as used in Turbo Basic and Turbo C, which is considerably more advanced than the editor used in Turbo Pascal 3.0. Tabs, for example, may be configured as either Turbo Pascal-style pseudo-tabs that move the cursor to the position beneath the beginning of the next word on the previous line, or true eight-position tabs that insert Ctrl-I characters into the edit file rather than space characters. This same assembly language editor is now available for inclusion in your own programs as part of the Turbo Pascal Editor Toolbox 2.0 (see "Exploring the Borland Binary Editor," elsewhere in this issue).

The output window is where ordinary text output is sent during compiled program (rather than environment) execution. Both the edit and output windows can be zoomed to occupy all screen space between the menu bar at the top of the screen and the prompt bar at the bottom. When sharing the screen with the edit window, the output window can be scrolled up and down to bring all portions of the output screen into view. The environment retains the last specified position of the cursor on the output screen, and the flashing text cursor will scroll faithfully into and out of view along with the other screen information.

The environment retains the names of up to the last eight files edited in a "pick" file, and when the environment is run, the last file edited is loaded into the editor on the assumption that it is the programmer's most likely choice. Failing that, one of the other recently edited files may be chosen from a pick menu, or some entirely new file loaded.

Most compiler switches and options may be set from an options menu. COMMAND.COM may be loaded as a child process, allowing the programmer to "EXEC to DOS" to perform DOS tasks or execute other applications without losing the current Turbo Pascal context.

Although all environment features are available through menus, the PC function keys are used to provide single-keystroke access to important features. F2 saves the current edit file; F3 brings up a prompt to load a new file; F5 zooms the currently active window to the full screen, and so forth. The several menus may be invoked from within the editor by a single keystroke Alt-key command, and the familiar F7 (Begin block) and F8 (End block) commands are present from the editor. By providing both menus and single-keystroke access to compiler features, neither newcomers nor longtime experts will feel shorted by the environment.

PROJECT MANAGEMENT

When a large application is divided into many small source code files, each of which may be separately compiled into a unit, there are suddenly choices as to when and how to compile and link the modules into the final .EXE file. Turbo Pascal 4.0 provides three separate commands that initiate compilation:

Compile compiles the current work file and any of its include files. No separately compiled units are recompiled during the operation.

Make compiles the current work file and include files, plus any separately compiled unit whose source was modified *after* it was last compiled. For example, a programmer might make a change to his application that requires modifying the source code of fifteen of the forty units comprising the application. Invoking **Make** would then compile *only* the main program and the fifteen modified units before linking the main program and all forty .TPU files into the final .EXE file.

Build recompiles *all* units along with the main program, regardless of time stamps.

UPHOLDING THE TURBO TRADITION

Remarkably, earlier versions of Turbo Pascal did little if any active optimization of code. The compiler was fast because it was written in assembly language, and its generated code was fast (compared to that of most other language compilers of its time) because that code was competent and not haywire.

Turbo Pascal 4.0 upholds the Turbo tradition. The newcompiler is still written in assembly language (the command line version is only 48K in size) but assembly language written with four year's experience in code generation techniques behind it. Compilation is a great deal faster than Turbo Pascal 3.0 To give you an idea just how fast, consider Table 2. MicroStar is a WordStar-like word processor consisting of 21,000 lines of Pascal code and distributed with the Turbo Pascal Editor Toolbox 2.0. This toolbox was completely rewritten for release with Turbo Pascal 4.0 and will not be available for Turbo Pascal 3.0. However, a copy of MicroStar 2.0 was modified to compile under Turbo Pascal 3.0 while being as close as possible syntactically to the Turbo Pascal 4.0 version. We recorded its compilation times under various machine environments, as shown in the table. Compilation is from four to six times faster under Turbo Pascal 4.0

Built into the compiler is a "smart" linker that will not link subprograms into an .EXE file if those subprograms are never called. This extends to the runtime library as well, making Turbo Pascal 4.0 .EXE files consistently smaller than those produced by other native code Pascal compilers for equivalent code. For example, the simple "null" program

```
PROGRAM Null;

BEGIN
END.
```

produces an .EXE file only 1052 bytes in size. Because 3.0 includes the entire runtime library in every .COM file it generates, the same program compiled under 3.0 occupies 11K bytes. In a more practical example, a 700-line "whereis" utility that compiles to a 16K file under 3.0 reduces to a 12K file under 4.0.

The generated code has the advantages of several kinds of optimization, including short-circuit Boolean expression evaluation and constant folding.

Short-circuit Boolean expression evaluation. Short-circuit Boolean expression evaluation ceases evaluation of a Boolean expression as soon as the runtime code determines that further evaluation cannot change the ultimate value of the expression. For example, in this statement

```
IF R<>0 AND ((Sagitta/R) > CA/4.0)
  THEN CalculateRatios;
```

an **R** equal to zero will stop evaluation of the expression, since **False** ANDed with any other Boolean value will still yield **False**. Quite apart from speed and space considerations, the sharp-eyed may see an additional value to this sort of evaluation: If **R** takes on a zero value and the expression

```
Sagitta/R
```

is evaluated, a Divide By Zero runtime error will result. Short-circuit Boolean expression evaluation may be turned on and off with either a compiler switch or through a menu option in the environment.

Constant folding. Constant folding avoids creating a separate instance of a named constant in the code each time that constant is used, if doing so will save space. This generally applies only to string constants, since Turbo-style typed and structured constants have always been stored only once and accessed through pointers when used.

Support code for the six-byte software-only reals is consistently about 30 per cent faster than it was under Turbo Pascal 3.0, and other generated code runs between 20 per cent and 35 per cent faster depending on the context.

CONCLUSION

The critical path in any software development scenario is not between source code and object code, but between the programmer's mind and the finished program. A great part of Turbo Pascal's original success lay in getting unnecessary obstacles out of this critical path. The enormous boost in programmer productivity that it provided more than compensated for the substitution of include files for separate compilation and the use of overlays instead of a large code model.

With Turbo Pascal 4.0, these compromises have been removed. It's fair to say that any user application that will function on a 640K DOS machine can be written effectively in 4.0, and that the compiler has joined the other three Turbo languages in the big time of DOS software development. ■

TURBOCHARGING: MOVING FROM 3.0 TO 4.0

Converting from 3.0 to 4.0 will be a little work, but the payoff is considerable.

Bruce Webster

PROGRAMMER

No pain, no gain.

Adding as much to the Turbo Pascal definition as does 4.0 means some retrofitting of older code will be required. Some of the changes involve moving the language definition more closely in line with the ANSI specification for Pascal, but most of them add real value to your code in terms of speed, modularity, or portability. The conversion process is not as painful as you might think, and Borland has provided some tools to help you.

UPGRADING YOUR PROGRAM

For starters, you don't have to do all the work yourself. Turbo Pascal 4.0 includes a program called UPGRADE.EXE. This utility takes your old 3.0 source files, reads them in, and writes out two new files. The first is an updated and commented version of your old file, with minor changes made where possible and comments inserted where potential problems were detected. The second is a "journal" file, which tells what UPGRADE noticed about your program and what steps you might take to make the conversion. If you print out the journal file, then refer to it as you go through the updated source file, you'll find that UPGRADE has probably done most of what you need to in order to convert.

UPGRADE has a second function to make life even easier: unit generation. By placing special directives (in comments) throughout your program, you can instruct UPGRADE to break your program up into units. Chances are you'll have to do a bit of post-editing, but the bulk of the work will be done for you.

BUT WHAT IF I LIKE VERSION 3.0?

If you do, there's still hope. Version 4.0 includes two units (program libraries) designed to help you support programs written for 3.0. These units contain constants, data types, variables, procedures, and functions that were predeclared in 3.0 but no longer exist (or have been redefined) in 4.0. In most cases, they have disappeared or been modified in an effort to make Turbo Pascal more logical and consistent. However, if your programs rely upon them heavily, you may want to continue to use them.

These units are called **Turbo3** and **Graph3**. They're located in a file called TURBO.TPL, which is Turbo Pascal's "resident library" file. If you're going to use them, place the statement

```
USES Crt,Turbo3,Graph3;
```

at the start of your program, following your program header. Let's take a quick look at each.

THE TURBO3 UNIT

The **Turbo3** unit restores some low-level I/O and system items found in 3.0 but not found (or redefined) in 4.0. These include the following:

Kbd. Not found in 4.0; replaced by the **ReadKey** function.

CBreak. Undocumented in 3.0; documented and renamed **CheckBreak** in 4.0.

AssignKbd. Not defined in 3.0, but allows you to set up any text file to act like **Kbd**.

MemAvail, MaxAvail. In 4.0, these are of type **LongInt** and return the quantity of available heap space in *bytes*; the functions in this unit are operationally identical to those in 3.0 that are of type **Integer**, and return the quantity of available heap space in *paragraphs*.

LongFileSize, LongFilePos, LongSeek. These are all of type **Real** and act just like 3.0 functions. Under 4.0 the functions **FileSize**, **FilePos**, and **Seek** all return type **LongInt**.

An important note: the unit **Turbo3** uses the unit **Crt**, so you have to list **Crt** in the **USES** statement before **Turbo3**:

```
USES Crt, Turbo3;
```

THE GRAPH3 UNIT

This unit restores all the basic, advanced, and turtle graphics items not found in the unit **Crt**. These are too lengthy to warrant being listed here; suffice to say that if you use **Graph3**, you have full access to all the constants, types, variables, procedures, and functions described in chapter 19 of the *Turbo Pascal Version 3.0 Reference Manual.*

As with **Turbo3**, **Graph3** uses the unit **Crt**, so you have to list **Crt** in the **USES** statement before **Graph3**:

```
USES Crt, Graph3;
```

OK, NOW WHAT?

Even with UPGRADE.EXE and the two units, you may still need to make changes in your source code. The larger, more complex, or trickier your program, the more likely you'll need some handcrafted changes. The purpose of the following information is to indicate what those changes are, how you might go about making them, and how vital they are to the functioning of your program.

When tasks are listed, they will be flagged as being one of these three types:

- **HELPFUL:** Takes advantage of some feature in 4.0 to make life easier. You don't need to do this; it's up to you.
- **RECOMMENDED:** Really should be done, though you may be able to get by without doing it. Ignore these at your own risk.
- **ESSENTIAL:** Must be done, or your program won't correctly compile and run under 4.0.

PREDEFINED IDENTIFIERS

Version 4.0 doesn't support all the predefined identifiers that 3.0 does. Some have been dropped because they no longer make sense; others have been superseded by new identifiers; still others have been moved into the units found in TURBO.TPL.

- Use **Turbo3** and/or **Graph3** as needed. Ultimately, you will want to convert completely to 4.0 identifiers. (HELPFUL)
- Take advantage of the new routines in the standard units, such as **ReadKey** (which returns a scan code). (HELPFUL)
- Convert completely to 4.0 identifiers. This weans you away from features no longer supported directly in 4.0 and increases your compatibility with revised Turbo Pascal-based products, future versions of Turbo Pascal, and (eventually) other Turbo languages. (RECOMMENDED)
- Run UPGRADE.EXE on your source code. This will make direct replacements wherever possible and give you comments on the rest. (RECOMMENDED)
- Use the appropriate units for certain data types, variables, procedures, and functions that were built into 3.0. For example, the procedures **Intr** and **MsDos** are no longer predeclared; instead, they are found in the **DOS** unit. Similarly, the **Lst** device (text file associated with the printer) is defined in the **Printer** unit. (ESSENTIAL)

DATA TYPES

Version 4.0 introduces a number of new data types and language functions involving data types. Many of these will help you to drop some of the "kludges" that you've had to use in the past.

- Use type casting in place of the **Move** routine to copy the contents of one variable into the space of another variable of an incompatible type. For example, use

  ```
  RealVar := Real(BuffPtr^);
  ```

  instead of:

  ```
  Move(BufferPtr^,
       RealVar,SizeOf(RealVar));
  ```

  With 4.0's new extended type casting, you can handle most of these transfers, so long as the destination is the exact same size as the source. (HELPFUL)
- Convert to new data types where appropriate and practical. These include **LongInt** and **Word** (to replace **Integer**);

Pointer (as a generic pointer type); and **String** (with an assumed maximum length of 255 characters). (RECOMMENDED)

- Be aware that hexadecimal (base-16) constants are considered to be of type **Word** (rather than of type **Integer**), so that the hex constant **$FFFF** represents 65535 instead of –1. You should consider converting any variables that are assigned hex constants to type **Word**. (RECOMMENDED)
- Likewise, be aware that 4.0 now allows you to assign –32768 to a variable of type **Integer**. Previously, the only way you could do that was by assigning it the hex constant $8000. However, that hex constant now represents the value 32768 (being of type **Word**); assigning it to an **Integer** variable will cause a compiler error. (RECOMMENDED)
- Use string library routines (such as **Length** and **Copy**) instead of directly accessing the internal string structure, such as **Ord(SVar[0])**, or absolute-addressed byte variables, on top of strings. This protects you against any future changes in how strings are implemented. (RECOMMENDED)
- Be aware that type checking in 4.0 is more strict on strings, characters, and arrays of characters. The assignment

```
CharVar := StringVar
```

is no longer acceptable, even if **StringVar** is declared as **String[1]**. The assignment **StringVar := ArrayVar** is still acceptable, but **ArrayVar := StringVar** is not. (ESSENTIAL)

- Version 4.0 enforces type checking more strictly on derived types, which means that variables must have identically named types or be declared together in order to be assignment-compatible. For example, given

```
VAR
  A      : ^Integer;
  B      : ^Integer;
```

then **A** and **B** are not assignment compatible (that is, the statement **A := B** will cause a compiler error), since they are separately derived types. In order to be assignment-compatible, they need to be declared together

```
VAR
  A,B    : ^Integer;
```

or they need to be of the same named data type:

```
TYPE
  IntPtr   = ^Integer;

VAR
  A        : IntPtr;
  B        : IntPtr;
```

Either of these solutions will work just fine; the second one is more general (allowing other variables, parameters, and functions to be of the same data type), and so is preferred. (ESSENTIAL)

- The **BCD** data type (and its associated **Form** routine) are not supported in 4.0. Consider using the **LongInt** data type; if you have a math coprocessor, then use the {**$N+**} directive and use the IEEE type **Comp** (8-byte integer). (ESSENTIAL)

LANGUAGE FEATURES

Version 4.0 introduces some restrictions and some enhancements. The restrictions are designed to help it conform to the ANSI standard definition of Pascal, while the enhancements are there to make your life as a programmer easier.

- Version 4.0 assumes *short-circuit Boolean evaluation*. This means that evaluation of Boolean expressions is halted as soon as possible. For example, consider the expression

```
IF expr1 AND expr2 ...
```

If **expr1** is **False**, then the entire expression will be **False**, regardless of the value of **expr2**. If short-circuit evaluation is enabled, then if **expr1** is **False**, **expr2** won't be evaluated. For example, if **expr2** contains a function call, then that function won't be called if **expr1** is **False**. You can enable standard (non-short-circuit) Boolean evaluation with the {**$B+**} compiler option. Be aware of the implications of enabling short-circuit evaluation. (HELPFUL)

- In line with the ANSI standard, 4.0 allows you to use only global and local variables as **FOR** loop control variables. For example, if the statement

```
FOR Indx := Start TO Finish ...
```

appears in a procedure (or function), then **Indx** must be declared either globally or within that procedure. **Indx** cannot be a formal parameter of that procedure, nor can it be declared within an enclosing procedure. (NECESSARY)

INPUT AND OUTPUT

Turbo Pascal 4.0 has made some significant changes in I/O handling, many of which are intended to increase ANSI compatibility. Most of these have to do with text files.

- **Read(IntVar)** now waits for an integer value to be entered; pressing Enter will no longer cause the program to continue, leaving **IntVar** unchanged. Revise your program appropriately. (RECOMMENDED)
- If you are reading and writing real number values with data files, be aware of the differences between the standard type **Real** (six bytes, compatible with 3.0) and the IEEE floating point types supported by the {**$N+**} directive (**Single**, **Double**, **Extended**, and **Comp**). Use the latter types only if you are sure that your program and any resulting data files will be used only on systems equipped with a math coprocessor. (RECOMMENDED)
- You can no longer directly declare variable-length buffers for text files in the format:

```
var F : text[length]
```

Instead, you use the predefined procedure **SetTextBuf**.

PROGRAM AND MEMORY ORGANIZATION

One significant change in 4.0 is the introduction of units. A unit is a collection of declarations and subroutines that can be compiled separately. A program can consist of many units and a main program; each unit, as well as the main program, can occupy a full 64K code segment, giving you essentially unlimited code size for your program. Units give you four important capabilities:

1. They allow you to create libraries (such as those in TURBO.TPL) that you can use in many different programs.
2. They allow you to break up a large program into manageable chunks by collecting related declarations and subprograms together.
3. They allow you to "hide" declarations and subprograms that you don't need (or want) to be "visible" to the rest of the program.
4. They allow you to break the 64K code barrier, since each unit can contain up to 64K of code.

As a consequence, significant changes have been made in memory organization as well. The user's guide explains more of the details; here are some of the tasks you need to consider:

- Convert your libraries from include files to units. This is by no means necessary, but it has several advantages. For one, you don't have to recompile the routines in the unit each time; for another, you can distribute your library routines without distributing source code. (HELPFUL)
- 4.0 has a new compiler directive, **{$M}**, allowing you to set the stack and heap sizes within a program. The format is

```
{$M stack size,heap min,heap max}
```

where all three values are in bytes. The default values are:

```
{$M 8192,8192,655360}
```

(HELPFUL)

- Convert large programs from overlays to units. You have to do this, since 4.0 no longer supports overlays. If you have been using overlays to get around the 64K code limit, then you won't have to worry any more: the main program and each unit can be up to 64K in size. If you've been using overlays because all your code couldn't fit into memory at once anyway, then you'll have to do some rewriting, since the main program and all units have to fit into memory at the same time. (ESSENTIAL)
- As mentioned earlier, be aware that **MemAvail** and **MaxAvail** are now of type **LongInt** and return their values in bytes instead of paragraphs. Either use **Turbo3** (which supplies the original 3.0-compatible versions of **MemAvail** and **MaxAvail**), or make the appropriate changes throughout your program. (ESSENTIAL)

COMPILER DIRECTIVES AND ERROR CHECKING

Version 4.0 heavily redefines the compiler directives and error codes. UPGRADE.EXE may help modify the compiler directives, but you are going to have to be sure you've caught all of them and that you've also changed over to the new error codes. Keep in mind:

- If an existing program doesn't work correctly, try turning off short-circuit Boolean evaluation with the **{$B+}** directive; the default is **{$B–}**. (HELPFUL)
- Range checking is now enabled by default; if you want it disabled, place a **{$R–}** directive at the start of your program. If you're unsure, leave enabled. If your program is halting with range-checking errors, either leave it enabled and figure out the problems, or disable it. (RECOMMENDED)
- Review *all* use of error codes, especially when the check is more than simply zero vs. nonzero. Define all error codes as constants in a global location so as to more easily deal with future changes. (ESSENTIAL)
- Review *all* compiler directives. Some are new, many have been redefined, others have been dropped altogether. (ESSENTIAL)
- **ErrorPtr** is gone; you should now use **ExitProc**. User-written error handlers must be modified. (ESSENTIAL)

ASSEMBLY LANGUAGE USAGE

Some significant changes have also been made on assembly language usage. Most notably, there is a new usage of the **INLINE** keyword which defines an inline macro rather than a separate, callable routine. This is in addition to the familiar 3.0 usage.

- For short assembly language code, consider using the **INLINE** directive (which differs from the **INLINE** statement). This generates actual inline macros in the resulting object code. (HELPFUL)
- Convert from **INLINE** to external subroutines where appropriate and practical; use **INLINE** only when necessary. (RECOMMENDED)
- The **INLINE** statement no longer allows references to the location counter (*), nor does it allow references to subprogram identifiers. (ESSENTIAL)
- External subroutines must be reassembled and incorporated in .OBJ format; if in doubt, use the Microsoft Macro Assembler (MASM) to do the reassembly. (ESSENTIAL)
- Typed constants now reside in the data segment (relative to DS) rather than the single 3.0 code segment (relative to CS). Therefore, they must be accessed differently by any external subroutines. (ESSENTIAL)

SUMMARY

This list is not exhaustive in either sense of the word. Many of your programs will run with little or no modification; others will work fine with just the processing UPGRADE.EXE performs. Likewise, this list doesn't cover all possible compatibility problems, since many Turbo Pascal programmers take advantage of undocumented or unsupported features of 3.0. Be sure to check the READ.ME file on your Turbo Pascal 4.0 distribution disk for any additional conversion.

Converting to 4.0 will probably be easier than you think. But even if it's a hassle, it'll be worth it in both the short and the long run. Version 4.0 compiles quicker, links smarter, produces smaller and faster code, and offers greater flexibility and power. At this point, you've probably got a good idea of what you need to do. Go to it! ■

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

GETTING TO KNOW UNITS

Separate compilation by way of units saves compile time and helps large projects come together smoothly.

Tom Swan

SQUARE ONE

One of Turbo Pascal 4.0's most useful new features is the *unit*, a collection of constants, types, variables, procedures, and functions compiled and stored in special files for programs to share. In this introduction, I explain what units are and show you how to use the precompiled units supplied on your Turbo Pascal diskettes. I also list a complete example to demonstrate how to create your own units.

WHAT IS A UNIT?

If a program is like a factory, a unit is like a warehouse, stocked with goodies ready for programs to put into production. Inside units are various raw materials—constants, types, variables, procedures, and functions—precompiled and stored in special .TPU (Turbo Pascal Unit) files.

To add a unit's features to programs, you tell the compiler to *use* that unit. Turbo Pascal then reads the .TPU file containing the unit and attaches all of the unit's declarations to the program, just as though the program had declared those same items itself.

Included on the Turbo Pascal master diskettes are eight precompiled units, ready for use. These units have two main purposes: they hold Turbo Pascal's standard library of routines, and they help smooth the transition from older Turbo Pascal compilers to the present 4.0 version.

You can also write your own custom units containing tested and debugged routines and data structures. This helps you to organize large programs into modules. For example, you might insert all of a program's disk I/O routines in one unit, put all the keyboard-input routines in another unit, and so on. Modular programs like these are easier to write, debug, and modify.

Units also help reduce compiling times. Turbo Pascal can read a unit's binary image from a .TPU file more quickly than it could recompile all the unit's routines from scratch. By compiling a program's many modules apart from one another, a technique known as *separate compilation*, you concentrate on individual sections without having to recompile the entire program for each modification you make. Separate compilation is especially important to full-time programmers who frequently work on programs containing tens of thousands of statements. With separately compiled units, Turbo Pascal 4.0 is well-suited to professional software development.

HOW TO USE UNITS

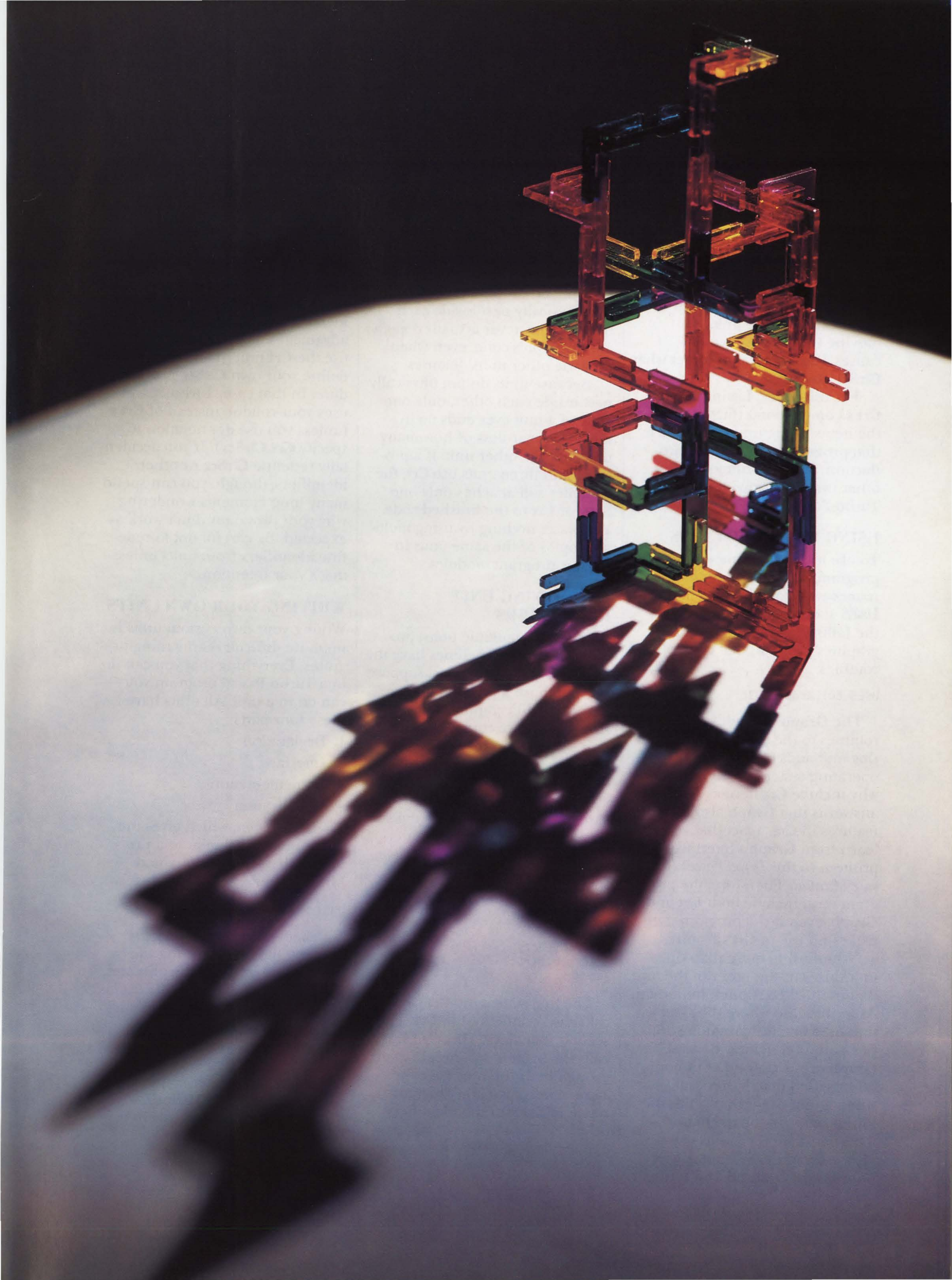
A simple example explains how to use units. Our goal is to write a program to center a line of text on display—nothing to shake the earth, but suitable for the demonstration. We need procedures to clear the screen, position the cursor, and display text. Here is the complete program:

```
PROGRAM MyTest;
USES CRT;
BEGIN
  ClrScr;
  GotoXY(25,12);
  Write
    ('Introducing Turbo Pascal Units')
END.
```

Following the **PROGRAM** declaration, a **USES** statement tells Turbo Pascal to use the **Crt** unit. If you have Turbo Pascal 4.0, type in this example and save as MYTEST.PAS. (Consult your reference manual and refer to the introductory article on Turbo Pascal 4.0 elsewhere in this issue for more information on typing programs.) Press Alt-R to compile and run. The first statement (the line after **BEGIN**) clears the screen by calling procedure **ClrScr**. The second statement positions the cursor at the location inside **GotoXY**'s parentheses, centering the string for **Write**.

For an experiment, remove the **USES** statement and type Alt-R to run. What happens? If you are following along, you receive the following error when

Bradley Ream



the compiler reaches **ClrScr**:

```
Error 3: Unknown Identifier
```

This error occurs because Turbo Pascal no longer knows the meaning of **ClrScr**. (You would experience a similar problem with **GotoXY** if the compiler got that far.) Because the **Crt** unit defines the meaning of **ClrScr** and **GotoXY**—plus several other procedures and functions—you must use **Crt** in programs that call the routines inside the unit. Removing the **USES** statement causes Turbo Pascal to forget what **ClrScr** means.

For reference, Listing 1 lists **Crt**'s complete interface—all of the items in the unit's warehouse that programs can put into production. Similar printouts for other units are provided with Turbo Pascal 4.0.

USING MULTIPLE UNITS

To use more than one unit in a program, separate multiple unit names with commas in a single **USES** statement. For example, the following statement starts a graphics program that calls DOS routines:

```
USES CRT,Graph,DOS;
```

The **Graph** unit adds graphics routines to the program while the **Dos** unit adds the ability to call operating system functions. But why include **Crt** here as well? The answer is that **Graph** also uses the features in **Crt**, a fact that you learn from **Graph**'s interface printout in the *Turbo Pascal Reference Manual*. Therefore, the program must include both **Crt** and **Graph**, even if the program itself makes no use of **Crt**'s features.

When using more than one unit this way, the order of the unit names in **USES** is often important. As in this case where the **Graph** unit uses **Crt**, the program specifies both unit names. Because **Graph** uses **Crt**, the **Crt** unit must be specified first. In general, when a unit uses other units, the lowest units on the totem pole—the ones that other units use—always come first. If unit **A** uses units **B** and **C**, and if unit **B** uses **C**, then this is the correct declaration order in **USES**:

```
USES C,B,A;
```

You might find this rule easier to memorize if you pretend that units nest inside each other. In this example, units **C** and **B** nest inside **A**. Unit **C** nests inside **B**. No units nest inside **C**. In the **USES** statement, units with deeper nesting levels come first. Those units on outer nesting levels (those that use other units) come last. Remember, though, that units do not physically nest inside each other. Units never actually *contain* another unit's code, even when they use other units' features.

Because units do not physically nest inside each other, only one copy of a unit ever ends up in memory regardless of how many units use the other unit. If a program and three units use **Crt**, the compiler still attaches only one copy of **Crt** to the finished code. You waste nothing by using multiple copies of the same units in various program modules.

REDEFINING UNIT IDENTIFIERS

When using multiple units, two identifiers will sometimes have the same name. For example, suppose you purchase a precompiled unit, perhaps a 3-D graphics module, from a software company. Inside unit **ThreeD** is a procedure, **ClrScr**—exactly the same name as the routine in **Crt**.

Although this is a hypothetical case, there is an easy fix. To distinguish two identical identifiers in different units, use *dot notation* as follows:

```
PROGRAM DotsTheWay;
USES CRT,ThreeD;
BEGIN
  CRT.ClrScr;
  ThreeD.ClrScr
END.
```

The first statement, **Crt.ClrScr**, calls the routine in unit **Crt**. The second statement, **ThreeD.ClrScr**, calls the routine in **ThreeD**. Even though both routines have the same name, attaching the unit name in front of the identifiers allows Turbo Pascal to resolve the ambiguity.

If you did not use dot notation as in this example, then which **ClrScr** routine would Turbo Pascal use? The answer is: the one most recently declared. In the example, a lone **ClrScr** would refer to the routine in **ThreeD** because that unit is declared after **Crt**. This is an important rule to keep in mind. If you redefine an identifier from a unit, then the most recent declaration takes precedence.

This ability to redefine unit identifiers has advantages and disadvantages. For example, you can use the **Crt** unit in a program, but define your own **ClrScr** procedure. In that case, Turbo Pascal uses your routine instead of **Crt**'s (unless you use dot notation to specify **Crt.ClrScr**). If you accidentally redefine **ClrScr** or other identifiers, though, you can spend many long moments wondering why your programs don't work as expected. Be careful not to redefine identifiers from units unless that's your intention.

WRITING YOUR OWN UNITS

Writing your own custom units is no more difficult than writing programs. Everything that you can do in a Turbo Pascal program you can do in a unit. All units have these four parts:

1. Declaration
2. Interface
3. Implementation
4. Initialization

The *Declaration*, which gives the unit a name, is similar to a program's header. A unit named **MyStuff** would begin like this:

```
UNIT MyStuff;
```

Usually, you should save the unit text file on disk using the same name for both the unit and file—MYSTUFF.PAS in this example. When Turbo Pascal compiles the unit, it creates a binary file named MYSTUFF.TPU containing the unit's compiled symbols and code. When you later compile a program that **USES MyStuff**, Turbo Pascal automatically looks for MYSTUFF.TPU on disk.

If you use a different file name for the unit, you must tell the compiler about the change. For example, if you store the **MyStuff** unit as STUFF.PAS and compile to STUFF.TPU, you would write the following statement to use the unit in a program:

```
PROGRAM SomeStuff;
USES {$U STUFF.TPU} MyStuff;
```

This tells the compiler that the **MyStuff** unit is actually in the file, STUFF.TPU. The *compiler directive*, **{$U STUFF.TPU}**, appears inside the **USES** statement just ahead of the unit name.

The second part of a unit is the *interface*. In this section, which begins with the keyword **INTERFACE**, are the declarations that you want programs to understand. There are never any programming statements in the unit's interface. There are only declarations of constants, data types, variables, procedures and functions. Returning to the earlier warehouse analogy, the unit's interface section is similar to an inventory sheet that tells you (and the compiler) what items the unit contains. The interface is the unit's documentation—the part that describes the items on the unit's warehouse shelves.

Suppose you decide to add a constant, a variable, and a procedure to a unit. The interface section then looks like this:

```
INTERFACE
CONST MyNumber 9;
VAR   MyVariable : Integer;
PROCEDURE
  MyProcedure(VAR X : Integer);
```

MyProcedure has no programming statements attached—only a declaration with the procedure's name and parameters in parentheses. (If a procedure or function has no parameters, then it simply ends with a semicolon after its name.)

Remember, the interface merely describes the items in the unit. The interface never contains any executable programming statements.

Following the interface section comes the juicy part—the *implementation*. As its name suggests, the implementation *implements* the items declared in the interface. Into the implementation go the programming statements that make the unit routines strut their stuff. Back at the warehouse, the implementation contains the stock—the actual items described in the interface's inventory. When Turbo Pascal compiles a program that uses a unit, it attaches the unit's implementation to the finished result.

One important rule to remember when writing your own units is that every procedure and function declared in the interface must have a completed routine in the implementation. Continuing our example, if **MyProcedure** adds constant **MyNumber** to parameter **x**, the complete implementation would be:

```
IMPLEMENTATION
PROCEDURE MyProcedure;
BEGIN
  X := X + MyVariable
END;
```

In the interface, **MyProcedure** includes a parameter list. In the implementation section the parameter list is missing. This is not a mistake. You can declare procedure and function parameters only in the unit's interface section. Those same parameters never repeat in the implementation. For reference, some programmers repeat the parameter list in the implementation, turning it into a comment with (* and *) or { and }. If you see this in a unit listing, don't be confused. It's just a trick to make life with units a little easier. For example, the implementation for **MyProcedure** could begin:

```
PROCEDURE
  MyProcedure{(VAR X : Integer)};
```

Of course, real units usually have many procedures and functions in their repertoire, not just one routine as in this example. In addition, a unit can have other supporting procedures and functions for its own use. Such routines also appear in the implementation part, but are not available to programs that use the unit because the routines have no corresponding declarations in the interface and, therefore, are completely hidden except to the unit itself.

The final part of a unit is the *initialization*, which resembles a Pascal program's main body. (Remember that units are *not* complete Pascal programs!) Suppose **MyUnit** must initialize **MyVariable** to the value of **MyConstant**. The initialization section would then have this design:

```
BEGIN
  MyVariable := MyConstant
END.
```

The statements in a unit's initialization run just prior to the first statement in the program that uses the unit. This allows the unit to perform start-up chores—such as initializing a variable as in this example—before the program begins. For units with no such chores to perform, just type **BEGIN** and **END** with no statements in between.

AN EXAMPLE UNIT

As a useful example, and to demonstrate a few other details, Listing 2, **Box**, contains a procedure and several variables that you can use to draw boxes on screen. Type the listing and save as BOX.PAS. Compile to disk, creating the file BOX.TPU. Notice that **Box**'s interface section begins with a **USES** statement, telling Turbo Pascal that **Box** uses **Crt**'s features. Because **Box** uses **Crt**, so must your program, which might begin this way:

```
PROGRAM MyBoxes;
USES CRT,Box;
```

Following **Box**'s interface section, eight **CHAR** variables tell the unit what characters to use to display boxes. **TopLine** through **RightLine** are for vertical and horizontal lines. **TopLeftCorner** through **BottomRightCorner** are for drawing box corners.

The last item in **Box**'s interface is a procedure, **DrawBox**, which has four integer parameters. To draw a box, call **DrawBox** with the coordinate values representing

LISTING 1: CRT.HDR

```
UNIT Crt;

INTERFACE

CONST

{ CRT modes }

  BW40 = 0;              { 40x25 B/W on Color Adapter }
  C40  = 1;              { 40x25 Color on Color Adapter }
  BW80 = 2;              { 80x25 B/W on Color Adapter }
  C80  = 3;              { 80x25 Color on Color Adapter }
  Mono = 7;              { 80x25 B/W on Monochrome Adapter }
  Last = -1;             { Last active text mode }

{ Foreground and background color constants }

  Black        = 0;
  Blue         = 1;
  Green        = 2;
  Cyan         = 3;
  Red          = 4;
  Magenta      = 5;
  Brown        = 6;
  LightGray    = 7;

{ Foreground color constants }

  DarkGray     = 8;
  LightBlue    = 9;
  LightGreen   = 10;
  LightCyan    = 11;
  LightRed     = 12;
  LightMagenta = 13;
  Yellow       = 14;
  White        = 15;

{ Add-in for blinking }

  Blink        = 128;

VAR

{ Interface variables }

  CheckBreak: Boolean;     { Enable Ctrl-Break }
  CheckEOF: Boolean;       { Enable Ctrl-Z }
  DirectVideo: Boolean;    { Enable direct video addressing }
  CheckSnow: Boolean;      { Enable snow filtering }
  TextAttr: Byte;          { Current text attribute }
  WindMin: Word;           { Window upper left coordinates }
  WindMax: Word;           { Window lower right coordinates }

{ Interface PROCEDUREs }

PROCEDURE AssignCrt(var F: Text);
FUNCTION KeyPressed: Boolean;
FUNCTION ReadKey: Char;
PROCEDURE TextMode(Mode: Integer);
PROCEDURE Window(X1,Y1,X2,Y2: Integer);
PROCEDURE GotoXY(X,Y: Integer);
FUNCTION WhereX: Integer;
FUNCTION WhereY: Integer;
PROCEDURE ClrScr;
PROCEDURE ClrEol;
PROCEDURE InsLine;
PROCEDURE DelLine;
PROCEDURE TextColor(Color: Integer);
PROCEDURE TextBackground(Color: Integer);
PROCEDURE LowVideo;
PROCEDURE HighVideo;
PROCEDURE NormVideo;
PROCEDURE Delay(MS: Integer);
PROCEDURE Sound(Hz: Integer);
PROCEDURE NoSound;
```

the box's top-left and bottom-right corners. For example, this outlines the entire screen:

```
DrawBox(1,1,25,80);
```

Next comes the unit's implementation, the part that fleshes out the interface's declarations. The eight constants here are the default values that the unit uses for **Box**'s global variables. Because these declarations appear in the unit's implementation part, they are invisible except from inside the unit. The unit itself can make use of the eight constants, but a program that uses the unit cannot do the same. Programs (and any other units that use **Box**) see only the items in the unit's interface.

Although **Box** declares only constants in its implementation, it could also declare data types and variables here as well. Those declarations would also be hidden from programs that use **Box**.

Some programmers call the declarations in the implementation *private*, because the items in this section are strictly for use by routines inside the unit. The declarations in the interface are *public*, visible to both the outside world and to the statements inside the unit. It's good programming practice to carefully restrict a unit's public parts to those items that programs and other units need to know.

Following the eight constants is the completed **DrawBox** procedure. For reference, parameters are converted into a comment. Remember, parameters to public routines can appear only in the interface section, never in the implementation section.

Inside **DrawBox** are three main sections comprising the actual programming that runs when you call this procedure. First, eight statements draw the box's corners. Next, two **FOR** loops fill in the top and bottom sides. Another **FOR** loop fills in the left and right sides, completing the box.

The unit's initialization comes last, between **BEGIN** and **END**. In this example, the unit's body initializes the unit's global variables, assigning the eight constants to the variables declared in the interface.

Initializing global variables in the unit's body is a typical job for a unit initialization to perform. Although not shown here, the unit body could also call procedures and perform other actions as part of its startup sequence.

USING THE BOX UNIT

As an example of using the **Box** unit, Listing 3, **TestBox**, draws two boxes on screen. The program begins with a **USES** statement, specifying both **Crt** and **Box**. **Crt** must precede **Box**. Do you remember why?

The program's body clears the screen before calling **DrawBox**. After that, three statements assign new values to a few of **Box**'s global variables. Notice that these variables—**BottomLeftCorner**, **BottomRightCorner**, and **BottomLine**—are not declared anywhere in **TestBox**. The variables appear in **Box**'s interface (see Listing 2). Because **TestBox USES Box**, it can use anything in the unit's interface as though the program had declared those same items itself.

As explained earlier, items declared only in the unit's implementation are private and are not visible to programs. As a demonstration of this rule, add the following line after **TestBox**'s **BEGIN**:

```
TopLine := DefTop;
```

Attempting to assign to **TopLine** the default value for box-top lines (**DefTop**) fails because **DefTop** is not visible to **TestBox**. The **DefTop** constant is declared in the **Box** unit's implementation and, therefore, only the statements inside the unit can use this constant's value.

LISTING 2: BOX.PAS

```
UNIT Box;

(*
 *      PURPOSE : A Unit for displaying boxes
 *      SYSTEM  : IBM PC/MS-DOS, Turbo Pascal 4.0
 *      AUTHOR  : Tom Swan
 *)


INTERFACE

USES    Crt;

VAR     TopLine:                  CHAR;   { Horiz. top line char }
        LeftLine:                 CHAR;   { Vert. left line char }
        BottomLine:               CHAR;   { Horiz. bottom line char }
        RightLine:                CHAR;   { Vert. right line char }

        TopLeftCorner:            CHAR;   { Top left corner char }
        TopRightCorner:           CHAR;   { Top right corner char }
        BottomLeftCorner:         CHAR;   { Bottom left corner char }
        BottomRightCorner:        CHAR;   { Bottom right corner char }


PROCEDURE DrawBox( Top, Left, Bottom, Right : INTEGER );

{ Call this procedure to draw a box, using the global character
  variables.  The four parameters are not checked.  Using values
  outside the legal display coordinates will produce strange-
  looking boxes! }


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

IMPLEMENTATION


CONST   DefTop=                   #205;
        DefLeft=                  #179;
        DefBottom=                #196;
        DefRight=                 #179;

        DefTopLeft=               #213;
        DefTopRight=              #184;
        DefBottomLeft=            #192;
        DefBottomRight=           #217;



PROCEDURE DrawBox(* ( Top, Left, Bottom, Right : INTEGER ) *);

{ Draw rectangular outline pegged to these coordinate values. }

VAR     x, y : INTEGER;    { Temporary screen coordinates }

BEGIN

        { First, draw the box's four corners }

        GotoXY( Left, Top );             { Top, Left }
         Write( TopLeftCorner );
        GotoXY( Right, Top );            { Top, Right }
         Write( TopRightCorner );
        GotoXY( Left, Bottom );          { Bottom, Left }
         Write( BottomLeftCorner );
        GotoXY( Right, Bottom );         { Bottom, Right }
         Write( BottomRightCorner );
        { Next, fill in the top and bottom sides }

        GotoXY( Left + 1, Top );
        FOR x := ( Left + 1 ) TO ( Right - 1 ) DO
                Write( TopLine );

        GotoXY( Left + 1, Bottom );
        FOR x := ( Left + 1 ) TO ( Right - 1 ) DO
                Write( BottomLine );


        { And then fill in the left and right sides }

        FOR y := ( Top + 1 ) TO ( Bottom - 1 ) DO BEGIN
                GotoXY( Left, y );
                Write( LeftLine );
                GotoXY( Right, y );
                Write( RightLine )
        END { for }

END; { DrawBox }


BEGIN   { Unit initialization }

        TopLine := DefTop;              { Assign default character }
        LeftLine := DefLeft;            {  values to global vars.  }
        BottomLine := DefBottom;
        RightLine := DefRight;

        TopLeftCorner := DefTopLeft;
        TopRightCorner := DefTopRight;
        BottomLeftCorner := DefBottomLeft;
        BottomRightCorner := DefBottomRight

END.    { Box unit }
```

LISTING 3: TESTBOX.PAS

```
PROGRAM TestBox;

USES    Crt, Box;

BEGIN

        { Draw a test box }

        ClrScr;
        DrawBox( 5, 8, 15, 64 );        { Use default values }

        { Draw a second box, this time with a double-line
          border on the top and bottom. }

        BottomLeftCorner  := #212;      { Change unit's variables }
        BottomRightCorner := #190;
        BottomLine        := #205;
        DrawBox( 3, 30, 20, 58 );       { Draw the box }

        GotoXY( 1, 24 ) { Position cursor before program ends }

END.
```

UNITS AND THE TURBO PASCAL PROGRAMMER

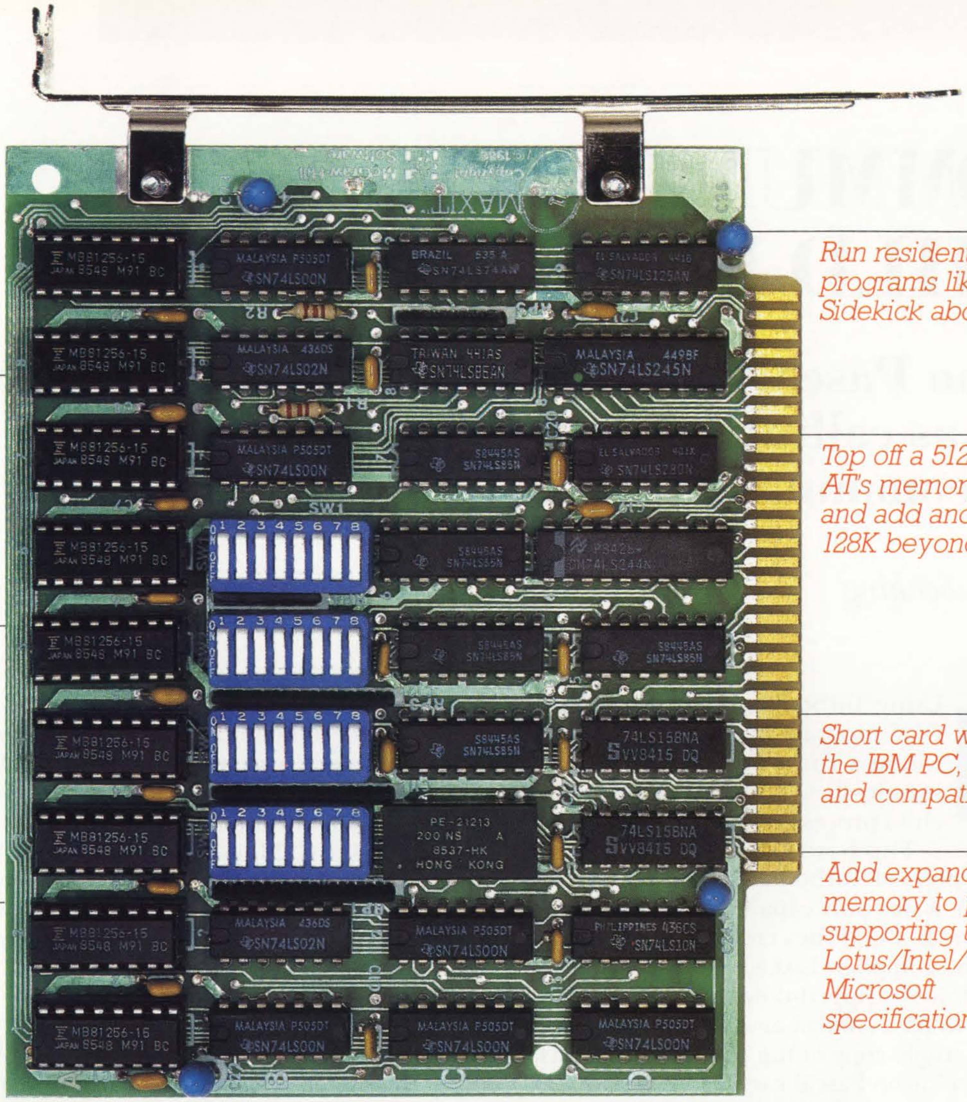
This introduction to units describes what units are and shows how to use units in Turbo Pascal programs. There is more to the unit story than I have room to cover here—and more information about units in your reference manual—but I've tried to touch on the major topics to give you a running start to using and writing units for your own programs.

Units can help you to write modular programs that are easier to design, code, debug and (over the long term) maintain. Because units are separately compileable, you are no longer required to compile all of a large application any time you need to compile a part of it. By allowing the program module interface to remain visible while hiding implementation details, units facilitate team programming and the distribution of proprietary subprogram libraries without revealing the source code. The precompiled units supplied with Turbo Pascal 4.0 provide excellent examples of units in action—be sure to examine them and experiment with them.

Units are like warehouses, stocked to the ceiling with goodies for programs to share. As programming tools, units promote modularity and give you the ability to separately compile large programs in pieces. Units are a welcome and an important new addition to our old friend, Turbo Pascal. ■

Tom Swan is the author of Mastering Turbo Pascal, Mastering Turbo Pascal Files *(Howard W. Sams), and* Programming with Macintosh Turbo Pascal *(John Wiley & Sons). Swan is currently burning the silicon candle at both ends, revising* Mastering Turbo Pascal *4.0 for release in 1988 while writing a new book about Turbo C.*

Listings may be downloaded from CompuServe as UNITS.ARC.



Add up to 96K above 640K to all programs, including PARADOX and 1-2-3.

Parity-checked 256K with a one-year warranty.

Run custom software or the 3270 PC Control Program above 640K.

Run resident programs like Sidekick above 640K.

Top off a 512K IBM AT's memory to 640K and add another 128K beyond that.

Short card works in the IBM PC, XT, AT, and compatibles.

Add expanded memory to programs supporting the Lotus/Intel/ Microsoft specification 3.2.

MAXIT™ memory works above 640K for only $195.

Break through the 640K barrier.

MAXIT increases your PC's available memory beyond 640K. And it does it for only $195.

MAXIT includes a 256K half-size memory card that works above 640K. MAXIT will:

- Add up to 96K above 640K to all programs.
- Run memory resident programs above 640K.
- Top off an IBM AT's memory from 512K to 640K.
- Expand 1-2-3 Release 1A or 2.0 worksheet memory by up to 256K.
- Add expanded memory above 640K to programs like Symphony 1.1.

Big gain — no pain.

Extend the productive life of your, IBM PC, XT, AT or compatible. Build more complex spreadsheets and databases without upgrading your present software.

Installation is a snap.

MAXIT works automatically. You don't have to learn a single new command.

If you have questions, our customer support people will answer them, fast. MAXIT is backed by a one-year warranty and a 30-day money-back guarantee.

Order toll free 1-800-227-0900.

MAXIT is just $195 plus $4 shipping, and applicable state sales tax. Buy MAXIT today and solve your PC's memory crisis. Call Toll free 1-800-227-0900 (In California 800-772-2531).
We accept VISA, MC, AE, and DC.

Mc Graw Hill

Osborne McGraw-Hill
2600 Tenth Street, Berkeley, California 94710

MAXIT is a trademark of McGraw-Hill CCIG software. IBM is a registered trademark of International Business Machines Corporation; 1-2-3 and Symphony are registered trademarks of Lotus Development Corporation; Sidekick is a registered trademark of Borland International, Inc; PARADOX is a trademark of ANSA Software.

COMMUNICATING WITH CHILD PROCESSES

Turbo Pascal 4.0 makes spawning a child process as easy as calling a library function. Here's how to facilitate communication with your offspring.

Neil Rubenking

WIZARD

Using Turbo Pascal 4.0, you can run *any* DOS program file as a child process from within a Turbo Pascal program and return seamlessly to your program when the child process ends.

This hasn't always been the case. Previous versions of Turbo Pascal had an **Execute** command, but it was only capable of executing other Turbo Pascal .COM files created with the same version of the compiler. Likewise, the **Chain** feature, which allowed stripped-down Turbo Pascal code modules to pass control among themselves while calling a single copy of the runtime code, was highly specific to Turbo Pascal itself.

The correct way to execute child processes is to use the DOS **EXEC** function call. This is difficult to set up, but an assembly language procedure called **Exec** in the standard DOS.TPU unit does all the work. It's easy. The syntax is:

```
Exec('PROGRAM','COMMAND LINE');
```

What more could you ask for?

Well, the **Chain** and **Execute** commands are no longer with us. Using **Chain** you were able to pass global variables from your main program to the chained program. The new-style **Exec** routine only allows you to pass as much data as you can fit on a command line. If you *need* to communicate between your programs, will you have to stay back with Turbo Pascal 3.0?

Fortunately, no. This article demonstrates two different methods for a parent program to communicate with its child processes.

THE SHARED DATA AREA METHOD

To use this method, you must collect all the global variables that the programs will share and create a record type to hold them. You need to be sure this type declaration is *exactly* the same in both programs, so make it a separate file and **$INCLUDE** it in both.

Declare a variable of the shared record type in the parent, and a pointer to that type in the child. You give the child process access to this shared area by simply passing the *address* of the shared area on the command line. The child process uses the **Ptr** function to point its pointer at the shared area. Now the child process can read *or* write data to the shared area.

THE COPIED DATA AREA METHOD

If you really want to pass a *lot* of data, it may be impractical or inelegant to put it all into one record. In this case, you let the parent and child each have its own data. You simply copy from parent to child when the child begins execution, and copy the data back to the parent when it ends.

Turbo Pascal's **Move** command copies bytes from one address to another. Once you know the address of the data area in the parent, you're ready to **Move**. Again, you'll need to be sure the set of variables you're passing is identical in both cases, so write it to a separate file and **$INCLUDE** it in both.

In this case, the communication is not as complete. The two data areas are separate, and you must copy one to the other whenever you make a change. However, this method is better suited to passing a large number of variables.

DEALING WITH ERRORS

The parent program needs to know how well its children have worked. When the child process terminates, it should tell the parent what's happened. The word variable **status** carries this information between child and parent. A value of 1 in **status** signals successful completion. If the child process crashes, a

LISTING 1: PARENT.PAS

```
PROGRAM Parent;

uses dos;

{$M 8192,8192,8192}
{$I LISTOF.VAR}
{$I PASSDATA.TYP}
{$I HEX.INC}

VAR
  P     : passdata;
  tempS : string[6];
  tempO : string[6];

BEGIN
  WriteLn('THIS is the PARENT program.');
  WriteLn('==========================');
  P.password := 'BORLANDint';
  P.status := 0;
  Write('What string shall I pass to the children?:');
  ReadLn(P.name);
  thing := P.name;
  Str(seg(P),tempS);
  Str(ofs(P),tempO);
  Exec('CHILDA.EXE',tempS+' '+tempO);
  WriteLn;
  WriteLn('NOW we are back to the parent program');
  CASE P.status OF
    0 : WriteLn('WHAT we have here is a failure to communicate');
    1 : BEGIN
          WriteLn('THE STRING we got is "',P.name,'"');
          WriteLn('THE INTEGER we got is ',P.number);
        END;
    2 : WriteLn('The user halted the child process with ^Break');
    3..$FF : WriteLn('OOPS! The child process crashed in an ',
                     'unexpected way.');
    ELSE
      Write('The child process crashed with error $');
      WriteLn(HexByte(P.status AND $FF));
  END;
  status := 0;
  keyword := 'BORLANDint';
  Str(seg(mark1),tempS);
  Str(ofs(mark1),tempO);
  Exec('CHILDB.EXE',tempS+' '+tempO);
  WriteLn;
  WriteLn('NOW we are back to the parent program');
  CASE status OF
    0 : WriteLn('WHAT we have here is a failure to communicate');
    1 : BEGIN
          WriteLn('THE LONGINT we got is ',L);
          WriteLn('THE STRING we got is "',thing,'"');
        END;
    2 : WriteLn('The user halted the child process with ^Break');
    3..$FF : WriteLn('OOPS! The child process crashed in an ',
                     'unexpected way.');
    ELSE
      Write('The child process crashed with error $');
      WriteLn(HexByte(status AND $FF));
  END;
END.
```

LISTING 2: CHILDA.PAS

```
PROGRAM A;

{$I PASSDATA.TYP}
{$I GETPARAM.INC}
{$I HEX.INC}

VAR
  P        : ^passdata; {defined in PASSDATA.TYP}
  Is_child : boolean;

  PROCEDURE Passback_Status(code : integer);
  BEGIN
    P^.status := Code;
  END;


  PROCEDURE Check_If_Child;
  (* ---------------------------------------------- *)
  (*  IF this program is running as a child process, *)
  (*  then there will be two valid WORDs on the      *)
  (*  command line.  Assuming we find these words,   *)
  (*  we set a pointer to the address they define    *)
  (*  and look for a particular "password" in that   *)
  (*  area.  IF we find it, then we have successfully *)
  (*  set up a shared data area with the parent.     *)
  (* ---------------------------------------------- *)
  VAR S, O : word;
  BEGIN
    Get_Parameters(S,O,Is_Child); {in GETPARAM.INC}
    IF Is_Child THEN
      BEGIN
        P := Ptr(S,O);
        IF P^.password = 'BORLANDint' THEN
          Passback_Status(3)
        ELSE
          BEGIN
            WriteLn('ERROR in address');
            Is_Child := false;
          END;
      END;
  END;

  {$I ERRORHAN.INC}

BEGIN
  ExitProc := @ErrorHandler;
  WriteLn;
  WriteLn('     THIS is program A');
  Check_If_Child;
  IF Is_Child THEN
    BEGIN
      WriteLn('      The name passed from PARENT was "',P^.name,'"');
      Write('     ENTER A STRING    : '); ReadLn(P^.name);
      Write('     ENTER AN INTEGER : '); ReadLn(P^.number);
      P^.status := 1;
      {status 1 = normal termination}
    END;
END.
```

LISTING 3: CHILDB.PAS

```
PROGRAM B;

  {$I LISTOF.VAR}
  {$I HEX.INC}
  {$I GETPARAM.INC}

VAR
  S,O,Len : word;
  Is_child : boolean;

  PROCEDURE Passback_Status(code : integer);
  BEGIN
    status := Code;
    MOVE(mark1, MEM[S:O], Len);
  END;

  PROCEDURE Check_If_Child;
  (* ---------------------------------------------- *)
  (*  IF this program is running as a child process,  *)
  (*  then there will be two valid WORDs on the       *)
  (*  command line.  Assuming we find these words,    *)
  (*  we set a pointer to the address they define     *)
  (*  and look for a particular "password" in that    *)
  (*  area.  IF we find it, then we have successfully *)
  (*  set up a shared data area with the parent.      *)
  (* ---------------------------------------------- *)
  BEGIN
    Get_Parameters(S, O, Is_Child); {in GETPARAM.INC}
    IF Is_Child THEN
      BEGIN
        Len := ofs(mark2) - ofs(mark1);
        MOVE(MEM[S:O], mark1, Len);
        IF keyword = 'BORLANDint' THEN
          BEGIN
            Is_Child := true;
            Passback_Status(3);
          END
        ELSE
          BEGIN
            WriteLn('ERROR in address');
            Is_Child := false;
          END;
      END;
  END;

 {$I ERRORHAN.INC}

BEGIN
  ExitProc := @ErrorHandler;
  WriteLn;
  WriteLn('     THIS is program B');
  Check_If_Child;
  IF Is_Child THEN
    BEGIN
      WriteLn('      The string passed from PARENT was "',thing,'"');
      Write('     ENTER LONGINT: '); ReadLn(L);
      Write('     ENTER STRING : '); ReadLn(thing);
      status := 1;
      {status 1 is normal termination}
    END;
  IF Is_Child THEN MOVE(mark1, MEM[S:O], Len);
END.
```

LISTING 4: PASSDATA.TYP

```
(* ======================================== *)
(* PARENT and CHILDA share this file.  This   *)
(* is necessary to be sure the fields in the  *)
(* pointer to the shared area will be correct. *)
(* ======================================== *)
TYPE
  passdata = RECORD
               password : string[80];
               name     : string[80];
               number   : integer;
               status   : word;
             END;
```

LISTING 5: LISTOF.VAR

```
(* ============================================== *)
(* PARENT and CHILDB share the list of variables  *)
(* in this file.  This sharing is necessary to    *)
(* make SURE the passed data will be correct.     *)
(* ============================================== *)
VAR
  mark1   : byte;
  L       : LongInt;
  status  : word;
  keyword : string[80];
  thing   : string[80];
  mark2   : byte;
```

LISTING 6: GETPARAM.INC

```
PROCEDURE Get_Parameters(VAR SegP, OfsP : word; VAR OK : boolean);
VAR code : word;
BEGIN
  OK := false;
  IF ParamCount <> 2 THEN
    BEGIN
      WriteLn('     Program running as a standalone');
      Exit;
    END;
  Val(paramStr(1), SegP, Code);
  IF Code <> 0 THEN
    BEGIN
      WriteLn('     ERROR in segment parameter');
      Exit;
    END;
  Val(ParamStr(2), OfsP, Code);
  IF Code <> 0 THEN
    BEGIN
      WriteLn('     ERROR in offset parameter');
      Exit;
    END;
  OK := true;
END;
```

LISTING 7: ERRORHAN.INC

```
{$F+}
{$I-}
  PROCEDURE ErrorHandler;
  VAR I : integer;
  BEGIN
    IF ErrorAddr = NIL THEN
      BEGIN
        IF ExitCode = $FF THEN
          BEGIN
            WriteLn('     USER BREAK');
            IF Is_Child THEN Passback_Status(2);
            {status 2 means  Break}
          END;
        Exit;
      END;
    I := IOresult;
    Assign(Output,'');
    ReWrite(Output);
    Write('     ERROR $',HexByte(ExitCode));
    WriteLn(' at ',hex(seg(ErrorAddr^)),':',hex(ofs(ErrorAddr^)));
    Close(OutPut);
    IF Is_Child THEN Passback_Status(ExitCode OR $100);
  END;
{$F-}
{$I+}
```

LISTING 8: HEX.INC

```
(*  HEX functions *)
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 : integer) : string4;
  begin
    Hex := HexByte(Hi(I)) + HexByte(Lo(I));
  end;
```

user-written error handler passes back that information in **status**, as I'll describe later. The only way **status** will still be 0 after the **Exec** call is if the call failed—e.g., if the program to be executed as a child process was not found on disk. Based on this status word, the parent can determine the outcome of its child processes.

THE SAMPLE PROGRAMS

The programs PARENT.PAS (Listing 1), CHILDA.PAS (Listing 2), and CHILDB.PAS (Listing 3) demonstrate these two techniques. CHILDA and PARENT have a shared data area whose data type declaration appears in the file PASSDATA.TYP (Listing 4). CHILDB and PARENT have a copied data area whose variable list is in the file LISTOF.VAR (Listing 5). Note that PARENT, CHILDA, and CHILDB are three independent programs that must be compiled separately to disk—the Make or Build options will not detect any relationship between them.

SETTING UP COMMUNICATION

In each method, the parent program creates a command line consisting of the segment and offset of the shared data in string form. For CHILDA, the address is that of the shared record variable **P**. For CHILDB, it is the address of the first variable in the list of copied variables, **mark1**. Both child programs use the same procedure to get and test the command line parameters—**Get_Parameters**, in file GETPARAM.INC (Listing 6). This procedure checks the command line. If there are exactly two parameters and each is a valid integer, it sets the Boolean variable **OK** to **True** and sets its arguments to the integer values.

If CHILDA receives a valid address on the command line, it sets its shared data pointer **P** to point to that address. A word of caution here—before *writing* anything to this area, CHILDA checks the password field. If the password doesn't match, then something went wrong. If it *does* match, you can now read and write the data through the pointer.

CHILDB uses the passed data address differently. It uses the **Move** statement to move data starting at **mark1**. The number of bytes to move is the difference between the offsets of the first shared variable and the final marker, **mark2**. Here again, it uses a password variable to confirm that the address was correct. Now the child program can *read* the shared data any time. In order for it to *write* data, it has to use **Move** again to copy the shared data area back to the parent.

Both child programs simply collect some data and pass it back to the parent. To see what happens when a child crashes, give a non-numeric entry when the program asks for a number, or press Ctrl-C. It works!

MISCELLANEOUS FILES

All three of the programs use the file HEX.INC (Listing 8). Routines in this file will translate a byte or a word into a two- or four-character hexadecimal string. This is handy for addresses and errors.

The child programs also share the same error handler, in ERRORHAN.INC (Listing 7). You'll notice some differences between this error handler and its 3.0 equivalent.

First, ERRORHAN begins with the compiler directive **{$F+}** and ends with **{$F–}**. When this directive is active, (**{$F+}**) the compiler generates a **FAR** procedure or function. Since the error handler may be called from anywhere in the program, it has to be declared **FAR**.

Second, the arguments to the procedure, usually named **ErrNum** and **ErrAddr**, are gone. The global variables **ErrorAddr** and **ExitCode** replace them, and **ErrorAddr** is a true pointer now. You install the error handler differently, too. In 3.0, you set the integer variable **ErrorPtr** equal to the offset of the handler. In 4.0, you set **ExitProc**, not **ErrorPtr**, and **ExitProc** is a full segment offset address. The new @ operator returns the address of its parameter, like the **Addr** function in 3.0. **@ErrorHandler** returns the address of procedure **ErrorHandler**.

An error handler can take care of any "cleanup" work you need to do when the program crashes. In our case, all we need to do is tell the parent program what happened. Since the two child processes communicate with the parent differently, the error handler calls a procedure **Passback_Status**. Each child program has its own version of this procedure.

In **Exec**, Turbo Pascal 4.0 offers an execute-program routine that is much more powerful than its predecessor in 3.0. A little pointer savvy will allow you to regain the power lost with the demise of Turbo Pascal 3.0-style **Chain** and **Execute**. ■

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

Listings may be downloaded from CompuServe as PROCESS.ARC.

EXPLORING THE BORLAND BINARY EDITOR

Add the text editor used in the Borland languages to your own applications. It will only cost you 14K . . . bytes, that is.

Jeff Duntemann

PROGRAMMER

Those of us who somehow managed not to be astonished by Turbo Pascal's speed back in 1984 did not fail to be astonished by its size. "Folded into hyperspace" was my favorite explanation for an editor/compiler/environment taking all of 36K—my own rather limited "smart" directory lister program (written in that other Pascal) was larger than 36K all by itself.

It wasn't hyperspace, of course—just assembly language. And although the Turbo Pascal compiler got most of the glory, the assembly language text editor hiding beside it in memory kept appearing in (and evolving through) other Borland products, from SideKick's Notepad through Turbo Prolog, Turbo Basic, Turbo C, and Eureka: The Solver.

As part of the major rewrite of Turbo Pascal's Editor Toolbox (coinciding with the release of Turbo Pascal 4.0), the same assembly language editor in binary unit form has been added to the Toolbox's considerable repertoire. By using unit **BinEd**, your own applications can now incorporate the Borland Binary Editor.

SMALL WONDER

The original Borland Binary Editor used in Turbo Pascal 1.0–3.0 was only 10K in size. The Binary Editor has grown some since then in both size and power, but it still adds only about 14K of code to applications that incorporate it.

Unlike the standard units included with Turbo Pascal 4.0, the implementation section of **BinEd** is not included as part of the product. The interface section stands alone in a file named BINED.HDR, given in Listing 1. BINED.HDR contains descriptions of the entry points to the several routines incorporated in the Binary Editor, along with a description of the data structures that pass information back and forth between the Binary Editor and your own applications.

The Binary Editor is a "plain ASCII" text editor, meaning it does not insert unprintable control codes in the edited file for formatting purposes. If a file is printable when loaded, it will still be printable when written back out to disk. The command set understood by the editor is a subset of WordStar's, (see sidebar) and depends on control-key combinations rather than the PC keyboard function keys or pull-down menus. A key-code installation utility included with the Binary Editor, however, allows you to set many of the commands to any keystroke combination you prefer.

The Binary Editor incorporates a number of intriguing features:

- It is self-windowing. Within the Editor Control Block (ECB) are coordinates for the upper left corner and lower right corner of a window. These coordinates are passed to the Binary Editor when it is invoked. The Binary Editor will display, clear, and scroll only within the window defined by those coordinates.
- A cursor position and marked block may be passed to the Binary Editor along with a text file. Again, the ECB contains fields specifying a cursor position offset in characters into the text file, as well as an offset for the start and end of a marked block. If some other program mechanism has examined a text file and located a position or region within the file that needs attention, the Binary Editor may be popped up with the cursor at that position, or with the block in question marked and highlighted.
- Multiple exit commands may be specified at runtime. In other words, although the Binary Editor will always exit on Ctrl-KD, other exit command

continued on page 38

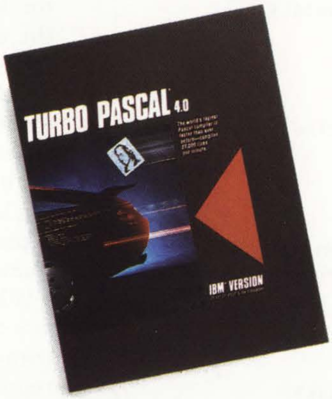
Program in the fast lane with Borland's new Turbo Pascal 4.0!

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

4.0 Technical Highlights:

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

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



4.0 breaks the code barrier

No more swapping code in and out to beat the 64K code barrier. Designed for large programs, Turbo Pascal 4.0 lets you use 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. 4.0 also includes an automatic project Make.

4.0's cursor automatically lands on any trouble spot

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

Only $99.95

*60-Day Money-back Guarantee***

For the dealer nearest you, or to order now,

Call (800) 543-7543

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

*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.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Copyright ©1987 Borland International, Inc. BI 1166

BORLAND

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

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

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

Name ______________________________

Ship Address ______________________________

City ______________________ State __________

Zip __________ Telephone ( ) ______________

This offer is limited to one upgrade per valid registered product. It is good until November 30, 1987. 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.

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

Please check box(es)	Suggested Retail	Upgrade Price†	Serial No.
□ Turbo Pascal 4.0 Compiler	$ 99.95	$ 39.95	________
□ Turbo Pascal 4.0 Developer's Library	395.00	150.00	
(Includes Turbo Pascal Tutor and all Toolboxes; must be ordered with compiler)			
□ 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 $ ________
Shipping and handling* $ ________
Total amount enclosed $ ________

Please specify diskette size □ 5¼" □ 3½"

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

Payment: □ VISA □ MC □ Check □ Bank Draft

Credit card expiration date: ________/________

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

*In US please add $5 shipping for each product ordered or $15 for the Compiler and Developer's Library. Outside US please add $10 shipping and handling for each product ordered or $25 for the Compiler and Developer's Library.

TT-11/87

LISTING 1: BINED.HDR

```
{$I-}
{$S-}
{$R-}

unit BinEd;
  {-The Borland binary editor interface for Turbo Pascal}

interface

const
  MaxFileSize = $FFE0;      {Maximum editable file size }
  EdOptInsert = $1;         {Insert on flag}
  EdOptIndent = $2;         {Autoindent on flag}
  EdOptTAB = $8;            {Tab on flag}
  EdOptBlock = $10;         {Show marked block}
  EdOptNoUpdate = $20;      {Don't update screen when entering editor}
  EventKBflag = 1;          {Scroll, num or caps locks modified mask}
  CAnorm = #255#1;          {Activates CRT "normal" attribute}
  CAlow = #255#2;           {Activates CRT "low"         -    }
  CAblk = #255#3;           {Activates CRT "block"       -    }
  CAerr = #255#4;           {Activates CRT "error"       -    }
  EdStatTextMod = 1;        {Text buffer modified mask}

type
  AttrArray = array[0..3] of Byte;
  ASCIIZ = array[0..255] of Char;
  ASCIIZptr = ^ASCIIZ;
  TextBuffer = array[0..$FFF0] of Char;

  CRTinsStruct =            {CRT installation structure}
  record
    CRTtype : Byte;         {1=IBM, 0=Non}
    CRTx1, CRTy1,
    CRTx2, CRTy2 : Byte;    {Initial window size}
    CRTmode : Byte;         {Initial mode 0-3,7 or FF(default)}
    CRTsnow : Byte;         {0 if no snow, don't care for mono}
    AttrMono : AttrArray;   {CRT attributes for mono mode}
    AttrBW : AttrArray;     {CRT attributes for b/w modes}
    AttrColor : AttrArray;  {CRT attributes for color modes}
  end;
  CIptr = ^CRTinsStruct;

  EdInsStruct =             {Command table installation structure}
  record
    ComTablen : Word;       {Maximum length of command table}
    ComTab : TextBuffer;    {Command table}
  end;
  EIptr = ^EdInsStruct;

  MIinsStruct =             {Main installation structure}
  record
    Ver : Byte;             {Main version}
    VerSub : Byte;          {Sub version}
    VerPatch : Char;        {Patch level}
    CPUmhz : Byte;          {CPU speed for delays}
    CIstruct : CIptr;       {Points to CRT installation record}
    EIstruct : EIptr;       {Points to Editor installation area}
    DefExt : ASCIIZptr;     {Points to ASCIIZ default extension}
  end;
  MIptr = ^MIinsStruct;
```

continued from page 36

strings may be passed to the Editor when it is initialized. The command which caused the Editor to exit is returned to the application using the Editor. This allows an application to "duck out" of the Binary Editor temporarily to take care of other tasks and reenter it later, perhaps without the user knowing that editing had been interrupted. For example, if an interactive help system must be available during editing, the F1 key could be defined as an exit command. When the user requests help by pressing F1, the Binary Editor would return control to the application program along with the F1 keystroke. The application would then branch to the help system based on the F1 command, and later reenter the editor to resume work where it had been left when F1 had been pressed.

- A "user event handler" may be defined. Every time the Binary Editor checks for a pending keystroke, it calls a procedure whose address has been passed to the Editor as a procedure pointer. (If the pointer passed is **NIL**, the call is not made.) This allows an application incorporating the Binary Editor to perform some "cooperative" multitasking, such as continuously maintaining a clock display in the corner of the screen, or sending characters from a file out to the system printer while another file is being edited.

BUILDING THE BINARY EDITOR INTO A PROGRAM

Considering all that the Binary Editor is able to do, application support for it is almost trivial. In fact, the smallest possible text editor incorporating the Binary Editor can be written in just 27 lines. This editor is given in Listing 2, SIMPLE.PAS. You can invoke SIMPLE.PAS from the command line with a text filename:

```
C>SIMPLE MYFILE.TXT
```

It edits the file, and saves the file to disk under the same file name when exited, with the original version of the file retained as a .BAK file. Because SIMPLE.PAS is uncluttered and easy to understand, we'll use it as an example in the following discussion.

To use the Binary Editor, your programs must **USE** both the **BinEd** and **Crt** units:

```
USES BinEd,CRT;
```

An ECB (whose type **EdCB** is declared in the BINED.HDR file shown in Listing 1) must be declared as a variable:

```
VAR EdData : EdCB;
```

Before you can load a file and call the Binary Editor for editing, the Editor must be initialized with a call to a function called **InitBinaryEditor**. The initialization task allocates buffers in memory and sets up certain fields in the ECB. The parameters to **InitBinaryEditor** are important, and bear examining in some detail:

EdData. ECB variable **EdData** goes to **InitBinaryEditor** "empty" and comes back ready to be passed to the editor proper.

DataLen. This **Word** variable gives the size of the workspace available to the Binary Editor. A constant named **MaxFileSize** may be imported from unit **BinEd**, as is done in SIMPLE.PAS. Its value, $FFE0, is the largest possible workspace that may be edited. However, if less space will do, a smaller value may be passed here. Because this workspace is subtracted from the heap, using a smaller workspace will make more memory available for dynamic variables used in other parts of the application calling the Binary Editor.

CX1 and CY1. These are the coordinates of the upper left corner of the window within which the Editor must work. **CX1** must be less

```
  EdCB =                      {Editor control block in detail}
  record
    x1, y1, x2, y2 : Byte;    {UL & LR corners of editor window}
    DataSeg : Word;           {Segment address of editor data area}
    DataSegLen : Word;        {Requested data area length (bytes)}
    Options : Word;           {Bit flags for editor options}
    FileStr : ASCIIZptr;      {Points to ASCIIZ filename}
    Commands : ASCIIZptr;     {Points to string of editor commands}
    Place1 : ASCIIZptr;       {Not used here}
    Place2 : ASCIIZptr;       {Not used here}
    Event : Pointer;          {Points to event handling procedure}
    Buffer : ^TextBuffer;     {Points to text area}
    BufSize : Word;           {Available size for text}
    MIstruct : MIptr;         {Points to main installation record}
    ComTab : ASCIIZptr;       {Points to terminate command table}
    EOtext : Word;            {Current number of chars in text buffer}
    CursorPos : Word;         {Current cursor position in buffer}
    BlockStart : Word;        {Start of marked block in buffer}
    BlockEnd : Word;          {End of marked block in buffer}
    Status : Word;            {Editor status}
    DataPtr : ^TextBuffer;    {Points to Turbo heap block  }
  end;                        {  allocated for text buffer}

const
  {CRT attributes for   normal low blk error}
  MonoArray : AttrArray = ($F, $7, $7, $70);
  BwArray : AttrArray = ($F, $7, $7, $70);
  ColorArray : AttrArray = ($E, $7, $3, $1E);

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

procedure CRTputFast(x, y : Word; s : string);
  {-Use binary editor services to write a string to the screen}
  {x in 1..25, y in 1..80}

function ExpandPath(Fname : string) : string;
  {-Return a complete path using the binary editor services}

function InitBinaryEditor(
    var EdData : EdCb;            {Editor control block}
    DataLen : Word;               {Size of binary editor workspace}
    Cx1 : Byte;                   {Editor window, upper left x 1..80}
    Cy1 : Byte;                   {Editor window, upper left y 1..25}
    Cx2 : Byte;                   {Editor window, lower right x 1..80}
    Cy2 : Byte;                   {Editor window, lower right y 1..25}
    WaitForRetrace : Boolean;     {True for snowy color cards}
    Coptions : Word;              {Initial editor options}
    DefExtension : string;        {Default file extension  }
                                  {  (must start with a period)!}
    var ExitCommands;             {Commands to exit editor}
    UserEventProcPtr : Pointer {Pointer to user event handler}
    ) : Word;

  {-Initialize the binary editor, returning a status code}
  {
  Status Codes -
  0 = Successful initialization
  1 = Insufficient memory space for text buffer
  }

function ReadFileBinaryEditor(var EdData : EdCb;
                              Fname  : string) : Word;
  {-Read a file into the binary editor buffer space, }
  { returning a status code }
  {
  Status codes -
```

```
    0 = Successful read
    1 = File not found, new file assumed
    2 = File too large to edit
  }

procedure ResetBinaryEditor(var EdData : EdCb);
  {-Call the editor reset procedure}

function UseBinaryEditor(var EdData       : EdCb;
                             StartCommands : string) : Integer;
  {-Edit file, using startcommands, and returning an exitcode}
  {
  Exit codes -
   -1 = Editing terminated with ^KD
    0 = Editing terminated with first user-specified exit command
    1 ...
  }

function ModifiedFileBinaryEditor(var EdData : EdCb) : Boolean;
  {-Return true if text buffer was modified during edit}

function FileNameBinaryEditor(var EdData : EdCb) : string;
  {-Return the current file pathname of the specified control block}

function SaveFileBinaryEditor(var EdData     : EdCb;
                                  MakeBackup : Boolean) : Word;
  {-Save the current file in the editor text buffer,
  {  returning a status code }
  {
  Status codes -
    0 = Successful save
    1 = File creation error
    2 = Disk write error
    3 = Error closing file
  }

procedure ReleaseBinaryEditorHeap(var EdData : EdCb);
  {-Release heap space used by a binary editor control block}
```

LISTING 2: SIMPLE.PAS

```
program Simple;  {SIMPLE.PAS: A simple editor using the BINED unit}

uses BinEd, Crt;

const ExitCommands : Char = #0;   {No extra exit commands}
var EdData : EdCB;              {Editor control block}

  procedure Abort(Msg : String);
  begin {Abort}
    GotoXY(1, 25); Write(Msg); Halt(1);
  end;  {Abort}

begin  {main}
  if (ParamCount = 0) then                      {No filename}
     Abort('Usage: SIMPLE filename.ext');
  if (InitBinaryEditor(EdData, MaxFileSize, 1, 1, 80, 25,
     True, EdOptInsert, '', ExitCommands, nil) <> 0) then
       Abort('Unable to load binary editor.');  {Couldn't load editor}
  if (ReadFileBinaryEditor(EdData, ParamStr(1)) > 1) then
     Abort('Unable to read ' + ParamStr(1));    {Couldn't read file}
  ResetBinaryEditor(EdData);                    {Reset for new file}
  if (UseBinaryEditor(EdData, '') = -1) then    {Edit the file}
     if ModifiedFileBinaryEditor(EdData) then   {Was it modified?}
        if (SaveFileBinaryEditor(EdData, True) <> 0) then {Save it}
           Abort('Error saving file.');
  GotoXY(1,25);
end.  {main}
```

than 80, but I have tested **CY1** out to 62 lines on the Genius VHR display.

CX2 and **CY2** are the coordinates of the lower right corner of the Editor's window. Again, **CY2** must be less than or equal to 80, but **CX2** works to at least 66 lines.

WaitForRetrace. This flag tells the Editor whether or not to wait for vertical retrace before refreshing the screen. IBM's original Color Graphics Adapter (CGA) displays "snow" when its buffer memory is accessed during display scan. Many CGA-compatible display boards use dual-ported memory and do not have this problem. For clean displays on all adapters this parameter should be set to **True**. However, if you only intend to run the editor on a snow-free adapter, setting **WaitForRetrace** to **False** will make for faster screen refresh.

COption. This set of bit flags is gathered together into a **Word**. The bit flags are Editor toggles that may be changed by various key commands during editing, and **Options** specifies the initial state of these toggles. The bit flags may be imported from unit **BinEd** as constants whose identifiers begin with **EdOpt**. (See the **CONST** section of BINED.HDR, Listing 1.) The easiest way to specify the bit flags is to use the **EdOpt** constants and add together any constants corresponding to the toggles you wish to assert. For example, to bring up the Binary Editor with both insert and autoindent on but with all other options off, add together the constants **EdOptInsert** and **EdOptIndent** and pass the resulting value in the **COptions** parameter.

DefExtension. This is a string specifying a default extension to be assumed for a file name specified without an extension. Turbo Pascal's well-known assumption of a .PAS extension for extensionless file names is this feature in action. Wildcard characters are not allowed. Pass an empty string (") to assume no extension.

ExitCommands. This parameter is slightly peculiar because it's an untyped **VAR** parameter. As happens with untyped **VAR** parameters, the actual parameter passed in **ExitCommands** is a pointer to a region of memory containing sequences of bytes terminated by a NUL character, binary 0. In C circles this would be called an "ASCIIZ string." This NUL-terminated string contains substrings that the Editor watches for in the stream of characters coming from the keyboard. Any sequence found in the stream causes the Editor to return control to the calling logic. The actual parameter passed in **ExitCommands** may be any type, but the most convenient thing to do is define a typed constant array of **Char** with the desired values inside. This is best shown as an example:

```
CONST
  ExitCommandArray :
    ARRAY[0..6] OF Char =
      (#2,^K,^X,#2,^K,^Q,#0);
```

Here, **ExitCommandArray** contains two separate exit commands: Ctrl-KX and Ctrl-KQ. The format for a command is: A binary byte specifying the length of the command in bytes, followed by the characters comprising that command. The entire array must be terminated by a binary 0. The **#2** characters in the array tell the Editor to watch for the following two characters as an exit command. There's nothing magical about a two-character sequence; if you wanted to use Ctrl-X to exit the editor, the sequence would be **#1,^X**. To search for an "extended" key, use an initial byte of **#0**. The character sequence for F10 would be **#2,#0,#68**.

Finally, **UserEventPtr** is an untyped pointer that is passed the address of a FAR-declared procedure that executes every time the

LISTING 3: DEMO0A.PAS

```
{ DEMO0.PAS Binary Editor 2.00A }
{Copyright 1986,87 (c) Borland International }
{Modified by Jeff Duntemann for Turbo Technix 8/31/87 }

program BinaryEditorDemo0;

uses
  bined,
  crt,
  dos;        {JD}

  {***************************************************************}
  {****************** demonstration follows **********************}
  {***************************************************************}
  {* This demonstration shows the use of one editor window which *}
  {********* works just like a standalone Turbo editor. **********}
  {***************************************************************}

const
  {Coordinates of the editor window}
  Windx1 = 1;
  Windy1 = 1;
  Windx2 = 80;
  Windy2 = 25;                   {Change to 43 for EGA 43-line operation}
  MakeBackup = True;             {True to create .BAK files}

var
  EdData : EdCB;                 {Editor control block}
  ExitCode : Word;               {Status code set by bin. ed. functions}
  ExitCommand : Integer;         {Code for command used to leave editor}
  Fname : string;                {Input name of file being edited}
  Junk : Boolean;
  XSave,YSave : Integer;         {JD}
  VidSegment : Word;             {JD}
  VideoBufferSize : Word;        {JD}
  SavePtr  : ^Word;              {JD}
  VideoPtr : ^Word;              {JD}
  VideoSeg : Word;               {JD}
  Now      : DateTime;           {JD}

const
  {Commands other than ^K^D to exit editor}
  ExitCommands : array[0..3] of Char =
  (#2, ^K, ^Q, #0);

  {Procedures and functions used as part of the demo}

  procedure WriteStatus(msg : string);
    {-Write a status message}

  begin                          {WriteStatus}
    GoToXY(1, Windy2);
    TextColor(White);
    Write(msg);
  end;                           {WriteStatus}

  procedure CheckInitBinary(ExitCode : Word);
    {-Check the results of the editor load operation}

  begin                          {CheckInitBinary}
    if ExitCode <> 0 then begin
      {Couldn't load editor}
      case ExitCode of
        1 : WriteStatus('Insufficient heap space for text buffer');
      else
        WriteStatus('Unknown load error');
      end;
      GoToXY(1, Windy2);
      Halt(1);
    end;
```

```
end;                            {CheckInitBinary}

procedure CheckReadFile(ExitCode : Word; Fname : string);
  {-Check the results of the file read}
var
  f : file;

begin                           {CheckReadFile}
  if ExitCode <> 0 then begin
    {Couldn't read file}
    case ExitCode of
      1 : begin
          {New file, assure valid file name}
          {$I-}
          Assign(f, Fname);
          Rewrite(f);
          if IOResult <> 0 then begin
            Close(f);
            WriteStatus('Illegal file name '+Fname);
          end else begin
            Close(f);
            Erase(f);
            Write('New File');
            Delay(2000);
            Write(^M);
            ClrEol;
            GoToXY(1, 1);
            ClrEol;
            Exit;
          end;
          {$I+}
        end;
      2 : WriteStatus('Insufficient text buffer size');
    else
      WriteStatus('Unknown read error');
    end;
    GoToXY(1, Windy2);
    Halt(1);
  end;
  GoToXY(1, 1);
  ClrEol;
end;                            {CheckReadFile}

procedure CheckSaveFile(ExitCode : Word; Fname : string);
  {-Check the results of a file save}

begin                           {CheckSaveFile}
  if ExitCode <> 0 then begin
    {Couldn't save file}
    case ExitCode of
      1 : WriteStatus('Unable to create output file '+Fname);
      2 : WriteStatus('Error while writing output to '+Fname);
      3 : WriteStatus('Unable to close output file '+Fname);
    else
      WriteStatus('Unknown write error');
    end;
    GoToXY(1, Windy2);
    Halt(1);
  end;
end;                            {CheckSaveFile}

function GetFileName : string;
  {-Return a file name either from the command line or a prompt}
var
  Fname : string;

begin                           {GetFileName}
  if ParamCount > 0 then
    Fname := ParamStr(1)
  else begin
```

Binary Editor checks for a pending keystroke. More on this later.

InitBinaryEditor. This function returns a value of **True** if the initialization has completed successfully. The ECB it returns must be passed to all subsequent Binary Editor routines. Once the Editor is initialized, a file may be read for editing. This is done with the function **ReadFileBinaryEditor**. It takes the initialized ECB as one parameter and a file name as the other, and returns a status code as a **Word** value. The status code will be one of three values: 0, indicating successful load; 1, indicating that no file is found and that a new file of that name will be created; and 2, indicating that the indicated file was too large to load into the current Editor workspace.

As long as the initialized ECB remains intact, the Binary Editor can be exited and reentered without disturbing the file under edit in the workspace. Therefore, each time the Editor is entered after a new file has been loaded into the workspace, the **ResetBinaryEditor** procedure must be called with the ECB as its parameter, to inform the Editor that the file in the workspace is in fact new.

To actually begin editing the loaded file, invoke the function **UseBinaryEditor** with the ECB as one parameter and a string containing one or more "start commands" as its other parameter. Any sequence of control-character commands may be passed in the start commands string, followed by a NUL character. For example, to move immediately to the end of the loaded file as soon as editing begins, pass Ctrl-QC in the **StartCommands** string.

The exit codes returned by **UseBinaryEditor** specify how the editor was exited. A value of -1 indicates that the Ctrl-KD command was used. Ctrl-KD is the default and only valid exit code, unless others are specified during the call to **InitBinaryEditor**.

Values of 0 or greater indicate which one of the user-specified exit codes was issued. A 0 indicates the first command in the **ExitCommands** ASCIIZ string, 1 the second command in the string, and so on. *The exit command itself is not returned.*

Exiting the Binary Editor does not automatically save any changes made to the file in the workspace. That must be done explicitly using function **SaveFileBinaryEditor**. Its first parameter is the ECB and its second is a Boolean value that if True, will cause the last saved version of the file to be retained as the backup version with a .BAK file extension.

A Boolean function, **ModifiedFileBinaryEditor**, is available that returns a True value if the file in the workspace was modified since the last time the file was saved. This allows an application to decide whether or not the workspace file must be saved.

A MORE SERVICEABLE EDITOR

SIMPLE.PAS clearly illustrates the essential logic in using the routines in **BinEd**, but it is a minimal editor at best, especially with respect to error handling. Listing 3 shows a much better text editor distributed with the Turbo Pascal Editor Toolbox. DEMO0.PAS has more of a user interface: If no file name is entered on the command line, it prompts for one. It also gives the user the option of not saving a file once it has been edited. Most important, when a file I/O error occurs, it provides detailed feedback in cases where SIMPLE.PAS can only shrug.

To demonstrate how easy it is to build upon the Binary Editor, I have added some features to DEMO0.PAS:

- It saves the DOS text screen to the heap before it alters the screen, and restores the text screen just before it returns to DOS. This can be useful when you need to edit a number of

```
    Write('Enter file name to edit: ');
    ReadLn(Fname);
  end;
  if Fname = '' then
    Halt;
  GetFileName := Fname;
end;                        {GetFileName}

function ExitBinaryEditor(var EdData      : EdCB;
                              ExitCommand : Integer) : Boolean;
  {-Handle an editor exit - save or abandon file}
var
  ExitCode : Word;

  function YesAnswer(prompt : string) : Boolean;
    {-Return true for a yes answer to the prompt}
  var
    ch : Char;

  begin                     {YesAnswer}
    WriteStatus(prompt);
    repeat
      ch := UpCase(readkey);
    until ch in ['Y', 'N'];
    Write(ch);
    YesAnswer := (ch = 'Y');
  end;                      {YesAnswer}

begin                       {ExitBinaryEditor}
  case ExitCommand of
    -1 :                    {^K^D}
      begin
        ExitCode := SaveFileBinaryEditor(EdData, MakeBackup);
        CheckSaveFile(ExitCode, FileNameBinaryEditor(EdData));
        ExitBinaryEditor := True;
        GoToXY(1, Windy2);
      end;

    0 :                     {^K^Q}
      begin
        if ModifiedFileBinaryEditor(EdData) then
          if YesAnswer('File modified. Save it? (Y/N) ') then begin
            ExitCode := SaveFileBinaryEditor(EdData, MakeBackup);
            CheckSaveFile(ExitCode, FileNameBinaryEditor(EdData));
          end;
        ExitBinaryEditor := True;
        GoToXY(1, Windy2);
      end;

    end;
  end;                        {ExitBinaryEditor}
```

```
{$F+} { All User-Event procesudures must be FAR calls!}
PROCEDURE Clocker(EventNo,Info : Integer);

VAR
  Hours,Minutes,Seconds,Hundredths : Integer;
  TimeBuf,TimeTemp : String;

BEGIN
  GetTime(Hours,Minutes,Seconds,Hundredths);
  Str(Hours:2,TimeBuf);
  Str(Minutes:2,TimeTemp);
  IF TimeTemp[1] = ' ' THEN TimeTemp[1] := '0';
  TimeBuf := TimeBuf+':'+TimeTemp;
  Str(Seconds:2,TimeTemp);
  IF TimeTemp[1] = ' ' THEN TimeTemp[1] := '0';
  TimeBuf := TimeBuf+':'+TimeTemp;
  CRTPutFast(65,1,TimeBuf)
END;
```

The Borland Binary Editor Command Set

CURSOR MOVEMENT COMMANDS

Character left	Ctrl-S or Left Arrow
Character right	Ctrl-D or Right Arrow
Word left	Ctrl-A or Ctrl-Left Arrow
Word right	Ctrl-F or Ctrl-Right Arrow
Line up	Ctrl-E or Up Arrow
Line down	Ctrl-X or Down Arrow
Scroll up	Ctrl-W or Ctrl-Up Arrow
Scroll down	Ctrl-Z or Ctrl-Down Arrow
Page up	Ctrl-R or Pg Up
Page down	Ctrl-C or Pg Dn
Beginning of file	Ctrl-QR or Ctrl-Pg Up
End of file	Ctrl-QC or Ctrl-Pg Dn
Beginning of line	Ctrl-QS or Home
End of line	Ctrl-QD or End
Top of screen	Ctrl-QE or Ctrl-Home
Bottom of screen	Ctrl-QX or Ctrl-End
Top of block	Ctrl-QB
Bottom of block	Ctrl-QK
Previous cursor position	Ctrl-QP
Jump to marker 0..3	Ctrl-Q0...Ctrl-Q3
Set marker 0..3	Ctrl-K0...Ctrl-K3

INSERT AND DELETE COMMANDS

New line	Ctrl-M or Enter
Insert line	Ctrl-N
Insert control character	Ctrl-P
EX: Ctrl-P G inserts BEL (Ctrl-G)	
Tab	Ctrl-I or Tab
Delete current character	Ctrl-G or Del
Delete character left	Ctrl-H or Backspace
Delete word	Ctrl-T
Delete to end of line	Ctrl-QY
Delete line	Ctrl-Y

BLOCK COMMANDS

Begin block	F7 or Ctrl-KB
End block	F8 or Ctrl-KK
Copy block	Ctrl-KC
Move block	Ctrl-KV
Delete block	Ctrl-KY
Hide block	Ctrl-KH
Mark single word	Ctrl-KT
Read block from file	Ctrl-KR
Write block to file	Ctrl-KW
Print block	Ctrl-KP

MISCELLANEOUS COMMANDS

Exit editor	Ctrl-KD
Toggle insert mode	Ctrl-V or Ins
Toggle autoindent	Ctrl-QI
Toggle fixed tabs/smart tabs	Ctrl-O F
Restore line	Ctrl-QL
Search for string	Ctrl-QF (See notes on options)
Search and replace string	Ctrl-QA (See notes on options)
Repeat last search operation	Ctrl-L

NOTES

SEARCH OPTIONS

B — *Searches backwards, from the current cursor position toward the beginning of the file.*

G — *Search globally, i.e., search the entire file starting at the beginning, or backward from the end of the file if combined with the B option.*

# (a number) — *Find the nth position of the search string, starting at the current cursor position.*

U — *Ignore case and treat all characters as uppercase.*

W — *Search for whole words only; skip patterns embedded in other text.*

REPLACE OPTIONS

N — *Replace without asking.*

files in a directory and don't want to have to display the directory again before each edit, just to make sure you've spelled a file name right.

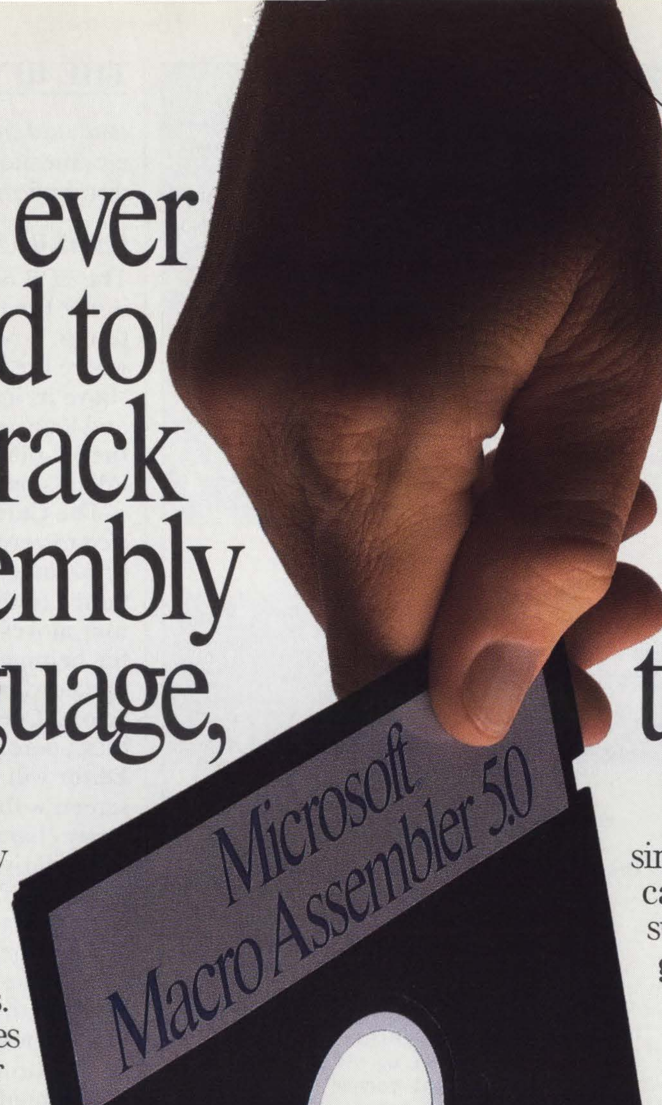
- It uses the user-event feature of the Binary Editor to maintain a clock display in the upper left corner of the screen. The address of procedure **Clocker** is passed to the Binary Editor during the call to **InitBinaryEditor** by using the **Addr** function. A user-event procedure must be declared as a FAR procedure by enclosing it between the {**$F+**} and {**$F−**} compiler directives. The procedure may be given any name, but it must be declared with two parameters (again, the names are not important) of type **Word**, as shown in Listing 3.

The shaded lines of code are the

continued on page 46

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.

```
{$F-}

{<<<< Monochrome >>>>}
{ From: COMPLETE TURBO PASCAL by Jeff Duntemann  }
{ Scott, Foresman & Co. 1986  ISBN 0-673-18600-8 }
{ Described in section 17.2 -- Last mod 2/1/86    }
{ HIGHLY specific to the IBM PC! }

FUNCTION Monochrome : Boolean;

VAR
  Regs : Registers;

BEGIN
  INTR(17,Regs);
  IF (Regs.AX AND $0030) = $30 THEN Monochrome := True
    ELSE Monochrome := False
END;


begin                         {Demo0}
  XSave := WhereX; YSave := WhereY;                {JD}
  VideoBufferSize := Windx2*Windy2*2;              {JD}
  GetMem(SavePtr,VideoBufferSize);                 {JD}
  IF Monochrome THEN VidSegment := $B000 ELSE      {JD}
    VidSegment := $B800;                           {JD}
  VideoPtr := Ptr(VidSegment,0);                   {JD}
  Move(VideoPtr^,SavePtr^,VideoBufferSize);        {JD}

  {Get a file name}
  Fname := GetFileName;

  {Initialize a window for the file}
  ExitCode :=
  InitBinaryEditor(
  EdData,                     {Editor control block }
  MaxFileSize,                {Size of data area to reserve for}
                              {binary editor text buffer, $FFE0 max}
  Windx1,                     {X of upper left corner; 1..80}
  Windy1,                     {Y of upper left corner}
  Windx2,                     {X of lower right corner}
  Windy2,                     {Y of lower right corner}
  True,                       {True = wait for retrace on CGA cards}
  EdOptInsert+EdOptIndent,    {Initial editor toggles}
  '.PAS',                     {Default extension for file names}
  ExitCommands,               {Commands which will exit the editor}
  Addr(Clocker));             {JD: Add a clock in the corner}
  CheckInitBinary(ExitCode);

  {Read the file}
  ExitCode := ReadFileBinaryEditor(EdData, Fname);
  CheckReadFile(ExitCode, FileNameBinaryEditor(EdData));

  {Reset the editor for the new file}
  ResetBinaryEditor(EdData);

  {Edit the file}
  ExitCommand :=
  UseBinaryEditor(
  EdData,                     {Editor control block for this window}
  '');                        {No startup commands passed to editor}

  {Handle the exit by saving the file or whatever}
  Junk := ExitBinaryEditor(EdData, ExitCommand);

  {Release heap space used by the editor data structure}
  ReleaseBinaryEditorHeap(EdData);

  Move(SavePtr^,VideoPtr^,VideoBufferSize);   {JD}
  FreeMem(SavePtr,VideoBufferSize);           {JD}
  GotoXY(XSave,YSave-1);                      {JD}
end.                          {Demo0}
```

THE BINARY EDITOR

continued from page 44

modifications made to the distribution file DEMO0.PAS.

OTHER CONSIDERATIONS

The ECB occupies a central place in the Binary Editor's scheme of things. It serves as a high shelf where the Editor can temporarily shove its important information, and it also offers the user certain special-purpose hooks into the Editor's operation.

The **CursorPos** field is initialized to zero, and when a file is first edited, the cursor is positioned to the offset given. As the user moves the cursor through the file being edited, the Editor updates **CursorPos** in the ECB. Setting **CursorPos** to some other offset before calling **UseBinaryEditor** will bring the file to the screen with the cursor offset that many characters into the file.

Similarly, the **BlockStart** and **BlockEnd** fields in the ECB are initialized to zero, but may be given values that specify the boundaries of an initially marked block. If the Binary Editor is entered with bit 4 of the **Options** field set to 1, the block will be highlighted. This bit is toggled by the Ctrl-KH command.

THE EDITOR AS A BASIC MECHANISM

In mechanical engineering there are certain fundamental mechanisms (lever, screw, inclined plane) that make up larger mechanisms. The same is true of large application programs. These programs contain basic mechanisms such as sort routines, tree managers, and text editors that have definitely "been done" and yet still must be there, and still must work correctly. I have written my own text editor (in interpreted BASIC, *mon Dieu!*) and would just as soon not have the pleasure again, thank you. The Borland Binary Editor provides a very serviceable editor —a basic mechanism in well-documented, bolt-on form. ■

Listings may be downloaded from CompuServe as EXBINED.ARC.

SENSE AND SEMICOLONS

Semicolons are for separating—that's the only rule you need to remember.

Jeff Duntemann

SQUARE ONE

Little things do count, and if the importance of little things is measured by the size of the headaches they cause, then in Pascal circles the humble semicolon stands apart. Newcomers to the language, desperate to find sense in Pascal's bewildering array of syntactic rules and regulations, too often see nothing but capriciousness in the placing of semicolons. There are places where they *must* go, places where they *can't* go, and places where it doesn't seem to matter whether they go or not. There are times when it can send you screaming out into the den to watch "Love Boat" reruns, sure that nothing makes sense in the world anymore.

Of course, like everything else in Pascal, semicolon placement *does* make sense. The problem is that the sense in semicolon placement proceeds from the larger structure of the Pascal language—and in first facing the language, newcomers tend to take arms against a sea of details, losing the big picture in the process. It's time to take a closer look at semicolons and the sense behind them.

THE ICE CREAM SODA METAPHOR

But first, a profound truth: The Turbo Pascal compiler drinks your program through a straw. It's an easy thing to forget. You look down at your neatly indented pages of source code and see a two-dimensional landscape of program structures. The compiler has no comparable view from a height. It opens your source file, reads the first character, and then the next character, and then the next, remembering certain essential things as it does its work. Still, its "view" of the language is essentially one-dimensional.

Also, remember that source code formatting is utterly ignored by the compiler. The two program listings in Figure 1 are completely identical from the compiler's perspective. Sadly, all your carefully considered indenting and spacing are left behind in the bottom of the glass as the program disappears up the straw. What we call "whitespace" (space characters, tab characters, carriage returns) is ignored beyond what it takes to delineate reserved words and identifiers from one another. In other words, one whitespace character is required to separate Figure 1's **BEGIN** from **ClrScr** (a carriage return) and the several space characters after the carriage return are ignored.

Formatting in Pascal is ignored to free the language from the tyranny of line structure, a bond that BASIC, FORTRAN, COBOL, and other early computer languages are only just now beginning to break. (Turbo Basic breaks it quite thoroughly, prompting some people to say it's becoming distinctly Pascal-ish.) To allow you to format source code as you please, the compiler must ignore any and all such formatting. The soda straw through which it sees your source code is a set of blinders that forces it to see only what matters.

But the problem remains: Whitespace isn't always enough to tell the compiler when a statement ends. Consider these two equally valid assignment statements:

```
Foo :=
       PI + 0.6

Foo := PI + 0.6 - Foo
```

Given that formatting is ignored, and that the compiler is examining these statements character by character, how does it know when they end? The compiler cannot "look ahead" to see what's coming up the straw and use that information to decide whether or not the statement it is currently compiling has ended. And because of the way it works, a "one-pass" compiler like Turbo Pascal cannot postpone a decision on ending a statement while it sucks up and examines a few additional characters. Once it passes a point in the source code, it never goes back.

Enter the semicolon: Its job is to tell the compiler when a statement is complete. When the compiler encounters a semicolon, it assumes that the current statement has ended, and that the next character up the straw begins a new statement. This is why we say

```
PROGRAM Rooter;

VAR
  R,S : Real;

BEGIN
  ClrScr;
  Writeln
    ('>>Square root calculator<<');
  Writeln;
  Write
    ('>>Enter the number: ');
  Readln(R);
  S := Sqrt(R);
  Writeln
    ('  The square root of ',R:7:7,' is ',S:7:7,'.')
END.
```

```
PROGRAM Rooter;VAR R,S : Real;BEGIN ClrScr;
Writeln('>>Square root calculator<<');Writeln;
Write('>>Enter the number: ');Readln(R);S := Sqrt(R);
Writeln('  The square root of ',R:7:7,' is ',S:7:7,'.')
END.
```

Figure 1. Two views of program Rooter.

that the semicolon is a statement *separator.*

THE FREIGHT CAR METAPHOR

When you have a series of statements, a semicolon must stand between one statement and the next. One way to think about this is to imagine a line of boxcars on a rail siding. Between every pair of boxcars is a pair of linked couplers. If Pascal statements were boxcars, each pair of couplers would be a semicolon.

Now, confusion may arise if you mistake **BEGIN** and **END** for boxcars. **BEGIN** and **END** are *not* statements. They are reserved words and act only as markers signaling the beginning and ending points of a compound statement. With that in mind, the semicolon situation in the following compound statement should make sense:

```
BEGIN
  I := 11;
  J := 17;
  K := 42
END
```

There is no semicolon after the **42** because there is no *statement* after the **42**. The statement **K:=42** doesn't need separating from anything, therefore, it has no semicolon after it.

When you have a series of statements, a semicolon must stand between one statement and the next.

THE INVISIBLE MAN RETURNS

If only it were that simple. Aggravated beginners will be quick to point out that Turbo Pascal will not complain if they put a semicolon after the **42** anyway. True, true. To explain the sense in that, I have to point out that the following will compile just as readily:

```
BEGIN
  I := 11;
  J := 17;
  K := 42;;;;;;;;;
END
```

Can there be couplers without boxcars? Yes, if they are *invisible* boxcars....

The culprit here is a Pascal abstraction called the *null statement.* Part of the syntactic rules of Pascal enable an "invisible" statement to exist anywhere an ordinary statement can exist. It's not a matter of some unseen special character the way a TAB or BEL character is unseen; it's very much there in your file. The null statement is simply nothing at all. Between each of the multiple semicolons after the **42** is a null statement. If there were only one semicolon after the **42** (where a newcomer frequently places one) it would serve to separate the statement **K:=42** from a null statement immediately following it. The null statement does nothing and generates no code.

Null statements seem pointless at first glance until you understand why they are necessary. The null statement is Pascal's NO OP (No Operation, from the famous 8086 machine instruction that does nothing.) It holds a place in a program where a statement can legally go but where no statement is wanted.

For example, consider this statement:

```
IF NOT DisketteIsInserted
  THEN RequestADisk
```

In a hypothetical program, the Boolean variable **DisketteIsInserted** contains a value of **True** any time there is a diskette in a given drive. The procedure **RequestADisk** asks the user to insert a diskette in the drive. The logic of the above statement is plain, but it would be plainer to eliminate the **NOT**:

```
IF DisketteIsInserted
  THEN {NULL}
  ELSE RequestADisk
```

This reads less like pig latin and better follows the conventional logic of human conversation: If the diskette is inserted, do nothing; otherwise insert a diskette.

The comment **NULL** is there for readability's sake only. The compiler ignores it. Between the **THEN** and **ELSE** reserved words, however, is a genuine null statement. Pascal allows them primarily to enable more flexibility in writing structured statements like the (admittedly simple) example.

The previous example only shows why the null statement is supported; it doesn't address semicolons. Because there are no other statements between **THEN** and **ELSE** there is no need for a semicolon.

A better, if more complex, example of null statements in use involves the **CASE** statement. Con-

continued on page 50

A DEAL YOU CAN'T REFUSE!

FREE* **Turbo C®** (Borland) or **FREE*** **QuickC™** (Microsoft) or **FREE*** **Norton** on Line Programmers Guide for C™

* *If you ALREADY own one, get a* ***FREE REFUND****... See special offer.*

C *WHY* YOU CAN'T REFUSE

- **A C COMPILER** *without a good add-on library is like a PC without a keyboard... it won't do what you want it to do.*
- **GAIN C POWER** *Add capabilities your compiler library does NOT have, e.g.:*
  - *Easy menuing*
  - *Flexible powerful windowing*
  - *Powerful cursor, video and attribute control*
  - *Time and date arithmetic*
  - *Sample code and abundant examples*
  - *Advanced string manipulation*
  - *Formatted data entry*
    - *— display default values*
    - *— calculator style entry option*
  - *500 functions you need*
- **SAVE TIME, TIME, TIME:** *man-years on development, calendar months on schedule!*
- **SMALLER PROGRAM SIZE:** *your application code can be up to 100% smaller!*
- **EASY** *for beginners!* **POWERFUL** *for professionals!*
- **INSTANT INSTALLATION UTILITY** *included!*
- **SUPERB DOCUMENTATION:** *time saving, helpful, clear, complete, instructive.*
- **BUSINESS USERS: FREE** *3 machine site license (C Library & Power Windows).*
- **FULL SOURCE CODE! NO ROYALTIES** *on products you develop.*
- **FREE UTILITY:** *To convert Turbo Pascal code to C code.*
- **FREE:** *Either Turbo C, or QuickC or Norton On-Line Programming Guide for C.*

SAVE MONEY! **SAVE TIME!** **DON'T WAIT** **ORDER NOW!**

SATISFACTION GUARANTEED (Limited time offer... Direct from Entelekon only) **CALL (713) 468-4412**

POWER WINDOWS™

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

Many types of menus. Highlighting. Move data between files, keyboard, program and windows. Status lines. Change size/location/overlapping. Move/add/delete/cascade windows.

5 disks........................$159.95

C FUNCTION LIBRARY

BEST YOU CAN GET
HUNDREDS OF FUNCTIONS
FULLY TESTED
BETTER FUNCTIONS

Most complete screen handling plus graphics; cursor/keyboard/data entry, 72 string functions with word wrap; status and control; utility/DOS BIOS/time/data functions; printer control & more. Special Functions.

7 disks........................$159.95

C BUSINESS LIBRARY

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

ALL for $299.95
(A $500.00 VALUE)

B-TREE LIBRARY & ISAM DRIVER

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

Fast B-tree indices. Add/remove keys Find first/last/next/any key. Find keys by Boolean selection. Read/write/delete or add records to file.

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

C STARTER PACKAGE

INCLUDES C FUNCTION LIBRARY, POWER WINDOWS, SUPERFONTS FOR C

ALL for $199.95
(A $370.00 VALUE)

* SPECIAL OFFER

Free Turbo C or QuickC or Norton On Line Programmers Guide for C with purchase of C Starter Package, C Business Library, C Function Library or Power Windows. Even if you already own Turbo C or QuickC or Norton On Line Programmers Guide for C, we will refund up to the full purchase price of one of these packages with the purchase of C Starter Package or C Business Library.

Entelekon™

SINCE 1982

12118 Kimberley, Houston, TX 77024
713-468-4412
VISA-MASTERCARD-CHECK-COD

sider this statement:

```
CASE CourseGrade OF
   'A': PrintCommendation;
   'B': ;
   'C': ;
   'D': PrintWarningNote;
   'F': PrintFailureNote
   ELSE GradingSystemError
END
```

In a hypothetical grading system for a school, this **CASE** statement decides what action to take based on a course grade value. An 'A' grade generates a printed commendation, while a 'D' generates a warning note and an 'F' a notification of course failure. No action needs to be taken for 'B' or 'C' grades. Grades other than 'A', 'B', 'C', 'D', and 'F' are undefined and generate an error message indicating the grade was entered incorrectly.

Note that for 'B' and 'C' grades there is only a semicolon after the case label. These semicolons are *required* to avoid a syntax error, because every case label must have a statement. Here, case labels 'B' and 'C' have only null statements, but null statements are just as valid as "real" statements. They indicate to the compiler (and to the human being who has to read the code) that nothing need be done.

Another interesting thing about the **CASE** statement example appears when you try to add multiple null statements after one of the existing statements, much like we did earlier after the **K:=42** statement:

```
CASE CourseGrade OF
   'A': PrintCommendation;
   'B': ;
   'C': ;
   'D': PrintWarningNote;;;;
   'F': PrintFailureNote
   ELSE GradingSystemError
END
```

Turbo Pascal will kick this out as a syntax error. Why? Well, within a **CASE** statement, each case label must have either a single statement or a compound statement bounded by **BEGIN** and **END**. **PrintWarningNote;** by itself is a single statement. **PrintWarningNote;;** (which has a null statement between the two semicolons) is neither a single statement nor a compound statement, and thus generates a syntax error.

Null statements are often visible only by virtue of a semicolon, so understanding null statements and where they are allowed can only help you understand the placement of semicolons. Again, as with the first example using the **IF** statement, I recommend using a comment like **{NULL}** to mark the position of a necessary null statement. It's rather like the bandages on the Invisible Man; they're there to keep the guy out of trouble.

IF, IF IF IS NECESSARY, YOU GO ASTRAY...

One of the most common semicolon errors, especially for Pascal newcomers, is in the **IF** statement. Like any statement, an **IF** statement must be separated from any following statement with a semicolon. Unlike most Pascal constructs, **IF** has an optional portion, signaled by the reserved word **ELSE**. The **ELSE** portion of the statement is often put on a separate line for neatness' sake, consequently newcomers sometimes code up something like this:

This rule, at least, is fairly simple: *A semicolon immediately before an* ELSE *keyword is always an error.*

```
IF TankIsFull THEN Agitate;
  ELSE AddReagent;
```

Tidy as it looks, and as much as it resembles normal Pascal practice, Turbo Pascal won't care for this at all. What we have here is *one statement*, and the semicolon after **Agitate** tries to separate it into two. The **ELSE** portion cannot stand alone and therefore becomes an error. This error is so common that it has been named "freestanding else" by harried CS101 professors.

This rule, at least, is fairly simple: *A semicolon immediately before an* **ELSE** *keyword is always an error.*

The **IF** statement situation is worsened by the fact that **IF** statements are frequently nested. Unless there are compound statements embedded within a nested **IF** statement, there is no need for semicolons. This may at times produce the somewhat anomalous situation where code runs for lines and lines and lines without any semicolons at all. Listing 1 is a good example. What looks like a fairly complicated function body is actually only two distinct statements: One short I/O port read followed by a whopper of a nested **IF** statement. The **IF** statement is the tinted portion of the listing.

Function **Button** reads the current state of the buttons attached to either of the two PC joysticks. (It is thus similar to the **STRIG** function in Turbo Basic.) There are two possible joysticks, with each supporting two possible buttons. The current state of all four possible buttons is kept in the high four bits of the byte stored at I/O port $201.

Button is passed a joystick number (1 or 2) in **StickNumber** and a button number (1 or 2) in **ButtonNumber**. What **Button** does, functionally, is single out the one requested bit of the four button bits and translate the state of that bit (either off or on) to a Pascal Boolean value of either **True** (if the button is pressed) or **False** (if the button is not pressed.) Numbers greater than two passed in either parameter will guarantee a Boolean return value of **False**.

The algorithm might be called "divide and question." We first ask of the parameters, "Is it stick #1?" If stick #1 is in fact the joystick we wish to test, we then ask, "well, then, is it button #1?" If **ButtonNumber** is indeed 1, we can then test the state of the bit belonging to stick #1, button #1 by masking out the appropriate bit and comparing the bit to zero. Note that the bits are "active low;" when a button is pressed, its bit is set to zero, *not* one.

If it is *not* stick #1, it will then be either stick #2 or an invalid stick number. And, if it is not button #1, it will be either button #2 or an invalid button number. For each of the tests there are two possibilities. This is a situation tailor-made for nested **IF** statements. Turbo Pascal "understands" when a statement in the next level of nesting begins because of the structure of the **IF** statement: After **THEN** begins a new statement; and after **ELSE** begins a new statement. And because the final "action" to convert a bit number to a Boolean value is a single assignment of a Boolean expression, no compound statements are involved. And no semicolons.

There is probably some additional semicolon arcana lurking about the corners of the Pascal language definition, but we've looked at most of it. To review:

1. Semicolons separate statements in a compound statement; that is, a series of statements bounded by **BEGIN** and **END**.
2. Line structure and source code formatting are ignored by the compiler. Carriage returns tell the compiler *nothing* about where statements begin and end. Semicolons do that.
3. The "null statement" is a symbolic abstraction that can exist legally anywhere a real statement can. This is why you can have a semicolon immediately before the **END** reserved word; the semicolon separates the last real statement in the compound statement from an "invisible" null statement.
4. A semicolon immediately before **ELSE** is *always* an error.

You needn't commit any of this to memory as long as you fathom the sense of what semicolons are for. Look for the higher principles in the language syntax, where you'll find that Pascal nearly always makes perfect sense. Conversely, trying to memorize a blizzard of detail usually leads to headaches. Finally, you can always learn these things (as I did) by tearing your hair out, so long as you remember (as I failed to do) that eventually you run out of hair. ■

Listings may be downloaded from CompuServe as SEMICLN.ARC.

LISTING 1: BUTTON.SRC

```
{----BUTTON.SRC---------------------------------------------------}
{                                                                 }
{    by Jeff Duntemann                                            }
{    Turbo Pascal 4.0 -- Last update 9/22/87                      }
{    For "Sense and Semicolons" -- TURBO TECHNIX V1#1             }
{    Adapted from TURBO PASCAL SOLUTIONS by Jeff Duntemann        }
{                                                                 }
{    BUTTON.SRC reads the state of the joystick buttons.          }
{    It does *NOT* read the XY value of the joystick itself--     }
{    for that you need assembly language.                         }
{                                                                 }
{    The button switch states are maintained in the high 4        }
{    bits of input port $201 for both PC-supported joysticks.     }
{    The bitmap looks like this:                                  }
{                                                                 }
{    |7 6 5 4 3 2 1 0|                                            }
{     | | | |                                                     }
{     | | | -----------------> Button #1, joystick #1             }
{     | | ------------------> Button #2, joystick #1              }
{     | --------------------> Button #1, joystick #2              }
{     ----------------------> Button #2, joystick #2              }
{                                                                 }
{    The low four bits are used to test the XY values; we         }
{    ignore them here.                                            }
{                                                                 }
{    One important thing to keep in mind is that a LOW (0) bit    }
{    indicates a button DOWN and a HIGH bit (1) indicates a       }
{    button UP.  That's why we test against a 0 bit rather        }
{    than a 1 bit.                                                }
{-----------------------------------------------------------------}

FUNCTION Button(StickNumber,ButtonNumber : Integer) : Boolean;

VAR
  PortValue : Byte;

BEGIN
  PortValue := Port[$201];    { Read the joystick I/O port }
  IF StickNumber = 1 THEN     { For joystick #1 }
    IF ButtonNumber = 1 THEN
      Button := ((PortValue AND $10) = 0)
      ELSE
        IF ButtonNumber = 2 THEN
          Button := ((PortValue AND $20) = 0)
            ELSE Button := False
  ELSE
    IF StickNumber = 2 THEN   { For joystick #2 }
      IF ButtonNumber = 1 THEN
        Button := ((PortValue AND $40) = 0)
        ELSE
          IF ButtonNumber = 2 THEN
            Button := ((PortValue AND $80) = 0)
              ELSE Button := False
    ELSE Button := False
END;
```

TAKING CHARGE OF DOS VOLUME LABELS

Handle disk volume labels without handles—use Extended FCBs instead.

Kent Porter

WIZARD

Of all the many features and conveniences DOS offers, probably none is as neglected as the volume label. It's not hard to see why; DOS works just as well with disks that don't have one as with those that do. In fact, DOS doesn't even do anything with the label except to display it (or a message saying there isn't one) when you list the directory, and there are no DOS calls that specifically manipulate the label. So why bother with it?

Labels can be useful for a number of things. One example is detecting when a diskette has been switched while files are still open. Another is in safeguarding sensitive information by refusing to write to a disk that lacks the right label. This article examines some uses for labels, and shows how to read, write, and modify them from within Turbo C programs.

First let's pin down what a label is, and how it's stored on disk. There's nothing magical about a label. It's simply a special entry in the disk's root directory. All file entries in DOS 2.0 and higher carry a byte describing the file's attributes (normal, hidden, read-only, etc.). A label is a directory entry with an attribute byte whose value is 08H. No space is allocated to the entry. It's much like a file that was created and then closed without any data being written to it. The only space a label occupies, then, is the 32 bytes taken by each directory entry.

This explains why a label is limited to 11 characters. In the directory, DOS reserves 11 bytes for a file name, with the name itself being left-justified and the extension right-justified. DOS bypasses this convention when storing and retrieving a label, instead left-justifying the label in the 11 characters and padding to the right with spaces to fill unused positions. You can have up to 11 characters in a label because that's how many bytes there are in a directory entry.

The attribute byte governs how DOS treats each entry. When you list the directory, you only see normal and read-only files. The DIR command skips over those with other attributes. That's why IBMBIO.COM, IBMDOS.COM, and other hidden and system files don't show up in the listing. Neither does the label, except in a special message at the top.

Most of the DOS file-management routines, and all of them in Turbo C except for **findfirst()** and **findnext()**, don't deal with files having unusual attributes. You have to work at it when fooling with the label, and writing a label should be considered armed and dangerous inasmuch as it can potentially maim the directory. Later we'll deal with safe methods for manipulating labels from within Turbo C programs.

Meanwhile, let's consider some uses for labels.

WHAT ARE THEY GOOD FOR?

The mainframe world has been using volume labels for years. No doubt that's where the idea came from when Microsoft set out to create DOS. They're particularly useful for identifying removable media such as disk packs and diskettes, but they can also serve to distinguish among fixed media such as multiple hard disks. In short, a label answers the question, "Who are you?"

There are several reasons you might want to ask this of a disk. One is to determine whether a diskette has been switched in the drive before writing to an open file. I have horror stories, and probably most other heavy PC users do too, about switching disks to save multiple copies of a file being edited. Later you discover that the disks are corrupted beyond redemption. If every diskette had a unique label and the program checked the label before writing, this wouldn't happen. Later we'll discuss the problem of unlabeled disks (which are indistinguishable from one another) and what to do about them in this context.

Another application for labels is to protect sensi-

Bradley Ream

LABEL

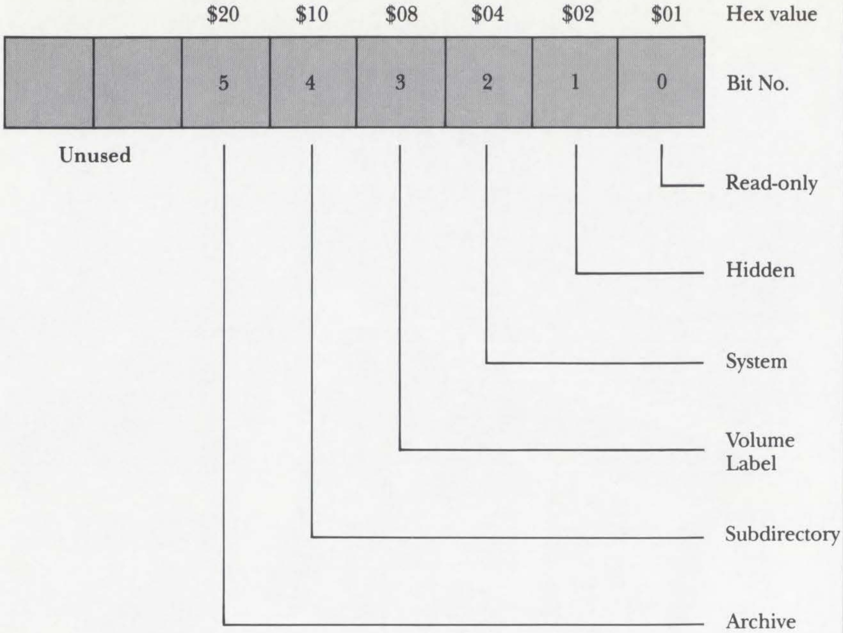


Figure 1. DOS file attribute bits.

tive data. Software, by and large, is naive about this. It cheerfully writes anything on any disk. An electronic prowler can thus bring up an application, save the company's most intimate secrets on any old disk, and take them away to be analyzed. Making a program label-sensitive isn't a cure-all for industrial espionage, but it can help. Develop a sensible, harmless-looking labeling system for diskettes containing sensitive data, keep those diskettes locked up, and make the program that reads and writes them check labels. If a program balks at writing to the intruder's diskette because it has the wrong label, he or she won't have a clue as to why. Especially since it never occurs to anyone using micros that disk labels might be important.

You can also effect a form of copy protection for your software using labels. It works like this: the first time the program runs, it checks the disk label. If there is no label, it creates one. In either case, it modifies itself by initializing a label field within the code space, then saves its own image to disk. Thereafter, each time the program starts, it checks the actual label against the one it expects, and if they don't match, the program aborts.

There are rules about labels, and DOS brooks no argument about them. If you violate the rules, DOS refuses to write out the label, but it doesn't explain why.

No doubt there are other uses for labels as well. Our purpose here is not to present an exhaustive list, but to give you some ideas about how to put this unappreciated DOS feature to work protecting your disks, data, and software.

WHAT TO DO ABOUT UNLABELED DISKS

One unlabeled disk looks the same as the next, and since few DOS users bother to label diskettes, this makes it impossible to detect if diskettes have been swapped in the drive. Consequently, if your program is going to check for switched diskettes before writing to an open file, it must first determine the label of the diskette from which it initially read the data. When no label exists, it must create one that has a good chance of being unique.

Suggestion: Make a string out of the system time and label the diskette with it. By going down to the level of hundredths of a second, there is virtually no chance that any two diskettes will ever be assigned the same label.

Having so tagged the diskette, the program can then check the current label against the "read-from" label before performing a write, and if they're not the same, tell the user to put the correct disk back in the drive. This is a sensible way to avoid the idiotic corruption of diskettes.

DOS IS FINICKY

There are rules about labels, and DOS brooks no argument about them. If you violate the rules, DOS refuses to write out the label, but it doesn't explain why. Its only response is to return 0FFH in the AL register, meaning that the operation was unsuccessful.

Because a label is a kind of file name, you can only use characters that are valid within the name of any DOS file. The DOS manual spells them out; in general they're any printable character, minus the wildcard characters "?" and "*". The 11-byte field must be padded with trailing spaces.

These restrictions pose a special problem for C programs, since C always terminates a string with an ASCII null or zero (**\0** in C notation) and leaves garbage after it. Consequently, after loading the label into the file control block and before asking DOS to put it on the disk, you have to replace the null terminator and any other garbage in the 11-byte field with spaces. The **pad()** function in Listing 1 performs this service.

A label can only have a meaningful existence in the root directory. Theoretically it's possible to copy one to a subdirectory, but

DOS would never notice it and your software would have to go to extraordinary lengths to find it. It's been reported here and there that DOS has bugs with regard to labels, which can cause it to garble directories, lose files, and even render the disk useless. For these reasons, it's advisable to limit a disk to one label properly placed in the root where it belongs. Manipulate the label using the "safe" procedures outlined next and shown in the VOLUME.C program in Listing 1.

OPERATING ON DISK LABELS

DOS provides two sets of functions for operating on files. One set came into existence with DOS 2.x and uses an integer called a file *handle* to identify a given open file; the handle routines accommodate tree-structured directories. The other set of file-handling functions are holdovers from DOS 1.x, which in turn were inherited from CP/M-80. It is these low-level functions, specifically 11H, 16H and 17H under interrupt 21H, that must be used to operate on labels. These functions only work on the root directory, which explains why a label is meaningless anywhere else.

The low-level functions use a structure called a *file control block* (FCB) instead of a handle. There are two types of FCBs, the standard format and a larger version called an extended FCB, or XFCB. The extended format is just an ordinary FCB preceded by an eight-byte header containing, among other things, an attribute byte. DOS functions 11H, 16H, and 17H recognize which FCB format they're dealing with by checking the first byte; it's always 0FFH in an XFCB.

You have to use an XFCB when working with a label, since attribute byte 08H is a necessary part of the label format. Turbo C furnishes definitions of the FCB and XFCB in the include file DOS.H, and the Turbo C function **findfirst()** can be used to read a label. However, there is no function provided with Turbo C to change a label. Because these are risky operations that require careful control, it's best to use the Turbo C **intdos()** function with appropriate register setups.

READING A LABEL

To read a label, use DOS function 11H. This function refers to an initialized XFCB to find out what you want to look for in the root directory. If it finds a match, it writes the corresponding XFCB to a separate 64-byte buffer called the DTA, for *disk transfer address*, and returns zero in register AL to indicate success. If there is no match (no label in this case), function 11H returns 0FFH and the DTA is unchanged. Following a successful read, you can fetch the label from offset 08H of the DTA.

You have to use an XFCB when working with a label, since attribute byte 08H is a necessary part of the label format.

Thus, before looking for the label, you have to do some setup. The first thing is to develop the DTA structure. For reasons we'll discuss later in connection with changing a label, this is not merely a copy of the XFCB definition given in DOS.H. An entry under GLOBALS in the heading of Listing 1 defines the structure variable **dta**.

Next you need to initialize a copy of the XFCB format as shown in the declaration of **struct xfcb fcbx** in the program. The variable **fcbx** thus has its first byte set to 0FFH to identify an XFCB to DOS, the reserved prefix field is set to zeros, and the attribute byte is set to 08H for a label (constant **FA_LABEL** is defined in DOS.H). Now you can begin execution.

The first executable step is to tell DOS where the DTA is. Do this by passing a far pointer to the Turbo C **setdta()** function, indicating the segment and offset of the structured **dta** variable. Next put the disk drive indicator into the **fcb_drive** field of structured variable **fcbx**. Finally, fill the file name and extension fields of **fcbx** with the wildcard character "?", which tells function 11H to return whatever name it finds in association with a label entry.

Now you can actually search for the label. Function **gotlabel()** in the program performs this task. Put the offset of **fcbx** into register DX, function 11H into AH, and call DOS with **intdos()**. If register AL contains 0FFH on return, DOS did not find a label, and a zero in AL means that it loaded the label's XFCB into the DTA.

Based on a simple test of register AL, you can either fetch the label name from **dta.oldlabel**, or determine that the disk is unlabeled. What you do from that point depends on your application: VOLUME.C reports the outcome and asks the user if he or she wants to write a new label, then proceeds based on the reply to the query, and on whether or not the disk is already labeled. A less interactive program might act according to different decision rules, simply storing the label if one exists and writing one if it doesn't, without asking the user's permission or preference.

CHANGING AN EXISTING LABEL

If you want to change an existing label to something else, proceed as though you're renaming a file using DOS function 17H. Turbo C has no equivalent that provides the necessary level of control for labels.

DOS function 17H requires deliberate corruption of the XFCB in the DTA. Write the new label starting six bytes after the end of the old label (field **newlabel** in the **dta** structure), and pad the remainder of the field with spaces.

LISTING 1: VOLUME.C

```
/* VOLUME.C: Reads and writes volume label */

/* INCLUDES */
#include <stdio.h>
#include <dos.h>
#include <string.h>

/* CONSTANTS */
#define PROMPT "\nLabel is limited to 11 characters:"
#define BEEP  7
#ifndef TRUE
#define TRUE  1
#define FALSE 0
#endif

/* LOCAL FUNCTION PROTOTYPES */
char gotlabel (void);
void change (void);
void addlabel (void);
void pad (char name[]);

/* GLOBALS */
union REGS   reg;                                       /* register set */
struct {                        /* simulated xfcb loaded by DOS fcn 11h */
  char      skip1[8];                             /* space to old label */
  char      oldlabel[11];                                  /* old label */
  char      skip2[5];                             /* space to new label */
  char      newlabel[11];                                  /* new label */
  char      skip3[29];                  /* to make struct 64 bytes long */
} dta;
struct xfcb fcbx = {{0xFF},         /* initialize xfcb for label search */
                      {"\0\0\0\0\0"},                 /* reserved zeros */
                      {FA_LABEL},              /* set label attribute */
                    };
/* -------------------------- */

main ()
{
char  exists;

/* INITIALIZE */
  setdta (MK_FP (_DS,(unsigned) &dta));    /* dta = scratch buffer */
  puts ("\nGet volume label from which drive?");
  fcbx.xfcb_fcb.fcb_drive = toupper (getch ()) - '@'; /* get drive */
  strcpy (fcbx.xfcb_fcb.fcb_name, "???????????");  /* set wildcard */

/* PROCESS */
  if (exists = gotlabel ())
    printf ("\nLabel of drive %c is %.11s",
               fcbx.xfcb_fcb.fcb_drive + '@', dta.oldlabel);
  else
    printf ("\nDrive %c has no label",
               fcbx.xfcb_fcb.fcb_drive + '@');
  puts ("\n\nDo you want to write a new label? (y/n)");
  if (toupper (getch ()) == 'Y')
    if (exists)
      change ();
    else
      addlabel ();
} /* ------------------------ */
```

DOS VOLUME LABELS

Because the DTA's **attribute** and **oldlabel** fields are already set (by function 11H), no other change to the DTA is required.

The **change()** function in the listing shows what to do next. Load the offset to the **dta** variable into register DX, DOS function 17H into AH, and call **intdos()**.

On return, register AL contains the result. It's zero if successful, and potentially a catastrophe if it's 0FFH. Any time this call is unsuccessful, immediately inspect the involved disk for damage to the FAT and root directory.

There are three probable causes for failure: you're trying to change a label that doesn't exist; your program set up the **newname** field in the DTA incorrectly; your copy of DOS is corrupted. In the first two cases, DOS simply refuses to process the request and never goes to the disk, so damage is unlikely.

On return, DOS returns zero in register AL for a successful save and 0FFH if unsuccessful.

LABELING AN UNLABELED DISK

Never assume that a disk is unlabeled. Instead check first for the existence of a label by attempting to read it. If you get 0FFH back in register AL, you know for certain that the disk is unlabeled. The reasons for this are twofold. DOS may suffer a nervous breakdown over having two labels in the same directory, and even if it doesn't, it will only recognize the first one it finds and ignore the second.

DOS function 16H creates a

new file, and that's the function you use to write a label. In changing the label as discussed above, you use the DTA; to write a new label, instead use the same XFCB structure you set up for the search with function 11H. As the listing for function **addlabel()** shows, copy the label into the file name field at offset 08H (field **fcbx.xfcb__fcb.fcb__name** in the program). Although this field is officially eight bytes long, followed by another three for the extension, Turbo C will let you copy up to the full 11 characters into the name field. Don't forget to pad the unused bytes with trailing spaces, lest DOS choke on the invalid null terminator. Nothing else needs to be done to prepare the XFCB.

Write the label into the root directory by placing the offset of the structured **fcbx** variable into register DX, DOS function 16H into register AH, and issuing **intdos()**.

On return, DOS returns zero in register AL for a successful save and 0FFH if unsuccessful. Failure usually occurs for one of two reasons: an improperly developed XFCB, or a full directory.

CONCLUSION

While at first glance labels don't seem to have much purpose, in fact they can be used creatively by advanced programs as a tool for protecting disks, data, and even software. Because DOS furnishes no specific calls for operating on labels, it's necessary to develop one's own. Writing labels to disk can be perilous unless you apply common sense and a bit of care, and then it can be a simple and safe operation. The methods furnished here should provide useful and reliable techniques for taking advantage of this ignored feature of DOS. ■

Kent Porter is a professional writer specializing in software. His most recent book is Stretching Turbo Pascal *(Brady/Simon & Schuster), and he's now at work on an advanced Turbo C book.*

Listings may be downloaded from CompuServe as DOSLABEL.ARC.

```
char  gotlabel (void)                              /* read label from disk */
                                         /* returns TRUE if found, else FALSE */
{
  reg.x.dx = (unsigned) &fcbx;                          /* point DX to fcbx */
  reg.h.ah = 0x11;                           /* DOS fcn: search for first */
  intdos (&reg, &reg);                                          /* call DOS */
  if (!reg.h.al)                                 /* AL = 0 when successful */
    return (TRUE);
  else                                    /* non-zero AL means no label found */
    return (FALSE);
} /* ------------------------ */

void  change (void)                                    /* change disk label */
{                                          /* same DOS call as renaming file */
  puts (PROMPT);
  scanf ("%11s", dta.newlabel);            /* ignore more than 11 chars */
  pad (dta.newlabel);                        /* pad with trailing spaces */
  reg.x.dx = (unsigned) &dta;                        /* point to dta buffer */
  reg.h.ah = 0x17;                                   /* DOS fcn: rename file */
  intdos (&reg, &reg);                                          /* call DOS */
  if (!reg.h.al)                                 /* AL = 0 when successful */
    printf ("\n\nLabel successfully changed to %.11s",
            dta.newlabel);
  else {
    puts ("\n\nUnsuccessful! Disk may be damaged");
    putchar (BEEP);
  }
} /* ------------------------ */

void  addlabel (void)                           /* label an unlabeled disk */
{                                       /* same DOS call as creating new file */
  puts (PROMPT);
  scanf ("%11s", fcbx.xfcb_fcb.fcb_name);                /* get 11 chars */
  pad (fcbx.xfcb_fcb.fcb_name);              /* pad with trailing spaces */
  reg.x.dx = (unsigned) &fcbx;             /* have to use xfcb for this */
  reg.h.ah = 0x16;                                   /* DOS fcn: create file */
  intdos (&reg, &reg);                                          /* call DOS */
  if (!reg.h.al)                                 /* AL = 0 when successful */
    printf ("\n\nDisk is now labeled %.11s", fcbx.xfcb_fcb.fcb_name);
  else {
    puts ("\n\nUnsuccessful! Root directory may be full");
    putchar (BEEP);
  }
} /* ------------------------ */

void  pad (char name[])               /* pad filename with trailing spaces */
{                              /* DOS chokes on C null terminator in filename */
int   p;

  if (strlen (name) < 11)
    for (p = strlen (name); p < 11; p++)
      name[p] = ' ';
} /* ------------------------ */
```

POINTERS IN TURBO C

The secret to efficient data access is simply this: point!

Mike Floyd

SQUARE ONE

Man has used them since the beginning of time. Trackers hacked them into the sides of trees so they wouldn't get lost. North American Indians used them as a road sign by stacking rocks to point the way. And early astronomers used the Big Dipper as one to locate Polaris (the North Star). So what are they? Pointers.

Pointers are markers that show where something is. In a programming language, a pointer is a variable whose value is the address of an object rather than the actual object. In a language like Turbo C, you will soon see that they can be an invaluable tool to the programmer.

This article covers some basics about memory addresses and how to use pointers. I'll show you how a pointer can be used to get a value as well as how to get the *address* of a value. Finally, I will talk about some of the reasons you would consider using them.

MEMORY ADDRESSING

Among novice C programmers, there's a great deal of confusion about pointers. Because many high-level languages don't allow the programmer to access addresses, even seasoned programmers find pointers a bit troublesome. To get a better understanding of how pointers work, let's first take a look at how values are addressed.

Figure 1 shows how RAM is divided. Notice that the operating system is located in the lowest portion of RAM. Next is what's known as the *transient program area* (TPA) where the executable program resides. Finally, any memory not used by the operating system or the program is left as free memory. As the program runs, some of this free memory is used for dynamic data storage (for variables, constants, etc.).

Remember that the smallest unit in this scheme is known as a bit, which has one of two values: 0 or 1. Also recall that eight bits make up a byte and two bytes make up a word. Further, four bytes make a long word and 16 bytes make a paragraph on the PC. As shown in Figure 2, each byte has an address.

As an example, consider the following program fragment where two values are declared: an integer **i**, and a character **c**:

```
int i;
char c;
i = 5;
c = 's';
```

Once declared, we assign the value **5** to the variable **i**, and the value **'s'** to the variable **c**. When these variables are declared, the compiler determines how much memory must be set aside to store these values. Figure 3 shows how these values are addressed in memory, assuming the starting address of integer **i** begins at 65000 and the address of character **c** starts at 65010.

USING POINTERS

As we said before, a pointer is simply a special kind of variable that contains the address of a value rather than the actual value. Pointers may be incremented, decremented, and assigned new values. Since we are accessing the value indirectly through the use of pointers, this process is called *indirection.* To declare a pointer, we use the indirection operator "*":

```
int *ip
```

The **int** tells the compiler what type of data the pointer will point to (integer in this case). The indirection operator tells the compiler to first create an address for the pointer, then mark the variable **ip** as a pointer.

Like any variable, a pointer must be initialized once it's declared. This is done by assigning the address of a variable to the pointer variable. The *address-of* operator "&" is used to get the address of the variable. So, given that the variables **i** and ***ip** have been declared, we initialize **ip** by assigning the address of **i** to it:

```
ip = &i
```

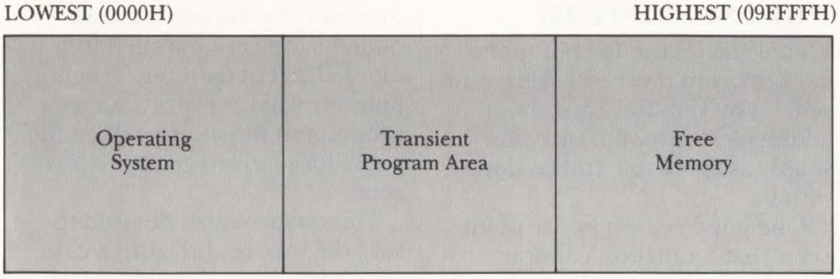


Figure 1. Memory layout on the IBM PC.

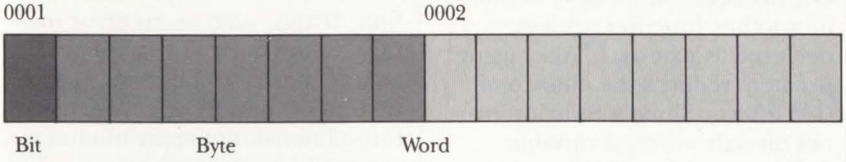


Figure 2. Basic units of memory.

65000
65001
65010
0
5
s

Figure 3. Storage and addressing of values in memory.

Now we can start using the variable **ip** in our program, but remember, **ip** contains an address. To get the actual value we must use the indirection operator "*", as shown here

```
main() {
   int i,  *ip;
   ip = &i;
   i = 5;
   printf
     ("The value of i= %d\n",*ip);
   printf
     ("The address of i= %p\n",ip);
}
```

which will print:

```
The value of i = 5
The address of i = FFD4
```

First we declare the variable **i** and the pointer variable **ip** to be an integer. Next we initialize **ip** to the address of **i**. This is a very important step, and one which many beginners forget.

Once the pointer variable has been initialized, we assign the value **5** to the variable **i** and print out the results. Notice that the first **printf** statement uses the indirection operator to get the value being pointed to by **ip**. Finally, the second **printf** statement prints out the contents of **ip**, which is the address of **i**. Notice in the output that the address is printed out in hex notation. This hex value is the equivalent of 65492.

This simple program demonstrates three important steps to working with pointers. First, declare your pointer variable. Second, initialize the pointer (or the results can be unpredictable). Finally, use the indirection operator "*" to get whatever your pointer is pointing to.

One good reason to use pointers is that a function cannot directly access a variable defined in another function.

FUNCTIONS AND POINTERS

With that little bit under our belts, we are ready to graduate to the next level. There really is no trick to using pointer variables within functions, but they are absolutely essential, as you will soon see. In terms of usage, there is little difference between the way pointer variables are treated and the way regular variables are treated. Consider the following function which calculates a factorial and passes back the result via a pointer variable.

```
void factorial(int num,
                    double *result)
{
   double x;
   int i;

   if(num < 1) {
      *result = 0.0;
      return;
   }

   for (x = 2.0, *result = 1.0;
      num > 1; num--, x = x + 1.0)
   *result = *result*x;
}
/* End factorial */
```

Note in our declaration of factorial that **result** is a pointer variable and that we are passing the value of that pointer (i.e., ***result**). This is essential whenever we wish to change the actual value of a variable defined outside of the function. The reason is that Turbo C, like most C compilers, passes a copy of the variable to the function when called. So, a function cannot directly access a variable defined in another function (unless declared as external). Therefore, without pointers, there would be no way to modify the variable within the function.

Finally, here's how factorial is called.

```
main()
{
   int x;
   double y;
```

```
    x = 6;

    factorial(x,&y);
    printf("%f",y);
}
/* End main */
```

POINTER ARITHMETIC

We can manipulate pointers as well as increment and decrement them. For instance, the pointer **ip** can be incremented

```
ip++;
```

where **ip** has been previously declared as

```
int *ip
```

Similarly, the same pointer **ip** can be decremented:

```
ip--;
```

As an example, consider the following function which passes a string in as an argument and returns the length of that string.

```
str_len(str)
char *str
{

    int i;

    for (i=0; *str != '\0'; str++)
        i++;
    return(i);
}
```

Here, the variable **str** is a pointer to characters. The **for** loop begins by initializing the counter *i* to 0. Next a test for the null character (designating the end of the string) is made and, until successful, increments **str** to point at the next character in the string. When the end of the string is encountered, the length of the string is returned through the function value.

Remember that when you increment a pointer, you are incrementing an address. Keep in mind that this has a different effect on different data types, depending on their storage requirements. In this case we are incrementing a pointer to a character. Since a character only requires one byte of storage, the next address will be the current address + 1. If, on the other hand, we were counting up the number of elements in a list of integers, the next address would be the current address + 2.

WHY USE POINTERS?

With a small taste for how pointers work, you must be asking yourself, "Why should I care about addresses?" After all, we could simply assign **5** to **i** and be done with it.

One good reason to use pointers is that a function cannot directly access a variable defined in another function (unless declared as external). Also, using pointers reduces the chance of side effects. Since a function cannot directly access a variable defined outside of its domain, there is less chance of "contaminating" other variables in the program.

The creation of new data objects is a very important reason to use pointers. Normally, we must declare data objects and structures in our program. Turbo C then looks at these declarations and allocates an appropriate amount of memory for them. This requires that our program know in advance what objects to expect. But Turbo C, through the use of **malloc**, **calloc**, and **free**, allows you to allocate and deallocate memory during the execution of your program. So, by using pointers, it is possible to create new data objects on the fly. This is essential for programming applications such as a relational database.

Many of the built-in library functions in Turbo C make use of pointer variables. If you're planning to take advantage of all of the features of Turbo C, you will want to learn how to use pointers.

The use of pointers can improve the efficiency of your program because they are treated internally by Turbo C. Consequently, they are handled in a more efficient manner. The use of pointers will also speed up your program in most cases, since the compiler tends to generate less code.

Pointers can also be used to save data space. Imagine a case where you must display one of four strings, such as an error message, to the user. For simplicity, let's assume that the string will be no longer than 80 characters. A four-dimensional array of characters could be defined as:

```
char msg[4] [80];
```

This automatically allocates space for a 4 × 80-character array, whether we need all that space or not. In other words, this guarantees that we will use up 320 bytes.

A more efficient way to do this is to declare a pointer to characters:

```
char *msg[4];
```

In this case, our program must allocate space for the array variable **msg**. The advantage is that we can allocate just enough space for the strings to be stored, and not waste unused space. In larger programs where memory space is at a premium, this approach can become very important.

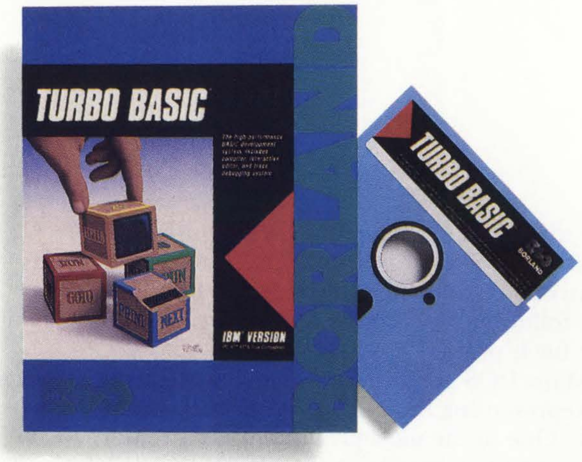
There is one other reason to use pointers in your programs. Many of the built-in library functions in Turbo C make use of pointer variables. If you're planning to take advantage of all of the features of Turbo C, you will want to learn how to use pointers.

There is considerably more to using pointers in Turbo C than we've covered here, but this should provide you with a good foundation. A careful reading of the Turbo C manual and a mastery of pointers will put you well on the way to becoming a C wizard. ■

REFERENCES

Purdum, Jack J. *C Programming Guide*, Indianapolis, IN: Que Corporation, 1987.

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

BASIC Benchmarks

		Turbo Basic 1.0	QuickBasic 3/87
Compile & Link to Stand alone. Exe		*3*	17
Size of .Exe		*32753*	41162
Execution Time	w/80287	*18 secs*	25 secs
	w/o 80287	*109 secs*	114 secs

Benchmark by Jerry Pournelle run on IBM PC/AT with 80287 at 8 MHz with IEEE floating point. Benchmark fills two floating-point matrices with 50 elements, multiplies the two matrices and sums the results. Sum = 23345440.1355113

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

THE END OF THE LINE

It's not easy knowing when you're at the end of the line if the line ends six ways....

Jonathan Sachs

Character I/O is one of the less standardized areas of computing. Because C was developed in Unix, where lines are ended by a single character (**newline**), C represents an end-of-line as a single character when reading or writing a character stream.

This presents a problem when C is ported to systems such as DOS, where a line normally ends with a "carriage return, line feed" (CR/LF) sequence. Simply reading the CR/LF makes sense to DOS programmers, but it doesn't make sense in terms of C. C programs developed in Unix have to be modified to work in DOS, and *vice versa.*

HOW C SOLVES THE PROBLEM...

Implementors of C resolve this problem by translating CR/LF to **newline** when reading a file, and **newline** to CR/LF when writing a file. This translation is done on files that are opened in text mode. For example, it's done when a file is opened with a 't' in the **mode** parameter of the **fopen** command, or when a file is opened without a mode specifier in **fopen**, and the global variable**_fopen** is set to **O_TEXT**. Thus, the statement

```
fputc('\n',file);
```

actually writes *two* characters, a CR and a LF, if **file** represents a text file.

Functions such as **fgetc** and **fgets** translate the other way when reading. Functions such as **fputc** and **fputs** translate **newline** to CR/LF when writing. Functions such as **putchar** and **puts** perform the same translation when writing to **stdout**.

...OR ALMOST SOLVES IT

But translating back and forth between CR/LF and **newline** doesn't end the problems associated with this difference between DOS and C. Unfortunately for C, a line in a DOS file may end in a variety of ways, and the differences among them are significant. For example, if a line ends with a CR alone, that means "carriage return without line feed." In other words, the following line will be superimposed on this one if the file is printed.

Many DOS application programs use different ways of representing lines to convey significant information. One of the most prominent is WordStar, which can end a line in at least six different ways. WordStar's most common distinction is between a hard carriage return, which ends a paragraph (0x0D/0x0A, CR/LF), and a soft carriage return, which may move whenever the text is edited or the margins are changed (0x8D/0x0A, CR/LF with the high-order bit turned on in the CR character).

Sometimes it's useful to be able to read a character stream and recognize all possible kinds of line endings. The **fgetln** function shown in Listing 1 does this.

fgetln COMPLETES THE SOLUTION

The essential purpose of **fgetln** is to read a line from a file. In this, it is similar to the standard library function **fgets**. Here is a summary of the differences:

- **fgets** does not recognize all possible ways of ending a line in DOS, and does not distinguish among the ones it does recognize.
- Because **fgetln** reads carriage returns and line feeds separately, its file handle must be opened in binary mode rather than text mode. Normally you can do this by omitting the 't' in **fopen**'s file mode parameter. If **_fmode** is set to **O_TEXT**, however, the default mode is "text;" then you must include a 'b' in **fopen**'s file mode parameter.
- **fgets** expects a pointer to the buffer where it is to store the line it reads. **fgetln** expects pointers to *two* buffers: one for the line, and the other for the characters that end the line. (The latter buffer must be three characters long, allowing for a maximum of two line-end characters and a NUL.)

- If a line is too long for the text buffer, **fgets** stops reading in the middle of the line and resumes the next time it is called. **fgetln** simply discards the excess. One can argue that the **fgets** approach is superior because data is never lost, but the **fgetln** approach seems to be more useful in most applications.
- **fgets** stores a **newline** at the end of the line it reads (assuming it could fit the whole line in the buffer). This is the only way you can tell whether or not **fgets** was able to read the whole line. **fgetln** does not store the line-end characters in the text buffer.

The program FGETDEMO.C (Listing 2) demonstrates the use of **fgetln**. It prompts you for a file name and opens the specified file. Then it reads each line and displays the line preceded by its length. After each text line it displays a separate line presenting the end-of-line character(s) in hexadecimal form.

HOW fgetln TREATS LINE ENDS

Here is a summary of the rules that **fgetln** uses to identify line ends:

- The following character sequences end a line: CR (0x0D); LF (0x0A); and CR followed by LF. The high-order bit may be on in either character or both.

LISTING 1: FGETLN.C

```
/* FGETLN.C -- read a line from an ASCII file, with limited length.
 It recognizes line-end characters with the parity bit on. */

#include "stdio.h"

#define NUL     ((char)0)
#define LF      ((char)0x0A)
#define CR      ((char)0x0D)
#define EOD     ((char)0x1A)

/*****
*Get a line from a file. This is like fgets(), but:
* 1. Reads to the end of the line even when the buffer won't hold it
*       all.
* 2. Returns the actual (untruncated) line length, not a pointer to
*       the line. Returns EOF if it encounters EOF with no preceding
*       data characters.
* 3. Recognizes line ends that consist of CR or LF alone, and
*       recognizes line end characters with the parity bit on.
* 4. Returns the end-of-line character(s) in a separate buffer.
*IN:    "lim" is the length of "bfr", including space for the
*        terminating NUL.
*       "fp" is the file handle. Note that the file should be opened
*        in binary mode!
*OUT:   "bfr" contains at most lim-1 characters of the next line, NUL-
*        delimited.
*       "eolbfr" contains the character or characters that ended the
*        line, NUL-delimited. If the line was ended by EOF (ctrl-Z or
*        physical end of file), "eolbfr" is null.
*RETURN:The actual (untruncated) length of the line, not including the
*        line end character(s); EOF if EOF.
*****/

int fgetln( bfr, lim, eolbfr, fp )
  unsigned char *bfr;
  unsigned lim;
  unsigned char eolbfr[3];
  FILE *fp;
{

int     i,                  /* A character read from file. */
        j;                  /* "i" with parity stripped. */
register char *cp;          /* Pointer to buffer. */
char    *ep,                /* Pointer to eolbfr. */
        *bfrlimit;          /* Limit of buffer, excluding final NUL. */

/* Set up buffer pointers. */
cp = bfr-1;                 /* Character before start of buffer. */
bfrlimit = cp+lim;          /* Last character of buffer (for NUL). */

/* Each loop reads one character into "bfr." The loop is ended by
 end-of-line or end-of-buffer. On exit, "cp" points to the last
 character stored; to bfr-1, if none. i & j contain the character
 after the last one stored. */
j = 0x7f & ( i = fgetc(fp) );
while (   i!=EOF && j!=EOD && j!=CR && j!=LF   &&   cp<bfrlimit   ) {
  *++cp = i;
  j = 0x7f & ( i = fgetc(fp) );
  }

/* If end of buffer, discard characters until another end condition
 occurs. "cp" is advanced to reflect length of line read, but no more
 data is stored. End status is the same as preceding loop. */
while (   i!=EOF && j!=EOD && j!=CR && j!=LF   ) {
  ++cp;
  j = 0x7f & ( i = fgetc(fp) );
  }

/* Seal end of line. */
*++cp = NUL;
/* Store end of line characters in "eolbfr." */
ep = eolbfr;
if ( j==EOD || j==CR || j==LF )
  /* No physical end of file. */
  {
  /* Store the first EOL character. If it's CR, read the next one to
   look for an LF. */
  *ep++ = i;
  if ( j==CR ) {
    j = 0x7f & ( i = fgetc(fp) );
    /* If the 2nd one is LF, store it too; else put it back. EOF
     isn't a character, & since it will repeat, need not be
     "put back." */
    if ( j==LF )
      *ep++ = i;
    else if ( i!=EOF )
      ungetc(i,fp);
    }
  /* Put back an EOD character at the end of a non-null line. Return
   EOF with a null line the next time this function is called. */
  else if ( j==EOD )
    ungetc(i,fp);
  }
/* Seal the EOF string. */
*ep = NUL;

/* If any characters were read, return the number. */
if (cp>bfr)
  return(cp-bfr);
/* No characters were read. If the EOL buffer is empty, or contains
 EOD, return EOF. */
if ( ep==eolbfr || *eolbfr==EOD )
  return(EOF);
/* We read a null line. */
return(0);
}
```

LISTING 2: FGETDEMO.C

```
/* FGETDEMO.C -- test & demonstrate fgetln(). */

#include "stdio.h"

int fgetln( unsigned char *bfr, unsigned lim,
                unsigned char *eolbfr, FILE *fp );


main()
{
FILE *fp;
unsigned char workarea[73],eolarea[3];
int len;

fputs( "What file shall I read? ", stdout );
gets(workarea);
fp = fopen(workarea,"rb");
if (!fp) {
  fputs( "That file doesn't exist!\n", stdout );
  exit(0);
  }
while (     ( len = fgetln(workarea,sizeof(workarea),eolarea,fp) )
                !=     EOF
          ) {
  printf( "%5d <%s>\n\t\t%02X %02X\n", len, workarea, eolarea[0],
                eolarea[1] );
  eolarea[0] = eolarea[1] = 0;
    }
printf( "***EOF***\n" );
}
```

END OF THE LINE

- An end-of-file condition or an end-of-data character (Ctrl-Z, 0x1A) also ends a line. 0x9A (Ctrl-Z with the high-order bit on) is treated as an ordinary data character.
- If **fgetln** encounters end-of-file or end-of-data at the end of a non-null line, it returns the line. For end-of-file, the end-of-line buffer contains a null string. For end-of-data, it contains a one-character string consisting of an end-of-data character. If **fgetln** encounters end-of-file or end-of-data *after* the end of a line, it returns an end-of-file condition immediately.
- When **fgetln** returns an end-of-file condition, the end-of-line buffer contains either a null string or a 0x1A to indicate whether the function encountered end-of-file or end-of-data in the file.

HOW fgetln WORKS

The overall strategy of **fgetln** is quite simple:

1. Read as much of the line as fits in the text buffer.
2. If the line overflows the buffer, read and discard the rest.
3. Analyze the end-of-line characters and store an appropriate value in the end-of-line buffer.

There is one unusual feature of the code that deserves mention: the character-reading loops start with the buffer pointer positioned to the character *before* the first character of the buffer, and end with the pointer at the *last* character position stored, rather than the position after it. The loop is slightly more efficient when written this way. Since repeated calls to **fgetc** make **fgetln** rather slow in any case, the trickiness of these loops seems worthwhile. ■

Jonathan Sachs has worked as a software developer and technical writer since 1971. He operates a consulting company near San Francisco.

Listing may be downloaded from CompuServe as LINEND.ARC.

IMPLEMENTING BINARY TREES

Plant a pointer, water recursively, grow a tree.

Kent Porter

PROGRAMMER

My wife, a student of human nature, is fond of observing that our greatest strengths are often our greatest weaknesses. And while my wife is a self-proclaimed computerphobe, she has unwittingly placed a finger precisely on the problem with C pointers; they're so flexible that it's often difficult to figure out how to work with them. That's why it's a challenge to implement dynamic data structures—whose lifeblood is pointers—in C. This article shows how to build, search, and traverse binary trees without getting lost among all those pointers.

A binary tree is a dynamic structure inherently ordered by keys into a hierarchy. The internal nodes of the tree consist of at least three fields: a data item serving as the key, and two pointers to other nodes of the same type. The pointers, called **LLink** and **RLink** by convention, point to children of lower and higher key values, respectively. The end nodes (commonly called leaves) have the same structure, but their pointers are **NULL**.

Aside from their efficiency, one of the advantages of binary trees is that it's possible to keep track of and work with a great deal of data using only one or two pointers within the program's data space. One points to the root (the highest-level node in the tree) and another might be needed to provide a means of keeping track of where we are as we move about.

For simplicity, suppose we have a tree that counts the frequency of letters in text. We can define the node type as follows:

```
typedef struct ntag {
   char       letter;     /* key field */
   int        freq;       /* frequency */
   struct ntag *LLink, *RLink;
} NTYPE;
```

Note that **LLink** and **RLink** must be declared as pointers to **struct ntag**, since the type identifier **NTYPE** has not yet been asserted when the compiler encounters them.

Now the only data item we need to declare in order to implement the tree is

```
static NTYPE  *root;
```

Declaring it as a static guarantees that the pointer is born initialized to **NULL**. That's important for determining whether or not the tree exists.

Let's say we build the tree from the string "ROGER SMITH." The resulting tree structure is shown in Figure 1, with the frequencies in parentheses. The root pointer points to the top node, R.

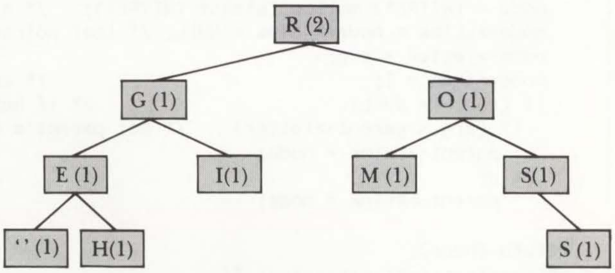


Figure 1. A binary tree based on the name "Roger Smith."

BUILDING THE TREE

The tree does not exist when the root pointer is **NULL**. Thus the first node to be allocated becomes the root, and the root pointer acquires its address. Thereafter this pointer is stable, providing entry into the tree.

A new node (whether the first or any subsequent) always begins life as a leaf, so its **LLink** and **RLink** are set to **NULL** and its data fields are initialized as appropriate. Nodes subsequent to the first have no enduring external effect; the root pointer is set only for the first. However, when a node becomes a parent, its **LLink** or **RLink** (depending on key relationships) is updated to point to the child. Thus, the root pointer is, in effect, the parent of the tree as a whole, and all nodes are within levels of descendancy from it.

Listing 1, CHARFREQ.C, contains the recursive function **newnode( )** that seeks the proper location and inserts the new node. By each invocation passing a pointer to itself in the parent argument, the child

LISTING 1: CHARFREQ.C

```
/* CHARFREQ.C: Analyzes frequency of printable characters in a text
               string, using a binary tree */

#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include <ctype.h>

/* NODE STRUCTURE */
typedef struct ntag {
  char         letter;                                /* key value */
  int          freq;                                  /* frequency */
  struct ntag *LLink, *RLink;                  /* linkage pointers */
} NTYPE;

/* GLOBAL */
static NTYPE  *root;                        /* entry to binary tree*/
/* ---------------------- LOCAL FUNCTIONS ----------------------- */
NTYPE  *found (char arg, NTYPE *node)   /* search for matching key */
{
  while (arg != node->letter)               /* loop while no match */
    if (node == NULL)                          /* if hit a leaf... */
      return (NULL);                             /* abandon search */
    else                        /* else go left or right from here */
      node = (arg < node->letter) ? node->LLink : node->RLink;
  return (node);             /* when found match, return node addr */
} /* ----------------------- */
NTYPE *newnode (char arg, NTYPE *parent, NTYPE *node)  /* add node */
{
  if (node != NULL)             /* recur to find location for node */
    if (arg < node->letter)      /* going left or right as appropr */
      node = newnode (arg, node, node->LLink);
    else
      node = newnode (arg, node, node->RLink);
  else {                            /* position found for new leaf */
    node = (NTYPE*) malloc (sizeof (NTYPE));   /* alloc heap space */
    node->LLink = node->RLink = NULL; /* leaf pointers always NULL */
    node->letter = arg;                                 /* set key */
    node->freq = 1;                            /* and count of one */
    if (root != NULL)                      /* if not first node... */
      if (arg < parent->letter)    /* set parent's correct pointer */
        parent->LLink = node;
      else
        parent->RLink = node;
  }
  return (node);
} /* ----------------------- */
void inOrder (NTYPE *node)             /* output tree by key order */
{
  if (node != NULL) {
    inOrder (node->LLink);                    /* keep working left */
    printf ("\n%c         %d",              /* then print contents */
             node->letter, node->freq);
    inOrder (node->RLink);                  /* now work to the right */
  }
} /* ----------------------- */
void postOrder (NTYPE *node)              /* delete tree from heap */
{
  if (node != NULL) {
    postOrder (node->LLink);                     /* do all to left */
    postOrder (node->RLink);                       /* all to right */
    free (node);                          /* and finally this node */
  }
```

can update its parent's linkage pointer at the time of creation.

SEARCHING A TREE

A binary tree is called *binary* because each node has, at most, two paths emanating from it: a less-than path and a greater-than path. Thus each node is effectively a fork in the road, and we pick which branch to take based on a comparison of the key with the argument. The search stops when: (1) we find the key we're looking for, or (2) we hit a leaf and can't go any further. In the second case, the search is unsuccessful.

A search function can signal the outcome by returning a pointer: either **NULL** if unsuccessful, or the node containing the sought key otherwise. That's what the **found( )** function performs in Listing 1.

TRAVERSING THE TREE

Traversing is the process of moving through the entire tree, usually for the purpose of doing something with all of its contents. Traversal always proceeds in the same physical order: root to lowest-keyed node to highest-keyed and then back to root.

Traversal of the left side of Figure 1 goes in the order **R**, **G**, **E**, **space**, **E**, **G**, **I**, **H**, **I**, **G**, **R**. Having traversed the left side, it then follows the sequence **O**, **M**, **O**, **S**, **T**, **S**, **O**, and returns back to **R**.

The apparently horrendous complexity of moving through a tree consisting of an unknown number of levels and nodes is solved by the use of recursion. Recursive functions use the stack to keep track of where they came from and thus don't explicitly need to store their path of retreat. The downside of recursion is that it quickly devours stack space.

If you have a good-sized tree, don't use Turbo C's tiny code model lest the tree and the stack collide and crash the program.

There are three kinds of tree traversal, called *pre-order*, *in-order*, and *post-order*. These methods differ not in the sequence of visiting nodes, but in the order in which they do something with the contents. Note that in Figure 1, **R**,

G, and **O** are all visited three times; **E**, **I**, **M**, and **S** twice, and the leaves only once. This gives us one to three opportunities to work on every node.

Pre-order processes the contents when it enters a node for the first time; in-order as it moves from the left branch to the right; and post-order when it leaves the node for the last time. Each has a unique purpose:

- Pre-order can be used to write contents to disk in a sequence that permits later reconstruction of the tree in its present form. Just read 'em back out and build the tree to achieve the same configuration.
- In-order is useful for outputting the tree's data sorted by key.
- Post-order is handy for deleting an entire tree from the heap.

Listing 1 contains the **inOrder( )** and **postOrder( )** functions, and Listing 2, PREORTRV.C, shows **preOrder( )**. To adapt them to your own uses, you need change only the processing statement, leaving the pointer-following statements intact.

SAMPLE APPLICATION

Now let's put the creation, searching, and traversal of a binary tree to work. CHARFREQ.C, accepts a line of user input. It then analyzes the frequency of printable characters by organizing them in a binary tree and counting occurrences, and produces a report in alphabetic sequence using in-order traversal. Afterwards it deletes the tree from the heap.

Binary trees are an excellent means for organizing and operating on large amounts of data at blazing speed. The methods covered here provide skeletons that you can adapt without suffering the heartbreak of pointer paralysis. ■

Kent Porter is a professional writer specializing in software. His most recent book is Stretching Turbo Pascal *(Brady/Simon & Schuster), and he's now at work on an advanced Turbo C book.*

Listings may be downloaded from CompuServe as BINTREES.ARC.

```
} /* --------------------- MAIN PROGRAM ------------------------ */
main ()
{
char    c, text [80];
int     n, p;
NTYPE  *current;

  do {
    puts ("\n\nType text to be analyzed:");
    gets (text);
    if ((n = strlen (text)) != 0) {      /* process if text typed */
      for (p = 0; p < n; p++) {
        c = toupper (text [p]);       /* get next letter, upshift */
        if (isprint (c)) {                    /* see if printable */
          if (root == NULL)             /* if first letter in text */
            root = newnode (c, NULL, NULL);         /* create root */
          else
            if ((current = found (c, root)) != NULL)
              current->freq++;              /* count if duplicate */
            else
              newnode (c, root, root);        /* else add to tree */
        }                                /* end of 'if printable' */
      }                                        /* end of for loop */
      /* PRINT FREQUENCY ANALYSIS */
      puts ("\nFrequency of letters used:");
      inOrder (root);

      /* DELETE TREE FROM HEAP */
      postOrder (root);
      root = NULL;
    }                                                /* end of if */
  } while (n != 0);                                     /* repeat */
}
```

LISTING 2: PREORTRV.C

```
void preOrder (NODETYPE *node)
{
  if (node != NULL) {
    /* insert your processing operation here */
    preOrder (node->LLink);            /* now go left */
    preOrder (node->RLink);         /* and then right */
  }
} /* end of function preOrder() */
```

JULIAN DAYS AND DATES

Make today's date a single number and spend less time manipulating it.

Marty Franz

In business programming, dates are often as important as dollars. Inventory shipments arrive on them, birthdays need to be remembered by them, and interest (or penalties) are figured with them. But as important as dates are, using conventional calendar dates can be cumbersome for the programmer. There's no way to directly subtract March 1, 1986, from January 28, 1987, to find the number of days that have elapsed. Instead, you end up loading an array with the number of days in each month and iterating through it—a tedious and needlessly bug-filled exercise.

The problem in computing with dates comes from our way of representing them, which combines three different numbering units: the number of days in a month, the number of months in a year, and the number of years that have elapsed since the start of the Christian era. To easily do any arithmetic with a "number" having three "digits," each in a different base (10, 28, 29, 30 or 31, and 12), you must first convert it to a single number having a single base. Typically, a number of days from a given date is used. Any date will do for computational purposes, so long as all dates you use are converted from the same base date.

The difficult part is figuring the number of days in a month (that's where you need the array mentioned earlier) and leap years, which adjust the number of days in the month of February. It would be easier if all the years, and all the months, had the same number of days. Then you could simply multiply each "digit" in a date by its appropriate radix, add them together, and have a valid base 10 date, and forget about arrays.

This can be done, but not with the Gregorian calendar (the one we use that has leap years every four years except every century year not divisible by four...got that?) This calendar was adopted in 1528 by Pope Gregory XIII to correct errors that had accrued in the earlier calendar, invented by Julius Caesar.

RENDERING UNTO CAESAR

The Julian calendar was based on a simpler 365-day year with a leap year every four years no matter what. We can convert dates into a workable format if we do our multiplying using fractional numbers (employing the astronomical fact that there are 365.25 days in a year and 30.6001 days in the average month). Then we adjust the month and year if necessary to always place the date after February to incorporate the leap day.

Using Julian dates is nothing new. Even before there were computers, astronomers needed to figure out the number of days between two dates, often for sightings separated in time by several years. They used an annual almanac that employed the Julian calendar and an accompanying set of formulas. (Those formulas are included in this article.) Because the early astronomers worked with a variety of calendars (some dating back thousands of years) and a variety of long-term observations, they decided on January 1, 4713 B.C., as their base date. All their observational dates were then calculated to be a number of days from this date. For example, January 28, 1987, is Julian day 2442824.

The modules provided here include functions that perform this calculation. In the functions, there are two formats for dates. The first is the conventional Gregorian calendar date expressed as a real number, with the components of the date arranged within the real number in the form YYMMDD. For example, the date May 1, 1987, would be kept as the single real number 870501.0. Dates in this format are easy to enter, edit, and keep in files. The various parts (year, month, and day) can be extracted by division and subtraction.

The second format used is the Julian date expressed as the number of days that have elapsed since January 1, 4713 B.C. You can use this format in computations involving dates in the future, or the number of days between two dates. You also can easily find the day of the week for a particular date, simply by dividing the Julian day number by seven, taking the remainder, and adding the day of the week for the base date.

The accompanying listings include versions of the Julian and Gregorian calendar functions for Turbo C, Turbo Basic, and Turbo Pascal. These functions are

LISTING 1: CHRONO.C

```
/*
CHRONO.C:       Julian date and time routines for C

version:        5-20-87
compiler:       Turbo C version 1.0
uses:           stdio.h, stdlib.h, dos.h
module type:    object

This file contains functions to work with calendar dates.  A
calendar date is a real number in the format YYMMDD.  For
example, 4/23/87 would be 870423.  Julian dates are "magic"
hashed versions of calendar dates that allow arithmetic,
determining the day of the week, etc.  without consulting an
actual calendar.

Functions in this file:

  year(D)          Returns the year part of a calendar date
  day(D)           Returns the day part of a calendar date
  month(D)         Returns the month part of a calendar date
  julian(D)        Converts a calendar date to a julian date
  calendar(J)      Converts a julian date to a calendar date
  dayofweek(J)     Returns day of week for a julian date
  today()          Returns today's date (from DOS) as calendar
                   date
*/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>

double year(), day(), month();
double julian(), calendar(), dayofweek();
double today();

double year(double cal)
{
        double floor();

        return floor(cal / 10000.0);
}

double day(double cal)
{
        double floor();

        return floor(cal - (floor(cal / 100.0) * 100.0));
}

double month(double cal)
{
    double floor();

    return floor((cal - (year(cal) * 10000.0) - day(cal)) / 100.0);
}


double julian(double cal)
{
    double m, y, floor();

    if (month(cal) > 2) {
            m = month(cal) + 1.0;
            y = year(cal);
    }
    else {
            m = month(cal) + 13.0;
            y = year(cal) - 1;
    }
    return floor(365.25*(1900.0+y))+floor(30.6001*m)+day(cal)+1720982;
}

double calendar(double jul)
{
        double m, d, y, dayno, floor();

        dayno = jul - 1720982;
        y = floor((dayno - 122.1) / 365.25);
        m = floor((dayno - floor(365.25 * y)) / 30.6001);
        d = dayno-floor(365.25 * y) - floor(30.6001 * m);
        m = (m < 14) ? m - 1 : m - 13;
        y = (m < 3) ? y + 1 : y;
        return (y - 1900) * 10000.0 + m * 100.0 + d;
}

double dayofweek(double jul)
{
        double dayno, x, fracx, floor();

        dayno = jul - 1720982;
        x = (dayno + 5.0) /7.0;
        fracx = x - floor(x);
        return floor(7 * fracx + 0.5);
}

/*
The today function is already available from Turbo C.
Here's a shell using getdate().
*/

double today()
{
    struct date d;

    getdate(&d);
    return (d.da_year - 1900) * 10000.0 + d.da_mon * 100.0 + d.da_day;
}
```

LISTING 2: CHRONO.INC

```
{
  CHRONO.INC:  Julian date functions

  version:     05-20-87
  compiler:    Turbo Pascal v3.0
  uses:        nothing
  module type: include file

  This file contains functions to work with calendar dates.  A
  calendar date is a real number in the format YYMMDD.  For
  example, 4/23/87 would be 870423.  Julian dates are "magic"
  hashed versions of calendar dates that allow arithme.ic,
  determining the day of the week, etc.  without consulting an
  actual calendar.

  Functions in this file:

    Year(D)        Returns the year part of a calendar date
    Day(D)         Returns the day part of a calendar date
    Month(D)       Returns the month part of a calendar date
    Julian(D)      Converts a calendar date to a julian date
    Calendar(J)    Converts a julian date to a calendar date
    DayOfWeek(J)   Returns day of week for a julian date
    Today          Returns today's date (from DOS) as calendar
                   date
}

function Year(Cal : real) : integer;
  {
    Return the year part of a date in the format YYMMDD.
  }
begin
  Year := Trunc(Cal/10000.0);
end; { Year }

function Day(Cal : real) : integer;
  {
    Return the day part of a date in the format YYMMDD.
  }
begin
  Day := Trunc(Cal-(Int(Cal/100.0)*100.0));
end; { Day }

function Month(Cal : real) : integer;
  {
    Return the month part of a date in the format YYMMDD.
  }
begin
  Month := Trunc((Cal-(Year(Cal)*10000.0)-Day(Cal))/100.0);
end; { Month }

function Julian(Cal : real) : real;
  {
    Convert a calendar date (YYMMDD) to a Julian date.
  }
var
  m, y : integer;
begin
  if Month(Cal) > 2 then
  begin
    m := Succ(Month(Cal));
    y := Year(Cal);
  end
  else
  begin
    m := Month(Cal)+13;
    y := Pred(Year(Cal));
  end;
  Julian := Int(365.25*(1900+y))+Int(30.6001*m)+Day(Cal)+1720982.0;
end; { Julian }
```

slightly different because of variations in each language's greatest integer (such as Turbo Basic's **INT** or equivalent) function. To use the Turbo Basic and Turbo Pascal versions, include the source file in your program. The Turbo C version should be compiled as an object module and then linked.

CALENDAR FUNCTIONS

The functions provided for each language are:

Year extracts the year from a date in Gregorian calendar format (YYMMDD).

Month extracts the month from a date in Gregorian calendar format (YYMMDD).

Day extracts the day from a date in Gregorian calendar format (YYMMDD).

Julian converts a date from Gregorian format to Julian format. The result is a real number that is the number of days that have elapsed since January 1, 4713 B.C.

Calendar converts a date from Julian format back to calendar (Gregorian) format. The result is a real number in the format YYMMDD.

DayOfWeek passes a date in Julian format, returns the day of the week as a number between 0 (for Sunday) and 6 (for Saturday).

Today returns the current date from the MS-DOS system clock in Gregorian format (YYMMDD).

The code in these functions is straightline. The only subtlety is

```
if Month(Cal) > 2 then
begin
  m := Succ(Month(Cal));
  y := Year(Cal);
end
else
begin
  m := Month(Cal)+13;
  y := Pred(Year(Cal));
end;
```

which adjusts the month and year so that January and February are always reckoned as if they were the thirteenth and fourteenth months of the prior year. This ignores leap years in the calculation. The reverse is done for the conversion back to Gregorian format.

For example, if you want to find the number of days between January 28, 1987 and March 1, 1986, in a Turbo Basic program, you can write

```
PRINT FNJulian(870128)-FNJulian(860301)
```

These functions are based on approximations and are useful between March 1, 1900 and February 28, 2100, so you won't have to leave the country when the year 2000 rolls around. In short, incorporating these Julian date functions in your programs will save you time and effort the next time you work with the calendar. ■

Marty Franz is a programmer who frequently writes on microcomputer topics. Marty lives in Kalamazoo, Michigan.

Listings may be downloaded from CompuServe as JULIAN.ARC.

THE DOS LEAP YEAR PROBLEM

Somewhere deep within DOS is a mechanism for calculating the day of the week given the date. When you read the date through DOS function call $2A, register AL contains a value ranging from 0 through 6, where 0 indicates Sunday, and 6 indicates Saturday.

So rather than implement tortuous algorithms like Zeller's Congruence for calculating the day of the week, you can read the current date and save it, set the DOS date to your input date, and then read back that date with function $2A—and get your day of the week in AL. (Then restore the current date that you saved.)

One problem is that DOS doesn't understand dates prior to 1/1/80—use this method only for dates since then. The other problem is that in leap years DOS's day-of-the-week is off by one. You must "rotate" the day of the week indicator downward on a leap year day, translating a 0 to a 6 and decrementing any other value by 1. The bug is quite consistent, and the workaround is reliable.

DateToDayOfWeek uses this method to return the day of the week for any given date. It corrects for the DOS leap year bug as well.

—*Jeff Duntemann*

```
FUNCTION DateToDayOfWeek(Year,Month,Day : Integer) : Integer;

VAR
  SaveDate,WorkDate : Registers;
  DayNumber         : Integer;
  LeapYearDay       : Boolean;

CONST
  DayArray : ARRAY[1..12] OF Integer =
    (31,28,31,30,31,30,31,31,30,31,30,31);

BEGIN
  LeapYearDay := False;
  IF (Month = 2) AND ((Year MOD 4)=0) AND (Day = 29) THEN
    LeapYearDay := True;
  IF (NOT LeapYearDay) AND (Day > DayArray[Month]) THEN
    DateToDayOfWeek := -1
    ELSE
      BEGIN
        WorkDate.AH := $2B;
        SaveDate.AH := $2A;  { Saves date encoded in registers }
        MSDOS(SaveDate);     { Fetch & save today's date }
        WITH WorkDate DO
          BEGIN
            CX := Year;      { Set the clock to the input date }
            DH := Month;
            DL := Day;
            MSDOS(WorkDate);
            AH := $2A;       { Turn around and read it back }
            MSDOS(WorkDate); {  to find the day-of-week indicator }
            DayNumber := AL; {  in AL. }
            IF LeapYearDay THEN { Correct for DOS's leap year bug }
              IF DayNumber = 0 THEN DayNumber := 6
                ELSE DayNumber := Pred(DayNumber);
            DateToDayOfWeek := DayNumber
          END;
        SaveDate.AH := $2B;  { Restore clock to today's date }
        MSDOS(SaveDate);
      END
END;
```

```
function Calendar(Jul : real) : real;
  {
    Convert a Julian date to a calendar date (YYMMDD).
  }
var
  m, d, y , DayNo : real;
begin
  DayNo := Jul - 1720982.0;
  y := Int((DayNo-122.1)/365.25);
  m := Int((DayNo-Int(365.25*y))/30.6001);
  d := DayNo - Int(365.25*y) - Int(30.6001*m);
  if m < 14 then
    m := m - 1
  else
    m := m - 13;
  if m < 3 then y := y + 1;
  Calendar := (y-1900)*10000.0 + m*100.0 + d;
end; { Calendar }

function DayOfWeek(Jul : real) : integer;
  {
    Returns the day of the week for a Julian date, with 0=Sunday.
  }
var
  DayNo : real;
begin
  DayNo := Jul - 1720982.0;
  DayOfWeek := Round(7 * Frac((DayNo+5.0)/7.0));
end; { DayOfWeek }

function Today : real;
  {
    Return today's date as a calendar date in YYMMDD format.
  }
const
  GetDate = $2A00;
var
  Regs : record
           AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : integer;
         end;
  y, m, d : integer;
begin
  with Regs do
  begin
    AX := GetDate;
    MsDos(Regs);
    y := CX;
    m := Hi(DX);
    d := Lo(DX);
  end;
  Today := (y-1900)*10000.0 + m*100.0 + d;
end; { Today }
```

LISTING 3: CHRONO.TB

```
' CHRONO.TB:    Julian Date functions
'
' version:      05-20-87
' compiler:     Turbo BASIC v1.0
' uses:         nothing
' module type:  include file
'
' This file contains functions to work with calendar dates.  A
' calendar date is a real number in the format YYMMDD.  For
' example, 4/23/87 would be 870423.  Julian dates are "magic"
' hashed versions of calendar dates that allow arithmetic,
' determining the day of the week, etc.  without consulting an
' actual calendar.
'
' Functions in this file:
'
' FNYear(D)          Returns the year part of a calendar date
' FNDay(D)           Returns the day part of a calendar date
' FNMonth(D)         Returns the month part of a calendar date
' FNJulian(D)        Converts a calendar date to a julian date
' FNCalendar(J)      Converts a julian date to a calendar date
' FNDayOfWeek(J)     Returns day of week for a julian date
' FNToday            Returns today's date (from DOS) as calendar
'                    date

DEF FNYear(Cal)
  ' Return the year in a calendar date.
  FNYear = INT(Cal/10000)
END DEF

DEF FNDay(Cal)
  ' Return the day in a calendar date.
  FNDay = INT(Cal-(INT(Cal/100)*100))
END DEF

DEF FNMonth(Cal)
  ' Return the month in a calendar date.
  FNMonth = INT((Cal-(FNYear(Cal)*10000)-FNDay(Cal))/100)
END DEF

DEF FNJulian(Cal)
  ' Convert a calendar date into a Julian date
  LOCAL m,y
  IF FNMonth(Cal) > 2 THEN
    m = FNMonth(Cal)+1
    y = FNYear(Cal)
  ELSE
    m = FNMonth(Cal)+13
    y = FNYear(Cal)-1
  END IF
  FNJulian = INT(365.25*(1900+y))+INT(30.6001*m)+FNDay(Cal)+1720982
END DEF

DEF FNCalendar(Jul)
  ' Convert a Julian date into a calendar date
  LOCAL m,d,y,DayNo
  DayNo = Jul - 1720982
  y = INT((DayNo-122.1)/365.25)
  m = INT((DayNo-INT(365.25*y))/30.6001)
  d = DayNo-INT(365.25*y)-INT(30.6001*m)
  IF m < 14 THEN
    m = m-1
  ELSE
    m = m-13
  END IF
  IF m < 3 THEN y = y + 1
  FNCalendar = (y-1900)*10000 + m*100 + d
END DEF

DEF FNDayOfWeek(Jul)
  ' Convert Julian date to day of week, 0=Sunday
  LOCAL DayNo,X,FracX
  DayNo = Jul - 1720982
  X = (DayNo+5)/7.0
  FracX = X - INT(X)
  FNDayOfWeek = INT(7*FracX+0.5)
END DEF

DEF FNToday
  ' Return today's date as a calendar date from MS DOS
  LOCAL m,d,y
  REG 1,&H2A00
  CALL INTERRUPT &H21
  y = REG(3)
  m = INT(REG(4)/256)
  d = REG(4) AND &H00FF
  FNToday = (y-1900)*10000 + m*100 + d
END DEF
```

Introducing the two on earth

The new COMPAQ DESKPRO 386/20™

The world now has two new benchmarks from the leader in high-performance personal computing. The new 20-MHz COMPAQ DESKPRO 386/20 and the 20-lb., 20-MHz COMPAQ PORTABLE 386 deliver system performance that can rival minicomputers'. Plus they introduce advanced capabilities without sacrificing compatibility with the software and hardware you already own.

Both employ an industry-standard 80386 microprocessor and sophisticated 32-bit architecture. Our newest portable is up to 25% faster and our desktop is actually up to 50% faster than 16-MHz 386 PC's. But we did much more than simply increase the clock speed.

For instance, the COMPAQ DESKPRO 386/20 uses a cache memory controller. It complements the speed of the microprocessor, providing an increase in system performance up to 25% over other 20-MHz 386 PC's. It's also the first PC to offer an optional Weitek™ Coprocessor Board, which can give it the performance of a dedicated engineering workstation at a fraction of the cost.

They both provide the most storage and memory within their classes. Up to 300 MB of storage in our latest desktop and up to 100 MB in our new portable.

It simply works better.

most powerful PC's and off.

and the new 20-MHz COMPAQ PORTABLE 386™

Both use disk caching to inject more speed into disk-intensive applications.

As for memory, get up to 16 MB of high-speed 32-bit RAM with the COMPAQ DESKPRO 386/20 and up to 10 MB with the COMPAQ PORTABLE 386. Both computers feature the COMPAQ® Expanded Memory Manager, which supports the Lotus®/Intel®/Microsoft® Expanded Memory Specification to break the 640-Kbyte barrier.

With these new computers plus the original COMPAQ DESKPRO 386™, we now offer the broadest line of high-performance 386 solutions. They all let you run software being written to take advantage of 386 technology. *And to prove it, from now until December 31, 1987, we're including Microsoft® Windows/386 Presentation Manager free with your purchase of any COMPAQ 386-based PC.* It provides multitasking capabilities with today's DOS applications to make you considerably more productive. But that's just the beginning. For more information, call 1-800-231-0900, Operator 40. In Canada, call 416-733-7876, Operator 40.

Intel, Lotus, Microsoft, and Weitek are trademarks of their respective companies.

COMPAQ®

A PROGRAMMER'S GUIDE TO THE PARALLEL PORT

Take the printer off your printer cable and it becomes a gateway to the Real World.

Bruce Eckel

PROGRAMMER

When IBM came out with their first parallel printer card, it was (of course) expensive. At those prices, one wouldn't think of using it for anything but a printer. The design wasn't complicated, however, and *everyone* has since cloned it, making it the cheapest adapter card available for the PC. With these prices, one starts thinking: "Why can't I use this thing in a burglar alarm? Or to control my Jacuzzi? Or to automate all manufacturing in the United States and eliminate the national debt?"

In this article, I will show you how to read from and write to the printer port on a pin-by-pin basis (since each pin seems to have its own personality). I will also describe some basics for interfacing external switches to the port.

FOUR CARDS

I studied the design of four different cards: an AST board and three low-cost clones (the cheapest one was $21 from Microsphere in Bend, Oregon). All were identical, which makes sense because any piece of software must see the same bits in the same I/O locations if it is to communicate correctly with the printer, regardless of the card's manufacturer.

Printer cards live in the "I/O space" (they are sometimes referred to as being "I/O mapped"). This means that to talk to them, you don't use Turbo C's **peek()** and **poke()** functions but instead the **inport()** and **outport()** functions. The card is accessed through three consecutive I/O addresses: **BASE** (the starting address), **BASE + 1**, and **BASE + 2**. **BASE** is set by dip switches or jumpers on your printer board; standard addresses are 0x378 for LPT1, and 0x278 for LPT2. Some cards (including IBM's original Monochrome Display and Printer Adapter) also support 0x3BC for LPT3.

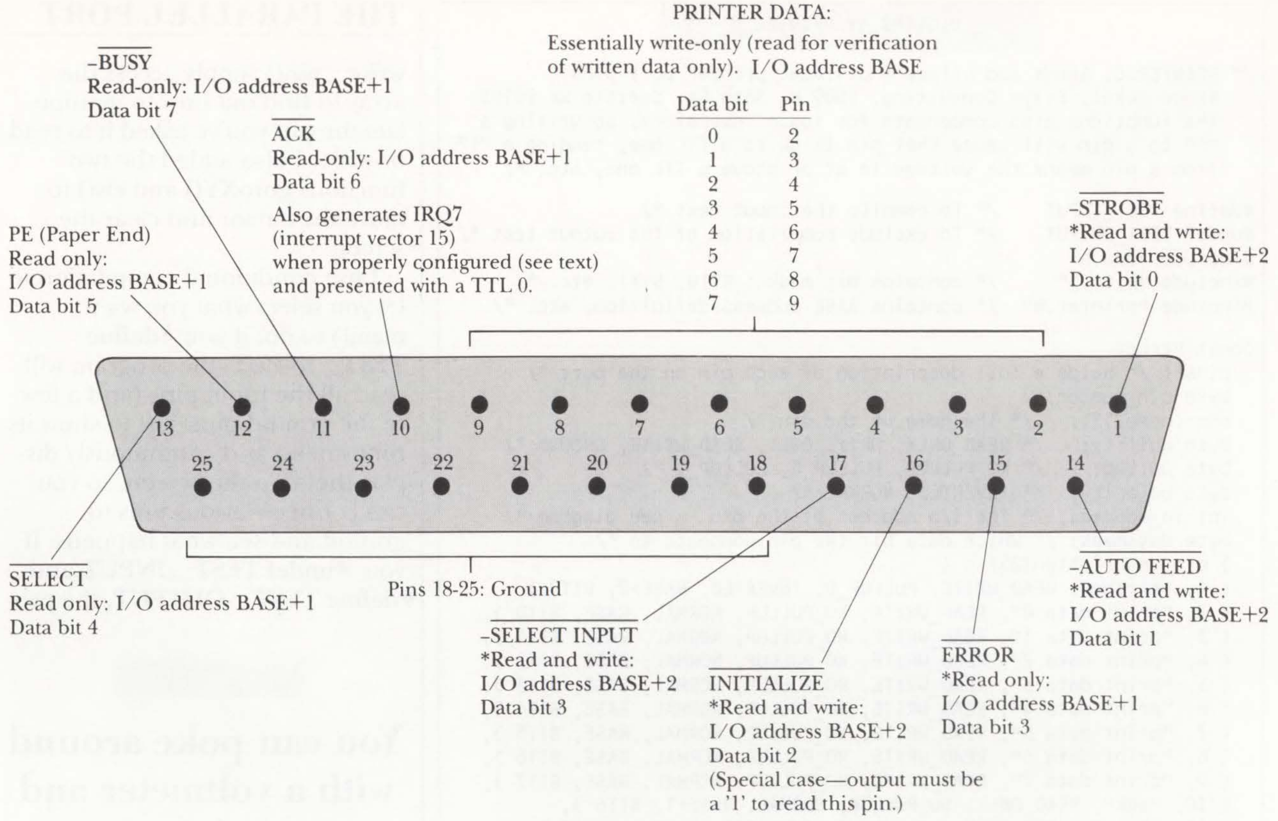
25 PINS

Usually, what you see of the printer port is a 25-pin female DB-25 connector sticking out of the back of your computer. Figure 1 names each pin and provides its port address, which data bit affects the pin, whether data is inverted between the data bus and the pin, and what the physical inputs and outputs look like electrically to the outside world.

The board's power supply requirements specify 0 volts for ground and 5 volts for V+, so you might think 0 volts is a logic zero, 5 volts is a logic 1, and anything else is uncertain. Actually the hardware devices used on the board (TTL) have a different concept of 0 and 1: less than or equal to 0.8 volts indicates a zero, and 2.7 volts or more indicates a 1. These are the voltages you will see coming out of pins 2-9 (the printer data pins), if you connect a voltmeter directly to the pins.

Pins 1, 14, 16, and 17 are a little different. They are the only pins that work both ways, i.e., data may be read into the computer from an external device, or data may be output to a device external to the computer. These pins use a kind of output called *open collector*. Depending on the logic state of the output, it will either be pulled low to ground or sent into a condition that resembles an electrical open circuit. The voltage on an open-collector output just floats around when it is set to 1, so if you want the output to go to a *real* 1, you must insert a resistor connected to 5 volts on the output pin. These resistors (typically 4700 ohms) "pull up" the output to 5 volts (instead of just TTL's normal 2.7 volts) when the output is 1, so they're called *pull-up resistors*. Pins 1, 14, 16, and 17 all have open-collector outputs with pull-up resistors, so they will output a solid 5 volts when set to 1.

The pull-up resistors make life a lot easier when reading a value from the outside world. Otherwise, unconnected TTL inputs tend to float high; you'll see when playing with PRINTER.C that they are generally



**To read data from the outside world, a "0" must first be written to this pin. The output has a pull-up resistor that pulls it up to five volts.*

NOTES: Signal names having a preceding minus sign (i.e., -STROBE) electrically invert an incoming voltage; i.e., a TTL zero on the pin will be inverted and sensed by the computer as a TTL one, and vice versa. The overbar indicates only the logical sense of the printer's signal—in other words, when the ERROR line goes to zero, the printer is signaling an error condition.

BASE is the starting I/O address of the block of three I/O addresses required by the parallel board logic (see text).

Figure 1. The PC parallel port.

high but some of them will rattle around a bit. It is tempting to rely on the inputs to be high when open and to pull them to ground with a switch to make them low, but this is a bad practice since open TTL inputs are susceptible to noise. If you want to hook a switch up this way, you should pull the input pin up to 5 volts (through a resistor—around 5K ohms), which means you must have your own external power supply.

Fortunately, the four pins 1, 14, 16, and 17 already have pull-ups on them, so if you can get away with only four input lines, all you need is a wire from the pin to a switch that is connected to one of the ground pins (18-24). No external power supply is needed. To read these pins, you must first let the open collector of the output driver for that pin float so that you can impress your external data on the pin—otherwise the pin will always be pulled down to zero. For the inverting pins, this means writing a zero to the output; for the noninverting, a 1.

The I/O address **BASE** reads and writes the eight bits of printer data on pins 2-9. Unfortunately, unless you physically modify the board, you can only read back what is presently written to the port, which means it's good for output, but virtually no good for input.

THE CODE

Listing 1, PRINTER.C, is a Turbo C program that allows you to separately read and write each pin (providing the pin in question has both capabilities) without affecting any of the others. Of course, you can manipulate things a byte at a time, but most people start out looking for a single switch closure or turning a single relay on and off; this should make it simpler to get started.

To be explicit, I created a constant array of structures called **pin_description[]** that contains all the information about each pin: where it lives, whether it inverts, how to read it, etc. The two functions **read_pin()** and

LISTING 1: PRINTER.C

```
/* PRINTER.C: Reads and writes individual printer port pins.
   Bruce Eckel, Eisys Consulting, 1009 N. 36th St. Seattle WA 98103.
   The functions also compensate for logic inversions, so writing a
   "1" to a pin will cause that pin to go to a TTL one, reading a "1"
   from a pin means the voltage is at or above a TTL one, etc. */

#define TEST_INPUT     /* To compile the input test */
#undef TEST_OUTPUT     /* To exclude compilation of the output test */
#include <dos.h>
#include "bits.h"      /* contains bit masks: BIT0, BIT1, etc. */
#include "printer.h"   /* contains BASE address definition, etc. */

const struct
  pins { /* holds a full description of each pin on the port */
  byte pin_number;
  char name[15];   /* The name of the pin */
  byte ability;   /* READ_ONLY, WRITE_ONLY, READ_WRITE, GROUND */
  byte pullup;    /* NO_PULLUP, PULLUP_0, PULLUP_1 */
  byte polarity;  /* INVERTED, NORMAL */
  int io_address; /* The i/o address of the pin -- see diagram */
  byte datamask; /* Which data bit the pin connects to */
  } pin_description[25] = {
  { 1, "strobe", READ_WRITE, PULLUP_0, INVERTED, BASE+2, BIT0 },
  { 2, "print data 0", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT0 },
  { 3, "print data 1", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT1 },
  { 4, "print data 2", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT2 },
  { 5, "print data 3", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT3 },
  { 6, "print data 4", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT4 },
  { 7, "print data 5", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT5 },
  { 8, "print data 6", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT6 },
  { 9, "print data 7", READ_WRITE, NO_PULLUP, NORMAL, BASE, BIT7 },
  { 10, "ack", READ_ONLY, NO_PULLUP, NORMAL, BASE+1, BIT6 },
  { 11, "busy", READ_ONLY, NO_PULLUP, INVERTED, BASE+1, BIT7 },
  { 12, "paper end", READ_ONLY, NO_PULLUP, NORMAL, BASE+1, BIT5 },
  { 13, "select", READ_ONLY, NO_PULLUP, NORMAL, BASE+1, BIT4 },
  { 14, "auto feed", READ_WRITE, PULLUP_0, INVERTED, BASE+2, BIT1 },
  { 15, "error", READ_ONLY, NO_PULLUP, NORMAL, BASE+1, BIT3 },
  { 16, "initialize", READ_WRITE, PULLUP_1, NORMAL, BASE+2, BIT2 },
  { 17, "select input", READ_WRITE, PULLUP_0, INVERTED, BASE+2, BIT3},
  { 18, "ground", GROUND, 0,0,0,0 },
  { 19, "ground", GROUND, 0,0,0,0 },
  { 20, "ground", GROUND, 0,0,0,0 },
  { 21, "ground", GROUND, 0,0,0,0 },
  { 22, "ground", GROUND, 0,0,0,0 },
  { 23, "ground", GROUND, 0,0,0,0 },
  { 24, "ground", GROUND, 0,0,0,0 },
  { 25, "ground", GROUND, 0,0,0,0 }
  };
byte oldvalue[3] = { 0,0,0 }; /* holds presently output values of
                                 bits, used to change one bit without
                                 disturbing the rest. */
byte read_pin(byte pin_number)
/* Returns 0 if voltage on pin is TTL zero, 1 if TTL one. */
/* These macros make things easier to read, and reduce typos. */
#define PIN pin_description[pin_number]
#define INDEX (PIN.io_address - BASE) /* tells us if it's BASE+1,
                                         +2 or +3 */
{
 if (PIN.ability == GROUND)
    return(0); /* Ground is always at zero, right? */
 if (PIN.pullup == PULLUP_0) {
    oldvalue[INDEX] &= ~PIN.datamask; /* set bit to 0 */
    outportb(PIN.io_address, oldvalue[INDEX]);
    }
 if (PIN.pullup == PULLUP_1) {
    oldvalue[INDEX] |= PIN.datamask; /* set bit to 1 */
    outportb(PIN.io_address, oldvalue[INDEX]);
    }
 if (PIN.polarity == INVERTED)
    return (inportb(PIN.io_address) & PIN.datamask ? 0 : 1 );
    /* note ternary if-then-else returns 0 if a 1 is returned at that
       bit position, and a 1 if a zero is returned (page 127 in user's
       guide). */
 else
    return (inportb(PIN.io_address) & PIN.datamask ? 1 : 0 );
   }
 byte write_pin(byte pin_number, byte value)
 /* If value is one, that pin goes to a TTL one; if zero, goes to
```

write_pin() simply access the array to find out how to manipulate the pin you've asked it to read or write. I also added the two functions **gotoXY()** and **cls()** to move the cursor and clear the screen.

I use conditional compilation to let you select what you want **main()** to do: if you **#define TEST_INPUT**, the program will read all the input pins (and a few of the ground pins, just to show its robustness) and continuously display them on the screen, so you can connect various pins to ground and see what happens. If you **#undef TEST_INPUT** and **#define TEST_OUTPUT**, the program will make all the pins (the ones it can write to) high, tell you about them and wait, then make everything low, etc. This way you can poke around with a voltmeter and assure that everything is properly toggling on and off.

You can poke around with a voltmeter and assure that everything is properly toggling on and off.

Since the header files (ending with .H) affect PRINTER.C, you should set up a project file (PRINTER.PRJ) and tell Turbo C you are using it. This way, if you change something in the header files, Turbo C will know to recompile PRINTER.C. The project file is simple; it need only contain

```
PRINTER (PRINTER.H BITS.H)
```

which means there is one file in this project called PRINTER.C (Turbo C assumes the .C), and it depends on PRINTER.H and

BITS.H. If you change either of those files, Turbo C recompiles PRINTER.C.

INTERRUPTS

The printer board also supports interrupts; when properly configured, pulling the ACK line (pin 10) to ground will cause IRQ7 to be generated. "Properly configured" is nontrivial, as you will see from TSR.C (Listing 2). This is a Terminate and Stay Resident (TSR) program written entirely in Turbo C (one of the wonders of Turbo C is you don't have to resort to assembly language to write a TSR or other interrupt routine—a marvelous reduction of programming hassles) that initializes all the hardware and installs itself in RAM to wake up with IRQ7. Configuring the printer board involves two steps: 1) setting a bit on the board to enable the interrupt; and 2) programming the 8259A Programmable Interrupt Controller to accept the interrupt. It's inappropriate to go into all the hardware details here; as long as you set the **BASE** address correctly, you can pull pin 10 to ground and generate the interrupt. (If you're hungry for more details, and want to know how to modify the board to get more out of it, check out my column in the October-November 1987 issue of *Micro Cornucopia*.)

TSR.C just gives the framework for the interrupt. You'll notice the only real function it performs in the interrupt is the call to **beep**(). You can add anything else you want as long as you don't make any calls to DOS or the ROM BIOS. This means anything like **printf**(), **scanf**(), **getch**()...well, that sort of limits you, doesn't it? Everyone who writes a "pop-up" program has to cope with this problem, which arises because DOS and the BIOS are not *reentrant*, meaning you can't call them if someone else (i.e., the process you're interrupting) has already called them. If Borland had written DOS and the ROM BIOS, we probably wouldn't have these problems. All this means you have to write your own keyboard and screen-control

```
    TTL zero. */
/* Returns WRITE_SUCCESS, READ_ONLY, or GROUND */
{
 if (PIN.ability == GROUND)
    return(GROUND);         /* bail out if we can't write to the pin */
 if (PIN.ability == READ_ONLY) return(READ_ONLY);
 if (PIN.polarity == NORMAL){
    if (value == 0)
       oldvalue[INDEX] &= ~PIN.datamask; /* set bit to 0 */
    else oldvalue[INDEX] |= PIN.datamask; /* set bit to 1 */
 }
 if (PIN.polarity == INVERTED){
    if (value == 0)
       oldvalue[INDEX] |= PIN.datamask; /* set bit to 1 */
    else
       oldvalue[INDEX] &= ~PIN.datamask; /* set bit to 0 */
 }
  outportb(PIN.io_address, oldvalue[INDEX]);  /* send the value out to
                                                  the pins */
  return(WRITE_SUCCESS);
}
void gotoXY(int x, int y)  /* Like Turbo Pascal. */
/*  This was taken from page 132 of the Turbo C Reference Guide */
#define VIDEO 0x10
{
 union REGS regs;
 regs.h.ah = 2; /* set cursor position */
 regs.h.dh = y;
 regs.h.dl = x;
 regs.h.bh = 0; /* video page 0 */
 int86(VIDEO, &regs, &regs);
}
void
cls()  /* quick-and-dirty clear screen */
{
#define SCREEN_BASE 0xb800   /* base address of color graphics card
                                (and EGA in color graphics mode  */
#define SCREEN_HEIGHT 25
#define SCREEN_WIDTH 80
#define SCREEN_CHARS (SCREEN_WIDTH * SCREEN_HEIGHT * 2)
                  /* number of chars and attributes in a screen */
unsigned int scr_index;
   for(scr_index = 0; scr_index < SCREEN_CHARS; scr_index += 2) {
      pokeb(SCREEN_BASE,scr_index,' ');
   }
}
main()
{
int pin_number;  /* Note the pin_number is off by one since arrays
                    in C are indexed from 0. */
cls();
#ifdef TEST_INPUT  /* selects the input test */
do {
    for (pin_number = 0; pin_number <= 20; pin_number++) {
        /* Most of this is just fooling around with tricks to make
           things come out nice on the screen. */
        gotoXY((pin_number % 2 ? 40 : 0),(2 *(pin_number /2) + 1));
        printf("pin %d: %s ", PIN.pin_number, PIN.name);
        if (PIN.ability == READ_ONLY)   printf("R");
        if (PIN.ability == WRITE_ONLY)  printf("W");
        if (PIN.ability == READ_WRITE)  printf("RW");
        gotoXY((pin_number % 2 ? 65 : 25), (2 * (pin_number /2) + 1));
        printf("value: %d", read_pin(pin_number));
    }
} while (!kbhit());  /* continue until any key is pressed */
}
#endif TEST_INPUT
#ifdef TEST_OUTPUT  /* selects the output test */
while(1) {
    for (pin_number = 0; pin_number <= 20; pin_number++)
        write_pin(pin_number,1);
    printf("all on\n");
    if(getch() != ' ')
       break; /* if anything but spacebar, bail out of "while" loop.*/
    for (pin_number = 0; pin_number <= 20; pin_number++)
        write_pin(pin_number,0);
    printf("all off\n");
    if (getch() != ' ') break;
}
}
#endif TEST_OUTPUT
```

LISTING 2: TSR.C

```
/* TSR.C:
   Terminate and Stay Ready (TSR) program written entirely in Turbo C.
   Bruce Eckel, Eisys Consulting, 1009 N. 36th St., Seattle, WA 98103.
   7/87.  The interrupt line on the parallel printer card (-ACK: pin
   10 on the DB-25 connector) is allowed through to IRQ7, and the
   8259A is configured to service the interrupt when -ACK is pulled to
   ground using a simple switch, TTL or CMOS logic.  You can test it
   just by poking a wire from pin 10 to one of the ground pins
   (18-25).  Each time you bring the line low, you will hear a brief
   beep.  More information can be found on pp 274-276 of the Turbo C
   user's guide, and under "keep" in the reference guide.
*/

#include <dos.h>
#define INT_NUMBER 15      /* interrupt number to install this function
                              into.  Note IRQ7 on the PC card bus
                              corresponds to interrupt handler 15 in the
                              interrupt vector table.  */
#define PROG_SIZE 0x620   /* Run the Turbo C compiler with the
                               options:linker:mapfile set to "segments."
                               Look at the mapfile generated for this
                               program.  The "stop" address for the
                               stack is the highest adress used -- set
                               PROG_SIZE to this value for use with the
                               "keep()" command */

#define PPORT_BASE 0x378 /* Parallel port board base address for LPT1.
                               This address is usually determined by
                               either dip switches or jumpers.  Change
                               this if you're using LPT2 or some other
                               address. */

#define PIC_OCW1 0x21      /* Address of 8259A Programmable Interrupt
                               Controller Operation Control Word 1.  */

void interrupt int_handler()

/* "interrupt" is a special Turbo C compiler directive which causes
all the registers to be pushed on entry and restored on exit.  In
addition, the special "return from interrupt" instruction is used to
exit the routine (and interrupts are re-enabled), instead of a normal
"return from subroutine" instruction, which a regular function call
uses.  The "interrupt" directive means you can write interrupt
routines without using assembly language (hooray for increased
productivity!).  */

{
extern void beep(int time); /* function prototype shows the compiler
                                 how a proper function call should
                                 look. */
int i;
beep(30);    /*  All we do is make a little noise, but you can add all
                 kinds of stuff here (as long as you don't make any DOS
                 or BIOS calls, since you might be interrupting one and
                 they aren't re-entrant.  */

/* tell the 8259A Interrupt Controller we're done executing IRQ7 */
outportb(0x20,0x67);    /* specific EOI for IRQ7 */

}

main()  /* this installs the interrupt, sets up the hardware, and
           exits leaving the program resident.  Main is never used
           again.  */
{
    setvect(INT_NUMBER,int_handler);  /* passes the ADDRESS of the
                                         beginning of the
                                         int_handler() function. */

   /* change the bit on the parallel board to allow the -ACK
      interrupt to pass through to IRQ7 on the PC card bus. */
      outportb(PPORT_BASE + 2, 0x10);

   /* zero top bit of OCW1 to allow IRQ7 to be serviced.  Note we get
      the current OCW1, force the top bit to 0 and put it back out --
      this retains the rest of the word (which affects other aspects
      of the machine) to prevent undesirable side effects).
```

routines, directly controlling the hardware so you don't meet a DOS call in a dark alley somewhere. Those subjects are large and complex, and must be handled elsewhere.

INTERFACING BASICS

To make anything happen, you need wires and details of how to connect to the card.

The quick-and-dirty approach to getting the wires you need is to cannibalize a printer cable. (They're pretty cheap, $15-20.) All you need to do is cut off the Centronics connector end, strip all the wires, and use a continuity checker to determine which wires go to which DB-25 pins. The other approach is to use a male DB-25 ribbon-cable connector and some ribbon cable. This has some advantage since the wires in the ribbon cable are all neatly lined up in order.

READING FROM THE OUTSIDE

As noted before, the simplest data-acquisition system you can make is to connect a switch between ground (pins 18-25) and any of pins 1, 14, 16, or 17. If you want switches on other readable pins, they need external pull-ups for reliable operation, ideally 4700 ohms to an external power supply.

You may also connect the readable pins directly to other electronic circuits. The most common circuits you will encounter are CMOS or TTL digital logic, both of which work fine. You can also use the inputs to look at analog circuits, but in no case should any of the circuits (TTL, CMOS, or analog) output more than five volts or less than 0 volts. The ground of the external electronic circuit needs to be connected to the ground of the printer board.

WRITING TO THE OUTSIDE

The writable output lines will each provide a few milliamps of current (enough to turn on a light-emitting diode, which can often be useful). If you want to control higher voltages, use optocouplers.

(I've provided an extensive introduction to control electronics in my *Micro Cornucopia* column.)

You can also drive other digital electronic circuits. TTL inputs can be connected directly to TTL outputs, but CMOS inputs (running at 5 volts, of course) require pull-up resistors on the lines to get the TTL outputs to rise to what CMOS considers an acceptable 1.

If you start trying to pass voltage levels across more than 15 or 20 feet, you may experience noise problems. This can be solved electronically (can't everything?) with high-current buffers, but it's much simpler just to move the computer closer.

If you start trying to pass voltage levels across more than 15 or 20 feet, you may experience noise problems.

CAUTION TO THE WIND...

Go ahead—working with the parallel port hardware is perfectly safe. You can't get shocked. You won't burn anything out as long as you don't put more than 5 volts or less than 0 volts on any of the pins. It's the least expensive and least complicated way to bring bits into your PC from the outside world. ■

Bruce Eckel has a BS in applied physics from the University of California, Irvine, and an MS in computer engineering from Cal Poly San Luis Obispo. He writes a hardware/software project column in Micro Cornucopia. *Contact Bruce at: Eisys, 1009 N. 36th St., Seattle, WA 98103.*

Listing may be downloaded from CompuServe as PARALLEL.ARC.

```
     I know all this writing-to-hardware stuff must seem mysterious,
     but if you really want to understand it you have to read the
     8259A manual (not pretty) and stare at the diagrams for the
     printer board (see text). */
     outportb(PIC_OCW1, inportb(PIC_OCW1) & 0x7f);

     keep(0,PROG_SIZE);  /* first parameter is exit status */
}

void
beep(int time)
/* beeps the speaker, carefully restoring the state of the control
   port.  ("Remember, scouts: leave things BETTER than you found
   them...").
   Taken directly from the Turbo C User's Guide, page 275.  */
{
char originalbits, bits;
int i,j;

   /* get the current control port setting */
   bits = originalbits = inportb(0x61);

   for (i = 0; i <= time; i++) {
   /* turn off the speaker for a while */
        outportb(0x61, bits & 0xfc);
        for (j=0; j <= 100; j++)
                ;
   /* now turn it on for awhile */
        outportb(0x61, bits | 2);
        for (j = 0; j <= 100; j++)
                ;
   }

   /* restore the control port setting */
   outportb(0x61, originalbits);
}
```

LISTING 3: PRINTER.H

```
/* PRINTER.H: #defines and typedefs used in printer.c */

#define BASE 0x378  /* base address of parallel printer card */
typedef int boolean;  /* makes usage a little more obvious,
                         like Pascal */
typedef unsigned char byte; /* if it's always positive and never more
                               than 255, why use extra space?
                               (then again, why not?) */

#define READ_ONLY 1      /* Indicates the pin can only be read from */
#define WRITE_ONLY 2     /* Indicates the pin can only be written to */
#define READ_WRITE 3     /* The pin can be read from or written to */
#define GROUND 4         /* This is a ground pin -- can't do either */
#define WRITE_SUCCESS 5 /* Returned by write_pin() upon successful
                           write */

#define NO_PULLUP 6      /* There is no pullup resistor on this pin. */
#define PULLUP_0 7       /* There is a pullup resistor on this pin.
                            You must set the output to '0' before
                            reading it. */
#define PULLUP_1 8       /* There is a pullup resistor on this pin.
                            You must set the output to '1' before
                            reading it. */

#define INVERTED 9  /* A TTL one will appear on the data bus as a 0 */
#define NORMAL 10   /* A TTL one will appear on the data bus as a 1 */
```

LISTING 4: BITS.H

```
/* BITS.H: defines bit patterns for selecting a bit within a byte. */

#define BIT0 0x01    /* bit masks */
#define BIT1 0x02
#define BIT2 0x04
#define BIT3 0x08
#define BIT4 0x10
#define BIT5 0x20
#define BIT6 0x40
#define BIT7 0x80
```

RECURSIVE DATA TYPES IN TURBO PROLOG

Turbo Prolog's ability to define data types recursively allows you to simply declare structures, such as trees, in your program.

Michael Covington

WIZARD

Turbo Prolog is the only widely used programming language that allows the user to define recursive data types. A data type is recursive if it allows structures to contain other structures like themselves. One of the most familiar recursive types is the tree shown in Figure 1. Crucially, each branch of the tree is itself a tree. Other recursive structures include lists—because the tail of every list is itself a list—and more complex linked structures.

Trees are easy to create in Lisp and standard (Edinburgh) Prolog. But in these languages, trees are not a *type*. There is no way to tell the compiler to expect a tree at a particular point in the program. Instead, every procedure must check the types of its arguments when they are actually passed to it. While this leads to greater versatility, it also consumes CPU time.

Turbo Prolog makes this checking unnecessary by requiring the programmer to declare the types of all arguments of procedures. This isn't as restrictive as it sounds because a user-defined type can consist of a set of alternatives rather than a single basic type. More importantly, type definitions can be recursive.

TREES AS A DATA TYPE

Recursive types were popularized by Niklaus Wirth in *Algorithms + Data Structures = Programs*. Wirth, you will recall, invented Pascal—not Prolog—nearly twenty years ago. He didn't implement recursive types in Pascal, but he did discuss what it would be like to have them. If Pascal had recursive types, we would be able to define a tree as something like this:

```
{ Not correct Pascal! }

TYPE treetype RECORD
                name : String[80];
                left,right : treetype
              END;
```

That is: "A tree consists of a name, which is a string, and the left and right subtrees, which are trees." The nearest we can come to this in Pascal is to use pointers and say:

```
TYPE treeptr = ^treetype;
     treetype = RECORD
                  name : String[80];
                  left,right : treeptr
                END;
```

But notice a subtle difference: Now we're talking about the memory representation of a tree, not the structure of the tree itself. We treat the tree as consisting of *cells*, each containing some data plus pointers to two more cells. Turbo Prolog allows truly recursive type definitions in which the pointers are created and maintained automatically. For example, we can define a tree as follows:

```
domains
  treetype = tree(string,treetype,treetype)
```

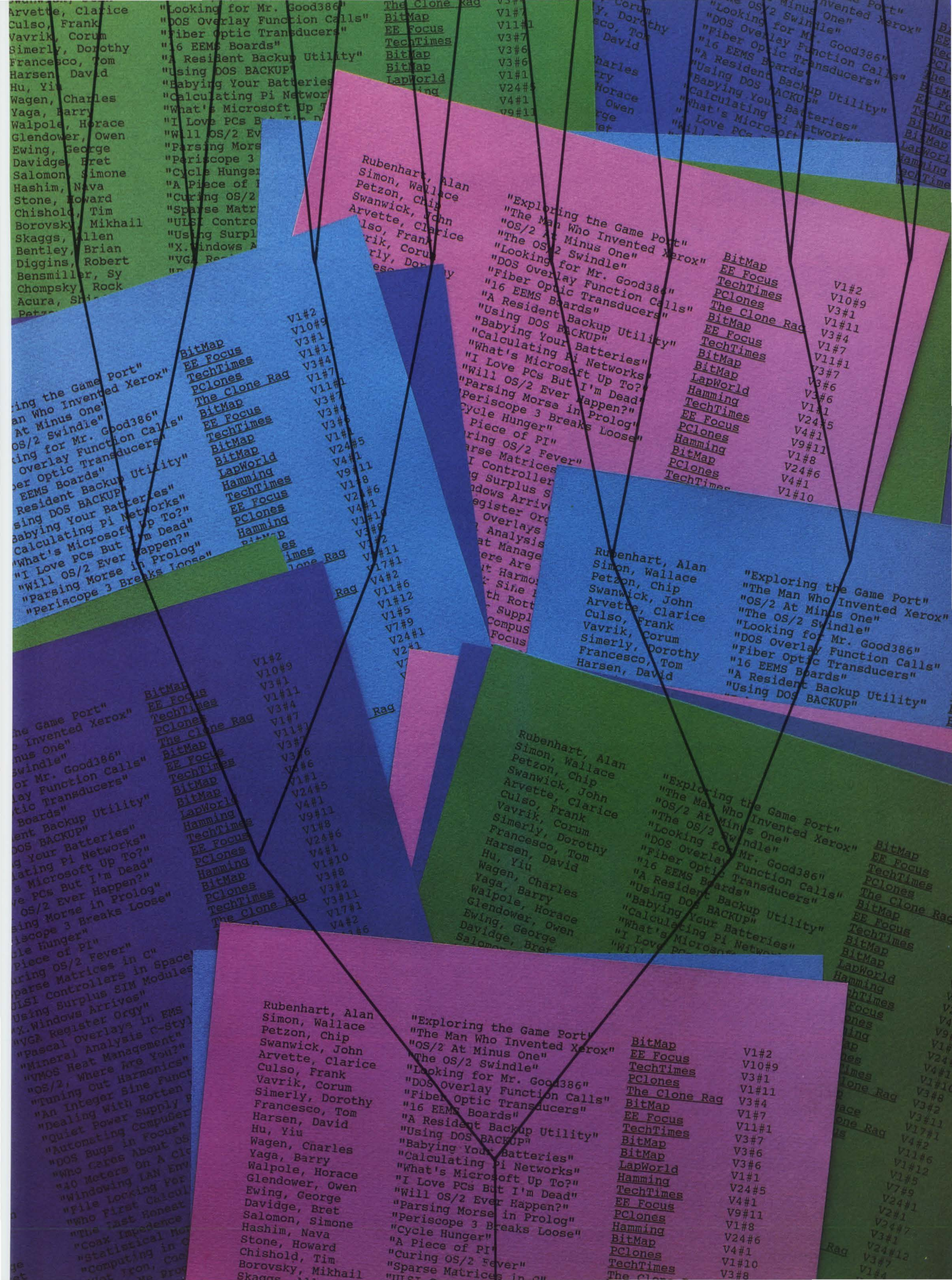
A *domain* is simply a user-defined type. This declaration says that a tree will be written as the type marker, or *functor*, **tree** whose arguments are a string and two more trees.

But this isn't quite right; it provides no way to end the recursion, and in real life, the tree does not go on forever. Some cells *don't* have links to further trees. In Pascal, we could express this by setting some pointers equal to the special value **nil**; here we don't have access to the pointers. The solution is to define two kinds of trees: ordinary ones and empty ones. This is done by allowing a tree to have either of two functors: **tree**, with three arguments, or **empty**, with no arguments.

```
domains
  treetype = tree(string,treetype,treetype); empty()
```

Notice that the names **tree** and **empty** are created by the programmer; neither of them has any predefined meaning in Turbo Prolog. We could have used **xxx** and **yyy** equally well.

Bradley Ream


Rubenhart, Alan
Simon, Wallace
Petzon, Chip
Swanwick, John
Arvette, Clarice
Culso, Frank
Vavrik, Corum
Simerly, Dorothy
Francesco, Tom
Harsen, David
Hu, Yiu
Wagen, Charles
Yaga, Barry
Walpole, Horace
Glendower, Owen
Ewing, George
Davidge, Bret
Salomon, Simone
Hashim, Nava
Stone, Howard
Chishold, Tim
Borovsky, Mikhail
"Exploring the Game Port"
"The Man Who Invented Xerox"
"OS/2 At Minus One"
"The OS/2 Swindle"
"Looking for Mr. Good386"
"DOS Overlay Function Calls"
"Fiber Optic Transducers"
"16 EEMS Boards"
"A Resident Backup Utility"
"Using DOS BACKUP"
"Babying Your Batteries"
"Calculating Pi Networks"
"What's Microsoft Up To?"
"I Love PCs But I'm Dead"
"Will OS/2 Ever Happen?"
"Parsing Morse in Prolog"
"Periscope 3 Breaks Loose"
"Cycle Hunger"
"A Piece of PI"
"Curing OS/2 Fever"
"Sparse Matrices in C"
BitMap
EE Focus
TechTimes
PClones
The Clone Rag
BitMap
EE Focus
TechTimes
BitMap
BitMap
LapWorld
Hamming
TechTimes
EE Focus
PClones
Hamming
BitMap
PClones
TechTimes
V1#2
V10#9
V3#1
V1#11
V3#4
V1#7
V11#1
V3#7
V3#6
V3#6
V1#1
V24#5
V4#1
V9#11
V1#8
V24#6
V4#1
V1#10
V3#8

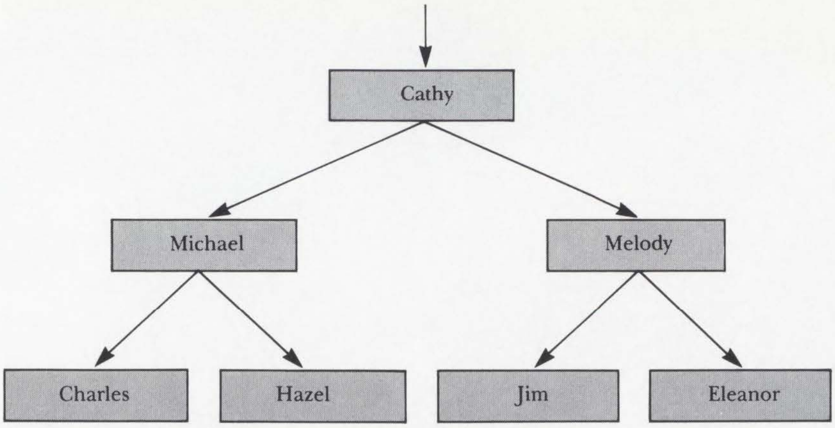


Figure 1. Part of the author's daughter's family tree.

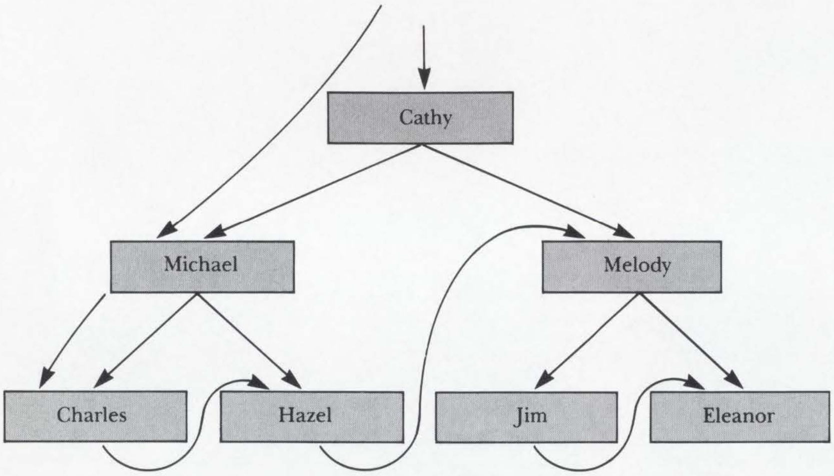


Figure 2. Depth-first traversal of the tree in Figure 1.

RECURSIVE TYPES

We can now write the tree in Figure 1 as it would appear in a Prolog program:

```
tree("Cathy",
  tree("Michael",
    tree("Charles",empty,empty),
    tree("Hazel",empty,empty)),
  tree("Melody",
    tree("Jim",empty,empty),
    tree("Eleanor",empty,empty)))
```

This is indented here for readability, but Prolog does not require indentation, nor are trees indented when you print them out normally. Note that this is not a Turbo Prolog clause; it is just a value to which a variable can be instantiated. Note further that it has no counterpart in Pascal, because in Pascal there is no written representation for a tree.

TREE TRAVERSAL

Before we discuss how to create trees, let's consider what to do with a tree once we have it. One of the most frequent operations is to examine all the cells and process them in some way, either searching for a particular value or collecting all the values. This is known as *traversing* the tree. One basic algorithm for doing so is the following:

1. If the tree is empty, do nothing.
2. Otherwise, process the current node, then traverse the left subtree, then traverse the right subtree.

Like the tree itself, the algorithm is recursive: it treats the left and right subtrees exactly like the original tree. Turbo Prolog expresses it with two clauses, one for empty and one for non-empty trees:

```
traverse(empty).  /* do nothing */

traverse(tree(X,Y,Z)):-
   /* do something with X */
   traverse(Y),
   traverse(Z).
```

A key concept of Prolog is that multiple definitions can be given for a single procedure. These are tried in sequence; if the first one does not match the actual arguments or cannot be carried out successfully, the next definition is used instead.

Our tree traversal algorithm is known as a *depth-first search* because it goes as far as possible down each branch before backing up and trying another branch (Figure 2). To see it in action, look at the program TREESRCH.PRO which traverses a tree, printing all the elements as it encounters them. Given the tree in Figures 1 and 2, TREESRCH.PRO prints:

```
Cathy
Michael
Charles
Hazel
Melody
Jim
Eleanor
```

Of course we could easily adapt the program to perform some other operation on the elements rather than printing them.

Depth-first searching is strikingly similar to the way Prolog searches a knowledge base, arranging the clauses into a tree and pursuing each branch until a query fails. If we wanted, we could describe the tree by means of a set of Prolog clauses such as:

```
father_of("Cathy","Michael").
mother_of("Cathy","Melody").
father_of("Michael","Charles").
mother_of("Michael","Hazel").
```

This is, in fact, preferable if the only purpose of the tree is to express relationships between individuals. However, it leaves us unable to treat the whole tree as a single data object, and as we shall see, complex data structures simplify difficult computational tasks.

CREATING A TREE

One obvious way to create a tree is to write down a nested structure of functors and arguments as we did in the example above. Ordinarily, however, trees are created by computation. In each step, an empty subtree is replaced by a non-empty one through Prolog's process of unification (argument matching). Creating a one-cell tree from an ordinary data item is of course trivial:

```
create_tree(N,
   tree(N,empty,empty)).
```

This says: "If **N** is a data item then **tree(N,empty,empty)** is a one-cell tree containing it." Building a tree structure is almost as easy. The following procedure takes three trees as arguments. It inserts the first tree as the left subtree of the second tree, giving the third tree as the result:

```
insert_left(X,tree(A,_,B),
  tree(A,X,B)).
```

All that Prolog has to do is match the arguments with each other in the proper positions, and the work is done.

Notice that the procedure has no body—there are no explicit "steps" in executing it. All that Prolog has to do is match the arguments with each other in the proper positions, and the work is done.

Suppose for example we want to insert

```
tree("Michael",empty,empty)
```

as the left subtree of:

```
tree("Cathy",empty,empty).
```

We just execute the goal

```
insert_left(tree("Michael,
   empty,empty),
   tree("Cathy",empty,empty),T).
```

and **T** immediately takes on the value:

```
tree("Cathy",tree("Michael",
 empty,empty),empty).
```

This gives us a way to build up trees step by step. Program MAKETREE.PRO demonstrates this technique. In real life the items to be inserted into the tree could come from external input.

Notice that there is no way to change the value of a Turbo Prolog variable once it's instan-

LISTING 1: TREESRCH.PRO

```
/* Traversing a tree by depth-first search
   and printing each element as it is encountered */

domains
  treetype = tree(string,treetype,treetype) ; empty()

predicates
  print_all_elements(treetype)

clauses
  print_all_elements(empty).

  print_all_elements(tree(X,Y,Z)) :-
    write(X),nl,
    print_all_elements(Y),
    print_all_elements(Z).

goal
  print_all_elements(tree("Cathy",
                        tree("Michael",
                             tree("Charles",empty,empty),
                             tree("Hazel",empty,empty)),
                        tree("Melody",
                             tree("Jim",empty,empty),
                             tree("Eleanor",empty,empty)))).
```

LISTING 2: MAKETREE.PRO

```
/* Simple tree building procedures

 create_tree(A,B) puts A in the data field of a one-cell tree giving B
 insert_left(A,B,C) inserts A as left subtree of B giving C
 insert_right(A,B,C) inserts A as right subtree of B giving C    */


domains
  treetype = tree(string,treetype,treetype) ; empty()

predicates
  create_tree(string,treetype)
  insert_left(treetype,treetype,treetype)
  insert_right(treetype,treetype,treetype)

clauses
  create_tree(A,tree(A,empty,empty)).
  insert_left(X,tree(A,_,B),tree(A,X,B)).
  insert_right(X,tree(A,B,_),tree(A,B,X)).

goal

  /* First create some one-cell trees... */

  create_tree("Charles",Ch),
  create_tree("Hazel",H),
  create_tree("Michael",Mi),
  create_tree("Jim",J),
  create_tree("Eleanor",E),
  create_tree("Melody",Me),
  create_tree("Cathy",Ca),

  /* ...then link them up... */

  insert_left(Ch,Mi,Mi2),
  insert_right(H,Mi2,Mi3),
  insert_left(J,Me,Me2),
  insert_right(E,Me2,Me3),
  insert_left(Mi3,Ca,Ca2),
  insert_right(Me3,Ca2,Ca3),

  /* ...and print the result. */

  write(Ca3),nl.
```

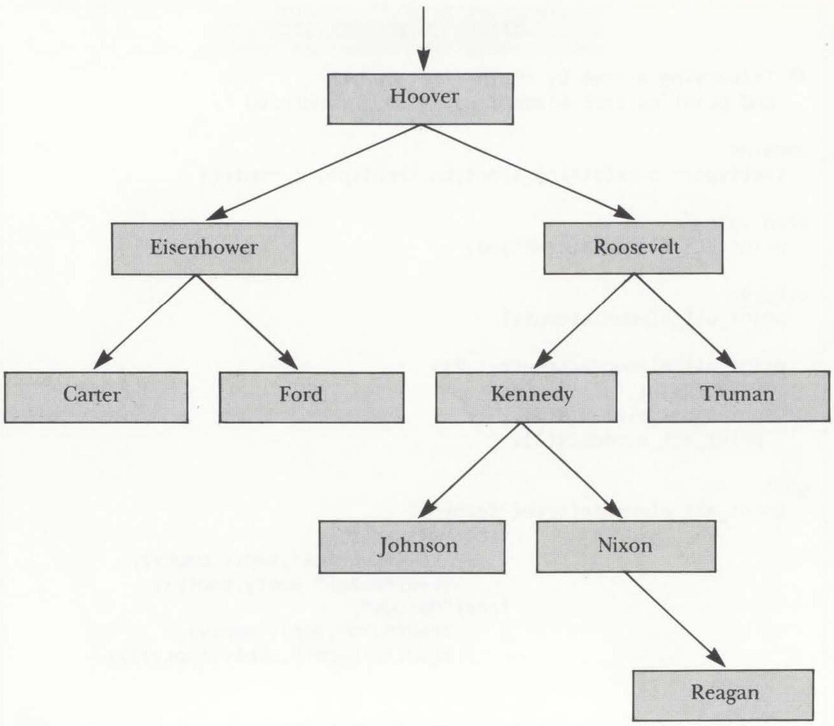


Figure 3. Binary search tree constructed by adding the names Hoover, Roosevelt, Truman, Eisenhower, Kennedy, Johnson, Nixon, Ford, Carter, and Reagan, in that order.

RECURSIVE TYPES

tiated unless backtracking occurs. That's why MAKETREE.PRO uses so many variable names; every time we create a new value we have to have a new variable. The superabundance of variable names here is unusual; more commonly, repetitive procedures obtain new variables by invoking themselves recursively, since each invocation has a distinct set of variables.

BINARY SEARCH TREES

So far, we have been using the tree to represent relationships between its elements. We have already noted that this is not the best use for trees, since Prolog clauses can do the same job. But trees have other uses.

In particular, trees provide a good way to store data items so that they can be found quickly. A tree built for this purpose is called a *search tree*, and from the user's point of view, the tree structure carries no information—the tree is merely a faster alternative to a list or array.

Recall that we traverse an ordinary tree by looking at the current cell and then at both of its subtrees. To find a particular item, we may have to look at every cell in the whole tree. Thus the time taken to search the tree with **N** elements is, on the average, proportional to **N**.

A binary search tree is constructed so that we can predict, upon looking at any cell, which of its subtrees our item will be in. This is done by defining an ordering relation on the data items, such as alphabetical or numerical order. We then place items in the left subtree if they precede the item in the current cell and in the right subtree if they follow it. Figure 3 shows an example. Notice that the same names, added in a different order, would produce a somewhat different tree. Notice also that although there are ten names in the tree, we can find any of them in at most five steps.

Every time we look at a cell during a search, we eliminate half of the remaining cells from consideration and the search proceeds very quickly. If the size of the tree were doubled, then only one extra step would typically be needed to search it. This means that the time taken to find an item is, on the average, proportional to $\log_2 N$ (or, in fact, proportional to log n with logarithms to any base).

To build the tree, we start with an empty tree and add items one by one. The procedure for adding an item is the same as for finding one: we simply search for the place where it ought to be, and insert it there. The algorithm is:

Recall that we traverse an ordinary tree by looking at the current cell and then at both of its subtrees. To find a particular item, we may have to look at every cell in the whole tree.

1. If the current node is an empty tree, insert the item there.
2. Otherwise, compare the item to be inserted with the item stored in the current node. Insert the item into the left subtree or the right subtree, depending on the result of the comparison.

In Prolog, this requires three clauses, one for each situation. The first clause is:

```
insert(NewItem,empty,
   tree(NewItem,empty,empty)):-!.
```

That is: "The result of inserting **NewItem** into **empty** is **tree(New Item,empty,empty).**" The exclamation mark is called a *cut*; it means that if this clause can be used successfully, no other clauses should be tried.

The second and third clauses take care of insertion into nonempty trees:

```
insert(NewItem,
   tree(Element,Left,Right),
   tree(Element,NewLeft,Right)):-
      NewItem < Element,!,
      insert(NewItem,Left,NewLeft).
```

```
insert
  (NewItem,
   tree(Element,Left,Right),
   tree(Element,Left,NewRight)) :-
   insert(NewItem,Right,NewRight).
```

That is, if **NewItem<Element**, we insert into the left subtree; otherwise we insert into the right subtree. Notice that because of the cuts, we get to the third clause only if neither of the preceding clauses has succeeded.

TREE-BASED SORTING

Once we have built the tree, it's easy to retrieve all the items in alphabetical order. The algorithm is again a variant of the depth-first search:

1. If the tree is empty, do nothing.
2. Otherwise, retrieve all the items in the left subtree, then the current element, then all the items in the right subtree.

Or, in Prolog:

```
retrieve_all(empty).
           /* Do nothing */

retrieve_all(tree(Item,
   Left,Right)):-
     retrieve_all(Left),

     <do something to item>

     retrieve_all(Right).
```

A sequence of items can be sorted by inserting them into a tree and then retrieving them in order. For N items, this takes time proportional to N log N, because both insertion and retrieval take time proportional to log N, and each of them has to be done N times. No faster sorting algorithm is known.

Program TREESORT.PRO is a useful DOS utility that uses this technique to alphabetize any standard DOS text file, line by line. To test it, I sorted the 428-line README file distributed with Turbo Prolog 1.1; TREESORT.PRO was more than five times as fast as SORT.EXE, the sort program provided by DOS. Clearly, tree-based sorting is efficient.

TREESORT.PRO has just one quirk. Turbo Prolog's built-in predicate **readln** expects every file to end with the same sequence as standard DOS text files: return (ASCII character 13), linefeed (character 10), and end-of-file character Ctrl-Z (character 26).

Files created with Borland editors often end with just Ctrl-Z at

LISTING 3: TREESORT.PRO

```
/* TREESORT.PRO   Michael A. Covington 1987

   Fast tree-based sort program.
   Reads lines from a file, sorts them in alphabetical
   order, and writes them on another file.

   The input file must be a normal DOS text file.
   That is, the end-of-file mark (Ctrl-Z) must be
   preceded by an end-of-line mark (Ctrl-M Ctrl-J).
   Otherwise the Ctrl-Z will be moved up into the
   middle of the file making it end prematurely.
*/

domains
  treetype = tree(string,treetype,treetype) ; empty()
  file     = infile ; outfile

predicates
  main
  read_input(treetype)
  read_input_aux(treetype,treetype)
  insert(string,treetype,treetype)
  write_output(treetype)

clauses

/*
 *   Main procedure, invoked by goal
 */

main :-
  clearwindow,
  write("Turbo Prolog Treesort / M. Covington 1987\n\n"),

  write("File to read:  "),
  readln(In),
  openread(infile,In),

  write("File to write: "),
  readln(Out),
  openwrite(outfile,Out),

  readdevice(infile),
  read_input(Tree),

  writedevice(outfile),
  write_output(Tree),
  closefile(outfile).

main :-
  /* Execution drops to this clause if   */
  /* anything in the preceding one fails */
  closefile(outfile),
  writedevice(screen),
  write("Unable to perform sort.\n"),
  write("Probable cause: Missing file.\n").
```

```
/*
 * read_input(Tree)
 *   reads lines from the current input device until EOF, then
 *   instantiates Tree to the binary search tree built therefrom
 */

read_input(Tree) :-
  read_input_aux(empty,Tree).

/*
 * read_input_aux(Tree,NewTree)
 *  reads a line, inserts it into Tree giving NewTree,
 *  and calls itself recursively unless at EOF.
 */

read_input_aux(Tree,NewTree) :-
  readln(S),
  !,
  insert(S,Tree,Tree1),
  read_input_aux(Tree1,NewTree).

read_input_aux(Tree,Tree).
  /* If the first clause failed, we are at EOF.          */
  /* So the second clause succeeds with no further action. */

/*
 * insert(Element,Tree,NewTree)
 *   inserts Element into Tree giving NewTree.
 */

insert(NewItem,empty,tree(NewItem,empty,empty)) :- !.

insert(NewItem,tree(Element,Left,Right),
               tree(Element,NewLeft,Right)) :-
     NewItem < Element,
     !,
     insert(NewItem,Left,NewLeft).

insert(NewItem,tree(Element,Left,Right),
               tree(Element,Left,NewRight)) :-
     insert(NewItem,Right,NewRight).

/*
 * write_output(Tree)
 *   writes out the elements of Tree in alphabetical order.
 */

write_output(empty).   /* Do nothing */

write_output(tree(Item,Left,Right)) :-
     write_output(Left),
     write(Item),nl,
     write_output(Right).


goal
  main.
```

the end of the final line. When this occurs, **readln** treats the Ctrl-Z as a readable character. During the sort process, TREESORT.PRO carries the Ctrl-Z to whatever position the line occupies in the final sequence of lines in the sorted file. The result is a file that seems to be the right size but ends abruptly in the wrong place when edited or printed, since the end-of-file marker has been moved to somewhere in the middle of the file.

This could be corrected by modifying the program to test whether each line ends in Ctrl-Z and, if so, delete the Ctrl-Z. This, however, would take extra CPU time. Hopefully, in the future Borland will modify **readln** so that Ctrl-Z is recognized as terminating the line as well as the file. ■

TREESORT.PRO was more than five times as fast as SORT.EXE, the sort program provided by DOS. Clearly, tree-based sorting is efficient.

SUGGESTED READING

Covington, Michael. "Procedural Algorithms in Prolog." *PC Tech Journal*, Vol. 5, No. 3 (March 1987), pp. 159-164.

Wirth, Niklaus. *Algorithms + Data Structures = Programs*, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

Michael Covington does artificial intelligence research at the University of Georgia and has written about 100 magazine articles relating to microcomputers.

Listings may be downloaded from CompuServe as RECURSE.ARC.

EXTRACTING ROUTINES FROM THE TURBO PROLOG TOOLBOX

If you don't need the entire toolbox, take what you need—it's easier than you might think.

Dan Shafer

PROGRAMMER

Learning how to use the Turbo Prolog Toolbox effectively and efficiently can greatly streamline your Turbo Prolog programming tasks. In this article, we will take a look at the basic concepts involved in using the Turbo Prolog Toolbox routines. We will examine differences in memory utilization between the two primary methods of using these routines. Finally, we will discuss some of the most useful Toolbox routines in depth.

TWO PATHS

There are at least two ways to use the routines supplied in the Turbo Prolog Toolbox. The routines are lumped together for convenience in files of related predicates. For example, the file MENU.PRO, which we will use extensively in this discussion, contains three predicates: **menu**, **menu_leave** and **menu_mult**.

In many situations, you may be content with simply using an include file, the method shown in the Turbo Prolog Toolbox manual. However, there are times when memory constraints can become significant, particularly if your program involves the need to include several of the Turbo Prolog Toolbox files. In that event, you may want to save space by retrieving only the specific routines you need for your program from the files supplied by Borland. Doing so requires a number of steps.

First, you should familiarize yourself with the predicates in each of the potential include files with which your program will work. If the application involves a complex usage of several include files, you may want to make a written list of the predicates in each for later reference.

Second, you should go through your code and be sure which of the predicates in each include file your program is actually using. Check them off the list as you find a need for them so that when you are done, you will have a complete list of the predicates your program needs from the include files.

Third, edit from each of the include files those clauses your program is not using. In doing so, be sure to remove *all* of the clauses for an unneeded predicate. Once you've removed the clauses for a given predicate, it's a good idea to immediately remove the predicates declaration. If you forget to remove this declaration you will receive an error message at compile time.

Also, you may wish to remove any domain declarations associated with these unneeded clauses. If you do this, though, be sure that the domain declarations are not used by other, needed predicates in your program or in the include file. Otherwise, you will once again run into apparent bugs at compile time.

It is a good idea, by the way, to save these modified include files under different names from those used by Borland on the distribution disks. That way, if you need the original include file in a later project, you won't find yourself wondering where the missing predicates went.

With all of this done, make backup copies of your newly modified files and then compile the code. If you have a multi-file project, be sure that an appropriate .PRJ file exists before you attempt to compile.

"STANDARD" INCLUDE FILES

There are two Turbo Prolog Toolbox files which every program that uses Toolbox routines generally incorporates: TPREDS.PRO and TDOMS.PRO. The former contains some frequently used predicates and the latter some almost universally required domain declarations.

However, just because Borland supplies these files and recommends their usage doesn't mean that they don't represent fertile ground when looking for more places to save space if you need it in your compiled Turbo Prolog program. If you can identify domains and predicates that are defined in those files and not needed in your program or in any of the include files from the Toolbox, you should feel free to modify them. Again, you should save them under different names.

LISTING 1: MYMENU.PRO

```
/**************************************************************
             Listing 1.
             Turbo Prolog Toolbox Article

        This program demonstrates the use of extracted
        predicates from files furnished on the Turbo Prolog
        Toolbox distribution disks.  It creates two new
        INCLUDE files from those on the distribution disks:
        MENPREDS.PRO from TPREDS.PRO and NEWMENU.PRO from
        MENU.PRO.  In both cases, unneeded predicates have
        been removed from the original files and the modified
        files saved separately.
**************************************************************/
include "tdoms.pro"     /* no modifications - nothing to delete */
include "menpreds.pro"  /* as modified for this program        */
include "newmenu.pro"   /* as modified for this program        */

predicates
        member(integer,integerlist)
        fun(integerlist)
        power_user(integerlist)
        developer(integerlist)
        fun_only(integerlist)
        all_work(integerlist)
        crazy(integerlist)
        write_it(integerlist)
goal

        clearwindow,
        menu_mult(10,10,7,7,["Arcade-style games","Databases",
                "Expert Systems Development","Math Research",
                "Programming","Spreadsheets","Telecommunications",
                "Text Adventure Games","Word Processing"],
                "Select all of the things for which you use a
                 computer",
                [],Choices),
        write_it(Choices).

clauses

/* These predicates permit us to categorize responses from the
menu in such a way as to make the computer's analysis and advice
more interesting and less expected.  To see what they are intended
to mean, look at the "write_it" clauses below.  All work basically
the same way.  The menu choices are categorized into "fun", "power
user", "developer" and "crazy."  Since the user is making a
multiple-choice selection, his/her replies are stored in a list
called Choices (as set up in the "menu_mult" predicate call in
the goal, above).  Then we just use the "member" predicate to
determine if a particular value is part of that list or not. */

        fun(Choices) :-
                member(1,Choices).

        fun(Choices) :-
                member(8,Choices).

        power_user(Choices) :-
                member(2,Choices).

        power_user(Choices) :-
                member(6,Choices).

        power_user(Choices) :-
                member(7,Choices).

        power_user(Choices) :-
                member(9,Choices).

        developer(Choices) :-
```

EXTRACTING ROUTINES

TPREDS.PRO is a particularly useful place to look when saving space. There are 14 predicates defined in the file, and they are logically grouped into these categories:

- miscellaneous
- letter-selection routines to scan a string
- predicates to adjust the locations of windows on the display
- keyboard input handlers

If your program doesn't need one or more of these categories of predicates, it's easy to edit the TPREDS.PRO file and remove the unneeded blocks.

WHAT CAN YOU SAVE?

Just to get an idea of the degree of compiled file-size savings attainable by the methods we've just described, let's take a look at the simple XSTATUS.PRO program on page 12 of the Turbo Prolog Toolbox manual. That program demonstrates the display and use of a status line. It uses three include files, the standard TDOMS.PRO and TPREDS.PRO, along with STATUS.PRO.

A look at XSTATUS.PRO reveals that, of the predicates defined in TPREDS.PRO, it only uses the keyboard reader routines. Thus we can eliminate all but the last section of code from that file, an action that results in a saving of 4,137 bytes, reducing the file from 5,377 bytes to 1,240 bytes.

As you might expect, the savings in TDOMS.PRO are less dramatic. We can eliminate only a small number of domain declarations: **ROW**, **COL**, **LEN**, **STRINGLIST** and **INTEGERLIST**. This editing creates a new file of 687 bytes, compared to the original 753 bytes in TDOMS.PRO.

Before any of these changes are made, compile the XSTATUS.PRO file as it's furnished by Borland, and you will find the resulting .EXE file occupies 42,594 bytes. After removing the unneeded predicates and domain declarations, the compiled result takes up 42,295 bytes, a savings of only 299 bytes.

Sometimes, of course, 300 bytes can be important, particularly when you are bumping against the upper limits of your PC's memory. We should also note that if the XSTATUS.PRO file didn't require all of the predicates in the STATUS.PRO file, the memory savings would be greater. The fact that XSTATUS.PRO is provided to demonstrate all of the predicates in STATUS.PRO means that we can't eliminate any of the predicates in STATUS.PRO.

But if you were using one of the MENU.PRO routines in your program, it's quite likely you would need only one of the three types of menu:

- a single-choice menu that disappears when the user makes a selection (using the **menu** predicate)
- a single-choice menu that stays on the screen after the user makes a selection (using the **menu_leave** predicate)
- a multiple-selection menu (using **menu_mult**).

In that event, you can delete the other two types of menu predicates from the MENU.PRO file, save it

continued on page 90

Borland's Turbo Prolog, the natural introduction to Artificial Intelligence

Nothing says Artificial Intelligence has to be complicated, academic or obscure. Turbo Prolog® proves that. It's intelligent about Intelligence and teaches you carefully and concisely so that you soon feel right at home.

Which is not to say that Artificial Intelligence is an easy concept to grasp, but there's no easier way to grasp it than with Turbo Prolog's point-by-point, easy-to-follow Tutorial.

Turbo Prolog is for both beginners and professional programmers

Because of Turbo Prolog's natural logic, both beginners and accomplished programmers can quickly build powerful applications—like expert systems, natural language interfaces, customized knowledge bases and smart information-management systems. Turbo Prolog is a 5th-generation language that almost instantly puts you and your programs into a fascinating new dimension. Whatever level you work at, you'll find Turbo Prolog both challenging and exhilarating.

Turbo Prolog is to Prolog what Turbo Pascal is to Pascal

Borland's Turbo Pascal® and Turbo C® are already famous, and our Turbo Prolog is now just as famous.

Turbo Pascal is so fast and powerful that it's become a worldwide standard in universities, research centers, schools, and with programmers and hobbyists. Turbo Prolog, the natural language of Artificial Intelligence, is having the same dramatic impact.

Borland's new Turbo Prolog Toolbox adds 80 powerful tools

Turbo Prolog Toolbox™ includes 80 new tools and 8000 lines of source code that can easily be incorporated into your own programs. We've included 40 sample programs that show you how to put these Artificial Intelligence tools to work.

Already one of the most powerful computer programming languages ever conceived, Turbo Prolog is now even more powerful with the new Toolbox addition.

The Critics' Choice

"I really wouldn't want to choose *the* most important MS-DOS product developed last year, but if I had to, I think it would be Borland's Prolog, which gives users a whole new way to think about how to use their computers.

Jerry Pournelle, 'A User's View,' InfoWorld

Turbo Prolog offers the fastest and most approachable implementation of Prolog.

Darryl Rubin, AI Expert"

Turbo Prolog Features:

- ☑ A complete development environment
- ☑ A fast incremental compiler
- ☑ A full-screen interactive editor
- ☑ Graphic and text window support
- ☑ Tools to build your own expert systems
- ☑ Full DOS access and support
- ☑ A free Tutorial
- ☑ The free GeoBase™ natural query language database
- ☑ An easy-to-understand 200-page manual

All this and more for only $99.95!

The new Turbo Prolog Toolbox includes:

- ☑ 80 tools
- ☑ 8000 lines of source code that can easily be incorporated into your own programs
- ☑ 40 sample programs
- ☑ Business graphics
- ☑ File transfers from Reflex,® dBASE III,® 1-2-3® and Symphony®
- ☑ Sophisticated user-interface design
- ☑ Screen layout and handling—including virtual screens
- ☑ Complete communications package including XMODEM protocol
- ☑ Parser generation
- ☑ Opportunity to design AI applications quickly
- ☑ 5th-generation language and supercomputer power to your IBM®PC and compatibles

Only $99.95!

BORLAND
INTERNATIONAL

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066
(408) 438-8400 TELEX: 172373

For the dealer nearest you or to order by phone call
(800) 255-8008
in CA (800) 742-1133 in Canada (800) 237-1136

All Borland products are trademarks or registered trademarks of Borland International, Inc. or Borland/Analytica, Inc. Other brand and product names and trademarks or registered trademarks of their respective holders. Copyright 1987 Borland International

BI-1120

```
                member(3,Choices).

        developer(Choices) :-
                member(5,Choices).

        crazy(Choices) :-
                member(4,Choices). /* Isn't any math researcher? */

/* If the user indicates only game choices, s/he's categorized as
   "fun_only" for our purposes. */

        fun_only(Choices) :-
                fun(Choices),
                not(power_user(Choices)),
                not(developer(Choices)).

/* Similarly, if the user indicates no fun choices at all, but only
   power user or programmer uses, s/he's categorized as
   "all_work". */

        all_work(Choices) :-
                not(fun(Choices)),
                power_user(Choices).

        all_work(Choices) :-
                not(fun(Choices)),
                developer(Choices).

        all_work(Choices) :-
                not(fun(Choices)),
                not(crazy(Choices)).

/* Here come the messages of advice -- words of wisdom, if you will -
   for each category of respondent. */

        write_it(Choices) :-
                fun_only(Choices),
                write("Maybe it's time you got serious about your
                        computer!").

        write_it(Choices) :-
                all_work(Choices),
                write("All work and no play makes for rich, but dull,
                        programmers!").

        write_it(Choices) :-
                crazy(Choices),
                write("You are definitely a strange duck!").

        write_it(Choices) :-
                not(fun_only(Choices)),
                not(all_work(Choices)),
                not(crazy(Choices)),
                write("You seem to be pretty well-balanced.").

/* Standard "member" definition, but not included with Turbo
   Prolog. */

        member(X,[X|_]).
        member(X,[_|Y]) :-
                member(X,Y).
```

continued from page 88

under a different name and eliminate enough file space to make the effort worthwhile.

A BRIEF MENU EXAMPLE

Let's create a small demonstration program to show how to include a single routine from the MENU.PRO file rather than reading in the entire file with an **include** statement.

Listing 1 displays a menu of choices from which the user makes multiple selections. Then the menu is removed and the program displays some advice based on the user's selections. In your own program, of course, you could do anything you wish with the user's reply.

Listing 2 is the MENU.PRO file provided on the Turbo Prolog Toolbox distribution diskette. The shaded areas indicate those lines removed to create the file MYMENU.PRO which is included in the NEWMENU.PRO program in Listing 1. These are lines that can be eliminated because we are only using one of the menu types.

Listing 3 shows the TPREDS.PRO file supplied by Borland with the Toolbox. Again, the shaded area represents the lines removed to create the MENPREDS.PRO file included in MYMENU.PRO. (The TDOMS.PRO file had no possible deletions in this application.)

Before any of the include files are changed, if we compile MYMENU.PRO, the resulting .EXE file occupies 44,730 bytes of space on the disk. After we make the changes indicated by the shading in the listings and then compile MYMENU.PRO, we create an .EXE file which only needs 44,671 bytes of space, a saving of only 59 bytes. This is despite the fact that the reduced file sizes are more than 1,750 bytes smaller after unneeded predicates are removed.

You will find that this pattern remains fairly consistent. For example, if we chose to implement a simple one-choice menu that goes away after the user's selection is made, we would find to our surprise that we save even less total compiled file size than when we do our multiple-choice menu (44 bytes rather than 59).

BE EFFICIENT

It's possible to save space by extracting specific predicates and groups of predicates from the files supplied on the Turbo Prolog Toolbox distribution diskettes. Often, the amount of space saved will not seem significant. But when larger files are included in your code, and your program makes use of small portions of their content, this is definitely a technique to keep in mind. ■

Dan Shafer is the author of the best-selling Turbo Prolog Primer *and the recently released* Advanced Turbo Prolog Programming, *both published by Howard W. Sams & Co. He lives in Redwood City, California.*

Listings may be downloaded from CompuServe as EXTRACTP.ARC.

LISTING 2: MENU1.PRO

```
/*****************************************************************

    Turbo Prolog Toolbox
    (C) Copyright 1987 Borland International.

                              menu
  Implements a popup menu with at most 23 possible choices.
  For more than 23 possible choices use longmenu.

  The up and down arrow keys can be used to move the bar.
  RETURN or F10  will select an indicated item.
  Pressing Esc aborts menu selection and returns zero.

  The arguments to menu are:

  menu(ROW,COL,WINDOWATTR,FRAMEATTR,STRINGLIST,HEADER,STARTCHOICE,
       SELECTION)

        ROW and COL determines the position of the window
        WATTR and FATTR determine the attributes for the window
                and its frame - if FATTR is zero there
                will be no frame around the window.
        STRINGLIST is the list of menu items
        HEADER is the text to appear at the top of the menu window
        STARTCHOICE determines where the bar should be placed.

  Ex:    menu(5,5,7,7,[this,is,a,test],"select word",0,CHOICE)

*****************************************************************/

PREDICATES
  menu(ROW,COL,ATTR,ATTR,STRINGLIST,STRING,INTEGER,INTEGER)
  menuinit(ROW,COL,ATTR,ATTR,STRINGLIST,STRING,ROW,COL)
  menu1(SYMBOL,ROW,ATTR,STRINGLIST,ROW,COl,INTEGER)
  menu2(KEY,STRINGLIST,ROW,ROW,ROW,SYMBOL)

CLAUSES
  menu(ROW,COL,WATTR,FATTR,LIST,HEADER,STARTCHOICE,CHOICE) :-
        menuinit(ROW,COL,WATTR,FATTR,LIST,HEADER,NOOFROW,LEN),
        ST1=STARTCHOICE-1,max(0,ST1,ST2),MAX=NOOFROW-1,
        min(ST2,MAX,STARTROW),
        menu1(cont,STARTROW,WATTR,LIST,NOOFROW,LEN,CHOICE),
        removewindow.

  menuinit(ROW,COL,WATTR,FATTR,LIST,HEADER,NOOFROW,NOOFCOL):-
        maxlen(LIST,0,MAXNOOFCOL),
        str_len(HEADER,HEADLEN),
        HEADL1=HEADLEN+4,
        max(HEADL1,MAXNOOFCOL,NOOFCOL),
        listlen(LIST,LEN), LEN>0,
        NOOFROW=LEN,
        adjframe(FATTR,NOOFROW,NOOFCOL,HH1,HH2),
        adjustwindow(ROW,COL,HH1,HH2,AROW,ACOL),
        makewindow(81,WATTR,FATTR,HEADER,AROW,ACOL,HH1,HH2),
        writelist(0,NOOFCOL,LIST).

  menu1(cont,ROW,ATTR,LIST,MAXROW,NOOFCOL,CHOICE):-!,
        reverseattr(ATTR,REV),
        field_attr(ROW,0,NOOFCOL,REV),
        cursor(ROW,0),
        readkey(KEY),
        field_attr(ROW,0,NOOFCOL,ATTR),
        menu2(KEY,LIST,MAXROW,ROW,NEXTROW,CONT),
        menu1(CONT,NEXTROW,ATTR,LIST,MAXROW,NOOFCOL,CHOICE).
  menu1(esc,ROW,_,_,_,_,CHOICE):-!,CHOICE=ROW+1.
  menu1(_,ROW,ATTR,_,_,NOOFCOL,CHOICE):-
        CHOICE=ROW+1,
        reverseattr(ATTR,REV),
        field_attr(ROW,0,NOOFCOL,REV).
```

```
  menu2(esc,_,_,_,-1,esc):-!.
  menu2(fkey(10),_,_,ROW,ROW,stop):-!.
  menu2(char(C),LIST,_,_,CH,selection):-tryletter(C,LIST,CH),!.
/*menu2(fkey(1),_,_,ROW,ROW,cont):-!,help.  If a help system is
  used */
  menu2(cr,_,_,ROW,CH,selection):-!,CH=ROW.
  menu2(up,_,_,ROW,NEXTROW,cont):-ROW>0,!,NEXTROW=ROW-1.
  menu2(down,_,MAXROW,ROW,NEXTROW,cont):-
        NEXTROW=ROW+1,
        NEXTROW<MAXROW,!.
  menu2(end,_,MAXROW,_,NEXT,cont):-!,NEXT=MAXROW-1.
  menu2(pgdn,_,MAXROW,_,NEXT,cont):-!,NEXT=MAXROW-1.
  menu2(home,_,_,_,0,cont):-!.
  menu2(pgup,_,_,_,0,cont):-!.
  menu2(_,_,_,ROW,ROW,cont).


/*****************************************************************/
/*                        menu_leave                             */
/* As menu but the window is not removed on return.              */
/*****************************************************************/

PREDICATES
  menu_leave(ROW,COL,ATTR,ATTR,STRINGLIST,STRING,INTEGER,INTEGER)

CLAUSES
  menu_leave(ROW,COL,WATTR,FATTR,LIST,HEADER,STARTCHOICE,CHOICE) :-
        menuinit(ROW,COL,WATTR,FATTR,LIST,HEADER,NOOFROW,NOOFCOL),
        ST1=STARTCHOICE-1,max(0,ST1,ST2),MAX=NOOFROW-1,
        min(ST2,MAX,STARTROW),
        menu1(cont,STARTROW,WATTR,LIST,NOOFROW,NOOFCOL,CHOICE).


/*****************************************************************
                          menu_mult
 Implements a popup-menu which allows a multiple number of
 selections.

 Each selection is made by pressing RETURN. All selections are
 then activated by pressing F10.

  The arguments to menu_mult are:

  menu(ROW,COL,WINDOWATTR,FRAMEATTR,STRINGLIST,HEADER,STARTLIST,
       NEWLIST)

        ROW and COL determine the position of the window
        WATTR and FATTR determine the attributes for the window
                and its frame - if FATTR is zero there
                will be no frame around the window.
        STRINGLIST is the list of menu items
        HEADER is the text to appear at the top of the menu window
        STARTLIST determines the items to be highlighted when
                  the menu is first displayed
        NEWLIST   contains the list of selections

  Ex: menu_mult(5,5,7,7,[this,is,a,test],"select words",[1],NEWLIST)

*****************************************************************/
```

```
PREDICATES
  menu_mult(ROW,COL,ATTR,ATTR,STRINGLIST,STRING,INTEGERLIST,
          INTEGERLIST)
  multmenu1(SYMBOL,ROW,ATTR,STRINGLIST,ROW,COL,INTEGERLIST,
          INTEGERLIST)
  highlight_selected(INTEGERLIST,COL,ATTR)
  handle_selection(INTEGER,INTEGERLIST,INTEGERLIST,COL,ATTR)
  try_del(INTEGER,INTEGERLIST,INTEGERLIST,COL,ATTR)

CLAUSES

  menu_mult(ROW,COL,WATTR,FATTR,LIST,HEADER,STARTCHLIST,CHLIST) :-
        menuinit(ROW,COL,WATTR,FATTR,LIST,HEADER,NOOFROW,NOOFCOL),
        multmenu1(cont,0,WATTR,LIST,NOOFROW,NOOFCOL,STARTCHLIST,
                CHLIST),
        removewindow.

  multmenu1(stop,_,_,_,_,_,CHL,CHL):-!.
  multmenu1(esc,_,_,_,_,_,[]):-!.
  multmenu1(selection,ROW,ATTR,LIST,MAXROW,NOOFCOL,OLDCHLIN,CHLOUT):-
        CHOICE=1+ROW,
        handle_selection(CHOICE,OLDCHLIN,NEWCHLIN,NOOFCOL,ATTR),
        multmenu1(cont,ROW,ATTR,LIST,MAXROW,NOOFCOL,NEWCHLIN,CHLOUT).
  multmenu1(cont,ROW,ATTR,LIST,MAXROW,NOOFCOL,CHLIN,CHLOUT):-
        reverseattr(ATTR,REV),
        highlight_selected(CHLIN,NOOFCOL,REV),
        cursor(ROW,0),
        readkey(KEY),
        menu2(KEY,LIST,MAXROW,ROW,NEXTROW,CONT),
        multmenu1(CONT,NEXTROW,ATTR,LIST,MAXROW,NOOFCOL,CHLIN,CHLOUT).

  highlight_selected([],_,_).
  highlight_selected([H|T],L,ATTR):-
        ROW=H-1,
        field_attr(ROW,0,L,ATTR),
        highlight_selected(T,L,ATTR).

  try_del(SELECTION,[SELECTION|REST],REST,LEN,ATTR):-
        ROW=SELECTION-1,
        field_attr(ROW,0,LEN,ATTR),!.
  try_del(SELECTION,[H|REST],[H|REST1],LEN,ATTR):-
        try_del(SELECTION,REST,REST1,LEN,ATTR).

  handle_selection(SELECTION,OLDCHIN,NEWCHIN,LEN,ATTR):-
        try_del(SELECTION,OLDCHIN,NEWCHIN,LEN,ATTR),!.
  handle_selection(SELECTION,OLDCHIN,[SELECTION|OLDCHIN],_,_).
```

LISTING 3: TPREDS1.PRO

```
/*****************************************************************
     Turbo Prolog Toolbox
     (C) Copyright 1987 Borland International.

 This module includes some routines which are used in nearly
 all menu and screen tools.
*****************************************************************/
/*****************************************************************/
/*              repeat                                           */
/*****************************************************************/

PREDICATES
  nondeterm repeat

CLAUSES
  repeat.
  repeat:-repeat.


/*****************************************************************/
/*              miscellaneous                                    */
/*****************************************************************/

PREDICATES
  maxlen(STRINGLIST,COL,COL)    /*The length of the longest string*/
  listlen(STRINGLIST,ROW)       /* The length of a list           */
  writelist(ROW,COL,STRINGLIST) /* used in the menu predicates    */
  reverseattr(ATTR,ATTR)        /* Returns the reversed attribute */
  min(ROW,ROW,ROW) min(COL,COL,COL)
  min(LEN,LEN,LEN) min(INTEGER,INTEGER,INTEGER)
  max(ROW,ROW,ROW) max(COL,COL,COL)
  max(LEN,LEN,LEN) max(INTEGER,INTEGER,INTEGER)

CLAUSES
  maxlen([H|T],MAX,MAX1) :-
        str_len(H,LENGTH),
        LENGTH>MAX,!,
        maxlen(T,LENGTH,MAX1).
  maxlen([_|T],MAX,MAX1) :- maxlen(T,MAX,MAX1).
  maxlen([],LENGTH,LENGTH).

  listlen([],0).
  listlen([_|T],N):-
        listlen(T,X),
        N=X+1.

  writelist(_,_,[]).
  writelist(LI,ANTKOL,[H|T]):-
        field_str(LI,0,ANTKOL,H),
        LI1=LI+1,
        writelist(LI1,ANTKOL,T).

  min(X,Y,X):-X<=Y,!.
  min(_,X,X).

  max(X,Y,X):-X>=Y,!.
  max(_,X,X).

  reverseattr(A1,A2):-
        bitand(A1,$07,H11),
        bitleft(H11,4,H12),
        bitand(A1,$70,H21),
        bitright(H21,4,H22),
        bitand(A1,$08,H31),
        A2=H12+H22+H31.


/*****************************************************************/
/*      Find letter selection in a list of strings               */
/*      Look initially for first uppercase letter.              */
/*      Then try with first letter of each string.              */
/****************************************************************/

PREDICATES
  upc(CHAR,CHAR)  lowc(CHAR,CHAR)
  try_upper(CHAR,STRING)
  tryfirstupper(CHAR,STRINGLIST,ROW,ROW)
  tryfirstletter(CHAR,STRINGLIST,ROW,ROW)
  tryletter(CHAR,STRINGLIST,ROW)

CLAUSES
  upc(CHAR,CH):-
        CHAR>='a',CHAR<='z',!,
        char_int(CHAR,CI), CI1=CI-32, char_int(CH,CI1).
  upc(CH,CH).

  lowc(CHAR,CH):-
        CHAR>='A',CHAR<='Z',!,
        char_int(CHAR,CI), CI1=CI+32, char_int(CH,CI1).
  lowc(CH,CH).

  try_upper(CHAR,STRING):-
        frontchar(STRING,CH,_),
        CH>='A',CH<='Z',!,
        CH=CHAR.
  try_upper(CHAR,STRING):-
        frontchar(STRING,_,REST),
        try_upper(CHAR,REST).

  tryfirstupper(CHAR,[W|_],N,N) :-
        try_upper(CHAR,W),!.
  tryfirstupper(CHAR,[_|T],N1,N2) :-
        N3 = N1+1,
        tryfirstupper(CHAR,T,N3,N2).

  tryfirstletter(CHAR,[W|_],N,N) :-
        frontchar(W,CHAR,_),!.
  tryfirstletter(CHAR,[_|T],N1,N2) :-
        N3 = N1+1,
        tryfirstletter(CHAR,T,N3,N2).

  tryletter(CHAR,LIST,SELECTION):-
        upc(CHAR,CH),tryfirstupper(CH,LIST,0,SELECTION),!.
  tryletter(CHAR,LIST,SELECTION):-
        lowc(CHAR,CH),tryfirstletter(CH,LIST,0,SELECTION).


/*****************************************************************/
/* adjustwindow takes a windowstart and a windowsize and adjusts */
/* the windowstart so the window can be placed on the screen.    */
/* adjframe looks at the frameattribute: if it is different from */
/* zero, two is added to the size of the window                  */
/*****************************************************************/

PREDICATES
  adjustwindow(ROW,COL,ROW,COL,ROW,COL)
  adjframe(ATTR,ROW,COL,ROW,COL)

CLAUSES
  adjustwindow(LI,KOL,DLI,DKOL,ALI,AKOL):-
                LI<25-DLI,KOL<80-DKOL,!,ALI=LI,AKOL=KOL.
  adjustwindow(LI,_,DLI,DKOL,ALI,AKOL):-
                LI<25-DLI,!,ALI=LI,AKOL=80-DKOL.
  adjustwindow(_,KOL,DLI,DKOL,ALI,AKOL):-
                KOL<80-DKOL,!,ALI=25-DLI, AKOL=KOL.
  adjustwindow(_,_,DLI,DKOL,ALI,AKOL):-
                ALI=25-DLI, AKOL=80-DKOL.
```

```
  adjframe(0,R,C,R,C):-!.
  adjframe(_,R1,C1,R2,C2):-R2=R1+2, C2=C1+2.


/****************************************************************/
/*                          Readkey                             */
/* Returns a symbolic key from the KEY domain                   */
/****************************************************************/

PREDICATES
  readkey(KEY)
  readkey1(KEY,CHAR,INTEGER)
  readkey2(KEY,INTEGER)

CLAUSES
  readkey(KEY):-readchar(T),char_int(T,VAL),readkey1(KEY,T,VAL).

  readkey1(KEY,_,0):-!,readchar(T),char_int(T,VAL),readkey2(KEY,VAL).
  readkey1(cr,_,13):-!.
  readkey1(esc,_,27):-!.
  readkey1(break,_,3):-!.
  readkey1(tab,_,9):-!.
  readkey1(bdel,_,8):-!.
  readkey1(ctrlbdel,_,127):-!.
  readkey1(char(T),T,_) .

  readkey2(btab,15):-!.
  readkey2(del,83):-!.
  readkey2(ins,82):-!.
  readkey2(up,72):-!.
  readkey2(down,80):-!.
  readkey2(left,75):-!.
  readkey2(right,77):-!.
  readkey2(pgup,73):-!.
  readkey2(pgdn,81):-!.
  readkey2(end,79):-!.
  readkey2(home,71):-!.
  readkey2(ctrlleft,115):-!.
  readkey2(ctrlright,116):-!.
  readkey2(ctrlend,117):-!.
  readkey2(ctrlpgdn,118):-!.
  readkey2(ctrlhome,119):-!.
  readkey2(ctrlpgup,132):-!.
  readkey2(fkey(N),VAL):- VAL>58, VAL<70, N=VAL-58, !.
  readkey2(fkey(N),VAL):- VAL>=84, VAL<104, N=11+VAL-84, !.
  readkey2(otherspec,_).
```

THINKING IN TURBO PROLOG

A programming language should be a reflection of the way the programmer thinks. To program in Turbo Prolog you have to think the part.

Alex Lane

SQUARE ONE

I remember my first experience with Prolog—in the pre-Turbo days—as being a struggle. My only clear recollection of that night was the realization that I was dealing with something "completely different." As moderator of the Prolog conference on the Byte Information Exchange (BIX), I see that my early experiences are being repeated by other programmers whose roots lie in the conventional-language world. Most certainly, the biggest obstacle to the mastery of the language is learning to think in Prolog.

Before actually sitting down to program in Turbo Prolog, it is important to get a sense of what Prolog is, and what place it occupies in the computing scheme of things.

PROLOG: A SYMBOLIC LANGUAGE

To date, most of the problems that computers have been programmed to solve involve manipulation of numbers. In such programs, problems are solved using addition, subtraction, and other familiar operations. For the past several decades, computer scientists have spent most of their time developing computer languages for better and faster numerical computation.

Some early computer scientists believed computers could be programmed to manipulate symbols as well as numbers. Such programs perform abstract operations on data structures. Unfortunately, such programs also required a lot of random access memory, which just wasn't available in those early computers. Although languages like LISP and Prolog provide tools for symbolic computation, the ability to write such programs was, until recently, restricted to those with access to multimillion-dollar computers.

Prolog is a language based on a branch of formal logic called *predicate calculus*. Predicate calculus provides us with a way to represent propositions, the relationship between propositions, and the means to infer new propositions from existing ones. Since Prolog is a symbolic language, it is particularly well suited for solving problems that involve relations between objects.

THE DECLARATIVE vs. PROCEDURAL DEBATE

The world of programming languages may be divided into two camps: the declarative and the procedural. Languages such as LISP are planted squarely on the declarative side of the fence while BASIC, Pascal, and C are lumped together on the procedural side. Prolog straddles the fence, being both semi-declarative and semi-procedural.

The declarative approach to programming emphasizes the relations defined ("declared") by the program. Declarative programming focuses on describing the problem, leaving out the procedures that tell the computer how to find the solution. In a purely declarative language, you'd simply enter a set of facts and relations about, say, the stock market. Then you'd be able to ask questions like, "What's a good stock to invest in?" and let the computer find an answer. While there are no "purely" declarative languages, Prolog does go a long way in bringing the idea to reality.

In the procedural—more properly called the imperative—approach to programming, the programmer must also describe the steps to be taken to arrive at an answer. If the declarative approach addresses the "what" of a program, the procedural approach addresses both the "what" and the "how."

In practice, the concepts of "declarative" and "procedural" are relative. For example, the BASIC expression

```
X = Y + Z
```

reflects both styles of programming, depending on your point of view. Viewed in the context of the overall program, the statement procedurally specifies an addition and assignment. Viewed in the context of the BASIC interpreter, the statement declaratively says: "Don't bother me with the details about registers, opcodes and the like, just add **Y** and **Z** and store the result in **X**."

While Prolog's declarative nature is normally a big plus in writing programs, it is an obstacle to those who insist on having a great deal of control over how their programs find answers. Just as diehard assembly language programmers decry the loss of control imposed by a language like Pascal, some Pascal programmers initially feel uncomfortable handing over control to Prolog's unification and backtracking mechanisms.

A NEW VOCABULARY

Like most programming languages, Turbo Prolog has its own terms and definitions. If you've read the definitions and found them a bit, well, alien, join the club. However, an understanding of these terms is essential, and is one of the first hurdles to overcome in learning Prolog.

Object. A very basic Prolog concept is the *object*. In the English language, an object would be called a noun. In other words, an object is the person, place, or thing that a relation describes. In Prolog terms it is a general term describing a single element of some type. By *type*, we mean either a standard Turbo Prolog *domain* type, such as a real number, a character or string, or some user-defined domain type. We'll pick domains up again shortly when we discuss Turbo Prolog program structure. For right now, remember: objects, and the relations between them, are what Prolog programming is all about.

Relation. A *relation* is a name that describes a collection of objects, for example:

```
has(bird,wings).
```

Here, the name of the relation is **has** and the objects in the relation are **bird** and **wings**. The number of objects in a relation is called the *arity* of the relation. In our example, the relation **has** has an arity of 2. By the way, a relation need not be associated with any objects at all. Such relations have an arity of 0.

A point that terrifies some novice Prolog programmers is deciding how to represent the relationship between objects. For instance, it may be decided, arbitrarily, that the relation "John likes Mary" is to be expressed in the form

```
likes(john,mary).
```

Having been presented with this (or a similar) example of expressing a relation, many beginners agonize over how to describe a similar relation, such as "John is Mary's father." Given the alternatives

```
father(john,mary).
father(mary,john).
```

must the relation be expressed a certain "correct" way? The answer is no. The relationship between a relation and its objects is pretty arbitrary and really exists only in the programmer's mind. What *is* important is for the relation to be expressed consistently in all clauses. Thus, the programmer must decide on a particular convention and stick to that convention throughout the program.

In addition, it helps to select a relation whose name is meaningful. For example, the expression

```
blivet(john, mary).
```

doesn't communicate anything about the relation between John and Mary. Comments also add to the clarity of the program. It never hurts to include a comment such as

```
/* father(Father,Offspring) :
   "Father is the father of
   Offspring" */
```

in the program source code so that the intended relationship is clear to anyone reading your code (including yourself six months down the line).

Predicate. *Predicate* is the technical term for a relation. In Turbo Prolog, predicates are declared. This declaration can be viewed as a blueprint for a relation. Once

KEYWORD	PURPOSE
domains	Declares user-defined domains (types).
global domains	When using domains in more than one module.
database	Declares database predicates.
predicates	Declares predicates used in clauses section.
global predicates	When using predicates in more than one module.
goal	Goal (or conclusion) to be proved (required).
clauses	Defines facts and rules used in the program.

Table 1. Program Structure in Turbo Prolog.

declared, the programmer can write facts and rules that define (or describe) the predicate.

Facts, Rules, and Clauses. *Facts* and *rules* are types of *clauses* (we'll get to clauses in a second). A rule consists of two parts. The first part, called the *head*, consists of a relation and is known as the conclusion. The second part, called the *body* of the rule, consists of one or a series of relations and statements called *subgoals*. The head and body are separated by the ":-" operator or the equivalent keyword **if**. Subgoals are separated by the "," or ";" operators, or the equivalent keywords **and** or **or**, respectively.

So, the basic form of a rule is

```
conclusion( is_true ) if
     subgoal1( is_true ) and
     subgoal2( is_true ) and
          .
          .
          .
     subgoalN( is_true ).
```

If the subgoals can be proved to be true, then the conclusion is also true. Rules, by the way, cannot be dynamically changed in Turbo Prolog the way facts can.

A Prolog fact consists of a relation followed immediately by a period. A fact is actually a special case of a rule. Thus, it can be thought of as a clause with a body but no head and is sometimes referred to as a headless clause. In a Prolog program, facts are true statements. Facts are also capable of being dynamically changed using the **assert** and **retract** relations.

With this basic collection of terms, we are prepared to talk about more advanced Turbo Prolog concepts, the most basic of which is program structure.

UNDERSTANDING PROGRAM STRUCTURE

In order to be compiled correctly, programs written in Turbo Prolog must conform to a relatively rigid structure. The structure is identified by the use of the keywords shown in Table 1.

Not all Turbo Prolog programs require all the sections shown in Table 1 to be included. The extreme case is to key "R" (for RUN) at Turbo Prolog's Main Menu with nothing in the editor. Despite not having declared any predicates, you can still use Turbo Prolog's built-in predicates. Since no goal has been declared, a Dialog window appears with a **Goal:** prompt. You could now carry on an interactive Prolog session indefinitely, but it wouldn't be too interesting.

The point of programming in Prolog consists of developing your own set of facts and rules, devising a goal, and setting the program running. As a very minimum, this requires you to get familiar with the **domains**, **predicates**, **goal**, and **clauses** program sections. As the table indicates, there are more sections than those mentioned here. But for our purposes, we will only concern ourselves with these four basic sections.

The **domains** section contains domain declarations. A *domain* is an object classification. There are five basic standard domains: **integer**, **real**, **char**, **string**, and **symbol**. (A sixth domain, **file**, deals with file input/output operations and is not considered here).

The first four basic domains are self-explanatory: they denote whole numbers, decimal numbers, individual characters, and strings of characters, respectively. The symbol domain is somewhat of a catch-all for a generalized object. As an example of a domain declaration, we can say

```
domains
   number = integer
   float  = real
```

which creates two new domains. The first declaration creates a **number** domain in terms of the built-in **integer** domain. It is worth noting that a **number** is now a separate domain from an **integer** and can only be used in that context. Similarly, a new domain, **float**, has been declared as a special kind of **real**, which is not to be confused with any other type of **real**.

These five domains are generally sufficient for simple Turbo Prolog programs, unless you plan to use lists. You may declare a list domain simply by defining it in terms of one of the standard domains, such as

```
charlist = char*
```

which says that the domain **charlist** (defined by you) describes a list of characters. Domain declarations can get pretty sophisticated and can be used to perform strict type checking. For the time being, limiting yourself to the standard domains plus user-defined list domains will get the job done.

The **predicates** section of a Turbo Prolog program is simply a list of the relations that will be used in the **clauses** section of the program. Each declaration consists of the name of a relation followed by the domain names of the objects in that relation. For instance, the declaration

```
predicates
   has(symbol,symbol)
```

declares the predicate **has** with two arguments, both in the symbol domain.

The **clauses** section of a Turbo Prolog program is where you describe the problem at hand. This section consists of all the facts and rules that define the relations declared in the **predicates** section. In our example, we can have a collection of facts that describe various animals as well as a char-

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*

JANUARY/FEBRUARY 1988
ISSUE CLOSING DATE: NOVEMBER 6

Turbo C is strong in floating point, and understanding the machinery behind floating point can help in using that machinery effectively . . . Using advanced features of Intel's microprocessors requires knowing what processor is installed in a user machine, and we show you how to find out . . . A complete expert system shell in Turbo Prolog is laid out in lights . . . And that's just a snapshot!

MARCH/APRIL 1988
ISSUE CLOSING DATE: DECEMBER 23

Artificial intelligence will require good natural language parsing, for which Turbo Prolog is a "natural" . . . Turbo Pascal proves its low-level smarts by spooling sequences of tone descriptors in memory and playing them in the background through the PC's speaker . . . Making the Turbo C/Reflex database connection is as easy as adding a short function library to your Turbo C Applications . . . 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

Assistant to the Publisher
Annette Fullerton

Western Office
(714) 586-1517
Janet Zamucen

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

Southern Office
(813) 394-4963
Megan Patti

acteristic that each has:

```
has(fish,scales).
has(bird,wings).
has(tiger,stripes).
```

The **goal** is what makes a Prolog program "go." In Prolog, the user specifies a goal, and the system tries to satisfy that goal (or prove it to be true). In Turbo Prolog, goals can either be internal (i.e. part of the program), or external, where the user is prompted for a goal through the Dialog window during a Turbo Prolog session. Goals are expressed in the form of a rule body: as either a single statement or as the conjunction of several subgoals. In our example, we could state the following external goal:

```
goal:   has(bird,wings).
```

Since there is a matching fact in the **clauses** section, the system will return a **True** response.

If you've ever programmed in Pascal, you might notice that Turbo Prolog programs superficially resemble Pascal programs in their structure. Turbo Prolog's **domains** and **predicates** declarations, with the requirement that the objects of relations have a specific type, are roughly analogous to the **LABEL**, **CONSTANT**, **TYPE** and **VAR** sections of Pascal programs. Similarly, Turbo Prolog clauses are analogous to the procedure and function declaration part of a Pascal program, while the Turbo Prolog goal is analogous to the main body of a Pascal program.

UNDERSTANDING VARIABLES

The most common method of setting up a goal is to represent one or more objects in a relation as a variable. Variables in Prolog have names just like symbols, except they begin with a capital letter. Having been given a goal with one or more variables in it, Prolog begins a process of scanning both the built-in and user-defined predicates for a matching relation. For example, if we had the goal

```
has(bird,What). /* Notice that
                   'What' is a
                   variable! */
```

Prolog would come up with the fact

```
has( bird, wings ).
```

and would *match* (or *unify*) the objects in the goal with those in the fact. In this simple example, **What** would be *instantiated* (or *bound*) to the value of **wings**, and the goal would *succeed.*

If, in our example, the goal had been

```
has(cow,What).
```

Prolog would not be able to come up with a match, because there is no relation about a cow. The way Prolog determines this is by examining each of the **has** relations sequentially. Prolog would first attempt to match the goal and the relation

```
has(fish,scales).
```

Since **cow** and **fish** do not match, the first relation would *fail.* Prolog would then backtrack to the next clause looking for an alternative solution. Once all the clauses had been scanned, the goal would also *fail.* (We'll get to failure in a moment).

An *uninstantiated,* or *unbound* variable is truly an unknown object in Prolog; it has no default value. Once a Turbo Prolog variable is bound through the process of unification, its value is set in concrete unless Turbo Prolog backtracks, in which case the variable reverts to the unbound state. A bound variable's scope extends only as far as the clause in which it appears. This is in sharp contrast to the time when BASIC was the only game in town, and all program variables were global in the sense that all variables were accessible from any place within the program. The advent of structured languages like Pascal and C gave rise to the concept of local variables—those accessible only from within a limited block of code—while preserving the ability to use global variables. In Turbo Prolog, however, there is no built-in facility for using global variables.

UNDERSTANDING FAILURE

Mention "failure" to a programmer steeped in traditional languages and the reaction is icy. If a traditional program fails, something is drastically wrong, either with the compiler, the program, or the computer. Failure means an error, and is to be avoided at all costs.

In Prolog, however, *failure* is a perfectly legitimate result and occurs when a search of a set of clauses comes up empty. When this happens, Prolog *backtracks* to the last place where it can find an alternative solution and starts moving forward again.

Backtracking is simply a way of searching for a solution, much the same way you might go about finding a path through a maze. At any fork, you might systematically choose to go down the left path. If at any point, you run across a dead end (this corresponding to a Prolog failure), you retrace your steps back to the last fork and choose to go down the next untraveled path to the left. If no such paths exist, you simply retrace your steps back to an earlier fork. If a solution exists, this technique will guide you through the maze, despite the number of dead ends (failures) you previously encountered.

PARTING WORDS

Learning to think and to program in Turbo Prolog may well be the most challenging computing task you choose to tackle. If you're like me, you might even feel discouraged at an apparent lack of progress. However, I can look back and recall the difficulties I had with C, and Pascal, and even with BASIC, and remember (with a chuckle) how those were overcome.

Prolog is—slowly—coming into its own. In the future, it will likely be the language of choice for solving a number of important tasks both in the field of Artificial Intelligence and more traditional areas. With a working knowledge of Turbo Prolog, you can be a part of that future. ■

Alex Lane is a software engineer living in Jacksonville, Florida. He is moderator of the Prolog conference on the Byte Information Exchange (BIX). Correspondence should be directed to him at RS&H, P.O. Box 4850, Jacksonville, FL 32201.

BIT BY BIT

Turbo Prolog's bitwise operators allow fine-tuning of your screen control.

Tom Castle and F. Barclay Shilliday

PROGRAMMER

If you have never worked with bitwise operations before, don't shy away. They're not hard. A bit is simply a physical switch representing a basic unit of memory in your computer. The switch can be on or off; nothing else. When a bit is on, we call it *set* or **TRUE**, giving the bit location a value of one. When a bit is off, we call it *clear* or **FALSE**, giving the bit location a value of zero. This is why the binary numbering system, which uses only zeros and ones to denote values, is so closely tied to the functions of a computer.

Working with bits alone is cumbersome and confusing, so most of the time we work with groups of bits. The most familiar group, a *byte*, is eight bits. There are also groups of four bits called *nibbles* and groups of 16 bits called *words*. Depending on the type of microprocessor you're using, the variable types you're used to seeing might represent a different number of bytes. Character variables are almost universally eight-bit values, but the move toward more powerful microprocessors may bring a 16-bit character set in the future. Integers are commonly represented as 16-bit values, but a move has already started to use 32-bit integers on some machines.

Figure 1 shows the values associated with each bit location of an integer when the bit is set. The total value of the integer can be obtained by summing the values of the set bits. The highest order bit, 15, is the sign bit. If set, it produces a negative number. If clear, the value is positive.

BIT MANIPULATIONS

Turbo Prolog includes the major bitwise operators you will need for most bit manipulations. But, the *Turbo Prolog Owner's Handbook* assumes that you understand the basics behind these functions. The bitwise operators **bitand, bitor, bitxor**, and **bitnot** use a system called *logical combinatorials* or *Boolean logic*. The operators **AND, OR, XOR**, and **NOT** are diagrammed in the truth tables of Figure 2. To read the table for a given operator, pick the value on the top and read down to the appropriate value on the left. The value inside the box is the logical result. For example, a logical **OR** between **TRUE** (corresponding to a value of 1) and **FALSE** (a value of 0) results in a value of **TRUE**. The operator **NOT** uses only one argument; **TRUE** becomes **FALSE** and **FALSE** becomes **TRUE**.

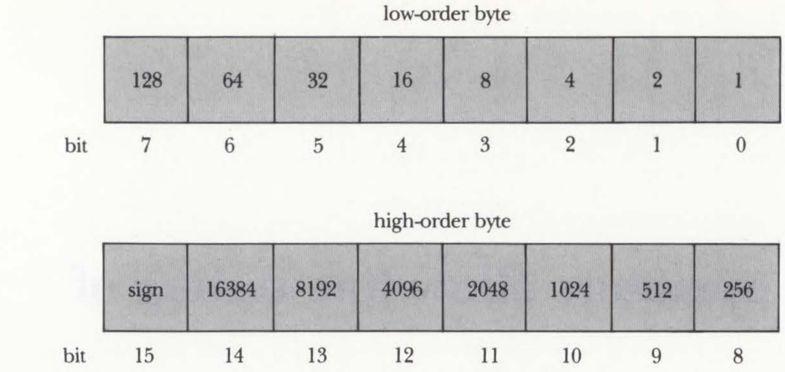
The predicates **bitand, bitor, bitxor**, and **bitnot** use the logic diagrammed in Figure 2 on each bit contained in the variables that are passed to them. So if you perform a logical operation between two integers, each of the 16 bits is subjected to the operation in the appropriate bit location.

Figure 3 schematically represents the workings of the logical operators **bitright** and **bitleft**. These functions move the contents of a variable's bits over a specified number of places to form a new value. A logical shift right by one place has the same effect as dividing by two and discarding any remainder. A logical shift left by one place has the same effect as multiplying by two.

Turbo Prolog does not support the use of unsigned integers. Such integers do not allow negative numbers and use bit 15 to hold the value 32768. This creates a range of integers from zero to 65535 instead of -32768 to 32767. Because the sign bit is always used, care must be taken not to shift inadvertently left into the sign bit. Bytes and words are much like the pre-Columbian view of a flat Earth. If you push something past the edge, it will fall off into oblivion and be gone forever.

USES OF BIT TWIDDLING

Bit manipulations are extremely important for some tasks. For instance, flags are values that provide status information about a given condition within a pro-



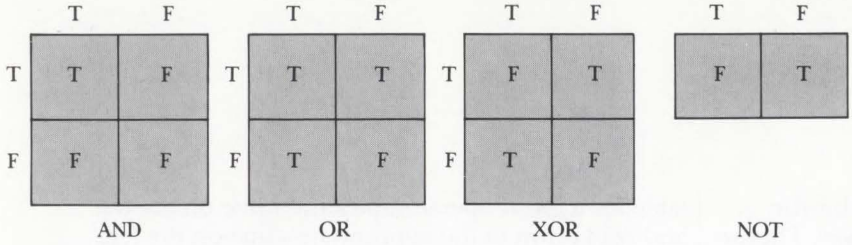
low-order byte

	128	64	32	16	8	4	2	1
bit	7	6	5	4	3	2	1	0

high-order byte

	sign	16384	8192	4096	2048	1024	512	256
bit	15	14	13	12	11	10	9	8

Figure 1. Values associated with each bit of an integer.



AND

	T	F
T	T	F
F	F	F

OR

	T	F
T	T	T
F	T	F

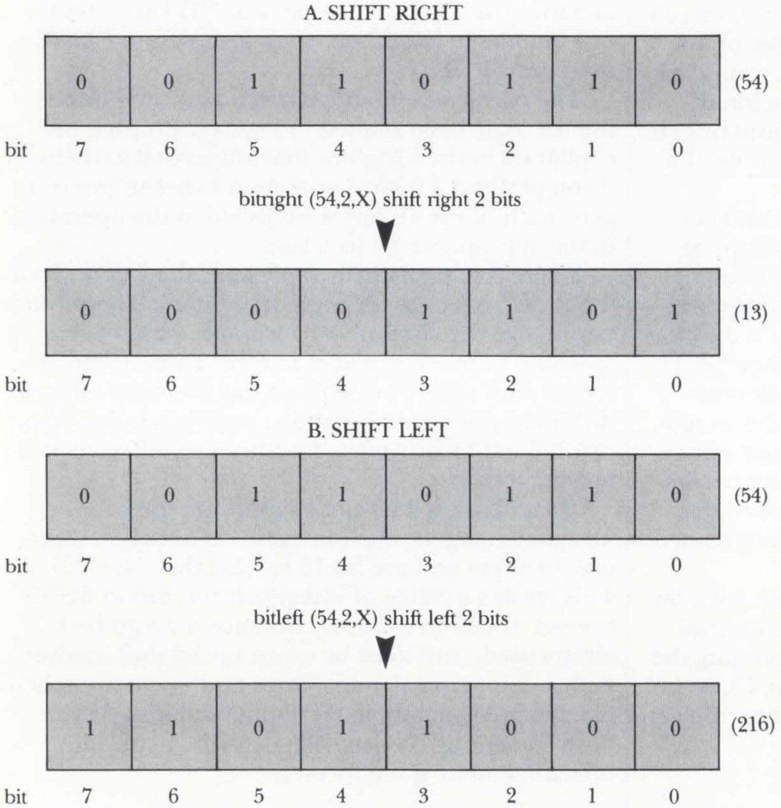
XOR

	T	F
T	F	T
F	T	F

NOT

T	F
F	T

Figure 2. Truth tables of Boolean operators and results.



A. SHIFT RIGHT

	0	0	1	1	0	1	1	0	(54)
bit	7	6	5	4	3	2	1	0	

bitright (54,2,X) shift right 2 bits

	0	0	0	0	1	1	0	1	(13)
bit	7	6	5	4	3	2	1	0	

B. SHIFT LEFT

	0	0	1	1	0	1	1	0	(54)
bit	7	6	5	4	3	2	1	0	

bitleft (54,2,X) shift left 2 bits

	1	1	0	1	1	0	0	0	(216)
bit	7	6	5	4	3	2	1	0	

Figure 3. Graphic representation of logical shifts.

gram. Each of the 16 bits of an integer can act as a flag for different information rather than wasting an entire integer on each flag.

To use each bit as a flag, we must have a convenient way of setting and clearing bits as required. Figure 4 demonstrates how to clear and set specific bits in a byte. For illustrative purposes, we randomly chose the starting values and the bits to manipulate. To clear bits, construct a mask that is clear only in those bits. A logical **AND** between the specified value and the mask produces a result that is clear in the proper bits but conserves all other information.

For each bit you want to set, set the mask in those bits and clear the rest. A logical **OR** ensures a set of the proper bits while conserving the other information.

Besides flags, you might be wondering what other uses can be made of bit manipulations. You can use bit manipulation in a text encryption scheme. In standard ASCII, setting bit 7 enables the extended characters. You might set bit 7 on every character you write to a disk file, making the file illegible to anyone but yourself. Alternatively, you could perform a logical shift left one place on the characters being written to disk for the same effect.

Bit manipulations can also change attributes on the IBM text screen. To see how, let's first examine the architecture of the screen.

THE IBM TEXT SCREEN

The memory address in which the physical text screen is mapped varies depending on the video adapter board used to control the monitor. The segment paragraphs of the base page address for the EGA, CGA, and monochrome adapters are A800H, B800H, and B000H, respectively.

The screens start at an offset of zero in the upper left corner of the screen. The first byte contains the ASCII code of the character displayed. The next byte contains the text attribute that holds the information about the blink, background color, foreground color, and intensity. Each 16-bit word consecutively contains the next pair of character and attribute bytes. The layout of the attribute byte is given in Figure 5.

REVERSE ATTRIBUTES

By knowing how the attribute byte is laid out, it's a simple matter of bit twiddling to produce an attribute where the foreground and background colors are exactly reversed except for the intensity of the foreground color. The job becomes slightly more complicated if we want to preserve the blinking or foreground intensity quality.

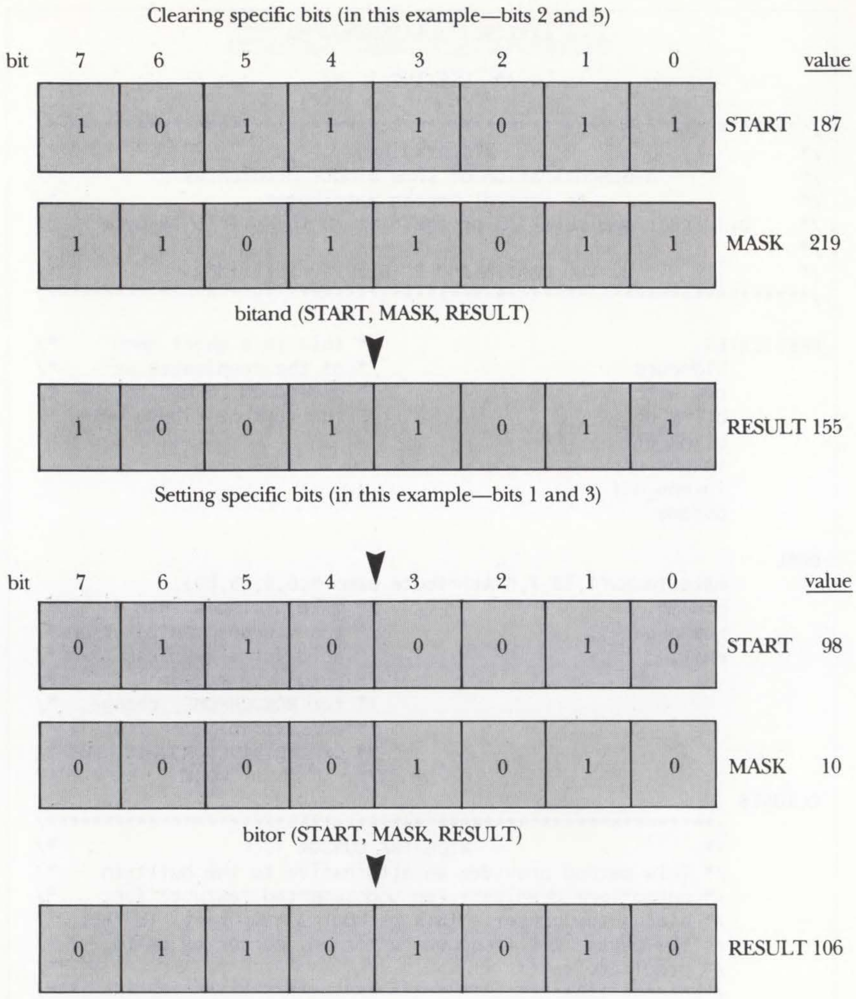
The most straightforward approach to reversing an attribute is shown as the **rev_attr** predicate in Listing 1. The first task is to safeguard the blink and intensity bits 7 and 3, respectively. We do this by query of the current attribute. A logical **AND** operation on that value with 136 (bits 3 and 7 set) results in a mask with bits 3 and 7 preserved and the rest of the byte cleared (equal to 0).

The second task is to perform some shift operations. We are going to make two new bytes by shifting four bits right or left, to make our new foreground and our new background, respectively. We then have to perform a logical **AND** between each new byte and 7 for the foreground (112 for the background). This is necessary to clear extraneous information from the old blink or intensity bit, so it doesn't corrupt our new values.

Our final task is to combine the new bytes together along with adding back the blink/intensity mask. This is done by two logical **OR** operations on the bytes involved. Setting this new attribute, which we label **REVATTR** in Listing 1, to the current attribute finishes the process of producing a reverse video effect on your next **write** or **field_str** call.

BLINK AND INTENSITY

By now you probably realize that turning the blink or the high-intensity foreground qualities on or off is a fairly simple matter.



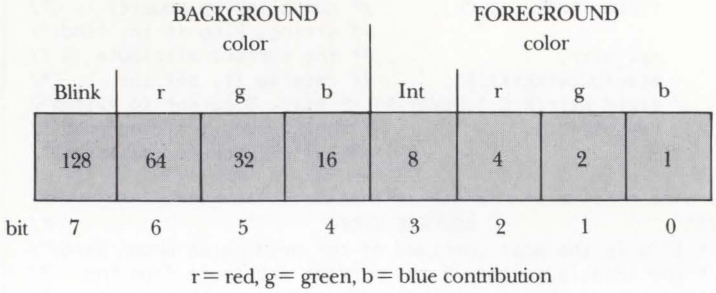
Clearing specific bits (in this example—bits 2 and 5)

bit	7	6	5	4	3	2	1	0		value
	1	0	1	1	1	0	1	1	START	187
	1	1	0	1	1	0	1	1	MASK	219
	1	0	0	1	1	0	1	1	RESULT	155

Setting specific bits (in this example—bits 1 and 3)

bit	7	6	5	4	3	2	1	0		value
	0	1	1	0	0	0	1	0	START	98
	0	0	0	0	1	0	1	0	MASK	10
	0	1	1	0	1	0	1	0	RESULT	106

Figure 4. Graphic representation of some useful bit manipulations.



BACKGROUND color				FOREGROUND color			
Blink	r	g	b	Int	r	g	b
128	64	32	16	8	4	2	1
bit 7	6	5	4	3	2	1	0

r = red, g = green, b = blue contribution

Figure 5. Layout of the attribute byte.

LISTING 1: ATTRDEMO.PRO

```
                              /* LISTING 1. */

/********************************************************************/
/*                           ATTRDEMO.PRO                           */
/*            a demonstration of some simple predicates             */
/*                  to control screen attributes                    */
/*   by direct manipulation of the bits in the attribute byte       */
/*                                                                  */
/*              by Tom Castle and F. Barclay Shilliday              */
/********************************************************************/

PREDICATES                              /* This is a short demo      */
        hidecurs                        /* of the predicates we      */
        rev_attr                        /* developed to manipulate   */
        blink_on                        /* the text attribute byte   */
        blink_off
        intens_on
        intens_off
        dodemo

GOAL
        makewindow(1,33,7," Attribute Demo ",0,0,25,80),
        dodemo,                         /* make a window then do the*/
        readchar(_),                    /* the various manipulations*/
        exit.                           /* Hold for a keypress then */
                                        /* return to editor         */
                                        /* for MONOCHROME, change   */
                                        /* scr attrib 33 -> 7 or 112*/
                                        /* OR THE HIDECURS WILL NOT */
                                        /*          WORK            */
CLAUSES
        /***********************************************************/
        /*                  HIDE THE CURSOR                        */
        /* This method provides an alternative to the built-in     */
        /* cursorform predicate (an undocumented feature) for      */
        /* hiding the cursor.  This method is temporary. In fact,  */
        /* the cursor will reappear after any cursor or write      */
        /* predicate calls.                                        */
        /***********************************************************/

        hidecurs:-
             cursor(R,C),               /* find where the cursor is */
             str_char(Ch,'\219'),       /* convert ASCII 219 (solid */
             field_str(R,C,1,Ch),       /* non-blinking square) to a*/
                                        /* string. Plop it in, find */
             rev_attr,                  /* the current attribute  & */
             attribute(REVATTR),        /* reverse it, set the      */
             field_attr(R,C,1,REVATTR),/* attr. @ cursor to switch*/
             rev_attr.                  /* background to foreground */
                                        /* Switch back for next call*/

        /***********************************************************/
        /*                   REVERSE VIDEO                         */
        /* This is the most involved of our predicates shown here.*/
        /* The idea is to obtain the current attribute from the    */
        /* standard unbound attribute function.  We then start     */
        /* constructing new bytes from various manipulations,      */
        /* splicing them back together, and installing the new     */
        /* value as the new current attribute.                     */
        /***********************************************************/

        rev_attr:-
             attribute(ATTR),           /* find the current attribut*/
             bitand(ATTR,136,Mask),     /* build a mask with only   */
                                        /* the blink and intens bits*/
             bitleft(ATTR,4,NB),        /* make fore -> background  */
             bitand(NB,112,Newback),    /* and clear any debris.    */
             bitright(ATTR,4,NF),       /* make back -> foreground  */
             bitand(NF,7,Newfore),      /* and clear any debris     */
             bitor(Newback,Newfore,Newattr), /* OR to make newbyte*/
```

BIT BY BIT

Those qualities can be changed by finding the current attribute, setting or clearing the appropriate bits, and setting the new attribute with the **attribute** predicate.

The blink is controlled by bit 7. It can be turned on by performing a logical **OR** with 128 (all bits clear except bit 7). This forces bit 7 to equal 1 while letting the others remain the same. The blink is similarly turned off by a logical **AND** with 127 (all bits set except bit 7) which forces bit 7 to equal zero.

The high-intensity quality of the foreground is controlled by bit 3. Intensity is controlled in much the same way as blink. To turn the high intensity on, perform a logical **OR** with 8 (bit 3 set). Likewise, turn high intensity off by using a logical **AND** with 247 (all bits set except bit 3).

You can redefine the blink and intensity predicates to return the new attribute value. This means that you must change the predicates declaration to something like:

```
predicates
   blink_on(integer)
```

Then you could pass the new attribute obtained from **blink_on** to a **field_attr** call.

HIDE THE CURSOR

There are many times when we want to program all input/output through field strings to retain more control over user interaction. One of the annoyances that may crop up when using field strings exclusively is the presence of that obtrusive, flashing cursor.

There is an undocumented way of turning off the cursor using the **cursorform** predicate: simply use values greater than 14. The trouble is that the change is permanent, even after you leave Turbo Prolog. If your program doesn't reset the cursor before exiting, the only way to regain the cursor is to reboot your system.

However, there is also a temporary method of hiding the cursor. We developed the **hidecurs** predicate in Listing 1 from a suggestion from p. 92 of Peter Norton's *Programmer's Guide to the IBM*

PC. The idea is to locate the cursor and plop in the ASCII value 219, a nonblinking rectangle that occupies the entire character box. To do this, first convert the ASCII value into a string so we can use the **field__str** predicate to poke it into the proper location. We tried several variations of this technique using the **membyte** predicate, but none worked as successfully as the method shown in Listing 1.

After string conversion, we then reverse the current attribute and install the new value with the **rev__attr** and **attribute** predicates. Using the **field__attr** predicate, we finish the deception. The final task is to again reverse the current attribute value for any future screen calls.

The cursor is not out of the way for good. It appears hidden only as long as you don't actually move it with the **cursor** or **write** predicates. You can use field strings and all will be well. Of course, if you need to move the cursor, you can always hide it again with another **hidecurs** call.

FINAL WORDS

If you are using a monochrome monitor, there are only a few useful attribute combinations. White on black is 7 (bits 0-2 set). Black on white is 112 (bits 4-6 set). Underline is 1. Invisible is 0. All other combinations are white on black. You can still twiddle the blink and intensity bits, though.

You now have a good foundation on which to experiment with bit manipulations. You'll be surprised at the interesting things that can be done. ■

Tom Castle is an MS chemist in Kalamazoo, Michigan. He writes software reviews for the Atari ST computer and studies Aikido in his spare time. F. Barclay Shilliday is also a chemist in Kalamazoo. He enjoys gardening, playing soccer, and fiddling, but not at the same time.

Listing may be downloaded from CompuServe as BITPRO.ARC.

```
          bitor(Newattr,Mask,REVATTR),/* add back the blink and*/
                                      /* intensity bits to form a */
          attribute(REVATTR).         /*reverse video. Set attrib */

/************************************************************/
/*                         BLINK                              */
/* The blink is determined by bit 7 of the attribute byte.*/
/* If the bit is set, blinking is activated.  If the bit   */
/* is clear, blinking is deactivated.  We will use the     */
/* method shown in Figure 4 to set and clear bit 7 to      */
/* control the blink.                                      */
/************************************************************/

blink_on:-
     attribute(ATTR),            /* find the current attrib  */
     bitor(ATTR,128,BLINK),      /* set bit 7 with AND 128   */
     attribute(BLINK).           /* plop in new attribute    */
blink_off:-
     attribute(ATTR),            /* find the current attrib  */
     bitand(ATTR,127,NOBLINK),/* clear bit 7 with OR 127  */
     attribute(NOBLINK).         /* plop in new attribute    */

/************************************************************/
/*                 FOREGROUND INTENSITY                     */
/* The foreground intensity quality is controlled by bit  */
/* 3.  Like the blink bit, a set bit will activate. Again,*/
/* the methods to selectively set and clear bit 3 are     */
/* generalized in Figure 4.                                */
/************************************************************/

intens_on:-
     attribute(ATTR),            /* find the current attrib  */
     bitor(ATTR,8,INTENS),       /* set bit 3 with OR 8     */
     attribute(INTENS).          /* plop in new attribute    */
intens_off:-
     attribute(ATTR),            /* find the current attrib  */
     bitand(ATTR,247,NOINTENS),/* clear bit 3 with AND 247*/
     attribute(NOINTENS).        /* plop in the new attribute*/

/************************************************************/
/*                         DEMO                               */
/************************************************************/

dodemo:-
     cursor(1,34),               /* we're just going to move */
     write("Normal text"),       /* the cursor near the top  */
     cursor(3,30),               /* center and start writing */
     rev_attr,                   /* text using the various   */
     write(" Now, reverse text "),  /* predicates.           */
     cursor(5,31),               /* all the predicates except*/
     rev_attr,                   /* the hidecurs can be used */
     write("now, normal again"),/* with write statements. */
     cursor(7,35),               /* Hidecurs must only be    */
     blink_on,                   /* with field_str since any */
     write("blink on"),          /* cursor movement with a   */
     cursor(9,35),               /* write call will show the */
     blink_off,                  /* cursor again.  So you    */
     write("blink off"),         /* would use hidecurs after */
     cursor(11,31),              /* any write call to keep it*/
     intens_on,                  /* hidden.  Intensity bits  */
     write("high intensity on"),  /* are not recognised by*/
     cursor(13,31),              /* all color displays. You  */
     intens_off,                 /* just need to experiment. */
     write("high intensity off"),
     hidecurs,
     field_str(15,35,10,"hidecursor"),
     readchar(_),                /* wait for a key press so  */
     cursor(17,34),              /* you can see that the     */
     write("show cursor"),       /* cursor is hidden and will*/
     readchar(_),                /* show again.              */
     cursor(19,35),
     write("all done").
```

STARTING OUT WITH THE TURBO BASIC DATABASE TOOLBOX

If that database manager you bought gets unmanageable, roll your own!

Peter Aitken

PROGRAMMER

As its name implies, the Turbo Basic Database Toolbox is intended primarily to simplify the development and support of database applications programs using Turbo Basic. Before describing the contents of the Toolbox and presenting a sample application program, let's review some database terminology.

A *database* is a collection of information that is stored in a fixed format. Perhaps the most common example is an address book, where the information for each person follows the same format: name, street address, city, etc. Each separate entry is called a *record*, while the individual pieces of information that make up a record are called *fields*.

In addition to storing information, a database program must provide reasonably fast access to that information. The simplest way to find a particular record—for example, John Smith's address—is to sequentially search the database, record by record, until a match is found. However, searching through a large database can be a time-consuming process even for today's speedy computers, so a faster method is needed. The Turbo Basic Database Toolbox speeds database searching and other maintenance tasks by means of keys contained in index files.

When you create a data file with the Database Toolbox, you specify one field that will be used as the key. For our address list example, the most logical key field might be last name. The Toolbox routines will then automatically create and maintain a sorted index file that consists of all key field entries—in our example, last names—with pointers to the locations in the data file where the associated records are located. Thus, when you want to look up John Smith's address, the program can quickly look up "Smith" in the index file and go directly to the proper record without having to search the data file at all.

```
'passes records to Turbo Sort:
SUB Sortload STATIC
    OPEN "DATAFILE.DAT" FOR INPUT AS #1
    WHILE NOT EOF(1)
        LINE INPUT #1, A$
        CALL SortRelease(A$)
    WEND
    CLOSE #1
END SUB

'record comparison function:
SUB LessThan(A$, B$, Result)
    Result = (A$ < B$)
END SUB

'prints sorted records:
SUB SortOut STATIC
    CALL SortReturn(A$, Done)
    WHILE NOT DONE
        LPRINT A$
        CALL SortReturn(A$, Done)
    WEND
END SUB
```

Figure 1. Custom routines for Turbo Sort.

```
CALL SaveScreenArea(UpperRow, LeftColumn,_
    NumberOfRows, NumberOfColumns,_
    SavedText$, SavedAttributes$)

CALL WriteScreenArea(UpperRow, LeftColumn,_
    NumberOfRows, NumberOfColumns,_
    HelpText$, HelpAttributes$)

delay 5    'wait a bit before restoring original screen

CALL WriteScreenArea(UpperRow, LeftColumn,_
    NumberOfRows, NumberOfColumns,_
    SavedText$, SavedAttributes$)
```

Figure 2. Saving and restoring text under a help screen.

Bradley Ream

LISTING 1: ARTICLES.BAS

```
'**********************************************************************
'Source file ARTICLES.BAS
'Demonstration program for some TURBO BASIC DATABASE TOOLBOX calls
'Maintains a flat-file database of magazine articles, indexed on
'article number and article topic.
'**********************************************************************

ON ERROR GOTO ERRORHANDLER

DEFINT A - Z                'all numbers to default to integers

CALL dbInit                 'initialization for Turbo Basic Access
CALL ScrnInit               'initialization for screen routines
CALL InitEntry              'initialization for keyboard entry routines

$INCLUDE "ARTICLES.INC"     'the file ARTICLES.INC was generated with
                            'the INCGEN utility program.  It creates or
                            'opens the data set and defines the fields
                            'for the variables.  See listing #2.

'As set up by the include file, the database is keyed on the
'articles.number field, which does not allow duplicate keys.
'We will now set up an auxiliary index file that will key on
'the articles.topic field and will allow duplicate keys.

AUX.KEY.LENGTH = 18         'length of articles.topic field
INCR LastFileNum
ARTICLES.AUX = LastFileNum
DUPS.ALLOWED = %YES

'Try to create the index, which succeeds only if it doesn't exist

CALL MakeIndex(ARTICLES.AUX, "ARTICLES.DB1", AUX.KEY.LENGTH,_
              "ARTICLES.DBD", ARTICLES.RLEN, DUPS.ALLOWED)

'If the index already exists, we can open it

IF dbStatus = %FileAlreadyCreated then_
    CALL OpenIndex(ARTICLES.AUX, "ARTICLES.DB1", AUX.KEY.LENGTH,_
              "ARTICLES.DBD", ARTICLES.RLEN, DUPS.ALLOWED)

'Data and index files are open - we can display main menu screen

DO UNTIL DONE

    CLS
    LOCATE 1,20  : PRINT "MAGAZINE ARTICLE DATABASE MAIN MENU"
    LOCATE 5,25  : PRINT "F1: MAKE NEW ENTRIES"
    LOCATE 7,25  : PRINT "F2: BROWSE BY NUMBER"
    LOCATE 9,25  : PRINT "F3: BROWSE BY TOPIC"
    LOCATE 11,25 : PRINT "F4: EXIT"

'Get keystroke

    DO
        CALL GetKeystroke(Ch)
    LOOP UNTIL (Ch > 313 and Ch < 318)          'accept F1-F4 only

    Ch = Ch - 313                   'now Ch = 1 for F1, 2 for F2 etc.

    ON CH GOSUB NEWENTRIES, BROWSENUMBERS, BROWSETOPICS, FINISH

LOOP        'end of main menu loop

'Program comes here when DONE = %YES to end program and exit

CALL dbClose(ARTICLES.FILE)
CALL CloseIndex(ARTICLES.AUX)
CLS : LOCATE 1,1 : END
```

THE B+ TREE STRUCTURE

The Database Toolbox maintains index files using a B+ Tree structure. The B+ Tree is a type of binary tree, a data structure that permits a program to rapidly and efficiently locate a data item, even in very large files. You don't need to know anything about B+ Trees to use the Toolbox, but the manual contains a good explanation of them for those who are interested.

A data file can be indexed on more than one field. For example, our address list could be indexed on zip code as well as last name. A data file with its associated index file(s) is called a *data set.*

The Turbo Basic Database Toolbox consists of several components. The major component is the Turbo Basic Access System, which contains the routines for working with data sets.

The Turbo Basic access calls are divided into two categories: high-level calls and low-level calls. The high-level calls are easier to use and more powerful than the low-level calls, but they are not as versatile.

The main functional difference between the two types of calls is that the high-level calls handle both the data file and the primary index file of a data set, while the low-level calls deal with only one or the other. The high-level calls actually use the low-level calls, combining them to provide the most frequently needed database functions in single calls. Many less complicated database applications can be programmed using only the high-level calls. The low-level calls are needed for more sophisticated programming situations. For example, you must use low-level calls to:

- Create more than one index file for a particular data file.
- Use extension records (associate more than one record with a key).
- Index a key that may have

```
'**************************** SUBROUTINES *****************************

SUB EntUserHook(Ch$)         'subroutine called by GetKeyStroke()
   Ch$ = Inkey$
END SUB

'-----------------------------------------------


FINISH:
   DONE = %YES
RETURN

'-----------------------------------------------


NEWENTRIES:

ExitKeys$  = chr$(13)             'ENTER key terminates entry
PromptAttr = 7                    'screen colors for prompt
DataAttr   = 112                  'and for data
EntryMode  = 3
Temp$      = ""
Nums$      = "0123456789"         'template for accepting numerals only
All$       = ""                   'template for accepting any character

'The next section of code gets user entries and LSETs them into
'fields (defined in ARTICLES.INC) in preparation for writing to disk

CLS
CALL PromptEntry(4, 5, Nums$, 5, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, "Article number: ", Temp$, Changed, ExitKey)

IF Temp$="" then RETURN              'we are done if nothing entered

LSET articles.number$ = Temp$ : Temp$ = ""

CALL PromptEntry(18, 18, All$, 6, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, " Article topic: ", Temp$, Changed, ExitKey)

LSET articles.topic$ = Temp$ : Temp$ = ""

CALL PromptEntry(62, 62, All$, 7, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, " Article title: ", Temp$, Changed, ExitKey)

LSET articles.title$ = Temp$ : Temp$ = ""

CALL PromptEntry(20, 20, All$, 8, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, " First keyword: ", Temp$, Changed, ExitKey)

LSET articles.keyword1$ = Temp$ : Temp$ = ""

CALL PromptEntry(20, 20, All$, 9, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, "Second keyword: ", Temp$, Changed, ExitKey)

LSET articles.keyword2$ = Temp$ : Temp$ = ""

CALL PromptEntry(60, 60, All$, 10, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, "       Authors: ", Temp$, Changed, ExitKey)

LSET articles.authors$ = Temp$ : Temp$ = ""

CALL PromptEntry(40, 40, All$, 11, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, "       Journal: ", Temp$, Changed, ExitKey)

LSET articles.journal$ = Temp$ : Temp$ = ""

CALL PromptEntry(4, 5, Nums$, 12, 1, PromptAttr,DataAttr,ExitKeys$,_
        EntryMode, "          Year : ", Temp$, Changed, ExitKey)
```

duplicate entries, such as a zip code in a large address list.

- Use index files to index something other than a data file, such as an array.

RELATIONAL DATABASES

Some of you may have heard of relational databases, and are wondering: A) What are they?, and B) Can they be done with the Turbo Basic Database Toolbox?

To answer the first question, a relational database program is one that can make use of the relationships between two or more separate data files that have one or more fields in common. For example, a business may maintain one data file of sales information, including the name of the salesman for each sale, plus a separate personnel data file that includes educational and salary information for each salesman. Using a relational database program, you could relate information between the two files based on the common field (name). One example would be correlating sales performance with salary history, to see if your best-paid employees are the best performing. Relatively few database situations really need a relational capability, but it can be very powerful when required.

As to whether the Turbo Basic Database Toolbox has relational capabilities, the answer is yes and no. No, because the Toolbox does not contain routines specifically designed for relating two separate data files. Yes, because the Toolbox routines allow you to have more than one data file open at a time. Therefore, although the programming would not be trivial, you could write your own code to relate records in separate data files.

The Toolbox also includes Turbo Basic Sort, a rapid sorting algorithm that can sort data on one or more keys. It is not limited to sorting data files created with the Toolbox, but can sort essentially any data. The flexibility of Turbo Sort is due to the fact that the user supplies the input, comparison, and output routines while

```
LSET articles.year$ = Temp$ : Temp$ = ""

CALL PromptEntry(20, 20, All$, 13, 1, PromptAttr,DataAttr,ExitKeys$,_
       EntryMode, "      Citation: ", Temp$, Changed, ExitKey)

LSET articles.citation$ = Temp$ : Temp$ = ""

'now we are ready to write the new record to disk.  First write the
'record with the unique key articles.number using the high-level call
'dbWrite.  If the key articles.number already exists, the old data
'are overwritten.

Newkey$ = articles.number$
CALL dbWrite(articles.file, Newkey$)

'the variables dbStatus and DataRef& are set by the dbWrite routine

IF (dbStatus = %KeyAlreadyExists) THEN_
    LOCATE 20,19 :_
    PRINT "Article number exists - record ";DataRef&;" updated"_
ELSE_
    LOCATE 20,24 :_
    PRINT "New article number - record added at # ";DataRef&

DELAY 1              'wait a bit

'now to add the secondary key articles.topic to the auxiliary
'index file, which does permit duplicate keys.

IF articles.topic$ <> "" THEN
    Newkey$ = articles.topic$
    CALL AddKey(articles.aux, DataRef&, Newkey$)
END IF

RETURN

'------------------------------------------------

BROWSENUMBERS:

CLS : LOCATE 24,10
PRINT "F1: next record   F2: previous record   F3: main menu";
CALL ClearKey(articles.file)
CALL dbNext(articles.file, keyval$)

DO
    GOSUB display.record        'display current record

'get keystroke from user

    DO
        CALL GetKeystroke(Ch)
    LOOP UNTIL (Ch > 313 and Ch < 317)        'accept F1-F3 only

    IF Ch = 314 THEN_               'if F1 get next record
        CALL dbNext(articles.file,keyval$)

    IF Ch = 315 THEN_               'if F2 get previous record
        CALL dbPrev(articles.file,keyval$)

    IF Ch = 316 THEN EXIT LOOP

    IF dbStatus = %EndOfFile THEN BEEP

LOOP
RETURN

'------------------------------------------------

BROWSETOPICS:

CLS : LOCATE 24,10
```

Turbo Sort does the actual sorting. This may sound intimidating, but the needed routines are typically very short and easy to write. For example, the three routines in Figure 1 read a list of data from a disk file, sort it alphabetically, and print the results.

These routines can be as simple or complex as your application requires. By writing them yourself, you gain great flexibility. For example, records could be received over a modem, sorted on several fields, and then passed to a database program.

Turbo Sort can sort a maximum of 32,767 records, and maximizes speed by doing all sorting, when possible, in memory. If there is insufficient memory, Turbo Sort's virtual memory manager will make use of disk space, treating it as an extension of RAM.

KEYBOARD ENTRY TOOLS

The Turbo Basic Database Toolbox also includes a set of miscellaneous tools that handle keyboard entry of data and sophisticated screen operations. These tools can be very useful in programming database applications, and you will also find them to be great timesavers in almost any other type of programming.

The primary function of the keyboard entry routines is to display data fields on the screen, accept user input, and return the data to the calling program. The routine **PromptEntry**, for example, displays a prompt and an entry field on the screen and permits the user to enter and/or edit a string. The routine allows control over all aspects of the process, such as the color, location, and size of the entry field, what characters will be accepted, and the key(s) that will terminate entry.

Using these routines, your programs can have the sophisticated and convenient data entry screens that usually are found only in expensive commercial programs.

SCREEN CONTROL ROUTINES

The screen control routines provide some low-level functions, such as setting the cursor size and position and setting or reading the current video mode. The high-level screen functions can scroll a rectangular area of the screen up or down, and can read or write text and color attributes directly from or to video memory.

These latter routines allow the screen area being read from or written to to be specified either in terms of absolute memory addresses or in terms of screen rows and columns. These functions greatly simplify programming of pop-up menus and help screens. For example, the three calls shown in Figure 2 will save the contents of an area of the screen, display some help information, and then restore the original text.

Speed-critical screen functions are written in inline assembler code for maximum speed. With these routines, windows appear and disappear almost instantaneously.

The Turbo Basic Database Toolbox also includes several utility programs that facilitate the development and support of database applications. There are file translation programs that will convert ASCII, dBase II, dBase III, and Reflex files to Turbo Basic Access format, plus an export program that will convert a Turbo Basic Access data file to an ASCII file. These translation programs make it possible for you to convert an existing application from almost any dedicated database program to Turbo Basic without fear that you will have to rekey data.

There is also a program that can repair damaged Turbo Basic Access data files. Damaged files are usually the result of the program terminating improperly, without having closed the data and/or index files. In such cases, REBUILD.BAS can repair the damaged files so the data in them are not lost.

Finally, there is a very useful programming utility called INCGEN.BAS which will generate the Turbo Basic code to open and define the fields for a particular database. The Turbo Basic code that INCGEN generates is in the form of an include file that can be incorporated into your program

```
PRINT "F1: next record   F2: previous record   F3: main menu";
CALL ClearKey(articles.aux)
CALL NextKey(articles.aux,DataRef&,keyval$)
CALL GetRec(articles.file,DataRef&)

DO
    gosub display.record

'now get keystroke

    DO
       CALL GetKeystroke(Ch)
    LOOP UNTIL (Ch > 313 and Ch < 317)        'accept F1-F3 only

    IF Ch = 314 THEN_
        CALL NextKey(articles.aux,DataRef&,keyval$)

    IF Ch = 315 THEN_
        CALL PrevKey(articles.aux,DataRef&,keyval$)

    IF Ch = 316 THEN EXIT LOOP

    IF dbStatus = %EndOfFile THEN_
        BEEP_
    ELSE_
        CALL GetRec(articles.file,DataRef&)

LOOP
RETURN

'************************************
DISPLAY.RECORD:

LOCATE 5,5  : PRINT "   Number: ";articles.number$
LOCATE 6,5  : PRINT "    Topic: ";articles.topic$
LOCATE 7,5  : PRINT "    Title: ";articles.title$
LOCATE 8,5  : PRINT " Keyword1: ";articles.keyword1$
LOCATE 9,5  : PRINT " Keyword2: ";articles.keyword2$
LOCATE 10,5 : PRINT "  Authors: ";articles.authors$
LOCATE 11,5 : PRINT "  Journal: ";articles.journal$
LOCATE 12,5 : PRINT "     Year: ";articles.year$
LOCATE 13,5 : PRINT " Citation: ";articles.citation$

RETURN

'**************************************************

ERRORHANDLER:

CLS
PRINT "Error ";ERR
PRINT "Aborting program"
CALL dbClose(ARTICLES.FILE)
CALL CloseIndex(ARTICLES.AUX)
END

'**************************************************

'these are the INCLUDE statements for the toolboxes

$INCLUDE "DBHIGH.BOX"
$INCLUDE "DBLOW.BOX"
$INCLUDE "ENTSUBS.BOX"
$INCLUDE "SCRNSUBS.BOX"
$INCLUDE "SCRNASM.BOX"

'****************   END OF ARTICLES.BAS   **********
```

LISTING 2: ARTICLES.INC

```
'*************************************************************************
'       Articles Database Include file for TURBO-BASIC ISAM
'
' Dataset reference No:  articles.File
' Data file name:        articles.DBD
' Data File length:       268
' Key File name:         articles.DBI
' Key length:              4
'*************************************************************************

defint a-z

if articles.File > 0 then goto EndarticlesInc 'skip if already open
  incr LastFileNum
  articles.File = LastFileNum
  articles.RLen =  268
  articles.KLen =  4

  call dbCreate(articles.File, FileNo, _
    "articles.DBI", articles.KLen, "articles.DBD", articles.RLen)

  if dbStatus = %FileAlreadyCreated then _
    call dbOpen(articles.File, FileNo,
     "articles.DBI", articles.KLen, articles.RLen)

  if articles.RLen <>  268 then
    print "articles.DBD length is inconsistent with this program."
    end
  elseif articles.KLen <>  4 then
    print "articles.DBI key length is inconsistent with this program."
    end
  end if

  field FileNo,_
    4 as articles.Skip$,_
    4 as articles.NUMBER$,_
    18 as articles.TOPIC$,_
    62 as articles.TITLE$,_
    20 as articles.KEYWORD1$,_
    20 as articles.KEYWORD2$,_
    60 as articles.AUTHORS$,_
    40 as articles.JOURNAL$,_
    4 as articles.YEAR$,_
    20 as articles.CITATION$


  field FileNo,  268 as articles.Buffer$

  gosub articlesClear
goto EndarticlesInc:

articlesClear:
  lset articles.Buffer$ = string$(  268, " ")
return 'articlesClear

EndarticlesInc: 'End of this include file
```

with an **$INCLUDE** statement. This can be very useful when you are using a number of programs to maintain several database files. For a particular program to access a particular database, all you need to do is **$INCLUDE** the code generated by INCGEN for that database.

All of the Toolbox routines are supplied as Turbo Basic source code. This allows you to study the code to see how things are done, and to modify it if you wish. Of course, the Borland technical support people cannot help you with Toolbox code that you've modified.

Take a look at the sample database application program, ARTICLES.BAS, written using the Turbo Basic Database Toolbox. ARTICLES.BAS creates and maintains a database file of magazine article citations, allowing you to browse through the file entries sorted by either the number or the topic of the articles. The source code is given in Listing 1, while the include file generated with the INCGEN utility is given in Listing 2. The source code is thoroughly commented, which should allow you to figure out how the program works.

ARTICLES.BAS is pretty much "bare-bones," lacking the fancy screen displays, sophisticated search routines, and printer output that you might want in your own database programs. Rather than being a finished product, it is intended only to demonstrate some of the capabilities of the routines in the Database Toolbox. These routines enabled me to write a fast, functional database management program using as little as 100 lines of code. If you do much database programming, and you value your time, this Toolbox should be of interest to you. ■

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.

Listing may be downloaded from CompuServe as TBDBASE.ARC.

DOS CALLS FROM TURBO BASIC

Here's how to determine where you are in a DOS directory structure by calling DOS Interrupt 21.

Ethan Winer

WIZARD

One of the significant enhancements to BASIC introduced with Turbo Basic is the **CALL INTERRUPT** statement. **CALL INTERRUPT** performs a direct call to any 8086 software interrupt, with necessary register values passed by way of the **REG** statement. All of the DOS functions can be performed by using **CALL INTERRUPT** to call DOS interrupt 21H, although as you might imagine, some are more complicated to set up than others.

Turbo Basic makes it easy for programmers to change the current default directory, but unfortunately, it provides no easy way to determine which directory is currently active. From the DOS prompt you simply issue a CD (or CHDIR) command without arguments, and the command will return the current directory. Listing 1 shows a Turbo Basic subprogram that uses DOS function 47H to retrieve the current directory and place it into a BASIC string.

GetDir is called by specifying a drive letter and passing a string variable that is to receive the returned directory information. Since a number of new concepts are used in this subprogram, let's take a closer look at how it works.

First, a string long enough to hold the largest possible path name is created. Not only must we tell DOS where to put the information it will return, but we must also ensure that other data isn't overwritten. If enough space has not been properly set aside, a system crash is the likely consequence.

Next, the drive letter being passed is checked to see if it is a null string, which we'll use to mean the default drive. Many of the DOS functions can be instructed to use drive A, drive B, and so forth, or the current default drive. In this case the drive designator is placed in the DL register, with a 1 indicating drive A, a 2 for drive B, and a 0 for the default. Notice how the drive letter is first forced to uppercase. This makes it easier for the calling program, by allowing it to use either uppercase or lowercase.

DIRECTING DOS

When we ask DOS to get the current directory for a given drive, we must also indicate where that information is to go, which in this case is the variable **Dir$**. But before we can know where **Dir$** is, we must get the pointers that Turbo Basic maintains for its own use. This is done with a combination of the **VARPTR** and **VARSEG** instructions.

VARSEG returns the segment that holds the string's descriptor, and **VARPTR** returns the address within that segment. Therefore, we must first make **Dir$**'s descriptor segment the current one for the **PEEK**s that are to follow. (We'll explain in detail how this is done below.) Once the descriptor address has been found, we can then locate the actual string.

A string descriptor is simply a four-byte table that contains a string's length and its address. We've already seen how to find where it is, but not what to do with it once we've found it. The first two bytes together contain the length of the string, and the next two bytes hold its offset within the string data segment. The "+2" and "+3" get the address by skipping over the length bytes.

Keep in mind that the string data segment is different from the string descriptor segment. One of Turbo Basic's greatest advances over earlier BASICs is its improved use of memory. Rather than place all of the strings and their descriptors within a single segment, they are instead kept in separate segments to allow more space. This makes things much more difficult when we have to find where a string is really located in memory.

The string data segment can be found by examining the very first two bytes in Turbo Basic's default data segment. That's the segment selected for subsequent **PEEK** statements when you execute a **DEF SEG** by itself with no arguments. By **PEEK**ing at these two bytes, we can obtain the segment that contains the actual string data. The offset within that segment is the one we got from the four-byte descriptor.

CALLING DOS

Once we've obtained both the segment and offset for **Dir$**, we are finally ready to call DOS. In fact, the rest

DOS CALLS

is easy after what we just went through! The segment of the string variable into which the path name will be placed is assigned to the DS register, and the string's offset within that segment goes into SI. The number indicating which drive to use is put into DL, and the Get Directory service number (47H) goes into AH, as usual. Finally, **CALL INTERRUPT** does the work of putting the current directory path into **Dir$**.

The last step is to isolate just the returned path from the rest of **Dir$**, which brings up another important issue. When a file or path name is sent to DOS, the end must be marked with a null, ASCII character 0. (In Turbo Basic, **CHR$(0)**). DOS doesn't use string descriptors like Turbo Basic's, and marking the end of data with a null is the only way DOS can tell when it has reached the end.

Likewise, when a string is being returned to us, DOS also marks the end with a null. For this reason, **INSTR** is used to locate the null character that DOS put there, and **Dir$** is then reassigned to keep only the part that matters.

One thing to note: when DOS returns a path name, it omits the drive letter, the colon, and the leading backslash. Therefore, if the current directory happens to be the root, DOS will simply return a single zero byte.

CONCLUSION

The trick in using DOS calls through **CALL INTERRUPT** is not in making the call itself. Instead, the key is in setting up information to be passed to the interrupt handler, and then retrieving information the handler returns to us.

Particularly, getting Turbo Basic's string information to and from interrupts is less than obvious. But with that understanding under your belt, you should be able to access any of the many DOS services from within Turbo Basic. ■

Ethan Winer owns Crescent Software, and is the author of the QuickPak utilities for Turbo Basic and Microsoft QuickBASIC.

Listing may be downloaded from CompuServe as TBDOSCAL.ARC.

LISTING 1: GETDIR.BAS

```
'*** GetDir.Bas - returns the current directory for a specified drive

CLS
LINE INPUT "Which drive? ", D$
CALL GetDir(D$, Directory$)
PRINT "The current directory on drive "  D$  " is \"  Directory$
END


SUB GetDir(Drive$, Dir$) STATIC

    LOCAL Drive, Descriptor!, Address!

    Dir$ = SPACE$(64)                   'make room for directory name

    IF Drive$ = "" THEN                 'a drive wasn't specified, so
       Drive = 0                        '  use the default drive
    ELSE
       Drive = ASC(UCASE$(Drive$)) - 64 'adjust so "A"=1, "B"=2, etc.
    END IF

    DEF SEG = VARSEG(Dir$)              'find Dir$ descriptor address
    Descriptor! = VARPTR(Dir$)
    Address! = PEEK(Descriptor! + 2) + 256! * PEEK(Descriptor! + 3)

    DEF SEG                             'find string data segment at
    REG %DS, PEEK(0) + 256! * PEEK(1)   'address 0, and put it in DS
    REG %SI, Address!                   'offset within segment in SI
    REG %DX, Drive                      'specify drive in DL
    REG %AX, &H4700                     'specify service &H47 in AH

    CALL INTERRUPT &H21                 'call DOS

    IF Dir$ = SPACE$(64) THEN           'error - indicate this to the
       Dir$ = "Error"                   '  caller with error message
       EXIT SUB
    END IF

    'keep only the name portion; DOS marks end with a CHR$(0)
    Dir$ = LEFT$(Dir$, INSTR(Dir$, CHR$(0))-1)

END SUB

$INCLUDE "RegNames.Inc"                 'this file defines registers,
                                        '  and is on the Turbo disk
```

Borland's new Turbo C: The most powerful optimizing compiler ever

Our new Turbo C® generates fast, tight, production-quality code at compilation speeds of more than 13,000* lines a minute!

It's the full-featured optimizing compiler everyone has been waiting for.

Switching to Turbo C, or starting with Turbo C, you win both ways

If you're already programming in C, switching to Turbo C will make you feel like you're riding a rocket instead of pedaling a bike.

If you've never programmed in C, starting with Turbo C gives you an instant edge. It's easy to learn, easy to use, and the most efficient C compiler at any price.

> "Turbo C does look like What We've All Been Waiting For: a full-featured compiler that produces excellent code in an unbelievable hurry . . . moves into a class all its own among full-featured C compilers . . . Turbo C is indeed for the serious developer . . . One heck of a buy—at any price.
>
> *Michael Abrash, Programmer's Journal* "

Join more than 100,000 Turbo C enthusiasts. Get your copy of Turbo C today!

Technical Specifications

- ☑ *Compiler: One-pass optimizing compiler generating linkable object modules. Included is Borland's high-performance Turbo Linker.™ The object module is compatible with the PC-DOS linker. Supports tiny, small, compact, medium, large, and huge memory model libraries. Can mix models with near and far pointers. Includes floating point emulator (utilizes 8087/80287 if installed).*
- ☑ *Interactive Editor: The system includes a powerful, interactive full-screen text editor. If the compiler detects an error, the editor automatically positions the cursor appropriately in the source code.*
- ☑ *Development Environment: A powerful "Make" is included so that managing Turbo C program development is highly efficient. Also includes pull-down menus and windows.*
- ☑ *Links with relocatable object modules created using Borland's Turbo Prolog® into a single program.*
- ☑ *Inline assembly code.*
- ☑ *Loop optimizations.*
- ☑ *Register variables.*
- ☑ *ANSI C compatible.*
- ☑ *Start-up routine source code included.*
- ☑ *Both command line and integrated environment versions included.*
- ☑ *License to the source code for Run-time Library available.*

Sieve benchmark

	Turbo C	Microsoft® C
Compile time	*2.4*	13.51
Compile and link time	*4.1*	18.13
Execution time	*3.95*	5.93
Object code size	*239*	249
Execution size	*5748*	7136
Price	*$99.95*	$450.00

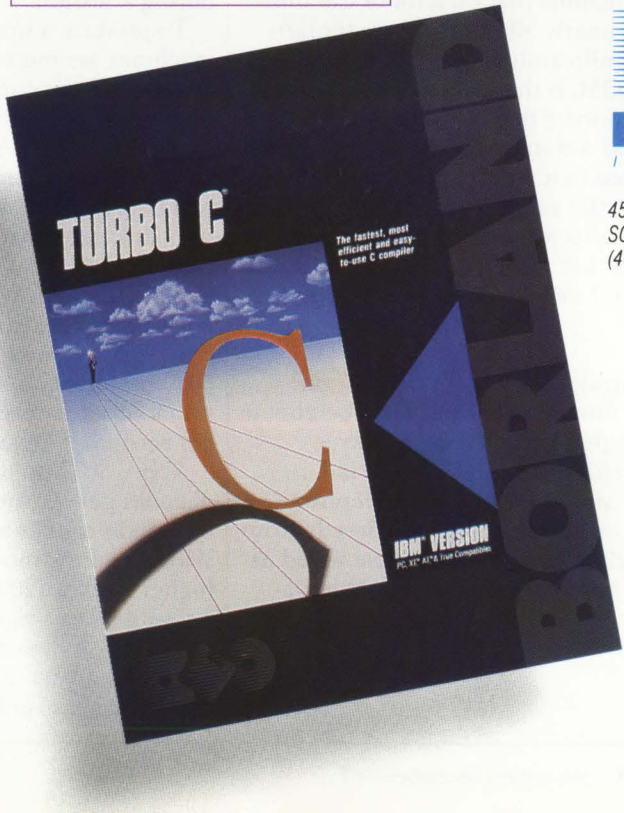
*Benchmark run on an IBM PS/2 Model 60 using Turbo C version 1.0 and the Turbo Linker version 1.0; Microsoft C version 4.0 and the MS overlay linker version 3.51.

Minimum system requirements: IBM PC, XT, AT, PS/2 and true compatibles. PC-DOS (MS-DOS) 2.0 or later. 384K.

For the dealer nearest you or to order by phone call

(800) 255-8008

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


TURBO C
The fastest, most efficient and easy-to-use C compiler
IBM VERSION
PC, XT, AT & True Compatibles
BORLAND

BORLAND
INTERNATIONAL

4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066
(408) 438-8400 TELEX: 172373

Only $99.95!

All Borland products are trademarks or registered trademarks of Borland International, Inc., or Borland/Analytica, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

BI-1136

TURBO BASIC COMMUNICATIONS

Hiding beneath a single Turbo Basic keyword is an interrupt-driven communications engine.

Reid Collins

PROGRAMMER

The field of data communications is a large one, and to most PC users, it is steeped in mystery, magic, and no small amount of frustration. If you have a serial port and a modem attached to your PC, perhaps you, too, have experienced some of the frustrations that are associated with making connections to the outside world.

This article looks at one small corner of data communications by taking a ground-level view of PC-based asynchronous communications through Turbo Basic. Its purpose is to lift the shroud of mystery and expose the essential elements of PC communications typified when a PC is used as a local terminal connected to a remote computer.

The programs presented here show how easy it is to write a useful communications program in Turbo Basic. This is because Turbo Basic incorporates a set of statements and functions that do a lot of the difficult work for you. In nearly all other computer languages, the task is significantly more complex. The first program, MINCOM, is the briefest communications program I could write that allows the PC to operate successfully as a terminal. An elaboration of MINCOM is presented in a second program, SIMCOM. It is essentially the same program as MINCOM, but SIMCOM adds a number of safety nets and convenience features that make data communications both more reliable and more pleasant for the user.

GROUND ZERO

Figure 1 shows the arrangement of hardware on which the programs function. The PC must be able to run DOS 2.0 or later and Turbo Basic. A Hayes-compatible modem, either internal or external, is connected to the PC and to an ordinary telephone line. Alternately, the connection may be directly from one machine to another using a short cable wired as a null-modem cable (also called a modem eliminator). The remote computer may be another PC or any other computer that is capable of answering a call.

An external modem requires a serial adapter in the PC and a connecting cable. If an internal modem is used, it combines the functions of a serial port and a modem into a single adapter.

The programs presented in this article are designed for full-duplex asynchronous communications with bulletin board systems and information utilities such as the Source, and any other system that supports this form of data communications. *Full duplex* means that unrelated data streams may be sent in both directions at the same time without interfering with each other. The term *asynchronous* means that the local and remote systems may transmit at any time, that there is no common clocking mechanism, and that neither system has advance knowledge of the other's plans to transmit. Each system must be ready to receive input at all times during a session.

To produce a workable connection between two machines, we must have an agreement between the two ends of the connection about the values of a set of communications parameters. Of primary importance is transmission rate. At rates of 300 bits per second (bps) and less, the rate may be expressed equivalently in baud or state transitions per second. This is the basic signaling rate of frequency-shift keying (FSK), that indicates the rate at which the signal switches between two audible frequencies. One baud is one state transition per second. (Note: the frequently used phrase "baud rate" is a redundancy. A baud specification is a rate, so 300 bps, for example, is the same as 300 baud. It makes no more sense to say "baud rate" than to say "Hertz per second" when speaking of frequency.)

Above 300 bps, the one-to-one correspondence of data bits per second to baud no longer holds because of the way signals are impressed on the carrier signal. With the phase-modulation techniques typically used at higher rates, a 1200 bps data rate is achieved on a signal with a basic signaling rate of 600 baud. Still higher transmission rates are accomplished by encoding greater numbers of bits per state transition. Always express the transmission rate in bits per second (rather

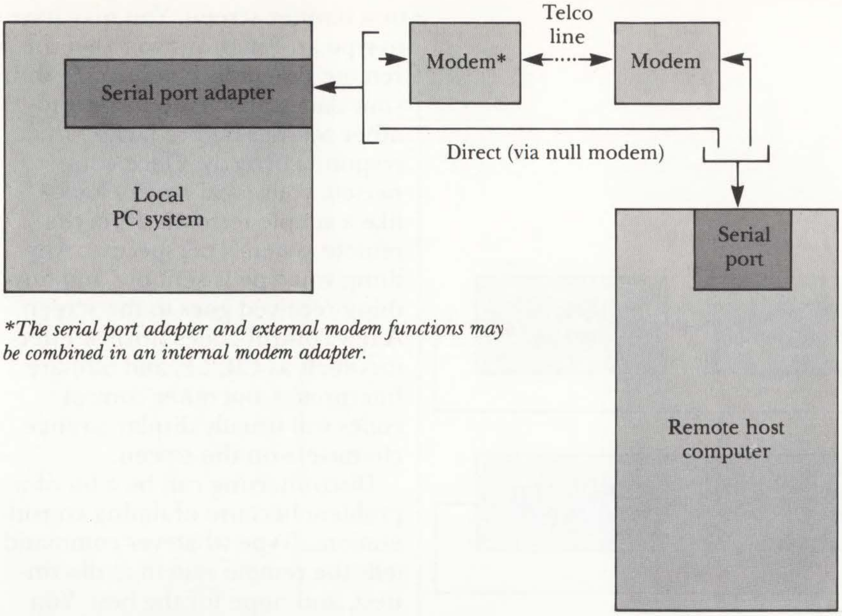


Figure 1. Typical PC-communications setups.

**The serial port adapter and external modem functions may be combined in an internal modem adapter.*

than baud or state transitions per second) regardless of the value because it is the number of bits per unit of time that really describes how fast data are being transmitted.

Another important communications parameter is *parity*. Parity provides a simple method of detecting single-bit errors by checking the number of bits set to logical 1 in a received character. If even parity is being employed, for example, the transmitter of data will set or clear the parity bit to produce a code that contains an even number of 1-bits. This code is then sent over the line, and if the receiver counts an odd number of 1-bits, it has reason to believe the received character was corrupted during transmission. However, even if a transmission error can be detected this way, no means of correcting the error exists in simple asynchronous transmission. Parity may be set to even, odd, none (the parity bit can arbitrarily take on any value or be absent), space (the parity bit always set to 0), or mark (the parity bit always set to 1).

The *number of data bits* in a transmitted character is another communications parameter that must be specified. This parameter indicates how many bits are used to form the codes that represent transmitted characters. Values from five to eight data bits are permitted. The ASCII character set requires seven bits, permitting codes ranging from 0 to 127. Eight-bit encoding can be used to express 256 distinct codes (0 to 255) and may be used to transfer extended codes found in IBM PC character graphics and true 8-bit data that comprise binary files such as executable programs. When eight data bits are used, the parity setting must be none.

Each transmitted character code is framed by one start bit and one or more stop bits. The number of stop bits is specified as either one or two. At speeds below 300 bps (typically the standard Teletype rate of 110 bps) specify two; otherwise use one. When the number of data bits is only five, specifying two stop bits results in a stop period equivalent to one and a half stop bits.

Of course, we must specify the serial port to use. On an IBM PC or compatible machine, the serial ports are identified as the primary adapter, COM1, and the secondary or alternate adapter, COM2. AUX may be used as a synonym for COM1 because it is initialized by DOS to be the same as COM1. The DOS MODE command can be used to assign AUX to COM2. Some hardware add-on boards use tricks to provide additional communications ports, but DOS does not support additional serial ports without the installation of special device drivers.

With the preliminaries out of the way, let's see what it takes to write a communications program in Turbo Basic. You may be surprised to find out how little programming is needed.

A MINIMAL COMMUNICATIONS PROGRAM

The four-line program in MINCOM.BAS (Listing 1) shows the simplest working communications program that can be written in any dialect of BASIC, to my knowledge. The program makes some assumptions that may not be correct for a given machine and communications channel; therefore, you may need to modify the source code (transmission rate, parity, etc.) for your system.

The logic of MINCOM is shown in Figure 2. The first step initializes the selected communications port for the required transmission rate, parity, and the number of data and stop bits, as well as setting up a communications buffer. Then the program enters an endless loop that alternates between sampling the characters from the port ("polling" received data) and the keyboard (data to be transmitted). In this simple program, anything that is received from the communications line is displayed on the local screen and anything keyed by the user is sent to the

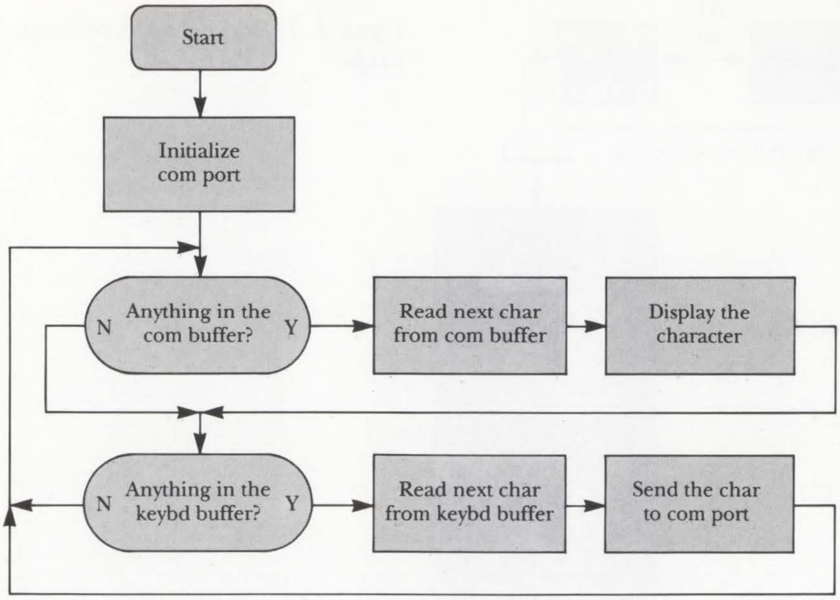

Start
Initialize com port
Anything in the com buffer?
N
Y
Read next char from com buffer
Display the character
Anything in the keybd buffer?
N
Y
Read next char from keybd buffer
Send the char to com port

Figure 2. The logic of a minimal asynchronous communications program.

BASIC COMMUNICATIONS

serial port for transmission to the remote system.

Care must be taken to avoid operations that stop the loop by waiting for either incoming or outgoing data. Therefore, a test is made to see whether the end of file (EOF) has been reached on the received data before an attempt is made to read anything. **EOF** returning a value of **TRUE** (−1) indicates that the receive buffer is empty. We'll examine the receive buffer in detail shortly.

The same practice prevents a blocking read from occurring on the keyboard. The **INSTAT** function returns a value of **TRUE** (−1) if any unread characters are in the keyboard buffer and **FALSE** (0) if there is nothing to read. If the buffer is not empty, **INKEY$** reads the next available character, which is immediately transmitted to the remote system.

USING MINCOM

MINCOM provides a real operating challenge because it does nothing to help the user. Dialing is entirely manual. There are no safety nets, so performance on noisy lines is likely to be poor. And making a clean disconnect is purely a matter of luck. Also, because Turbo Basic adds a line feed to any received carriage return character, displayed output is always double spaced. But the program works and is simple enough to be totally understandable, so it's a good starting point.

You can run MINCOM from within the Turbo Basic environment or as a standalone executable program. In either case, you should select the option that enables keyboard break (using the Ctrl-Break combination) before compiling and running the program. Otherwise it may be difficult to get out of the polling loop once you're finished using the program.

The following operating instructions assume that you are using a Hayes-compatible modem and a Touch-Tone telephone. They also assume that you have compiled MINCOM to a standalone .EXE file. To make a call, turn on the modem and start the program by typing MINCOM followed by an Enter. To dial, type the Hayes command prefix AT (in uppercase) followed by DT (the Hayes commands meaning "dial using Touch-Tones"), followed by the number. Terminate the command with an Enter. You may need to modify this procedure to suit your operating conditions (modem and telephone types).

If all goes well with the call, you should get some indication that the remote system is ready to communicate, such as a log-in prompt or a banner screen. You may have to type an Enter or two to get the remote system to synchronize with your data transmission rate and other settings before it can respond correctly. Once connected, your local system looks like a simple terminal from the remote system's perspective. Anything you type is sent out and anything received goes to the screen. A few control codes (format effectors such as CR, LF, and tab) are interpreted, but other control codes will usually display strange characters on the screen.

Disconnecting can be a bit of a problem because of timing considerations. Type whatever command tells the remote system to disconnect, and hope for the best. You may be able to get back to DOS by pressing Ctrl-Break. Often, however, MINCOM will hang the local system awaiting input from the communication line that never arrives. If this happens to you, do a warm reboot (Ctrl-Alt-Del) to restart your computer.

COMMUNICATIONS SUPPORT

Although it appears to be a very simple program, MINCOM is, in fact, a rather complex piece of code at the machine level. Hidden from your view as a Turbo Basic programmer is all the work that is done by the **OPEN COM** statement and the statements that test and read the internal buffers established by Turbo Basic and DOS. Indeed, this automatic buffering is MINCOM's key to success. Without it, the program would not work very well at all.

Figure 3 depicts an implementation of a receive buffer. Incoming data from the communications lines is automatically buffered by the runtime code. Unless told otherwise, Turbo Basic establishes a 256-byte circular buffer (I'll explain why it's called "circular" shortly) and installs an interrupt-driven interface routine. A Turbo Basic metastatement may be used to specify a different buffer size

```
$COMn size
```

where **n** is the port number and **size** is the buffer size in bytes.

Upon receipt of a character in the port's receive data register (which can hold only one character—read it or lose it!), the interrupt-routine adds the character to the receive buffer and advances a pointer—the write pointer—to the next location where a character will be written.

A read pointer that points to the buffer location where the next read-character operation will be attempted is also maintained by the interface routine. An **INPUT$** function call can be used to extract a string of a specified length from a communications buffer beginning at the current read position. After a successful read operation, the read pointer is advanced to the next location in the buffer, marking the point at which the next call to an input function will begin reading.

If the write and read pointers are equal (pointing to the same place) the input buffer is empty. An attempt to read a character or string from an empty buffer, with **INPUT$** for example, will apparently hang the program by waiting for something to arrive. A program should use the end-of-file **(EOF)** function to determine whether there is anything to read to prevent this sort of freeze-up. The **EOF** function returns a **TRUE** (−1) value when the buffer is empty and **FALSE** (0) otherwise.

The buffer is described as circular although it is really just a linear array of character-sized storage locations. The routines that control reading and writing operations automatically move their pointers back to the beginning of the buffer if an attempt is made to increment past the end of the buffer. Logically, the buffer is like a ring—it has no beginning or end.

COMMUNICATIONS EVENT TRAPPING

MINCOM, when compiled by Turbo Basic and run on a 4.77 MHz IBM PC, will keep up handily with a 1200 bps input stream.

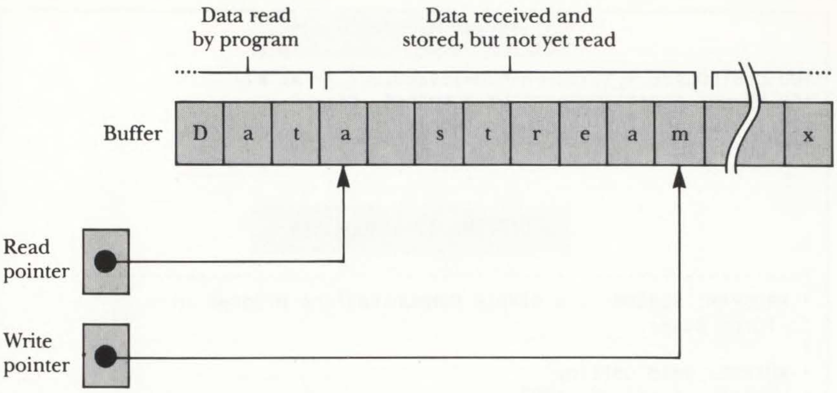


Figure 3. Received data buffering.

At higher transmission rates, problems can arise when characters are arriving rapidly in long, continuous streams, and especially when a color/graphics adapter is being used and the screen must be scrolled following a CR/LF on the bottom line of the screen.

The interrupt-driven interface put into place by the call to **OPEN COM** takes care of preserving the incoming characters in the buffer, but our programs still have to empty the buffer as rapidly as possible. If characters are not taken out fast enough, data will eventually be lost because the write pointer will overtake the read pointer, writing over data that has yet to be read. The use of event trapping can help to preclude data loss.

Event trapping is a feature of Turbo Basic that causes a program to check between the execution of program lines (or optionally between statements) to see whether an event has occurred. In addition to checking for keypresses (**ON KEY**), joystick button presses (**ON STRIG**), and a variety of other "ON somethings" that can specified, Turbo Basic can be told to check for the arrival of new characters at a specified communications port.

Two statements are used to control event trapping: **COM** and **ON COM**. The **COM** statement turns trapping on and off, and **ON COM** declares the subroutine that will handle trapped events.

The **COM** statement has the following general form

```
COM(n) { ON | OFF | STOP }
```

where **n** is 1 or 2 and the braces and vertical bars are notation that means "select one and only one of the enclosed choices." **COM ON** enables trapping, so that any newly received characters set an internal flag indicating activity. **COM OFF** disables trapping, thus causing any activity to be completely ignored.

A **COM STOP** instruction halts trapping but remembers that a flag was set. A subsequent **COM ON** instruction results in an immediate trap to the specified service routine. The service routine is identified by an **ON COM** statement of the form

```
ON COM(n) GOSUB label
```

where **n** is either 1 or 2 and **label** is the mnemonic label or line number that is associated with the service routine. If **label** is set to line number 0, trapping is disabled.

When a character arrives, the trap service routine is executed. Turbo Basic clears the flag and does an immediate **COM STOP** just before entering the service routine. This prevents unwanted additional checking and event trapping while the routine is executing. When the trap service routine returns, Turbo Basic executes a **COM ON** to enable event trapping again.

COM and **ON COM** permit us

to write simple programs that respond quickly to asynchronous events. Let's put them to work.

A SIMPLE COMMUNICATIONS PROGRAM

SIMCOM is an elaboration of MINCOM. It adds several important features that make the original program both easier to use and more reliable. It also works around a few problems that result from the way Turbo Basic does certain things.

The SIMCOM program is a full-duplex asynchronous communications program that features user-definable configuration from an external initialization file. Listing 2, SIMCOM.BAS, contains the source for the program. SIMCOM may be run within the Turbo Basic environment or as a separate executable program file. To produce SIMCOM.EXE, read in the source file and select the EXE mode under the Options menu. The Keyboard break option should be OFF, forcing the user to exit the program by using an approved quit command rather than typing Ctrl-Break.

The operating instructions for SIMCOM are similar to the ones for MINCOM. However, when the program starts, it looks in the current directory for the file SIMCOM.INI, which is the initialization file that lets a user customize the communications parameters used by SIMCOM. Listing 3 contains the author's initialization for a system equipped with a Hayes 1200B internal modem.

If the file exists, the default parameter values are replaced by those in the file. If the file is not found, the error recovery routine, specified by the call to **ON ERROR**, bypasses the statements that read and close the file.

Error detection and recovery in SIMCOM is simple but effective. In addition to the disk error procedure just described, the error handler responds to all other errors by jumping back into the keyboard loop. This procedure allows the user to continue operating or at least exit gracefully after an error.

LISTING 1: MINCOM.BAS

```
100 CLS:LOCATE 1,1,1:OPEN "COM1:1200,E,7,1" AS #1
110 IF NOT EOF(1) THEN PRINT INPUT$(1, #1);
120 IF INSTAT THEN PRINT #1, INKEY$;
130 GOTO 110
```

LISTING 2: SIMCOM.BAS

```
'-----------------------------------------------------------------
' PROGRAM: SIMCOM -- a SIMple COMmunications program in
'  Turbo Basic.
'
' AUTHOR: Reid Collins
' WRITTEN: April 25, 1987
'
' DESCRIPTION:
'  SIMCOM is a simple program that provides a basic level of
'  full-duplex asynchronous communications support with a few
'  amenities to enhance its usefulness.
'
'-----------------------------------------------------------------

'-----------------------------------------------------------------
' Initialization -- default values and possible override of the
' communications parameters from a disk init file.
'-----------------------------------------------------------------
CLEAR           ' initialize memory; clear event traps
CLOSE
DEFINT A-Z      ' all vars are ints unless noted otherwise

'--- Boolean values ---
False = 0
True = NOT False

'--- set up an error handler ---
ON ERROR GOTO ErrorHandler

'--- define screen colors ---
Fgnd = 2
Bkgnd = 0

'--- default communications parameters ---
Portnum = 1
Port$ = "COM1"
Rate$ = "1200"  ' specify as bits per second
Parity$ = "E"   ' even parity
Databits$ = "7" ' ASCII characters only
Stopbits$ = "1" ' 1 or 2 stop bits

'--- set screen and cursor ---
SCREEN 0, 1, 0, 0
COLOR FGND, BKGND
CLS
LOCATE 1, 1, 1

'--- flow control variables and constants ---
Paused = False
PausedCount = 0
WarningLevel = 128
Waiting = 0     ' comm input queue size (used for testing)

'--- read init file, if any ---
10 OPEN "SIMCOM.INI" FOR INPUT AS #2
   INPUT #2, Portnum, Port$, Rate$, Parity$, Databits$, Stopbits$
   CLOSE #2
```

continued on page 122

The keyboard loop understands only two commands. These commands are invoked by keys that produce two-character sequences in which the first is a NUL byte (a character with a value of 0) and the second is usually the scan code for the key. The two accepted commands are:

```
Ctrl-End (NUL + 117)--
   Quit (return to TB.EXE or DOS)
F1 (NUL + 59)--
   Send a hardware BREAK signal
```

In addition, the Del key on a PC keyboard issues a NUL byte followed by scan code 83. This sequence is converted to an ASCII DEL code (127) and sent to the remote system.

All standard ASCII characters (7-bit) and IBM Extended ASCII characters (8-bit) typed at the keyboard are sent to the remote system unchanged. The extended characters may be transmitted by using the PC's Alt-number method. For example, the block character (code = 219) is transmitted by holding Alt and keying in 2, 1, and 9 on the numeric keypad, and then releasing the Alt key. Keep in mind that this method of typing IBM extended characters is part of the PC's BIOS keyboard support and *not* SIMCOM or Turbo Basic.

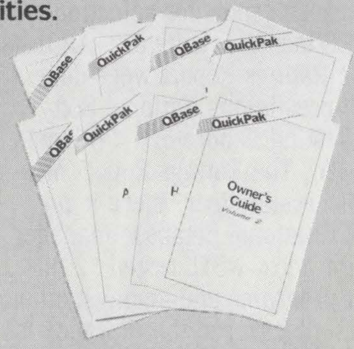
When a character is received from the remote system and stored in the receive buffer, SIMCOM jumps to the **GetComInput** subroutine, which checks to make sure there is really something to read. If so, the subroutine extracts a character from the buffer and processes it. It then branches back to the beginning of the subroutine to see whether there is anything else to read (another character may have arrived while one was being processed). Only when the receive buffer has been completely drained does the **GetComInput** subroutine return control to the keyboard loop.

FLOW CONTROL

At transmission rates of 1200 bps and above, it's possible for the transmitting system to get ahead of the receiving system. Control sequences that cause the receiving system to do time-consuming tasks, such as scrolling the screen or writing data to disk, are usually responsible for the inability to keep up. The receiving system needs a way to tell the transmitting system to stop sending until it is able to catch up.

Flow control is one possible answer to this problem. In a full-duplex setting, the receiving system can send an XOFF character (ASCII code 19) to tell the transmitting system to suspend transmissions. Usually, the receiving system monitors its input buffer. If it gets filled beyond some threshold (**WarningLevel** in SIMCOM), an XOFF is sent. After its receive buffer has been drained completely, or below some minimum level, the receiving system sends XON to signal the transmitting

continued on page 122

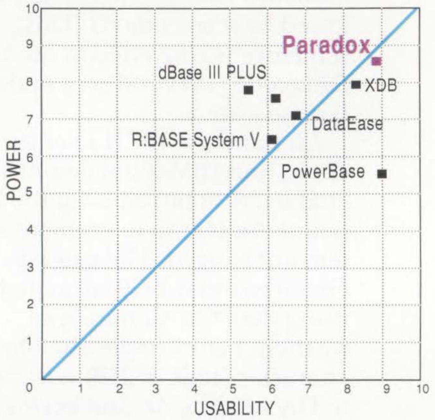


Paradox: the top-rated relational database manager in the world

"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.

*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

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.

How to make your network work

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 1.45 or higher
- AT&T Starlan Network with AT&T PC 6300 Network Program
- Other network configurations that are 100% compatible with DOS 3.1 and one of the listed networks above

The merger of Ansa with Borland is subject to the final approval of shareholders and regulatory authorities.

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 unit return of program control were recorded and averaged.

BI-1158

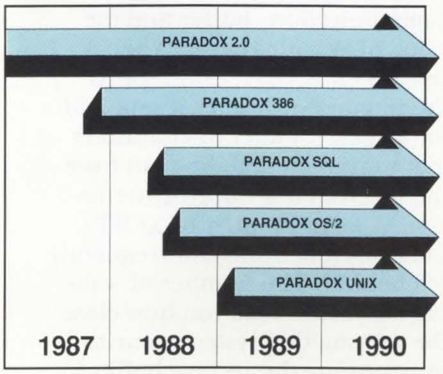
If within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

Paradox makes your network run like clockwork

Paradox is just as valuable to multi and network users as it is to single users. It runs smoothly, intelligently and so transparently that multiusers can access the same data at the same time—without either being aware of each other or getting in each other's way. It works exactly the same way whether you're flying solo or as part of the crew.

> “ Paradox was a delight to use, both as a standalone product and from a local area network server
>
> *Don Crabb, InfoWorld* ”

Paradox saves you from Future Shock



Your investment today in Paradox applications is protected as new generations of hardware emerge. Paradox 2.0 applications will run unchanged on Paradox 386, Paradox OS/2, Paradox Unix and Paradox SQL.

> “ Paradox 2.0 will do for the LAN what the spreadsheet did for the PC
>
> *David Schulman, Bendix Aerospace* ”

> “ Anyone who hasn't seen the network version of Paradox should take a look. Ansa has dramatically advanced the state of the art in multiuser network databases
>
> *Phil Lemmons, BYTE* ”

Paradox updates automatically

Changes made by anyone are automatically updated to everyone. While more than one person can be working in the same table at the same time, there are safeguards that prevent, two users from making changes to the same record at the same time.

Special Offer!

We're making a Special Offer on all three versions of Paradox. Mail in your proof of purchase, dated between September 15, 1987 and December 15, 1987 and your signed registration form for any of the three, and we'll mail you a $100.00 rebate.** It's that simple.

- Paradox 1.1, suggested retail, $495.00
- Paradox 2.0, suggested retail, $725.00
- Paradox Network Pack, suggested retail, $995.00 (each pack supports up to 6 users)

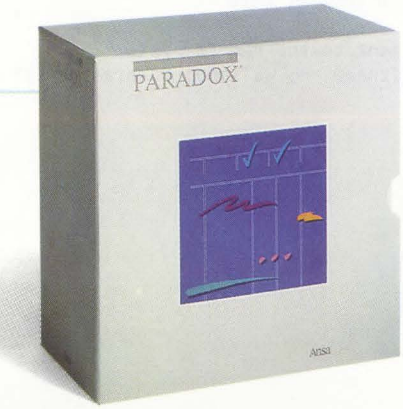
60-Day Money-Back Guarantee

For a brochure or the dealer nearest you call (800) 255-8008, in California (800) 742-1133, in Canada (800) 237-1136.

PARADOX

by Ansa

A Borland Company



```
'--- enable communications event trapping ---
20 IF Portnum < 0 OR Portnum > 2 THEN
     PRINT "Bad port specification: COM"; Portnum
     END
   END IF
   COM(Portnum) ON
   ON COM(Portnum) GOSUB GetComInput

'--- define values for a BREAK signal ---
IF Portnum = 1 THEN
  PortAddr = &H3FB          ' COM1 control port address
ELSE
  PortAddr = &H2FB          ' COM2 control port address
END IF
BreakBit = &H40             ' bit pattern for BREAK on/off control
BreakPeriod! = 0.5          ' seconds to hold a SPACING signal

'--- open the needed data streams ---
COMPARM$ = Port$+":"+Rate$+","+Parity$+","+Databits$+","+Stopbits$
PRINT "SIMCOM Parameters: " + COMPARM$
OPEN COMPARM$ AS #1     ' serial port default setup

'-------------------------------------------------------------------
' Keyboard dialing reoutine -- get a number from the user and call
' the remote system (configured for Hayes-compatible tone dialing).
'-------------------------------------------------------------------
INPUT "Number to call: ", Number$   ' get number for the host
PRINT #1, "ATDT" + Number$   ' call using tone dialing

'-------------------------------------------------------------------
' Main loop -- enter an interactive session with the remote system.
' The program spends most of its time waiting for the user to type
' something.  When data arrives at the com port, it is read and acted
' upon immediately, interrupting anything the keyboard routines may
' be doing at the time.
'-------------------------------------------------------------------
MainLoop:
  WHILE True
   _' process keyboard input, if any
    IF INSTAT THEN
      ' got something -- check for special keys
      K$ = INKEY$
      IF LEN(K$) = 2 THEN
        GOSUB ExtendedCode
      ELSE  ' send anything else to the remote host
        PRINT #1, K$;
      END IF
    END IF
  WEND

'-------------------------------------------------------------------
' Extended code subroutine -- process selected extended key codes.
'-------------------------------------------------------------------
ExtendedCode:
  SELECT CASE ASC(RIGHT$(K$, 1))  ' read the scan code
  CASE 59   ' F1 key pressed -- send a hardware BREAK signal
    GOSUB SendBreak
  CASE 83   ' Del key pressed -- send a real ASCII DEL code
    PRINT #1, CHR$(127);
  CASE 117  ' Ctrl-end pressed -- return to BASIC (or DOS)
            ' after reporting flow control data
    CLS
    PRINT PausedCount; "XOFF(s) sent to the remote system"
    PRINT "Longest waiting data stream ="; Waiting; "Character(s)"
    PRINT "Bye from SIMCOM"
    END
  END SELECT
  RETURN
```

system that it's alright to start sending again.

Although flow control via ASCII XOFF/XON control codes is widely used, it is not universal. Some systems will not honor an XOFF request. It may be necessary to use a lower transmission rate with such systems or break the work into smaller chunks to prevent overflow. Alternatively, a larger buffer can be specified, but this method only delays the buffer overflow and resultant lost data; it doesn't prevent overflow. For simplicity's sake, SIMCOM does not honor flow control requests from the remote system, but you can add it easily if you need to communicate with a slow remote system or another copy of SIMCOM.

SIMCOM incorporates a monitoring feature to help you determine an optimum size for the communications buffer and the value of **WarningLevel**. When a session ends, SIMCOM reports the number of XOFFs it sent and the largest number of characters that were waiting to be read from the receive buffer during the session. A large number of XOFFs indicates that SIMCOM frequently fell behind. The number of waiting characters tells you how close the transmitting system came to overrunning the receive buffer.

A HARDWARE BREAK SIGNAL

Transmitting a hardware BREAK signal requires fiddling with the bits of an I/O port on the serial adapter of the active serial port. SIMCOM initializes several values that are needed to generate the BREAK signal correctly. BREAK is not a normal character. It is a signal—an interruption in the normal flow of data—and its purpose is to get the attention of the remote computer system.

When the user presses the F1 key, SIMCOM imposes zeros on the communications line for a half-second period. It does so by setting and clearing bit 6 of the line control register. On the primary serial adapter (COM1), the line control register is at I/O address (port) 3FBH. On the secondary serial adapter (COM2), it is located at I/O address 2FBH.

To set the bit, the Turbo Basic function **INP** is used to read the port value. The value is ORed with a mask value of 40H. The **OUT** statement, therefore, sets bit 6 in addition to those that were already set. The Turbo Basic **DELAY** statement provides a machine-independent time delay, effectively holding the communications line in the spacing state for a period determined by the **BreakPeriod!** variable (the exclamation point indicates that the variable contains a single-precision floating point value). At the conclusion of the delay period, ANDing the port's contents with the bitwise complement of the mask effectively clears the BREAK bit without affecting any of the other bits.

CONCLUSION

SIMCOM acts like a simple hard-copy terminal. It works well with bulletin board systems and other services that don't require special capabilities. To make it really useful, you will probably want to add a dialing directory, file transfer capabilities, maybe a video terminal emulation, and a friendly user interface. SIMCOM shows, in its simplicity, how to use several capable Turbo Basic statements and functions as the essential building blocks of a communications program. All of the user-convenience features and safety nets can be added easily with only a modest amount of programming. Try your hand at it. It's fun, and you can save yourself $150 or more on the cost of a commercial communications package. ■

Reid Collins is a senior programmer for an aerospace company.

Listings may be downloaded from CompuServe as BASCOMM.ARC

```
'----------------------------------------------------------------------
' Minimal error-handler routine -- a disk error means there is no
' init file.  Anything else causes an immediate return to the
' keyboard input routine (so the user can try to recover)
'----------------------------------------------------------------------
ErrorHandler:
  IF 10 = ERL THEN ' no initialization file found
    PRINT "Using default communication parameters"
    RESUME 20
  ELSE  ' go back to reading the keyboard
    RESUME MainLoop
  END IF

'----------------------------------------------------------------------
' Received data subroutine -- process communications line data
' coming in from the host.
'----------------------------------------------------------------------
GetComInput:
  IF EOF(1) THEN
    IF Paused THEN
      Paused = False            ' OK for host to send again
      PRINT #1, CHR$(17);       ' send an XON to the host
    END IF
    RETURN
  END IF
  ' next two lines are for testing the communications input buffer
  InputBufLen = LOC(1)
  IF InputBufLen > WarningLevel AND not Paused THEN
    Paused = TRUE               ' input buffer filling up
    PRINT #1, CHR$(19);         ' send an XOFF to the host
    PausedCount = PausedCount + 1
  END IF
  IF InputBufLen > Waiting THEN Waiting = InputBufLen
  ' read from the communications buffer
  I$ = INPUT$(1, #1)
  IF I$ = CHR$(8) THEN
    ' simulate a non-destructive backspace
      IF POS(0) > 1 THEN LOCATE , POS(0) - 1
  ELSEIF I$ = CHR$(13) THEN
    LOCATE , 1  ' simulate a lone carriage return
  ELSE
    PRINT I$;   ' display anything else unchanged
  END IF
  GOTO GetComInput              ' see whether there's any more input

'----------------------------------------------------------------------
' BREAK signal subroutine -- holds the communication line at logical
' zero for a period of time that exceeds one normal character
' transmission period (~1/2 second is commonly used).
'----------------------------------------------------------------------
SendBreak:
  OUT PortAddr, (INP(PortAddr) OR BreakBit)       ' set BREAK bit
  DELAY BreakPeriod!
  OUT PortAddr, (INP(PortAddr) AND NOT BreakBit) ' clear BREAK bit
  RETURN
```

LISTING 3: SIMCOM.INI

```
1
COM1
1200
N
8
1
```

EXPLORING THE CIRCLE STATEMENT

From pie charts to Pac-Man, Turbo Basic's CIRCLE statement does it all. Here's how.

Peter Aitken

SQUARE ONE

One of the reasons for the popularity of Turbo Basic is the wide variety of graphics statements built into the language. Using these statements, programmers can produce impressive business graphics, user-friendly interfaces, dazzling games, and most any other sort of screen display that can be imagined.

The **CIRCLE** statement, used to draw circles and ellipses, is one of the most useful graphics commands. This article provides a step-by-step explanation of the **CIRCLE** statement, and then applies the statement in a pie-chart subroutine that can be incorporated into your own programs. **CIRCLE** is a graphics statement, and therefore can be used only if you have a graphics adapter, and only when the adapter is in a graphics mode. Some of the arguments that **CIRCLE** takes refer to screen units, and will consequently have different meanings depending on the screen mode.

In all graphics modes, coordinates of 0,0 refer to the top left corner of the screen, with X coordinates increasing toward the right, and Y coordinates increasing toward the bottom. The lower right corner of the screen has the maximum X and Y coordinates for any graphics mode: these are 320,200 for medium-resolution mode (**SCREEN 1**), 640,200 for high-resolution mode (**SCREEN 2**, **7**, or **8**), 640,350 for enhanced high-resolution EGA mode (**SCREEN 9** or **10**), and 640,480 for VGA enhanced high-resolution (**SCREEN 11** and **12**). With different resolutions, the same X,Y coordinates will refer to different screen locations in different graphics modes.

KNOWING THE MODE

As with all graphics statements, programming with the **CIRCLE** statement requires knowledge of the screen mode that will be in effect when the statement is executed. (See the *Turbo Basic Owner's Handbook* for additional information on graphics screen modes in the reference section under the **SCREEN** statement, p. 333.)

The syntax of the **CIRCLE** statement is as follows:

```
CIRCLE [STEP] (X,Y),Radius[,Color,Start,Stop,Aspect]
```

Arguments in square brackets are optional; the only arguments that *must* be included whenever **CIRCLE** is called are **X**, **Y**, and **Radius**. These required arguments determine where on the screen the circle is located, and how big it is. **X** and **Y** are the coordinates of the center of the circle, in the units of the current screen mode. **Radius**, as you may have guessed, determines the circle's radius.

The **Color** argument determines the color used to draw the circle. The colors available, and the default value used if no **Color** argument is given, depend on the type of graphics adapter you have and on the specific graphics mode that is currently active. (For further details, consult the sections on the **Color** and **SCREEN** statements in the *Turbo Basic Owner's Handbook.*)

The **Start** and **Stop** arguments are used when you want the **CIRCLE** statement to draw only part of a circle. **Start** and **Stop** are angular measurements, that is, they give the angles at which the drawing of the circle starts and stops. However, they use a method of specifying angles that may be unfamiliar to some of you. To be able to effectively use the **CIRCLE** statement, it's necessary to understand how start and stop angles are specified.

Most of us are used to measuring angles in degrees. A complete circle contains 360 degrees. Using a clock for illustration (not a digital clock, but the old-fashioned kind with hands!), at one o'clock the angle between the hands is 30 degrees, at three o'clock it is 90 degrees , at four o'clock it is 120 degrees, at six o'clock it is 180 degrees, and so forth.

The **CIRCLE** statement does not use degrees to specify angles, but uses a unit of angular measurement called the *radian.* One radian is defined as the angle at the center of a circle that subtends an arc equal in length to one radius. Now, you may recall from high school geometry that there is a fixed relationship between a circle's diameter and its circumference such that:

```
circumference/diameter = PI
```

PI is a constant, the same for all circles, and equal (approximately) to 3.14159265. Since diameter = 2 × radius, a circle's circumference has a length equal to 2 × PI × radius. Therefore, the angle of a full circle (i.e., 360 degrees) is equal to 2 × PI radians. The relationship between radians and degrees is that 1 radian equals approximately 57.3 degrees, and 1 degree equals 0.0175 radian.

With the **CIRCLE** statement, 0 radians refers to the point on the circle directly to the right of the center—the "3" on a clock face. From zero, radians increase counterclockwise: PI/2 radians at "12," PI radians at "9," and 3 × PI/2 radians at "6." **CIRCLE** works in a counterclockwise direction also, drawing counterclockwise from **Start** to **Stop**. Thus, the expressions **Start=0 Stop=PI/2** cause **CIRCLE** to draw one-quarter of a circle, while **Start=PI/2 Stop=0** cause **CIRCLE** to draw three-quarters of a circle. Examples of the circles that result from various **Start** and **Stop** values are shown in Figure 1.

If **Start** and/or **Stop** is given as a negative value between 0 and −2×PI, the angle is still interpreted as positive but a radius is drawn from the center to the start or stop point. This permits wedges to be drawn, a feature that will be used in the following pie-chart subroutine. To draw a radius at 0 radians, you should specify a very small negative value rather than "minus zero" (**Start** = -0.0001 rather than **Start** = -0.0).

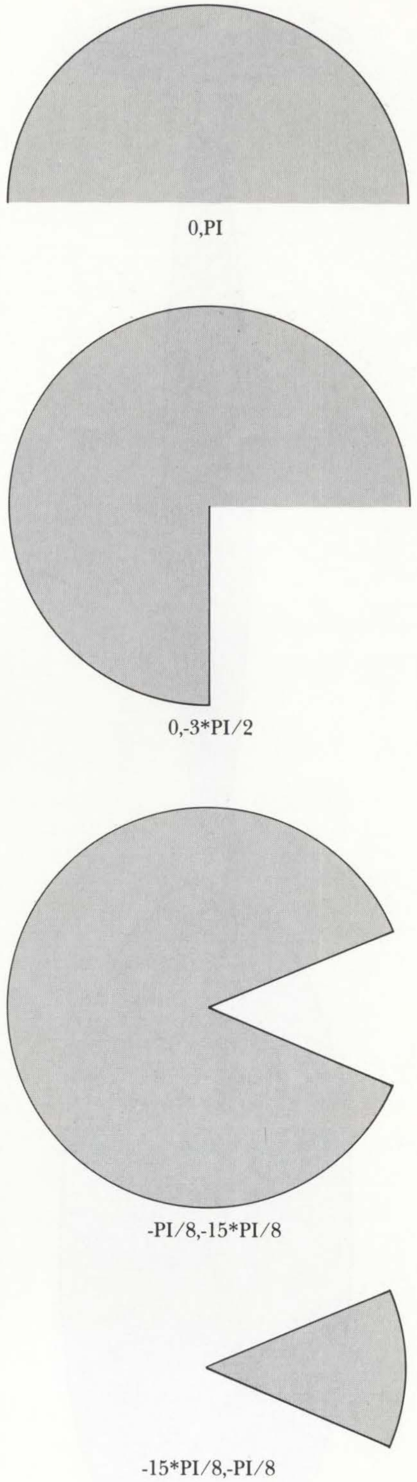
THE ASPECT ARGUMENT

It was mentioned earlier that the **CIRCLE** statement can draw ellipses as well as circles. This is where the **Aspect** argument comes in. In a circle, the radius is the same for all angles. An ellipse is sort of a "squashed" circle, so that the radius in the horizontal, or **X**, direction is different from the radius in the vertical, or **Y**, direction. The **Aspect** argument gives the ratio of the **Y** radius to the **X** radius.

If the **Aspect** argument is omitted, a true circle is drawn. This does not, however, mean that the default aspect is 1. Because of the way graphics units are represented on the screen, a vertical line of a certain number of graphics units will not have the same actual length as a horizontal line of the same number of graphics units. This can be illustrated by running the following short program, which draws vertical and horizontal lines, each 100 graphics units long.

```
SCREEN 2
DRAW "BM 50,50 D100 R100"
```

After running this program, use a ruler to measure the line lengths on your screen. You'll find that the horizontal line is 5/12 the length of the vertical line. If you change the program to run in



0,PI

0,-3*PI/2

-PI/8,-15*PI/8

-15*PI/8,-PI/8

Figure 1. The circles shown here result from executing a **CIRCLE** *statement with the* **Start** *and* **Stop** *values shown* (**PI = 3.14159265**). *Note that the bottom two figures use the same two angles for* **Start** *and* **Stop**, *but are reversed in position.*

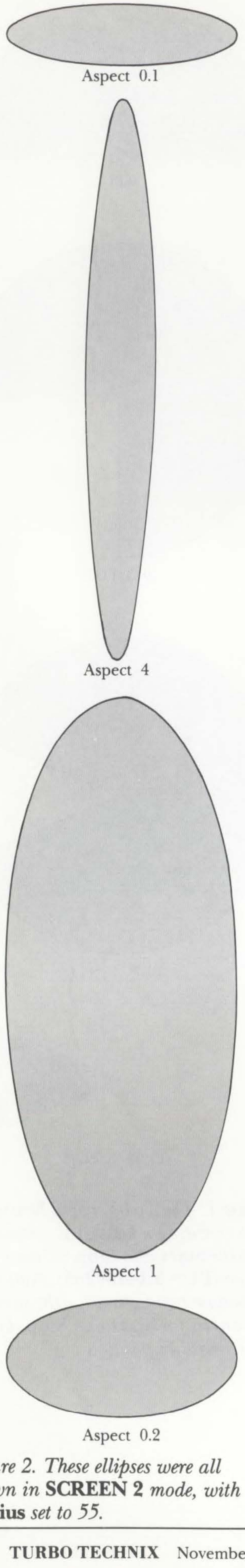


Figure 2. These ellipses were all drawn in **SCREEN 2** *mode, with* **Radius** *set to 55.*

SCREEN 1 mode, the horizontal/vertical ratio will be 5/6. The ratio between the **Y** and **X** lengths is the default aspect ratio used by the **CIRCLE** statement when an **Aspect** argument is not specified. The default aspect ratio is different for different graphics modes, and is automatically determined by Turbo Basic.

If graphics units are different in the **X** and **Y** directions, then how is the **Radius** argument interpreted? If no **Aspect** argument is given, or if an **Aspect** less than 1 is given, then the **Radius** argument specifies the radius in the **X** direction. If an **Aspect** greater than 1 is specified, the **Radius** argument specifies the radius in the **Y** direction. The effects of different **Aspect** arguments are shown in Figure 2.

The **STEP** keyword determines whether the **X** and **Y** coordinates of the circle's center are interpreted in absolute or relative coordinates. If the **STEP** keyword is included in the **CIRCLE** statement, **X** and **Y** are interpreted as relative to the last point referenced, or LPR. The LPR is set to the center of the screen upon entering a graphics mode with a **SCREEN** statement, and is moved by subsequent graphics statements. For example, after a **CIRCLE** statement is executed, the LPR is at the center of the circle. (For further information on LPR, see the discussion in the Graphics section of Chapter 4, p. 110 of the *Turbo Basic Owner's Handbook.*)

If the **STEP** keyword is not included, **X** and **Y** are interpreted in terms of absolute screen coordinates. To illustrate the effect of **STEP**, compare the plot created by

```
SCREEN 2
CIRCLE (320,100),50
CIRCLE (50,50),25
```

with that created by

```
SCREEN 2
CIRCLE (320,100),50
CIRCLE STEP (50,50),25
```

The first program draws the smaller circle at 50,50 relative to the top left corner of the screen (absolute coordinates). The second program draws the smaller circle at 50,50 relative to the center of the large circle (the LPR).

The **CIRCLE** statement is very forgiving in terms of accepting out-of-range arguments. In fact, the **X**, **Y**, **Radius**, **Start**, and **Stop** arguments can accept essentially any value without causing an error. The center, and all or part of the circumference of a circle, can be outside the screen boundaries. Only that part of the circle that falls within the screen boundaries will be drawn. To illustrate, run this program:

```
SCREEN 2
  FOR I=0 TO 39 STEP 3
  CIRCLE (20*I,5*I),10*I
NEXT I
```

As for **Start** and **Stop** arguments, the only values that are meaningful to the **CIRCLE** statement are those between -2 × PI and 2 × PI. If values outside this range are given, Turbo Basic, in effect, divides them by 2 × PI and uses the remainder. Thus, a **Start** argument of 9 × PI is equivalent to PI, 4 × PI is equivalent to 0, −5 × PI/2 is equivalent to−PI/2, and so on.

THE PieChart SUBROUTINE

Now that we have thoroughly explored the components of the **CIRCLE** statement, let's take a look at a subroutine that uses **CIRCLE** for drawing pie charts on the screen. **PieChart** is a simple subroutine, and will draw only one pie chart with a maximum of ten sections, without labels. It's passed the values to be graphed in a one-dimensional integer array **Pie%()**, with **Pie%(0)** containing the number of sections and **Pie%(1)** through **Pie%(Pie%(0))** containing the values for each section.

The line numbers in the subroutine should not be entered when you key in the program—they are included for reference purposes only. **PieChart** is provided within a simple demonstration program called **PieDemo** (you can figure out the workings for yourself if you're interested).

The first line defines the subroutine **PieChart** and indicates that it will be passed a one-dimensional integer array as a parameter. Line 10 declares local variables that are used in the subroutine, and line 20 declares the variable PI, defined in the main program body, as a shared variable. Lines 30-60 assign values to local variables that determine the location and size of the pie chart and the starting point of the first wedge. Line 70 clears the screen and sets graphics mode 2 (640 x 200 resolution).

Lines 80-130 are a loop that executes once for each section of the pie chart. Line 90 determines the stop angle for the wedge (the start angle is set to 0 for the first wedge). Since the full pie chart will be an angle of 2 × PI, one percent of the chart will be 0.01 × 2 × PI or 0.02 × PI, and the total angle for the Nth wedge will be **Pie%(N) × 0.02 × PI**. The stop angle is obtained by adding the total angle to the start angle.

Line 100 draws the wedge. By specifying the stop angle as a negative quantity, a line will be drawn from the stop end of each arc to the center of the circle.

Line 110 sets the start angle for the next wedge equal to the stop angle for the preceding wedge. The loop then cycles to plot the next wedge. Once all wedges are plotted, line 130 waits for a keypress before returning. Since **INSTAT** only detects a keypress but does not remove it from the keyboard buffer, line 140 is included to read the keypress, which would otherwise remain in the keyboard buffer and might cause problems in other parts of the program.

The **PieChart** subroutine can be used as it is, but it also offers considerable room for enhancement. To increase your familiarity with the **CIRCLE** statement, you could try some additional programming exercises. For example, you could modify **PieChart** to produce a so-called "exploding" pie chart, where one wedge is emphasized by being slightly separated from the others, or to use ellipses to make the pie chart appear three-dimensional. ■

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 CIRCLE.ARC

LISTING 1: PIEDEMO.BAS

```
REM - program PIEDEMO
REM - demonstrates PieChart subroutine

      PI=3.141592
      DIM Pie%(10)

Again:
      CLS
      INPUT "HOW MANY CATEGORIES (MAX = 10, 0 TO EXIT)";Pie%(0)
      IF Pie%(0)=0 THEN END
      IF Pie%(0)<2 OR Pie%(0)>10 THEN BEEP : GOTO Again
      Sum=0

      FOR I=1 TO (Pie%(0)-1)
        LOCATE I+10,20
        PRINT "ENTER PERCENTAGE FOR CATEGORY ";I;
        INPUT Pie%(I)
        Sum=Sum+Pie%(I)
      NEXT I

      IF Sum >= 100 THEN
        CLS
        BEEP
        LOCATE 10,20
        PRINT "PERCENTAGES TOTAL MORE THAN 100 - TRY Again"
        FOR I = 1 TO 1000 : NEXT I
        GOTO Again
      ELSE
        Pie%(Pie%(0))=100-Sum
      END IF

      CALL PieChart (Pie%())

      GOTO Again

REM - subroutine starts here

SUB PieChart (Pie%(1))

10    LOCAL XCenter,YCenter,Radius,StartAngle,StopAngle,Counter
20    SHARED PI

30    XCenter    = 0
40    YCenter    = 0
50    Radius     = 160
60    StartAngle = 0
70    CLS : SCREEN 2   'Use SCREEN 9 for EGA color; 10 for EGA mono

80    FOR Counter = 1 TO Pie%(0)
90      StopAngle = StartAngle + Pie%(Counter)*0.02*PI
100     CIRCLE STEP (XCenter,YCenter),Radius,,StartAngle,-StopAngle
110     StartAngle = StopAngle
120   NEXT Counter

130   WHILE NOT INSTAT : WEND
140   K$ = INKEY$

END SUB
```

THINKING IN PAL

The Paradox Application Language can automate any Paradox interactive session.

Todd Freter and Ken Einstein

The paradox of Paradox is that it can at once be both powerful and easy to use. Much has been made of its approachability by unsophisticated users, but far less has been said about one of the facets of its power as a database manager: PAL, the Paradox Application Language. In this article, we'll present an overview of PAL and the entire Paradox applications development environment and provide a brief code example to give you a taste of what it's like to think and program in PAL. In future issues we'll be presenting more detailed exercises with an eye toward improving your PAL skills.

SPEAKING IN PARADOX

Paradox is a fully relational database management system designed to meet the needs of users at all levels of experience. PAL is a complete structured programming language that resides within Paradox and lets you customize the very general Paradox environment into smoothly integrated applications. Other tools in the Paradox package include:

- A built-in editor that is totally integrated with the rest of the environment;
- A powerful debugger;
- The Personal Programmer, a menu-driven application generator that programmers can use both as a prototyping tool and as an automatic code generator for simple to fairly complex applications; and
- A runtime module that can be used to distribute database applications in a cost-effective way.

PAL, a complete programming language in itself, is the most sophisticated of these tools. In style, it resembles a cross between the popular dBase language and C, but with much tighter integration between the language and the database manager's own interactive features. For example, in PAL, programs can directly access Paradox menus or can emulate their operation through a group of primitives called *abbreviated menu commands.* The interactive nature of the product is always available to the programmer.

In fact, the relationship between the interactive use of Paradox and its use through the embedded programming language is the reverse of that typically found in database systems. Usually, the interactive model is based on programming language syntax; interactive use, such as with the dBase Assistant, is considered a way to help non-programmers navigate through the system. In Paradox, virtually all of the functionality of the program is available interactively through menus. It is the interactive use of the product, and not the syntax of the programming language, that governs. In fact, because of this tight integration, it is often useful to think of PAL as an automated Paradox user.

To appreciate PAL as a language it is essential to first understand the nature and capabilities of Paradox itself.

PARADOX FROM A HEIGHT

Paradox is object-oriented. It lets users directly interact at a high level with objects such as tables, forms, and reports. These objects can quickly be created interactively and then directly referenced by applications programs. Because Paradox allows access to these objects at a high level, neither interactive users of the program nor applications developers need be concerned with lower-level operations such as opening and closing files, explicitly saving changes, and so forth.

The Paradox report generator has a highly visual, what-you-see-is-what-you-get interface, allowing very complex reports without programming. It supports up to 16 levels of grouping, a full range of summary calculations, and provides special support for mailing labels and other free-form output. It is almost never necessary to write code solely to generate printed output.

The Paradox query language is a true "query by example" system that allows both programmers and interactive users to construct query statements in a visual, non-procedural manner. When a query is executed at run time, Paradox internally translates the

query statement into procedural code and then optimizes that code to produce the desired results in the shortest possible time. Query statements developed interactively in Paradox can be stored as program modules that can be directly incorporated into applications.

Paradox's virtual memory management system allows it to manipulate large files (limited only by disk capacity) at RAM speed. It automatically pages data between disk and RAM.

Paradox supports both primary and secondary indexes. Indexes are automatically maintained and used by the program, therefore neither interactive users nor applications developers need worry about index maintenance.

Sequences of interactive keystrokes can be recorded as scripts or programs. Recorded scripts can be incorporated directly into more elaborate PAL applications.

PARADOX DEVELOPMENT STRATEGIES

Developing an application under the Paradox environment is essentially a three- (or four-) step process:

1. Because PAL takes full advantage of Paradox, you can first use Paradox to create the tables, queries, forms, reports, and other objects needed for an application.
2. You can then record scripts to capture interactive operations as program modules. You can also use the Paradox Personal Programmer to automatically generate code for operations that can't be done interactively in Paradox, such as constructing custom menus.
3. You can use PAL's Script Editor to write whatever PAL code is needed to establish the flow of control, to customize screen and keyboard I/O, to fine-tune performance, and to tie everything together.
4. Finally, you have the option of using the Paradox Runtime to package and distribute the application to your users.

THE PAL LANGUAGE

In addition to being able to take full advantage of all of the functionality of Paradox, PAL gives you access to additional features and tools:

- Unlimited variables and arrays: The number of variables and arrays you can define is limited only by system memory. Built-in primitives for using arrays to manipulate entire records enable you to move records between tables quickly.
- Procedures: In a manner very similar to C, PAL allows you to create procedures consisting of sequences of commands. Procedures may or may not take arguments or return values. PAL places no restrictions on the number of user-defined procedure definitions that can be active in memory simultaneously, since at run time PAL can automatically swap active procedures into and out of memory if resources become tight. PAL procedures also allow for private and global variables, recursion, nesting of scripts and procedures, and dynamic scoping of variables. You can also create libraries of preparsed procedure definitions.

In addition to traditional control structures such as **IF-THEN-ELSE**, **WHILE-ENDWHILE**, **FOR-ENDFOR**, and **SWITCH-CASE**, PAL has a powerful **SCAN** construct that lets you perform sequences of operations on each record of a table very quickly. Complete control over screen and keyboard I/O is possible through PAL, as is full password encryption: Data can be selectively encrypted, and you have control over user access of data down to the field level.

PAL includes more than 100 built-in functions, including:

- Mathematical, statistical, and transcendental functions;
- Financial functions, such as mortgage-payment calculation and computation of present and future value;
- Powerful string parsing, matching and formatting functions;
- Functions to manipulate and determine the status of Paradox objects.

PAL IN ACTION

In order to appreciate PAL in action, let's look at the definition of a procedure called **CheckPass** that performs password checking for an application. **CheckPass** gives the user of the application three chances to first present a valid user name and then a password that matches the user name. If the name and password presented are valid, the procedure returns the logical value **True**; otherwise it returns **False**.

Note: **CheckPass** as given here contains syntax specific to Paradox 2.0 and will not operate correctly under Paradox 1.1.

The code for **CheckPass** is given in Listing 1. In this example, PAL keywords appear in uppercase to highlight them; in practice, keywords may be uppercase, lowercase, or mixed case as desired.

CheckPass assumes that the user names and passwords are stored in a Paradox table called **secrets**. The table and some sample data are shown in Figure 1. As you can see, the table has two fields: **name** and **password**. **CheckPass** also assumes that the table itself has been password protected and that "dontshowit" is the required password for the table. The table can be created quickly, filled with data and protected by using Paradox interactively.

```
SECRETS====Name====Password==
       1 | Bill   | AC327    |
       2 | Jill   | OPENUP   |
       3 | April  | Sparky   |
       4 | Mike   | 46352    |
       5 | George | Hayward  |
```

Figure 1. The secrets table.

A PAL procedure definition is enclosed in a **PROC-ENDPROC** block within the script. The second line of the procedure declares three variables that are **PRIVATE** to it. These variables are not typed in this declaration; PAL automatically types variables dynamically as values are assigned to them.

Next, the **PASSWORD** command is used to present Paradox with the password "dontshowit" needed to gain access to the protected **secrets** table. The **VIEW** command places the **secrets** table in the Paradox workspace so that its data can be accessed by the PAL program logic. One significant difference between using Paradox interactively and controlling it through PAL is that the workspace can't be viewed on the screen during the running of an application unless the programmer explicitly makes it visible. Thus, although the **VIEW** command places the **secrets** table in the workspace, it is not visible to users of the application.

The **MOVETO FIELD** statement makes the **name** field current. Interactively, this would be done by using the arrow keys on the numeric keypad to move the cursor to the various fields of the table. The **MOVETO** command lets the programmer make a field current without having to worry about where the cursor is located.

The next 16 lines of the program contain a **FOR** loop that gives the user three chances to enter a valid name. At the top of the loop, the statement "@**2,4**" positions the cursor at row 2, column 4 and the "?" command outputs the specified string to the screen at that position. The **ACCEPT** command is a powerful primitive that controls user input. In this case, the program expects the user to enter an alphanumeric value of up to 15 characters; whatever the user enters will be assigned to the private variable **username**. **ACCEPT** automatically performs validity checking on the values entered by the user. For example, if the user were to enter an alphanumeric value into an **ACCEPT** that was expecting a number, PAL would automatically generate an error message to the user.

The **LOCATE** command is then used to check the value in the **username** variable against the values in the name column of the **secrets** table. If the name is present in the table, **LOCATE** makes the record containing that value current. Also, as a side effect, the value returned in a special system variable called **retval** is set to **True**; if the name is not present in the table, **retval** is set to **False**.

The procedure tests the value of **retval** to determine whether or not the entered user name is valid. If the name is valid, the private variable **nameok** is set to **True**, and PAL breaks out of the **FOR** loop with the **QUITLOOP** statement. If the value in **username** is not valid, the user hears a beep, a message is output to the screen and the first command in the loop is reexecuted.

The first **FOR** loop ends with the **nameok** flag set to **True** or **False**. The **IF** in the next statement tests the value of the flag; if equal to **True**, then the user will be asked to provide a valid password associated with that name. A new prompt for a password appears where the prompt for username was displayed, and PAL **ACCEPT**s the user's input and assigns it to the private variable **pass**. The program then checks the password's validity.

This checking is done using a powerful PAL construct called a *field specifier*. Remember that the **LOCATE** command positions the cursor to the record containing the name entered by the user.

PAL allows you to reference each field in the current record by merely enclosing the name of the field in square brackets. Thus the statement

```
IF [password] = pass
```

checks to see whether the value in the **pass** variable is equal to the value in the password field of the current record. If there is not a match, a beep sounds, and a message indicates that the password is invalid. If this is not the user's third password challenge, another attempt may be made to enter the password.

If a correct password is entered, then the **CLEARALL** statement removes the **secrets** table from the workspace, and the **UNPASSWORD** statement withdraws the previously presented password so that the table is protected from any further access. Then the **RETURN** command is used to assign a value of **True** to the **CheckPass** procedure. After a **RETURN** command, all further statements in the procedure are skipped, and the procedure is exited.

If the user fails the three attempts to supply the correct password, or if a valid user name is not entered, then the Paradox workspace is cleared and password access to **secrets** is with-drawn as above, but a value of **False** is returned by the procedure.

When PAL encounters the **PROC-ENDPROC** block in a program, it parses the entire procedure definition and then loads it into memory. It is then available to be called.

The last three lines of code in Listing 1 show how the **CheckPass** procedure could be used in a program once it's been loaded into memory. The **IF** statement in the code calls **CheckPass**. If the value returned by **CheckPass** is **False**, then the abbreviated menu command **EXIT** is issued. **EXIT** has the same effect as an interactive user selecting **Exit** from the Paradox menu—it exits the program and returns to DOS. Therefore, users who are not able to provide a valid name and password won't be able to continue with the application.

As noted above, procedure definitions can also be be stored in libraries. When stored in libraries, procedures may be loaded and run much more quickly, since they do not have to be compiled again when loaded. Also, placing a procedure in a library is one way to encrypt it; in the case of **CheckPass** this would obviously be necessary to prevent enterprising users from reading the PAL source code to learn the specified password to the **secrets** table.

THINK IN PARADOX

Part of the secret of thinking in PAL is, in fact, to keep thinking in Paradox: the syntax of the language follows the interactive use of Paradox very closely. Aim your design at the natural flow of control the user would encounter in getting his or her work done. Anything that the user can do in Paradox, PAL can automate. Define the job to be done, and with PAL, most of the work is over. ■

Ken Einstein is manager of documentation and interface design at Ansa Software. Todd Freter is senior writer/editor at Ansa Software.

Listings may be downloaded from CompuServe as THINKPAL.ARC.

LISTING 1: PASSWD.SC

```
; Checkpass procedure to check passwords and user names

PROC Checkpass()                ; header contains name of proc
PRIVATE nameok, pass, username  ; variables private to proc
PASSWORD "dontshowit"           ; present password for protected
                                  ;  "secrets" table
VIEW "secrets"                  ; places secrets table on workspace
MOVETO FIELD "name"             ; makes "name" field current
FOR i FROM 1 TO 3               ; top of FOR loop to check name
  @2,4 ? "Enter your name: "    ; prompt user
  ACCEPT "A15" TO username      ; get input
  CURSOR OFF
  @5,5 CLEAR EOS
  CURSOR NORMAL
  LOCATE username               ; is name in the table?
  IF retval                     ; Yes, so
    THEN nameok = True          ;  go on to the
      QUITLOOP                  ;  next step
    ELSE                        ; No, so
      BEEP                      ;  tell the user about it
      MESSAGE "That name can't be found"
      nameok = False
    ENDIF
ENDFOR

IF nameok                     ; was a valid name presented?
  THEN                        ; Yes, so
    FOR i FROM 1 TO 3         ; check for valid password
      @2,4 ? "Enter your password: "
      ACCEPT "A15" TO pass
      IF [password] = pass      ; password is good
        THEN
          CLEARALL        ; clear table from workspace
          UNPASSWORD "dontshowit" ; reprotect table
          RETURN True     ; set value and return
        ELSE              ; password no good
          BEEP
          MESSAGE "Invalid password"
       ENDIF
    ENDFOR
    CLEARALL                  ; user failed after 3 tries
    UNPASSWORD "dontshowit"
    RETURN False
  ELSE                        ; user presented an invalid name
    CLEARALL
    UNPASSWORD "dontshowit"
    RETURN False
ENDIF
ENDPROC                       ; end of procedure definition


IF NOT Checkpass()            ; call Checkpass
  THEN EXIT                   ; if not True, then exit ENDIF
```

BINARY ENGINEERING

Divide and rule

Bruce Webster

A true story: In the finals of a state high school programming competition, the contestants were given the task of writing a program to generate all the prime numbers within a certain range and having certain characteristics. Only two contestants came up with a working program. One wrote a program that ran for around three hours before they could verify that it would indeed solve the problem; had they let it run to completion, it would have taken about six hours. The other wrote a program that completely and correctly generated the list of desired primes in about three *minutes*. The first contestant won the competition. Why? Because he took a minute or two less to write his program than the second contestant did.

An observation: Two books sit side by side on a shelf near me. One is titled *The Craft of Programming*, the other *The Science of Programming*. At the other end of the shelf are three more volumes, all of which carry the subtitle, *The Art of Computer Programming*. Between them stand other books: *Elements of Programming Style*, *Structured System Design*, *A Discipline of Programming*, *The Mythical Man-Month*, and one slim volume called *Software Engineering for Micros*, among others. What is programming—art, craft, science, or discipline? Do any of these titles apply? Do all of them? Does it matter to micro programmers?

Well , yes and no. Most of us can cook to one degree or another. Some of us can cook well enough to keep from starving; opening a carton of milk and a box of Cheerios is within our grasp. Others of us can open cans, sauté mushrooms, and brown meat. Yet others can prepare a variety of delicious, attractive meals for ourselves and our families. And there are those who cook well enough to get paid for it. Does it matter how well you can cook? Depends upon what you want to do with it. Is there an advantage in improving your cooking skills, even if you don't want to become a paid chef? It might make your own life more pleasant. And what is cooking, anyhow—art, craft, science, or discipline?

This time, we're going to talk about methodical problem solving: how to attack it step by step.

Programming, like cooking, is done to solve problems. You cook to feed yourself, to feed others, to entertain, to have fun, to justify that expensive stove-top grill you bought. You program to balance books, to contact others, to entertain, to have fun, to justify that expensive desktop system you bought. And, like cooking, programming is faster if you buy ready-made solutions, but cheaper (usually) if you do it yourself.

The point is this: no matter how well you know how to program now, it won't hurt you to learn more. And since you're holding this magazine in your hands, you want to program. We're here to help you to learn how.

A BRIEF ASIDE

Me? My name's Bruce Webster. I program some, and I write some, and I teach some. Keeps me busy. Yes, I do have a degree in computer science and actually worked out in industry for several years.

STEPS TO PROBLEM SOLVING

The essence of programming is problem solving. Yes, it is possible to program just for fun or for whatever reasons, but the reason behind the actual program is to solve the problem. To learn to program well—to learn to be a software engineer—you must learn how to solve problems.

This time, we're going to talk about methodical problem solving. There is a creative side to problem solving, the "Aha!" effect, that leads to sudden breakthroughs. In my experience, that happens most often in the context of step-by-step work. So let's look at some steps to problem solving.

STEP 1: UNDERSTAND THE PROBLEM

Before you can solve a problem, you have to know what the problem is. The difficulty is this—what you think the problem is may not actually be the problem at all.

Case in point. A firm wanted to expedite orders by creating a computerized invoicing system; previously, the secretary had typed each invoice by hand. So, one of the in-house programmers was assigned to develop the system. She developed specifications, consulted with the secretary, designed the program, coded, debugged, and installed it. The result: the secretary got the invoices done more quickly, but orders still weren't expedited. Why? Because the bottleneck was in the shipping room, where the invoices were now piling up.

At the risk of sounding radical, the moral here is to question authority. If your boss comes to you with a problem to solve, ask enough questions (and do enough research) to assure yourself that: 1) the problem indeed needs to be solved, and 2) that the problem has some bearing on the real issues.

STEP 2: ANALYZE THE PROBLEM

Once you've picked the right problem to solve, you need to analyze it to determine the solution(s), if any. If there are no solutions, then you need to go back to Step 1 and find a problem you *can* solve. If there are many solutions, you need to look at each and decide which is most likely to meet your needs.

The answer to solving a problem with a computer almost always involves *output*. Of course, you will immediately start thinking up exceptions. I, of course, will counter by redefining the word "output," which, after all, means far more than numbers on a screen or words on a page. So, rather than debate semantics, let's agree to call a program's solution "output," and go from there.

Why start with the output? Because that determines everything else. A computer program has four basic elements: input (to get the data), data structures (to hold it), algorithms (to manipulate it), and output (to write it out). But since we know, or at least suspect, what the output should be, we work backwards from there.

But (and be honest now) how many of you have had the phrase "balance my checkbook" or "organize my recipes" drift through your mind when contemplating programming on your micro?

An important part of this step, if not the most important part, is to be sure that a computer should be part of the solution. Years ago, in the days of mainframes and punched cards, a computer science professor was approached by a colleague in a different discipline. This colleague was interested in using one of the large computers on campus to develop and maintain a list of names and addresses. This would involve punching several cards for each entry, writing a program to read in and format the data, and printing it out to put in a binder. He felt that this would make it easy to update, add, and remove addresses as needed. He asked the CS professor how he could best go about doing this. The CS professor replied that the best solution would be to buy a small notebook, some paper for it, and a pencil. Write each name and address on a separate piece of paper, and put them all in the notebook. That way, changes could be made by adding or removing sheets of paper, or erasing and rewriting to make modifications. All things considered, the CS professor continued, this would be a far easier solution than what the colleague was proposing.

Given the accessibility and ease of use of most micros, the low-tech approach may no longer be the best solution for this particular problem. But (and be honest now), how many of you have had the phrases "balance my checkbook" or "organize my recipes" drift through your mind when contemplating programming on your (or someone else's) micro? Worse yet, how many of you actually spoke these phrases aloud in justifying your purchase of a computer? (And how many of you were too chicken to give the real reason: to play games and otherwise entertain yourself?)

Well, don't feel too embarrassed; you're in plenty of company. And chances are that you've already reached the conclusion that a pencil and calculator, or a 3 × 5 box and some index cards, solve those problems far better than a computer. Which entitles you to give a small, superior smile when your neighbor gives the same justifications for the computer system he just bought.

STEP 3: DECOMPOSE THE PROBLEM

Some years ago—back in 1965, to be exact—Edsger Dijkstra published an article entitled "Programming Considered as a Human Activity" (*Proceedings of the 1965 IFIP Congress*, pp. 213-217). In it, he discusses the growing complexity of computer programs, the limits of the human mind to comprehend such programs, and then makes the following statement:

> "The technique of mastering complexity has been known since ancient time: *Divide et impera* (Divide and rule)."

What may seem obvious today was innovative and controversial twenty years ago, but time has been on Dijkstra's side. His theme, expanded in a later paper called "The Humble Programmer" (*Communications of the ACM*, Vol. 15, No. 10, pp. 859-66), is that the best way to deal with large, incomprehensible problems is to break them down into small, understandable ones. Furthermore, those smaller problems should be clearly understood and analyzed, and they should stand relatively independent of one another. And once you've solved all the little problems, the big problem has also been solved.

Man-years and megabucks have been lost by an unwillingness or inability to spend time on these first three steps, usually due to a desire to "see results" quickly. And, sadly, managers are some of the worst offenders. I know that I caused a fair amount of hair-pulling because of my approach.

If there's a moral, it's this: The longer you wait before coding, the quicker you'll get the coding—and the project—done.

When I'd get a new project to work on, I'd listen to the requirements, look at any pertinent data, and then go into "ponder" mode: I'd put my feet on my desk, fold my arms across my chest, lean back in my chair, and stare at the wall (or ceiling, depending upon how far back I could lean). Occasionally, I'd close my eyes. Depending upon the size and complexity of the project, this could go on for just a few minutes or last (off and on) for days. In my mind, I would think about the problem, break it down into subproblems, consider solutions, and try to visualize the consequences of those solutions. Once I had a clear idea of the finished result, I would start designing the program. I know there were a few managers who would have liked to fire me, but I kept on completing projects.

STEP 4: WRITE THE PROGRAM USING PSEUDO-CODE

At this point, you're probably chomping at the bit, anxious to sit down at a computer and start churning out code. You *are* ready to start programming, but not in the language you think. You're going to start by writing your program in *pseudo-code.*

What's pseudo-code? The prefix *pseudo* comes from Greek and means "fake" or "false." In other words, you're going to write in a phony programming language. The idea is to code in a language that you (and presumably others) can understand: English, or a structured subset thereof. For example, Figure 1 shows a pseudo-code implementation of a selection sort. Of course, you can make it look more like a "real" programming language if you want, as shown in Figure 2.

```
find the lowest value in the list
swap with the top value
move the top down by one
continue until at end of the list
```

Figure 1. English-like pseudo-code for the selection sort.

Why use pseudo-code? There are basically three reasons. First, you've produced a version of the program that almost any programmer can read, regardless of what languages she (or he) does or doesn't know. Besides, it beats the heck out of flowcharting (gag!) for readability and ease of modification.

Next, it allows you to focus on solutions without worrying too much about implementation details. The language(s) we use can blind us to certain approaches or considerations; by delaying the actual coding as long as possible, we avoid that.

Finally, it lets you quickly move from one language to another. You can take the pseudo-code

```
{ Selection sort by Bruce Webster       }
{ Pseudo-code -- Last modified 9/23/87 }

subroutine SelectionSort(List=array,Count=list size)

variables:  Top, Min, K = index values

  for Top going from 1 to Count-1 by 1 do
    set Min to Top
    for K going from Top+1 to Count by 1 do
      if List[K] is less than List[Min] then
    set Min to K
      end if
        end for

    if Min is different from Top then
      swap List[Min] and List[Top]
    end if
  end for
end Selection
```

Figure 2. Program-like pseudo-code for the selection sort.

```
/* Selection sort by Bruce Webster        */
/*   Turbo C -- Last modified 9/23/87     */

void selection(int *list,int count)

{
  int   top,min,k,temp;

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

Figure 3. The selection sort as implemented in C.

above and translate it into any language as needed.

STEP 5: TRANSLATE INTO THE TARGET LANGUAGE(S)

Once you've gone through all this and have your problem solved in pseudo-code, then you start translating it into your target language. Even if you don't know the target language well, you can translate a good pseudo-code implementation by browsing the reference manual for that language. For example, Figure 3 shows a Turbo C version of the selection sort.

Note that we had to adjust the starting and ending values in the **for** loops, since arrays in C start with an index value of 0 (instead of 1, as we had used in the pseudo-code above).

Will you always use pseudo-code? As you get better and better in a given language, the pseudo-code step becomes tedious and often unnecessary. When that happens, you can switch to *stepwise refinement* for Steps 4 and 5. Stepwise refinement simply means that you start off coding in your target language, but use comments and dummy subroutine calls in place of pseudo-code instructions. You then replace each comment or dummy subroutine with the actual code for that function, making each replacement work before going on to the next item.

If your target language is assembly, you could even use a high-level language for your pseudo-code. Years ago, I got the job of writing a hyphenation program for a spell-checking package. The algorithm was printed in a book and described in convoluted English; the target language was 8080 assembler under CP/M. I had never programmed under CP/M and had only done a little Z-80 coding (and none in 8080). So I wrote the program, using pencil and paper, in Pascal, going over it again and again to make absolutely sure it corresponded to the English description. Then I sat down and "hand-compiled" it into 8080 assembly language, looking up instructions and CP/M system calls as I needed to. I finished the entire program in less than three weeks, despite holding down a full-time job.

THE MORAL

If there's a moral to the column, it's this: the longer you wait before coding, the quicker you'll get the coding—and the project—done. Assuming, of course, that you spend that time designing.

That's it for this issue. Feel free to write me with requests for topics; however, due to the volume of mail and my own crummy letter-writing habits, I can't guarantee individual replies. Until then, I'll see you on the bitstream. ■

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

The Turbo Prolog—Turbo C Connection

Gary Entsminger

The connection between Turbo Prolog and Turbo C is something I'm very excited about, having used it to write (along with my friends Larry Fogg at *Micro Cornucopia* and Mollie Messimer at the University of Virginia), several "smart" programs—including an expert system development tool and a statistical inference system.

In this issue, I'll go over the details of connecting these two snappy languages, and next issue, I'll begin detailing some very practical uses of this connection.

Each language provides the programmer with certain advantages. Since we want to get the most out of each language, it might be worthwhile to first explore some of these advantages. So, let me cover a few of the issues and conclude with a prediction: If you're a Prolog or C programmer, you're going to believe that Turbo Prolog and Turbo C go together like Chuck Berry and a rock and roll rhythm in 4/4.

LISTS VS. ARRAYS

Typically (for convenience and efficiency), we collect objects in single structures. A simple (and powerful) structure in Pascal and C is the array, which is simple for two reasons: 1) its contents are of one type; and 2) memory is automatically allocated by our programs for storing those contents (we don't have to worry about low-level stuff like dynamic memory allocation and deallocation).

This second advantage turns out to be an array's main disadvantage. Memory is automatically allocated, but in advance. Since our program has to know the size of an array when it begins executing, we can't dynamically create new data objects; we can only fill the array. And, it turns out, dynamically creating new data objects is essential for programming relational databases, expert systems, and the like.

Fortunately, programming languages came out of the first dark ages a while ago, and high-level languages (like Pascal and C) supply a variable, called "pointer" for skirting this limitation. (A pointer holds the address of some data rather than the data itself.)

We use pointers to create more complex structures, for example, the linked list, which also has two main advantages: 1) its contents can be of more than one type, and 2) data objects can be dynamically created while our program is running.

We can dynamically create objects while our program is running, but we must allocate memory for them. Turbo C provides the functions—**malloc**, **calloc**, and **free** for this purpose. But it still requires this extra (low-level) step.

The designers of Turbo Prolog decided to take care of this low-level step for us, by including the dynamic memory allocation step in a very powerful pair of dynamic structures—the *list* and the *functor*. In combination, they provide the power of a linked list but require a lot less programming expertise (it's as close to a more-for-less situation as you're likely to encounter).

A *list* (like an array) is a collection of similar objects (or types), but with two primary advantages:

1. Unlike the array in Pascal or C (which Prolog lacks), we don't have to predetermine its size;
2. Unlike the Pascal linked list and C structure we don't have to allocate storage for its contents.

The link between items in the list is built-in.

REPRESENTING STRUCTURES

From Turbo C's point of view, a Prolog list is a recursive structure containing three elements—a **type** (**type**=**1** if it's a list element; or **type**=**2** if it's at the end of a list), the value of the element (which is the same as the Turbo Prolog type), and a pointer to the next node (**node**=**1** if there's a next element; or **node**=**2** if it's at the end of the list or it's an empty list). A list of reals looks like this in Turbo C:

```
struct real_list {
  char functor;
  double val;
  struct real_list *next;
};
```

A Prolog *structure* is an object which contains a set of objects (called components). A *functor* provides a general description of the structure. In Prolog, a functor is represented as

```
f(object)
```

and the C structure is represented as

```
struct real_func {
  char type;
  real value;
};
```

So, a linked list of Pascal- or C-like records might look like this in Prolog

```
record(Rec_Num,List_of_items)
```

where **Rec_Num** can be any number of records containing a list of indeterminate size. Prolog handles all the memory allocation! So we can add record after record (data object after data object) without worrying (much!) about low-level details. This functor-list combo is a very powerful feature that makes Prolog very high-level indeed, and invaluable for many projects (often flying under the AI banner) that depend on dynamic data object creation.

TURBO PROLOG AND THE MEAN

The mean (or central tendency) is an impressively useful statistic, used in many different fields—the sciences (for statistical inferences and as the basis for more sophisticated inferences); in sports (batting averages, shooting percentages, betting odds, etc.); in government, in business, and so on.

The mean is useful, and it's trivial to calculate, which qualifies it beautifully for an example to consider alternative programming strategies.

The mean is equal to the sum of cases (a list of numbers) divided by the number of cases. In Listing 1, we pass the function,

Tail Recursion and the Mean

The code in Listing 6 (another version of mean in Turbo Prolog) is recursive and very fast, but requires so much memory that it can only solve a short list (a few thousand reals on a 640K/286-based system). In order to understand why the code uses so much memory, let's go down a level and see what's happening with the stack (that very important last-in, first-out, volatile bundle of memory).

Each time a program calls a function, it puts the return address of the next function (and other pertinent parameters) on the stack. This enables it to continue processing after it returns from a function. So each time **mean** calls itself, it remembers that it must eventually divide by the count.

Listing 6 is similar to Listing 1 except that it calculates the mean value in the main clause rather than in the terminating clause. So, the recursive call to **mean** appears in the middle of the clause. The final step of our mean calculation, dividing by the count, occurs after the recursive call:

```
mean(H|T]):-
   ...
   mean(T,Y,N2),
   Z = Y/N,
   ...
```

We can change our program's strategy (without changing the actual process of calculation) by moving the division to another variant of **mean,** in effect, concealing the division from the central variant (this is how we came up with Listing 1).

The central variant ends by calling itself, and therefore does not have to save all those addresses (one each time it calls itself). This programming strategy, called tail recursion, is essential for efficient Turbo Prolog programming.

By making our clause tail recursive (i.e. the recursive call is the last call in the clause), Turbo Prolog is able to optimize the internal code generated by the compiler. This technique is known as *tail recursion elimination.* Tail recursion elimination is a method of optimizing your program by replacing recursion with iteration. This technique not only eases the demands placed on the stack, but can greatly improve the speed of execution.

The code in Listing 1 uses tail recursion elimination by eliminating all calls after the recursive call to **mean.**

On a 10MHz 80286-based Multitech-AT (without a numeric coprocessor), Listing 1 calculates the mean of a list of 7000 reals in 1.21 seconds.

Well, 7000 reals in 1.21 seconds seems pretty fast, but I'm never sure how fast "fast" really is, so I decided to test the mean by writing the fastest real processor I could in the fastest high-level procedural language I have, Turbo C, and then comparing it to Turbo Prolog.

The program in Listing 7, the fastest I created in Turbo C, gets its input (7000 reals) from an array. On the same system (same time of day, same current), it takes 2.53 seconds.

In other words, with the right conditions—if you define a relationship (or problem) succinctly and eliminate tail recursion—Turbo Prolog might just be the fastest high-level number crunching language on the PC.

The catch is, of course, the right conditions. And unfortunately, describing a problem succinctly and eliminating tail recursion are suddenly formidable tasks when things get complicated. Eventually, I'm betting, if you program long enough in Prolog, your descriptive prowess or your tail recursion elimination will quit meshing. ■

LISTING 1: MEAN.PRO

```
/* Listing 1 - Prolog module to find the mean using tail recursion
               elimination */

domains
   real_list = real*

database
   answer(real)              /* Create storage for answers. */

predicates
   process(real_list)
   mean(real_list,real,integer)

clauses
   process(List):-
      N=0,                      /* Initialize Count to 0.          */
      S=0,                      /* Initialize temp storage var     */
      mean(List,S,N),           /* Pass "mean" the list and vars. */
      answer(Answer),           /* Get answer from storage.        */
      write(Answer).

   mean([H|T],S,N):-            /* "mean" recursively processes    */
      Y=H+S,                    /* the list --adding each member   */
      N2=N+1,                   /* to the sum of the others and    */
      mean(T,Y,N2).             /* keeping track of the no of      */
                                /* members.                        */
   mean([],S,N):-               /* When the list is empty,         */
      Z=S/N,                    /* divide the total by the count   */
      assert(answer(Z)).        /* and store the answer.           */
```

LISTING 2: MEAN-P.PRO

```
/* Listing 2 - Turbo Prolog module to call C to find the mean,
               which is passed back in a functor. */

domains
   ilist=real*
   ifunc=f(real)

global predicates
   cpinit language c         /* declare Turbo C initialization */
                             /* module, cpinit                 */
   mean(ilist,ifunc) - (i,o) (o,i) language c
                             /* declare Turbo C module, mean.  */
predicates
   process(ilist)

goal
   cpinit,
   process([1.0,5.0,10.0,14.0]).

clauses
   process(List):-
      mean(List,Ans),     /* Call Turbo C module to calculate mean */
      write(Ans),nl.
```

mean, a list of numbers to be summed one by one until the list is exhausted. We count the number of cases (numbers), and eventually divide the sum of the cases by the count. The summation of cases can be represented in Prolog as

```
mean([H|T],S,N):-
   Y = H+S,
   N2 = N+1,
   mean(T,Y,N2).
```

which pulls the first element out of the list, adds it to the current sum **S** and assigns it to **Y**. **N** is used to count the number of elements summed up thus far. So, **N** is incremented and the result is assigned to **N2**. Finally, **mean** is called to add the next element in the list to the total sum.

We are finished with the summation when there are no more elements in the list. So the terminating condition becomes

```
mean([],S,N):-
   Z = S/N,
   asserta(answer(Z)).
```

which says that when the variable list is empty ([]), calculate the mean and assert the result in the database.

TURBO PROLOG TO TURBO C

The program in Listings 2 and 3 is another version of **mean** which sets the stage in Turbo Prolog, calls Turbo C to calculate, and then returns to Prolog for the finishing touches.

This code (with the problem "described" in Turbo Prolog and "solved" in Turbo C) takes 2.63 seconds to calculate 7000 reals. In this case it's slightly slower than Turbo Prolog or Turbo C alone (see sidebar) but impressive nonetheless. We get the descriptive power of Prolog, the processing power of C, and the flexibility of using two qualitatively different languages in one program.

Let's look more closely at Listings 2 and 3 for the details of interfacing between these two languages. In the Turbo Prolog code (Listing 2), note the following points.

- The C functions being called must be declared as global predicates. As with all global declarations, the I/O flow patterns must be declared explicity. In addition, the language to be interfaced (in this case C) must be specified.
- CPINIT is called first in the goal section in order to set memory allocation compatibility between Turbo Prolog and Turbo C (if you're using integers exclusively, you can skip this step).

In the Turbo C code (Listing 3), note the following points.

- The C function **(mean)** which is called by Turbo Prolog is suffixed with **_0**. This corresponds to the **(i,o)** flow pattern specified in the Turbo Prolog module. Thus, the suffix refers to a specific flow pattern and is generated internally by Turbo Prolog. The suffix must be incremented by 1 for each additional flow pattern. For instance, if a second flow pattern were specified, such as **(o,i)**, a second mean function would have to be defined in C and would include the **_1** suffix.
- There is no C main module; it's replaced by the main module (containing the **goal**) in Turbo Prolog.

When compiling your C module, keep these points in mind.

- Compile using the large memory model (the only memory size Turbo Prolog compiles to).
- Compile with register allocation turned off (-r-).
- Compile with generate underbars turned off (-u-).

When linking, remember these points.

- INIT (Turbo Prolog's initialization module) must be the first object file linked.
- CPINIT must be the second object file linked.
- .OBJ modules can follow CPINIT in any order.
- The PROLOG.SYM file must be the last module linked.
- Specify the output (.EXE) file name.

LISTING 3: MEAN-C.C

```
/* Listing 3 - C function called by Turbo Prolog to find the mean.*/

struct ilist {            /* Declare a Turbo Prolog list in C */
    char functor;
    double val;
    struct ilist *next;
};

struct ifunc {           /* Declare a Turbo Prolog functor in C */
    char type;
    double value;
};

void mean_0(struct ilist *in, struct ifunc **out) {
    int count = 0;
    double y = 0, z = 0;

    if (in->functor !=1)
    fail_cc();                  /* 1 indicates a list element. */

    while(in->functor !=2) {    /* 2 indicates an empty list.  */
       y = y + in->val;         /* Keep a running sum of the list.  */
       count = count+1;
       in = in->next;           /* Get the next member of the list. */
    }

    z = y/count;                /* z = the mean.  */

    *out = (struct ifunc *) palloc (sizeof(struct ifunc));
    (*out)->value = z;
    (*out)->type = 1;
}
```

LISTING 4: VARIANCE.PRO

```
/* Listing 4 - Turbo Prolog main module to call Turbo C to calculate
               the variance. The variance is passed back to Turbo
               Prolog as a real */

domains
   ilist=real*
   ifunc=real

database
   answer(real)

global predicates
   cpinit language c              /* declare Turbo C initialization     */
                                  /* module, cpinit  */
   variance(ilist,ifunc) - (i,o) language c
                                  /* declare Turbo C module, variance   */

predicates
   main
   process(ilist)

goal
   cpinit,      /* cpinit must be invoked before we call the */
   main.        /* TC function. */


clauses
   main:-
      List = [1.0,5.0,10.0,14.0],
      process(List).
process(List):-
   variance(List,Answer),    /* Pass the list to variance */
   Answer > 10,              /* Then, the condition succeeds */
   write(Answer).
process(_):-              /* Else, say it doesn't */
   write("Condition fails.").
```

LISTING 5: VARIANCE.C

```
/* Listing 5 - Turbo C function to be called from Turbo Prolog
               to calculate the variance. */

struct ilist {          /* Declare a Turbo Prolog list in C. */
    char functor;
    double val;
    struct ilist *next;
};

struct ifunc {
   char type;
   double value;
};

void variance_0(struct ilist *in, struct ifunc **out) {
    int count = 0;
    double y = 0, dev = 0, square = 0, squares=0, z = 0, var = 0;
    struct ilist *dup_list;

    if (in->functor !=1) fail_cc();
    dup_list = in;            /* Save the address of the head */
                              /* of the list. */

    while(in->functor !=2) {  /* Find mean first. */
       y = y + in->val;
       count = count+1;
       in = in->next;
    }
    z = y/count;
                              /* Then find variance. */
    while(dup_list->functor !=2) {
       dev = z - dup_list->val;
       square = dev * dev;
       squares = squares + square;
       dup_list = dup_list->next; /* Get next list element. */
    }

    var = squares/count;      /* Get var and return it. */

    *out = (struct ifunc *) palloc (sizeof(struct ifunc));
    (*out)->value = var;
    (*out)->type = 1;
}
```

- Finally, list the libraries beginning with PROLOG.LIB. (See the Turbo C manual for specific C library link order.)

My LINK line for **mean** looked like this:

```
TLINK INIT CPINIT MEAN-P MEAN-C
  MEAN-P.SYM,MEAN, ,Prolog+
  EMU+MATHL+CL
```

MEMORY MANAGEMENT

Since we're connecting Turbo Prolog and Turbo C, we're combining two different approaches to the dynamic structure/memory problem. How do we handle it?

When we pass complex structures between Turbo Prolog and Turbo C, we note whether Turbo Prolog will handle the memory management for us or not. And the rule for noting is simple: if the structure originates in a Turbo Prolog module, we don't have to allocate memory for it. If it originates in Turbo C, we do.

In Listings 2 and 3, when our Turbo Prolog module passes the list to our Turbo C module, it's passing a Turbo Prolog list, which means space has been allocated for it. However, when our Turbo C module is finished, it passes its results back in a functor which it has created dynamically while Turbo Prolog was away, so to speak. The size of the return structure (because it is a structure) is unknown to the Turbo Prolog calling module. So, we have to allocate space for it in our Turbo C module.

But, our Turbo Prolog module is in control of our computer's memory once it begins executing. So, our Turbo C function must allocate memory within the Turbo Prolog system, by Turbo Prolog rules. Fortunately, Turbo C and Turbo Prolog know these rules, and we can use two functions for the connection—the **sizeof** function that sizes the structure

```
sizeof(struct ifunc)
```

and **palloc** (one of the special memory management functions contained in CPINIT which allocates storage on the stack for the structure:

```
palloc sizeof(struct ifunc))
```

Objects of known size (reals, chars, and the like) don't need our memory management since Turbo Prolog knows in advance the size of the objects it's getting in return. So returning nonstructures is a piece of cake. See Listing 5 where our Turbo C function calculates and returns the variance via a simple pointer.

The variance is another very useful descriptive statistic, equal to the sum of the deviating squares of a list divided by the number of cases (or observations). (I'll spare you these details, since most of you probably would rather live without them. However, if you're interested in making sense of variance, deviating squares, and so forth, check out the reference to Bradley at the end of this column.)

This is a more complicated problem because we need to calculate the mean before we can calculate the variance. So we'll have to process the list twice in the same function. A second pointer points to our original list and makes it a snap to reprocess the list again quickly in Turbo C.

In Turbo Prolog, however, this would be a little tricky—I'll leave this problem for a homework assignment. ■

REFERENCES

Acquired Intelligence. *"micro einstein" User's Manual and Reference,* 1987.

Borland International. *Turbo Prolog Owner's Handbook,* 1986.

Borland International. *Turbo C User's Guide,* 1987.

Bradley, J. and J. McClelland. *Basic Statistical Concepts,* 1978.

Bratko, I. *Prolog Programming For Artificial Intelligence,* 1986.

Entsminger, GL "Game Theory Modeling In Prolog and C". *Micro Cornucopia,*#31.

Kernighan, B. and D. Ritchie. *The C Programming Language,* 1978.

Gary Entsminger writes on artificial intelligence topics, and is an associate editor of Micro Cornucopia .

Listings may be downloaded from CompuServe as LCV1N1.ARC.

LISTING 6: MEAN2.PRO

```
/* Listing 6 -  Mean in PROLOG without tail recursion elimination */

domains
   real_list = real*

database
   answer(real)
   data(integer,real_list)

predicates
   process(real_list)
   mean(real_list,real,integer)

clauses
   process(List):-
      N=0,
      S=0,
      mean(List,S,N),   /* Call Turbo PROLOG function.         */
      answer(Answer),   /* to calculate the mean.              */
      write(Answer).    /* Get the answer from the answer and */
                        /* report.                             */

   mean([],_,_).
   mean([H|T],S,N):-
      Y=H+S,
      N2=N+1,
      mean(T,Y,N2),
      T = [],              /* These lines force mean to       */
      Z=Y/N2,              /* remember an address each time */
      assert(answer(Z)). /* it calls itself recursively.  */
   mean([_|_],_,_).        /* We need this line to make       */
                           /* mean always succeed.            */
```

LISTING 7: MEAN3.C

```
/* Listing 7 -- a fast mean function in Turbo C */

#include <stdio.h>
void mean(double *list);

main()
{
   double list[7000];             /* Create a list of reals to process */
   double i;                      /* The time to do this isn't included */
                                  /* in the benchmark. */

   for(i = 1; i < 7000; i++)
   list[i]=i;
   mean(list);                    /* Call subroutine, mean  */
                                  /* to calculate.          */
}
   void mean(double *list) {
      int i;
      double x = 0, z;

      puts("start");                /* Start timing here */
      for(i = 1; i < 6999; i++){
      x = list[i] + x;            /* Add each element of */
   }                              /* the array.          */
   z = x/i;                       /* Divide by count */
   printf("%f\n",z);              /* Print result */
}
```

ARCHIMEDES' NOTEBOOK

Flexible curve-fitting with Eureka

Namir Clement Shammas

Eureka's powerful optimization engine can solve a variety of least-squares curve-fitting problems. This article looks at using Eureka: The Solver for what are commonly called noniterative and iterative curve-fitting problems. Of course, Eureka itself solves all of the problems via iterative optimization techniques.

LINEAR AND POWER MODELS: RELATING TWO VARIABLES

The first category of fitted mathematical models simply relates two observed variables, call them **Y** and **X**. The variable **Y** (i.e., the dependent variable) is a measured response to the changes in variable **X** (i.e., the independent variable). The simplest relation between any two variables is the *linear model*:

```
Y(X) = a * X^b
```

Despite its simplicity, many physical and chemical laws, as well as practical correlations, are linear (or are approximated to linearity for practical ranges of values). Nevertheless, there are numerous cases where nonlinear models must be used. For example, the *power model* correlates the variables **X** and **Y** using:

$$Y = a\,X^b \qquad (1)$$

Traditionally, to use the above model with statistical packages you must transform equation (1) into:

$$\log Y = \log a + b \log X \qquad (2)$$

However, when using Eureka you need not resort to equation (2). Instead, the nonlinear form of equation (1) will do just fine. In Eureka, equation (1) is expressed in a slightly modified form:

```
Y(X) = a * X^b                        (3)
```

This tells Eureka that **X** is the independent variable, not **a** or **b.** Similarly, other popular nonlinear models (such as the logarithmic, exponential, reciprocal, and square root models, to name a few) can be used in their direct form. Keep in mind, however, that Eureka does not support mathematical transformations with function definitions. For instance, the following function definition is illegal:

```
ln(f(x)) = a0 + a1 + x
```

Correlating two variables is extended to a popular category of functions, namely, polynomials. A quadratic fit is expressed in Eureka as:

```
Y(X) = a0 + a1 * X + a2 * X^2   (4)
```

Higher-order polynomials are similarly written. In addition to polynomials, you may use any arbitrary function that you feel is meaningful to correlate variables **X** and **Y**. Listing 1 shows a Eureka file that lists the two categories of functions used in correlating **X** and **Y**. In the listing, I use **f(x)** instead of **Y(X)** in expressing the dependent variable. The sample problem in Listing 1 fits a power model into the given data. The data reveal that the fitted curve approximates a square function.

MULTIPLE VARIABLE MODELS

The power and versatility of Eureka enables you to tackle the correlation of more than two observed variables. Using linear models, this class of problem is known in statistics as *multiple linear regression.* Eureka is able to handle simple and complex models that involve multiple variables. In the category of simple multivariable models, the observed response is assumed to be the sum of the linear combination of the observed dependent variables. The general model is:

$$f(x_1,\ldots,x_n) = a_0 + a_1 f_1(x_1) + \ldots + a_n f_n(x_n) \qquad (5)$$

Notice that each right-hand term uses a single-variable function. No cross-product terms are used. Examples of models in this category are:

```
f(X1,X2) = a0 + a1 * X1 + a2 * X2

f(X1,X2)
 = a0 + a1 * log(X1) + a2 / X2

f(X1,X2,X3)
 = a0 + a1 * X1 + a2 * X2 + a3 * X3

f(X1,X2,X3)
 = a0 + a1 * sqrt(X1) + a2
 * log(X2) + a3 * X3
```

Eureka is able to handle cross-product models with equal ease. The general model is:

$$f(x_1,\ldots,x_n) = a_0 + a_1 f_1(x_1,\ldots,x_n) + \ldots + a_n f_n(x_1,\ldots,x_n) \qquad (6)$$

Among the most popular equations in this category are the surface-fitting models. Listing 2 contains commented models for both categories. The model for the three-dimensional curve is also shown, along with an arbitrarily chosen model. The sample problem fits the given data with a double quadratic model.

It is important to point out that multivariable correlation models need not be a strictly linear combination. I have used the linear form due to its popularity. Eureka is able to handle nonlinear forms as well. For example, you may write an empirical equation for three independent variables as:

LISTING 1: CFIT1.EKA

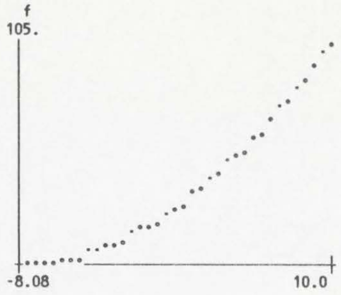
```
; CFIT1.EKA
;
; version 1.0
; August 10, 1987
; Copyright (c) Namir Clement Shammas
;
; Least-square fitting between two variables
;
; general model is:     f(x) = expression of variable x
;
; examples:
;
; Simple models
; -------------
;
;  linear      fitting  --->   f(x) = a0 + a1 * x
;  power       fitting  --->   f(x) = a0 * x^a1
;  logarithmic fitting  --->   f(x) = a0 + a1 * Ln(x)
;  exponential fitting  --->   f(x) = exp(a0 + a1 * x)
;  reciprocal  fitting  --->   f(x) = a0 + a1 / x
;  square-root fitting  --->   f(x) = a0 + a1 * sqrt(x)
;
; Advanced models
; ---------------
;
;  quadratic fitting  --->   f(x) =  a0 + a1 * x + a2 * x^2
;  cubic     fitting  --->   f(x) =  a0 + a1 * x + a2 * x^2 + a3 * X^3
;  hybrid fitting e.g.:
;       f(x) = b1 / x + b0 * Ln(x) + a0 + a1 * x + a2 *x^2

$ substlevel = 0

; state selected model here
; Model: power
f(x) = a0 * x^a1

f(1) = 1
f(2) = 4.1
f(3) = 8.9
f(4) = 16
f(5) = 25
```

LISTING 2: CFIT2.EKA

```
; CFIT2.EKA
;
; version 1.0
; August 10, 1987
; Copyright (c) 1987 Namir Clement Shammas
;
; Least-square fitting between three or more variables
;
; general model is:
;
;     f(x1,...,xn) =  a0 + a1 f1(x1,...,xn) + a2 + f2(x1,...,xn) +
;                     an fn(x1,...,xn)
;
; examples:
;
; Simple models: no cross-product terms
; ------------------------------------
;
;   f(x1,x2)        = a0 + a1 * x1 + a2 * x2
;   f(x1,x2,x3)     = a0 + a1 * x1 + a2 * x2 + a3 * x3
;   f(x1,x2)        = a0 + a1 * Ln(x1) + a2 * sqrt(x2)
;   f(x1,x2,x3)     = a0 + a1 * x1^2 + a2 * x2 + a3 * Ln(x3)
;
; Advanced models: with cross-product terms
; ----------------------------------------
;
;  3-D surface
;   f(x1,x2) = a0 + a1 * x1 + a2 + x1^2 + b1 * x2 + b2 * x2^2 + c1
;
;   f(x1,x2,x3) = a0 + a1 * x1 * sqrt(x2) + a2 * Ln(x2) * x3^2

$ substlevel = 0

; state selected model here
f(x1,x2) = a0 + a1 * x1 + a2 * x1^2 + b1 * x2 + b2 * x2^2

f(1,1) = 11
f(1,2) = 17
f(1,3) = 27
f(1,4) = 41
f(2,1) = 7
f(2,2) = 14
f(2,3) = 25
f(2,4) = 40
f(3,1) = -1
f(3,2) = 7
f(3,3) = 19
f(3,4) = 35
f(4,1) = -13
f(4,2) = -4
f(4,3) = 9
f(4,4) = 26
```

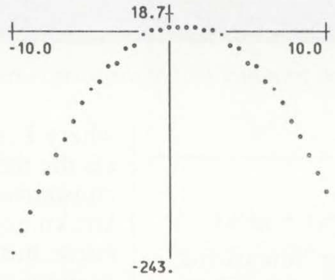
PLOT1: CFIT1.PLT

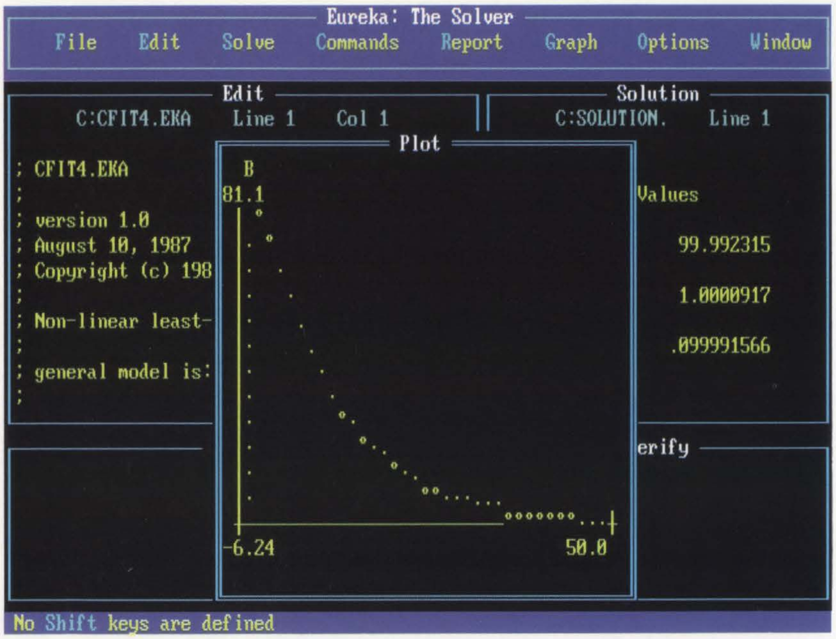
```
Solution:

 Variables     Values

 a0            =       .99806750

 a1            =       2.0009539


Maximum error is      .10508944
```



PLOT2: CFIT2.PLT

```
Solution:

 Variables     Values

 a0            =       3.7500000

 a1            =       3.5000000

 a2            =      -2.0000000

 b1            =       1.5000000

 b2            =       2.0000000


Maximum error is      2.2500000
```





```
; CFIT4.EKA
;
; version 1.0
; August 10, 1987
; Copyright (c) 198
;
; Non-linear least-
;
; general model is:
;
```

```
99.992315

1.0000917

.099991566
```

No Shift keys are defined

Figure 1. Plotting the curve in a text window.

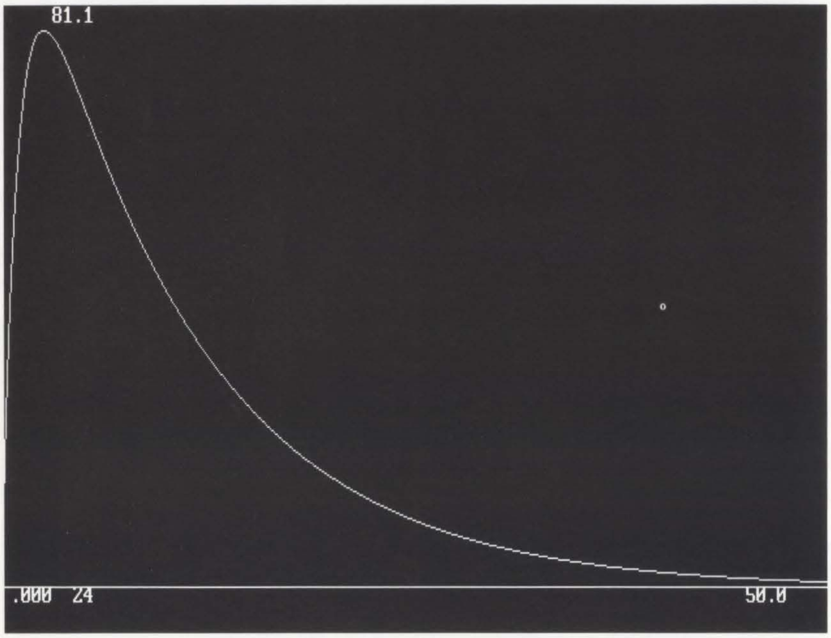


Figure 2. A full-screen graphics plot of the same curve.

NOTEBOOK

```
f(x1,x2,x3)
 = a0 * x1^a1 * x2^a2 * x3^a3
```

without resorting to "linearizing" the above equation using a logarithmic transformation.

NONLINEAR CURVE-FITTING

The next two popular problems traditionally fall into the iterative curve-fitting category. The first deals with measuring the time response of a first-order system:

$$Y = Y_0 (1 - e^{-kt}) \qquad (7)$$

where Y is the measured response, t is the time, and Y and k are both constants. Equation (7), also known as the crescent-shaped curve, has popular manifestations in physics, chemistry, and engineering. Consider a modified form of equation (7), appropriately called the *modified crescent-shaped curve:*

$$Y = Y_0 - Y_0' \; e^{-k\,t} \qquad (8)$$

Y_0' is approximately equal to Y_0. The explicit distinction between Y_0 and Y_0' gives the model more flexibility to handle errors in the data.

Listing 3 shows a Eureka program that fits a given set of data with the model in equation (8). The data represents the rise in the concentration of oxygen in water that was initially oxygen-free. The solution computed by Eureka, without explicit initial values, comes very close to the exact values of the coefficients.

The second case of nonlinear curve fitting seeks to find the unknown coefficients, k_1 and k_2, in the following system of differential equations:

$$\frac{dA}{dt} = -k_1 A$$

$$\frac{dB}{dt} = k_1 A - k_2 B$$

solving the above equations for the condition $t = 0$, $A = A_0$, and $B = 0$ results in:

$$A = A_0 \; e^{-kt} \qquad (9)$$

$$B = \frac{A_0 k_1}{(k_2 - k_1)(e^{-k_1 t} - e^{-k_2 t})} \qquad (10)$$

Chemistry offers an application for equations (9) and (10). These equations represent a solution to two consecutive first-order reactions that convert an initial compound, *A*, into an intermediate compound, *B*, before yielding the final product, *C*. Handling equation (9) is very easy. Hence, I will concentrate on obtaining the values of k_1, k_2, and A_0 using equation (10), given the observed data for variables *B* and *t*. Listing 4 shows a Eureka program with sample data using the model in equation (10). Notice that I have included an explicit initial guess value for A_0. The nature of the problem enables you to consistently provide such a guess. This is beneficial in obtaining the solution.

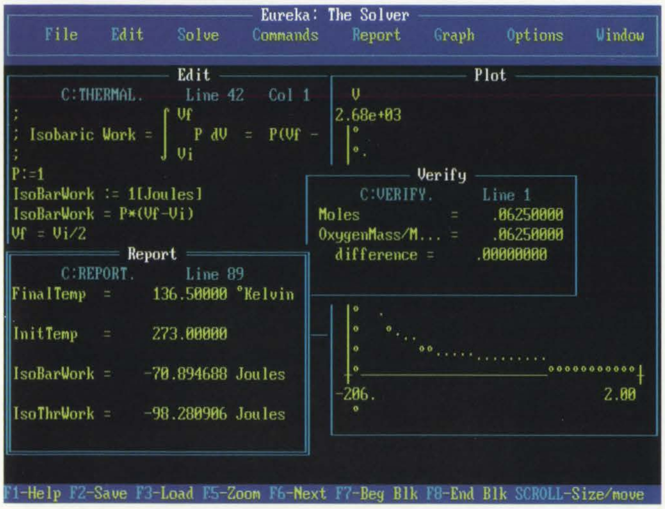
The last line of Listing 4 includes the results of running the example, and shows the close agreement between the exact solution and that obtained by Eureka. Keep in mind that the data are ideal and contain no measurement errors. The graph of the fitted function is also included, showing that the function has a maximum. ■

Namir Clement Shammas is editor of the Turbo Tech Report *newsletter, and a columnist for* Dr. Dobb's Journal *and* PC AI.

Listings may be downloaded from CompuServe as CURVEFIT.ARC.

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™



```
                 Edit
   C:THERMAL.     Line 42   Col 1
;
; Isobaric Work = ∫ Vf P dV  =  P(Vf -
;                  Vi
P:=1
IsoBarWork := 1[Joules]
IsoBarWork = P*(Vf-Vi)
Vf = Vi/2
```

```
                 Verify
      C:VERIFY.        Line 1
Moles               =     .06250000
OxygenMass/M... =     .06250000
  difference =      .00000000
```

```
              Report
   C:REPORT.        Line 89
FinalTemp   =     136.50000 °Kelvin

InitTemp    =     273.00000

IsoBarWork =    -70.894688 Joules

IsoThrWork =    -98.280906 Joules
```

```
F1-Help F2-Save F3-Load F5-Zoom F6-Next F7-Beg Blk F8-End Blk SCROLL-Size/move
```

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

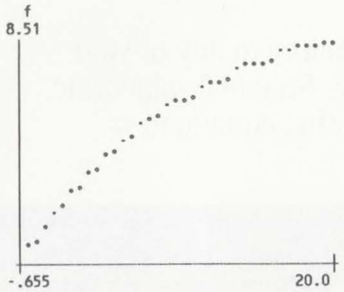
In CA: (800) 742-1133;
In Canada: (800) 237-1136

LISTING 3: CFIT3.EKA

```
; CFIT3.EKA
;
; version 1.0
; August 10, 1987
; Copyright (c) 1987 Namir Clement Shammas
;
; Non-linear least-square fitting of a modified crescent shaped model
;
; general model is:  f(t) = a0 - a1 * exp(-k * t)
;
; where a0, a1 and k are constants, & a0 is approximately equal to a1

$ substlevel = 0

; state selected model here
f(t) = a0 - a1 * exp(-k * t)

f(1) = 0.86
f(2) = 1.65
f(3) = 2.39
f(4) = 3.00
f(5) = 3.58
f(6) = 4.11
f(7) = 4.58
f(8) = 5.01
f(9) = 5.40

; Solution is a0 = a1 = 9.1, k1 = 0.1
```

PLOT3: CFIT3.PLT

```
Solution:

 Variables     Values

 a0        =       9.6658878

 a1        =       9.6107779

 k         =      .090974651

Maximum error is     .039362152
```

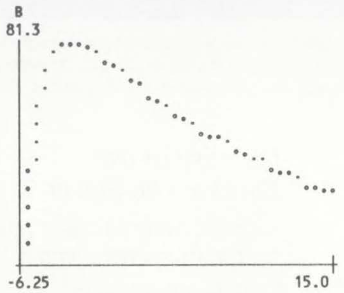


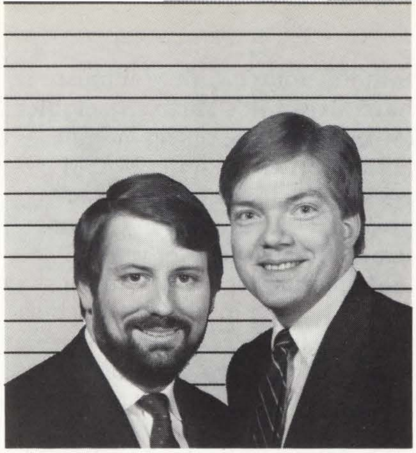
LISTING 4: CFIT4.EKA

```
; CFIT4.EKA
;
; version 1.0
; August 10, 1987
; Copyright (c) 1987 Namir Clement Shammas
;
; Non-linear least-square fitting
;
; general model is:
;
;  B(t) = A0 * k1 /(k2 - k1) * (exp(-k1 * t) - exp(-k2 * t))
;

$ substlevel = 0

; state selected model here
B(t) = A0 * k1 /(k2 - k1) * (exp(-k1 * t) - exp(-k2 * t))

B(1) = 59.66
B(2) = 75.93
B(3) = 76.78
B(4) = 72.44
B(5) = 66.64
B(6) = 60.70
B(7) = 55.07
B(8) = 49.89
B(9) = 45.16

; initial guess for A0
A0 := 110

; Solution is A0 = 100, k1 = 1, k2 = 0.1
```

PLOT4: CFIT4.PLT

```
Solution:

 Variables     Values

 A0        =       99.992302

 k1        =       1.0000918

 k2        =      .099991530

Maximum error is     .0032416853
```





TALES FROM THE RUNTIME

Getting Started

Mark L. Van Name and Bill Catchings

Borland is offering its customers the unusual opportunity to license the source code to its Runtime Library routines for Turbo C. In this column we typically will examine components of that Runtime source code and explore both how they work and how you can modify them. This time around, however, we'll take a look at the Runtime's basic structure and provide some DOS batch files that will help you work on the source code. (To follow common usage, we refer to the Runtime Library simply as the Runtime.)

When you use a programming language, you rarely think only of the language itself, its syntax and semantics. Instead, you must concern yourself with the entire environment in which it operates, from the operating system to the compiler to the linker to the very functions it provides. Many of these support functions typically are not a part of the language's actual syntax. Yet the language would be almost useless without them. This is particularly true of C, which was designed to be a small, portable language.

THE TURBO C RUNTIME FUNCTIONS

A C compiler's runtime library is composed of these support routines. The C language proper includes no interfaces to the outside world. There are no I/O routines or operating system functions. The typical C runtime library fills this gap and provides other useful functions. The Turbo C Runtime offers over 300 functions that help to make it a very useful development tool. Some of the major areas addressed by Turbo C Runtime functions include the following:

- correct setup of the execution environment
- retrieval of command line arguments
- stack and memory manipulation
- operating system interface
- basic input and output
- process management
- string manipulation
- basic math functions
- sorting and searching

Virtually all C compilers provide at least a minimal runtime library, but few give their users the chance to license their runtime source code. Despite the unusual nature of this practice, there are very good reasons you might want this code. It can serve as a useful learning aid, because aside from actual programming, there are few better ways to learn to use a programming language than to read good code written by others. Also, by having the Turbo C Runtime source code, you can better understand exactly what the support routines do and how they do it. This understanding is particularly useful when you must be sure of what a function does, as you might need to be when, for example, you are debugging your own code or trying to write a similar routine. But the most important reason for getting this source code is the opportunity it gives you to modify the capabilities of Runtime routines. You might wish only to replace a Runtime function with one that does something a bit differently, or one that adds a few new features, or simply one that is more efficient for your most common uses. No matter how well a general function performs, you can usually replace it with one that is better tailored to your needs if you have special knowledge of your application.

In general, a runtime library's performance is one area that can be easily affected. For example, the Turbo C Runtime was compiled with options that stressed small code size over speed. You could make the opposite choice. Also, to make it useful on the broadest range of PCs, the Runtime interacts with the user through DOS routines. At the expense of portability, you can improve performance by changing the Runtime I/O routines to work with the BIOS or even go directly to the hardware. In some cases, notably when you are developing a Terminate and Stay Resident (TSR) application, such as SideKick, you would have to change the I/O routines or not use them. This change is required because DOS is not reentrant and so cannot be called from within a TSR. Regardless of the reason you buy this source code, once you load it onto your disk and start looking around you will find that there is a lot of it! Because of the size and number of files, we will assume that you have a hard disk and that Turbo C is already on your system in a top-level directory called TURBOC. For convenience, in both our text and batch files we refer to the hard disk as drive C. You can change those references if necessary to reflect your own disk structure.

If you have Turbo C, you already have the Runtime, of course. You just do not have the source code.

The Runtime library is composed of several libraries and objects in the C:\TURBOC\LIB directory. There are two libraries for floating point: FP87.LIB and EMU.LIB. FP87.LIB is used when you have a math coprocessor (8087, 80287, or 80387) on your system. When you do not, EMU.LIB provides software floating point functions. You will rarely want to change these libraries. In fact, the Runtime source does not include the code for FP87.LIB, although it does have the source for EMU.LIB.

The Runtime also includes three other standard parts: a math library, a library of general C support functions, and a start-up routine. Because of the 8086 family's memory architecture, however, there are several files for each of these. In fact, Turbo C offers six different memory models: tiny, small, compact, medium, large, and huge. Corresponding to these six memory models are six start-up files, named CO<x>.OBJ, where <x> is one of the letters **t**, **s**, **c**, **m**, **l**, or **h**.

THE MATH LIBRARIES

There are also five math libraries, MATH<x>.LIB, and five general C libraries, C<x>.LIB, where <x> is defined as one of the letters listed above except that there are no specific libraries for the tiny memory model. This is because code that uses the tiny and small memory models shares the small (with the specifier **s**) libraries. All programs, regardless of the memory model they employ, use the same floating point libraries (FP87.LIB and EMU.LIB).

The source for the Runtime comes in four directories: MATH, CLIB, EMULIB, and INCLUDE. The first three contain the source for the MATH<x>.LIB, C<x>.LIB, and EMU<x>.LIB libraries, respectively. While there are different finished libraries for each memory model, there is only one set of sources. You use command line options to build the different memory model libraries from the single source set. The fourth directory contains include files that are used in many of the Runtime sources. In all, there are about 275 source code files and 10 include files.

There are three major types of source files, and you can identify them by their file name extension.

The Runtime also includes three other standard parts: a math library, a library of general C support functions, and a start-up routine.

The .C files are completely C code; those that end in .ASM are completely assembler. Taking advantage of Turbo C's ability to include inline assembly code, the .CAS routines contain both C and assembly code, although typically they are nearly all in assembler. The frequent use of assembly code in the Runtime has the advantages of smaller and more efficient Runtime routines than would otherwise be possible. This brings us to a crucial point: If you do not want to mess with assembly code, you will have trouble with the Runtime source code. It contains a great deal of assembly code, including several large, important functions. To work with this source code you must have Microsoft's Macro Assembler (MASM) and the object module librarian (LIB) that the MASM package includes. In fact, if you present TCC (the Turbo C command line executable) with an .ASM module or include inline assembly code in a .C file (such as the .CAS files), TCC will call MASM. We suggest that you put the MASM and LIB executables in a directory that is included in your path. Likewise the C:\TURBOC directory should be in your path so that TC (the Turbo C environment executable) and the other Turbo C executables can always be found.

THE MATH AND CLIB DIRECTORIES

The meat of the Runtime source is in the MATH directory, with about 40 modules, and the CLIB directory, which contains over 230 modules. Both of these directories also contain DOS batch files named MATH.BAT and CLIB.BAT, respectively, that rebuild the libraries. These batch files in turn each depend on a file that lists the files to be linked into the library (MATH.RSP and CLIB.RSP). Both rebuild all five memory model libraries. They use TCC options that optimize for size (-O) and turn on register optimization (-Z).

While all of this sounds reasonable, we suggest that you not use these batch files because of the directory structure that they follow. MATH.BAT and CLIB.BAT assume that they are executed from the directory that contains the sources, that objects will go into a subdirectory of the source directory called OBJ, and that your include files are in

```
C:\INCLUDE
```

and

```
C:\LIBRARY\INCLUDE
```

Few programmers are likely to organize their disks in this fashion.

We suggest an alternative directory structure and we have provided two batch files, NEW_MATH.BAT and NEW_CLIB.BAT, that use these structures (see Listings 1 and 2). First, create a directory for the entire Runtime source in your Turbo C directory, C:\TURBOC\RUNTIME. Under it put the four release directories, now

```
C:\TURBOC\RUNTIME\MATH
C:\TURBOC\RUNTIME\CLIB
C:\TURBOC\RUNTIME\EMULIB
C:\TURBOC\RUNTIME\INCLUDE
```

Then create a fifth directory,

```
C:\TURBOC\RUNTIME\OBJ
```

that will hold the objects that result from your work.

It is seldom a good idea to trash the original copies of anything.

While the libraries that are shipped with Turbo C are in

```
C:\TURBOC\RUNTIME\LIB
```

we feel that you should avoid placing your changed versions there. It is seldom a good idea to trash the original copies of anything. Instead, create a directory,

```
C:\TURBOC\RUNTIME\LIB
```

that will contain the new libraries you build. Start it out with copies of all of the libraries and start-up objects from the standard library directory, and then change them as you desire.

To get Turbo C to use the libraries in these new locations you must inform both the TC environment executable and the TCC command line version. For TCC, either use the argument

```
-LC:\TURBOC\RUNTIME\LIB
```

or add the line to the appropriate TURBOC.CFG. To instruct TC to

LISTING 1: NEW_MATH.BAT

```
ECHO OFF
ECHO ***
ECHO *** Building MATH Library
ECHO ***
ECHO ***
ECHO *** Step 1:  Compiling Model-Independent Modules
ECHO ***
CD \TURBOC\RUNTIME\MATH
DEL *.OBJ
DEL ..\OBJ\*.OBJ
masm FLAGS87 /MX;
masm FTOL /MX /E;
COPY *.OBJ ..\OBJ
ECHO ***
ECHO *** Step 2:  Building SMALL and TINY Memory Model Math Library
ECHO ***
tcc -I\TURBOC\INCLUDE -I\TURBOC\RUNTIME\INCLUDE -O -Z -c -d -mt *.C*
DEL ..\LIB\MATHS.*
lib ..\LIB\MATHS @MATH.RSP
ECHO ***
ECHO *** Step 3:  Building MEDIUM Memory Model Math Library
ECHO ***
DEL *.OBJ
tcc -I\TURBOC\INCLUDE -I\TURBOC\RUNTIME\INCLUDE -O -Z -c -d -mm *.C*
COPY ..\OBJ\*.OBJ
DEL ..\LIB\MATHM.*
lib ..\LIB\MATHM @MATH.RSP
ECHO ***
ECHO *** Step 4:  Building COMPACT Medium Model Math Library
ECHO ***
DEL *.OBJ
tcc -I\TURBOC\INCLUDE -I\TURBOC\RUNTIME\INCLUDE -O -Z -c -d -mc *.C*
COPY ..\OBJ\*.OBJ
DEL ..\LIB\MATHC.*
lib ..\LIB\MATHC @MATH.RSP
ECHO ***
ECHO *** Step 5:  Building LARGE Memory Model Math Library
ECHO ***
DEL *.OBJ
tcc -I\TURBOC\INCLUDE -I\TURBOC\RUNTIME\INCLUDE -O -Z -c -d -ml *.C*
COPY ..\OBJ\*.OBJ
DEL ..\LIB\MATHL.*
lib ..\LIB\MATHL @MATH.RSP
ECHO ***
ECHO *** Step 6:  Building HUGE Memory Model Math Library
ECHO ***
DEL *.OBJ
tcc -I\TURBOC\INCLUDE -I\TURBOC\RUNTIME\INCLUDE -O -Z -c -d -mh *.C*
COPY ..\OBJ\*.OBJ
DEL ..\LIB\MATHH.*
lib ..\LIB\MATHH @MATH.RSP
lib ..\LIB\MATHH -FLAGS87;
ECHO ***
ECHO *** Finished Building MATH Library
ECHO ***
```

use the new libraries, follow this menu path within TC: Options to Environment to Library directory. From the Library directory, enter:

```
C:\TURBOC\RUNTIME\LIB.
```

If you use the Turbo C linker TLINK directly, you must supply the exact name, including the path, of every library that you wish to use in the link.

Our library batch files, NEW__MATH.BAT and NEW__CLIB.BAT, assume this directory structure and also inform you of their progress as they work.

The easiest way to use these batch files is to put them in your C:\TURBOC directory, so that they're on your path along with the other Turbo C executables and .BAT files. They connect to their source directory

```
C:\TURBOC\RUNTIME\LIB

C:\TURBOC\RUNTIME\MATH
```

for

```
NEW_MATH.BAT,

C:\TURBOC\RUNTIME\CLIB
```

for

```
NEW_CLIB.BAT
```

and then remove all objects there and in our proposed object directory. The batch files then rebuild all memory model libraries. This is a time-consuming process; it took about 45 minutes to build CS.LIB on an 8 MHz AT clone, and that is the library for only one of the five memory models.

Consequently, you rarely will want to build only the C<x> libraries completely from scratch. Rather, you will want to change only a few routines and avoid the high cost of rebuilding all of the libraries.

For just these occasions we have written two DOS batch files, UPDC.BAT and UPDASM.BAT, that allow you to change a single .C (or .CAS) or .ASM file, respec-

LISTING 2: NEW_CLIB.BAT

```
ECHO OFF
ECHO ***
ECHO *** Building CLIB
ECHO ***
ECHO ***
ECHO *** Step 1:  Compiling Model-Independent Modules
ECHO ***
CD \TURBOC\RUNTIME\CLIB
DEL *.OBJ
DEL ..\OBJ\*.OBJ
masm EMUINIT     /MX;
masm LDIV        /MX;
masm LRSH        /MX;
masm LXMUL       /MX;
masm OVERFLOW    /MX;
masm PADA        /MX;
masm PADD        /MX;
masm PCMP        /MX;
masm PINA        /MX;
masm PSBP        /MX;
masm SCOPY       /MX;
masm SPUSH       /MX;
COPY *.OBJ ..\OBJ
ECHO ***
ECHO *** Step 2:  Building SMALL and TINY Memory Model Library
ECHO ***
masm C0,C0T      /D__TINY__  /MX;
masm C0,C0S      /D__SMALL__ /MX;
COPY C0T.OBJ ..\LIB
COPY C0S.OBJ ..\LIB
masm SETARGV     /D__SMALL__ /MX;
masm SETENVP     /D__SMALL__ /MX;
masm EXEC        /D__SMALL__ /MX;
masm SPAWN       /D__SMALL__ /MX;
masm CVTFAK      /D__SMALL__ /MX;
masm REALCVT     /D__SMALL__ /MX;
masm SCANTOD     /D__SMALL__ /MX;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -mt *.C*
DEL ..\LIB\CS.*
lib ..\LIB\CS @CLIB.RSP
ECHO ***
ECHO *** Step 3:  Building MEDIUM Memory Model Library
ECHO ***
DEL *.OBJ
masm C0,C0M      /D__MEDIUM__ /MX;
COPY C0M.OBJ ..\LIB
masm SETARGV     /D__MEDIUM__ /MX;
masm SETENVP     /D__MEDIUM__ /MX;
masm EXEC        /D__MEDIUM__ /MX;
masm SPAWN       /D__MEDIUM__ /MX;
masm CVTFAK      /D__MEDIUM__ /MX;
masm REALCVT     /D__MEDIUM__ /MX;
masm SCANTOD     /D__MEDIUM__ /MX;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -mm *.C*
COPY ..\OBJ\*.OBJ
DEL ..\LIB\CM.*
lib ..\LIB\CM @CLIB.RSP
ECHO ***
ECHO *** Step 4:  Building COMPACT Memory Model Library
ECHO ***
DEL *.OBJ
masm C0,C0C      /D__COMPACT__ /MX;
COPY C0C.OBJ ..\LIB
masm SETARGV     /D__COMPACT__ /MX;
masm SETENVP     /D__COMPACT__ /MX;
masm EXEC        /D__COMPACT__ /MX;
masm SPAWN       /D__COMPACT__ /MX;
masm CVTFAK      /D__COMPACT__ /MX;
masm REALCVT     /D__COMPACT__ /MX;
masm SCANTOD     /D__COMPACT__ /MX;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z -c -mc *.C*
COPY ..\OBJ\*.OBJ
DEL ..\LIB\CC.*
lib ..\LIB\CC @CLIB.RSP
ECHO ***
ECHO *** Step 5:  Building LARGE Memory Model Library
ECHO ***
```

tively. These batch files appear in Listings 3 and 4. As usual, we suggest that you put them in C:\TURBOC so that they are always available. They work only on the C<x>.LIB libraries, but you could copy them and make minor modifications to have batch files for the MATH<x>.LIB libraries as well. Each one updates all five memory model C<x>.LIB libraries in the directory C:\TURBOC\RUNTIME\LIB. Each requires one argument, the base name (for example, FOO, but not FOO.C or FOO.ASM) of the source file to be changed in the library. They assume that you are running them from the directory that contains the source file and that TCC, MASM, and LIB are in the current directory or on your path.

An alternative to this approach would be to construct MAKE files that contain the instructions for all of the files in each of the source directories. We did not follow this approach because of the large number of files, but for dedicated MAKE users it probably would be well worth the trouble.

With these four batch files and our new directory structure in place, we are ready to get into the code itself. In future columns we will do just that. We will look at routines as diverse as **C0** (the start-up routine) and **printf**, probably starting with the addition of file name template, or wildcard, abilities to the standard Turbo C command line argument processor (**__SETARGV**). Until then, we encourage you to browse through the Runtime source and experiment with this exciting and useful new product. ■

Mark L. Van Name is co-founder and vice president of research and development at Foresight Computer Corp. He is also a freelance writer. Bill Catchings is a software engineer at Data General Corp and a freelance writer.

Listings may be downloaded from CompuServe as RSV1V1.ARC

```
DEL *.OBJ
masm C0,C0L      /D__LARGE__ /MX;
COPY C0L.OBJ ..\LIB
masm SETARGV     /D__LARGE__ /MX;
masm SETENVP     /D__LARGE__ /MX;
masm EXEC        /D__LARGE__ /MX;
masm SPAWN       /D__LARGE__ /MX;
masm CVTFAK      /D__LARGE__ /MX;
masm REALCVT     /D__LARGE__ /MX;
masm SCANTOD     /D__LARGE__ /MX;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z [illegible]
COPY ..\OBJ\*.OBJ
DEL ..\LIB\CL.*
lib ..\LIB\CL @CLIB.RSP
ECHO ***
ECHO *** Step 6:  Building HUGE Memory Model Library
ECHO ***
DEL *.OBJ
masm C0,C0H      /D__HUGE__ /MX;
COPY C0H.OBJ ..\LIB
masm SETARGV     /D__HUGE__ /MX;
masm SETENVP     /D__HUGE__ /MX;
masm EXEC        /D__HUGE__ /MX;
masm SPAWN       /D__HUGE__ /MX;
masm CVTFAK      /D__HUGE__ /MX;
masm REALCVT     /D__HUGE__ /MX;
masm SCANTOD     /D__HUGE__ /MX;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z [illegible]
COPY ..\OBJ\*.OBJ
DEL ..\LIB\CH.*
lib ..\LIB\CH @CLIB.RSP
ECHO ***
ECHO *** Finished Building CLIB
ECHO ***
```

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 [illegible]
lib \TURBOC\RUNTIME\LIB\CS -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z [illegible]
lib \TURBOC\RUNTIME\LIB\CM -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z [illegible]
lib \TURBOC\RUNTIME\LIB\CC -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z [illegible]
lib \TURBOC\RUNTIME\LIB\CL -+%1;
tcc -I\TURBOC\RUNTIME\INCLUDE -I\TURBOC\INCLUDE -O -Z [illegible]
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 [illegible]
ECHO *** CLIB Libraries
ECHO ***
```

CRITIQUE

QuickPak for Turbo Basic

QuickPak for Turbo Basic
Crescent Software
64 Fort Point Street
East Norwalk, CT 06855
(203) 846-2500
List Price: $69.00 ($3 s/h)

Crescent Software's QuickPak for Turbo Basic is a set of subroutines and documentation for Turbo Basic. Originally released for Microsoft's QuickBASIC, it was converted for the Turbo Basic environment by its author, Ethan Winer.

QuickPak consists of a single 5.25-inch diskette and three laser-printed, softbound manuals. The first manual is the software owner's guide, the second an assembly language tutor, and the third a collection of hints on using BASIC more effectively. All are written in a clear, humorous manner and interspersed with little cartoons: a welcome change from the very serious Microsoft manuals, but not as necessary with Borland's lighter documentation.

The assembly language tutor included as part of QuickPak is intended to help the casual or novice programmer write portions of BASIC programs in assembler, for more speed or access to the PC's DOS and BIOS services. As such, it is not a comprehensive tutorial, but more a chapter in a larger assembly language primer. With this limitation in mind, it does a good job of explaining the basics of writing assembly language subroutines.

The software shipped as part of QuickPak consists of BASIC and assembly language routines which perform some little things that enhance both QuickBASIC and Turbo Basic. These routines include file manipulation, fast sorting, video services, and getting the current directory of a drive into a string so it can be restored later.

Because Turbo Basic does not use a linker, the routines are included in source programs using the **$INCLUDE** metastatement. The dependencies between source files are not documented in the manual, so you need to inspect the source to make sure all of the required subroutines are included. Since many of the subroutines have both a BASIC "wrapper" and an assembler **$INLINE** that does the real work, care must be taken to ensure that both source files are accessible or compilation errors will occur. One way around this problem is to modify the BASIC functions to use the same directory for both the assembler and BASIC files, instead of QuickPak's default INCLUDE directory for the assembler files.

Making this modification, and working with the package in general, is easy: all of the subroutines (including the parts in assembly language) are generously commented and clearly formatted. While Winer's formatting style is not the same as Borland's examples, even beginners should have little trouble figuring out these subroutines and changing them to suit their needs. All of the source files have long commented prologues that clearly explain their calling parameters and shared variables.

Included in QuickPak are most of the pieces needed to write business applications, including input routines (text, date, and numbers), menus (Lotus-style, pulldown, and conventional), an assortment of string functions, date and time routines, and others. Some of the routines are not as robust as they might be: for example, the menu routines display only nine items and won't scroll through a display of a larger number of items.

All of the routines have been designed to minimize the necessary calling parameters. Although this makes the routines easier to use, it also means that changes must be made for more complex uses.

Many of the major subroutines have demonstration programs that can be immediately compiled and evaluated. A minor quibble is that there is no standard header file of commonly used descriptive constants that can be placed at the start of a program to simplify using the routines.

Taken together, the routines supplied in QuickPak are more topical than comprehensive. That is, they address specific needs within the Turbo Basic and QuickBASIC environment rather than providing a unified toolbox for business software developers. Some routines are far more useful than others. For example, the sort package is a fast, general-purpose quicksort suitable for business use, while the menu routines require substantial cosmetic reworking and are more an example or scaffold than a production program.

QuickPak clearly was a hasty conversion from Microsoft QuickBASIC: the Turbo Basic-applicable portions of the documentation are found only in a small addendum pamphlet. In the assembly tutor especially, these changes should have been incorporated into the text for clarity and readability. Further, given Turbo Basic's radical memory organization, most of the hints in the hint booklet are effectively worthless.

This criticism in mind, QuickPak is still a good value, especially for those needing to understand the code they use. Its clear source code is well worth examining by programmers who want to learn Turbo Basic or build their own software toolbox. Combined with a file access method, QuickPak would be a good start for business programmers who need a toolbox of routines they can easily modify. With improved, Turbo Basic-oriented documentation, QuickPak will be a very useful product. ■

— *Marty Franz*

TECH Help!

TECH Help! Version 3.2a
Flambeaux Software
1147 East Broadway, Suite 56
Glendale, CA 91205
(818) 500-0044
List price: $69.95

It's late at night, and you're putting the final touches on your magnum opus, a programming masterpiece that will assure your fame and fortune. Just a few more final touches—but you can't remember the details of a certain DOS video call. As you reach for a reference book, it happens. You knock over a cup of coffee, sending a high-caffeine river flowing into your brand new 80486 machine.

As the motherboard melts and the hard disk starts sounding like a garbage compactor, you congratulate yourself for having stashed numerous backup copies of the program around your house. At this moment, the last trickle of coffee finds its way through the floor to the basement fuse box, where it causes a short circuit. While you manage to escape the resulting fire, the entire house—including all program backups and your autographed photograph of Philippe Kahn—burns to a crisp.

Could this nightmare scenario have been avoided? With TECH Help!, that fateful reach for the reference book never would have happened. TECH Help! is a memory-resident technical reference "book" that can be popped up within applications programs. TECH Help! makes available most of the technical information that would be found in the *DOS Technical Reference Manual*, the *IBM PC/XT/AT Technical Reference Manual*, and an assembly language reference book. TECH Help! is intended primarily for programmers, who can access its [illegible] from within their programs.

[illegible] VS. MEMORY [illegible]

[illegible] used as a standalone [illegible] called from [illegible] command prompt. It [illegible] however, when used [illegible] resident pop-up [illegible] TECH Help! is [illegible] display "engine" and [illegible] material are made [illegible]. The main data [illegible] remains on disk, is [illegible] contains some 400 [illegible] in compressed [illegible]. There are two options for [illegible] Help! memory resident. [illegible] option, which takes [illegible] memory and installs it [illegible] it cannot be [illegible] of rebooting); and [illegible] which permits [illegible] be removed from [illegible] requires an additional [illegible] the /X option is [illegible] Help! must be the [illegible] resident program [illegible]

[illegible] is not copy protected [illegible] installation consists [illegible] two files to the [illegible] directory. There is a [illegible] program that need [illegible] you want to [illegible] display colors. While [illegible] can be run from [illegible] disk is recommended. [illegible] Help! requires [illegible] and a minimum [illegible]

[illegible] is popped up with [illegible] hotkey combinations. [illegible] which cannot be [illegible] Left Shift. The [illegible] specified when [illegible] installed, and is [illegible]

[illegible] activated, TECH Help! [illegible] entire screen and [illegible] main menu, which [illegible] online help plus five indices to the TECH Help! topics. The [illegible] in which TECH [illegible] is one of its [illegible] allowing several quick routes from the main menu to the specific topic you are interested in. For example, one index lists all 269 TECH Help! topics alphabetically, starting with

About DOS Functions
Access Mode / Open Mode
ANSI Console Escape Sequences

and ending with:

Video Service Directory
XT Hard Disk Ports
XT Switch Settings.

In contrast, the General Index lists topics by category. The first category is ANSI.SYS, under which the topics ANSI Console Escape Sequences and The CONFIG.SYS File are listed; the second category is ASCII, with topics ASCII and ASCIIZ, Box Line and Special Characters, Character Set Matrix, Epson/IBM Printer Control Codes, and Extended ASCII Keystrokes.

Accessing a specific topic sometimes requires moving through one or two submenus before reaching the explanatory text. The name of the current submenu or topic is displayed at the top of the screen, and navigation commands are displayed at the bottom.

In both menus and text, topic keywords are distinguished by color or, with a monochrome display, intensity. As you scroll through the text, a highlight bar moves from keyword to keyword. At any time, pressing Enter takes you to the highlighted topic. The Esc key backs you up, in order, through previous topics. This system makes it fast and easy to move around in TECH Help!. In addition, access to information is reasonably fast, even though data must be read from disk. Using an XT with a hard disk, TECH Help! screens came up almost instantaneously in most cases and never took longer than about half a second.

WELL BEHAVED IN MEMORY

In memory-resident mode, TECH Help! is well behaved, meaning that it can happily coexist with other memory-resident programs. I installed TECH Help! in all possible permutations with SuperKey and SideKick—before them, between them, and after them—and all combinations functioned properly. These programs can also be popped up in almost any order.

Thus, for example, while writing in Sprint I could pop up TECH Help! to find some information, pop up SuperKey to write a macro, and then activate the SideKick dialer. A few taps of the Esc key would then back me out of all three. The only exception I found was when SuperKey is popped up from the DOS prompt. In this situation, TECH Help! cannot be popped up on top of SuperKey. From inside an application, however, TECH Help! can be popped up on top of SuperKey. I don't know the reason behind this incompatibility, but in any event, it's minor and unlikely to be of concern to anyone. The important point is that I was never left with a locked keyboard or crashed system.

As well designed as TECH Help!'s user interface is, it would not be of much use if it presented incorrect information. Although I could not check everything, I did crosscheck several dozen items against my own knowledge and other reference sources. I did not find a single error in TECH Help!, which suggests that it can be counted on to provide accurate information.

TECH Help!'s major flaw is that it takes up the entire screen when activated and disappears completely when exited. This makes it difficult to transfer information from TECH Help! to the program or document you are editing. A variable size window that pops up away from the current cursor location, and then remains visible temporarily after TECH Help! is exited, would be an improvement. Overall, however, this is an excellent product, and I have become quickly addicted. I highly recommend TECH Help! to serious programmers or anyone else who needs fast, online access to technical information about MS-DOS and PCs. ■

—Peter Aitken

BOOKCASE

MEMORY RESIDENT UTILITIES, INTERRUPTS, AND DISK MANAGEMENT WITH MS & PC DOS

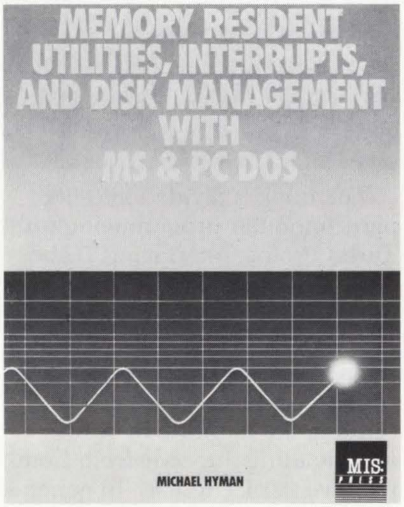
Michael Hyman, Management Information Source, Inc., Portland, OR: 1986, ISBN 0-943518-73-3, 373 pages, perfect bound, $22.95, disk $20.00

The stated purpose of this book is to provide advanced DOS programmers with a "detailed understanding of the sophisticated aspects of DOS programming." It is doubtful that any one book could completely cover such a broad topic, but for the person interested in advanced DOS subjects, Michael Hyman's attempt is well worth the price.

The book, like its title, is divided into three sections that deal with disk drives, interrupts, and memory residency. Each section is comprised of several small chapters.

Each subject is supported with example routines in either Turbo Pascal or assembly language source form. This is one of the strong points of the book: The practical nature of real-world programming is illustrated much more effectively when honest program examples are used to support the concepts under discussion. Thus, Hyman avoids the "lead them to water but don't let them drink" attitude that certain more famous programmer's guides seem to favor.

The author recommends Turbo Pascal and Microsoft Macro Assembler for compilation of the example listings. A disk containing both the source code and compiled programs is available from Princeton Software for $20.


MEMORY RESIDENT UTILITIES, INTERRUPTS, AND DISK MANAGEMENT WITH MS & PC DOS
MICHAEL HYMAN
MIS:

Section I thoroughly explains disk drives and files. Subjects include the hard disk partition record, the boot record, the file allocation table, the root directory, subdirectories, file erasure, and the recovery of erased files.

To reinforce what is learned in this section, Hyman encourages the reader to type in, compile, and run the Turbo Pascal program EXPLORER, which is a fairly powerful disk editor. As each subject is covered, modules are added or enhanced until the program can read and write disk sectors, view the boot record and root directory, change file attributes and unerase files. At the end of each lesson, the author suggest experiments with EXPLORER to enhance comprehension.

Section II covers operating system interrupts and function calls. Screen, keyboard, mouse, and light pen control are discussed, as are files and directories. The descriptions of text and graphics techniques, however, are somewhat skimpy. The author apparently recognizes this, because he recommends another of his books, *Advanced IBM PC Graphics,* in several places.

The chapter on conventional and expanded memory usage is quite good. Two example programs are included, both written in assembly: MOVE, which renames files and allows the moving of files between subdirectories without copying and erasing the originals, and DIR2, which is an enhanced directory utility similar to the DOS command DIR.

Memory resident and "pop-up" utilities are the subject of Section III: how to write them, which operating system calls to use, and what to avoid. Two assembly language programs are used to illustrate these concepts: VIDEOTBL, a small program that replaces the video parameter table and remains in memory, and PROTECT, a memory-resident utility that prevents hard disks from being formatted by DOS or BIOS calls.

Memory Resident Utilities, Interrupts, and Disk Management with MS & PC DOS assumes some knowledge of operating systems in general, and DOS in particular. Given the subject of the book, this is not a bad assumption. The book is not fancy or smooth, but it is straightforward, unpretentious, and clear. The examples given bear the mark of experience, and the warnings supplied could only come from a programmer who has successfully attempted what he claims is possible.

Michael Hyman has a good attitude and his book reflects it; I recommend it to anyone who is interested in exploring advanced topics in DOS programming. ■

—Rick Ryall

ADVANCED TURBO PROLOG PROGRAMMING

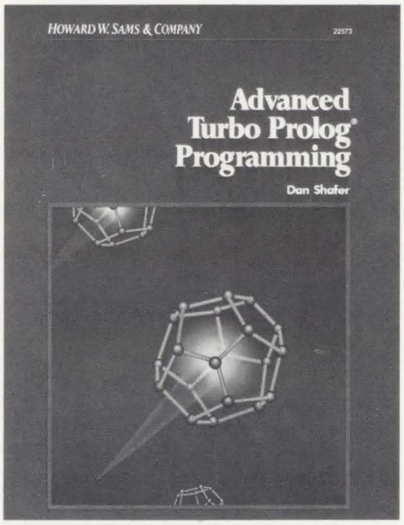
Dan Shafer, Howard W. Sams & Company, Indianapolis, IN: 1987, ISBN 0-672-22573-5, 251 pages, softcover, $19.95, Disk $10.00

Turbo Prolog is a language in search of applications. Since its birth, this unique declarative language has been tempting software developers with its AI programming features and its built-in symbolic processing capabilities. Unfortunately, very little has been written to show programmers how to develop solid applications. Most of the books written on Turbo Prolog only go as far as explaining what's in the user's manual.

Fortunately, Dan Shafer, in his latest Prolog book, has taken a different approach. Hopefully, this book is the beginning of a new wave of useful, practical, and well-designed advanced Turbo Prolog programming books.

Advanced Turbo Prolog Programming is designed for programmers who have bought Turbo Prolog, read most of the tutorial books, and are now ready to take the next step—to create real programs. The author's goal is to dispel the myth that Turbo Prolog is a toy language; thus, he emphasizes the important programming concepts and techniques and uses solid examples.

The book is unique in two respects. First, it presents important software development issues, such as modular programming, interfacing to other languages, and constructing utilities, that most Prolog books usually ignore. Second, many of the example programs are written by a variety of programmers; thus, the reader benefits from the knowledge and programming experience of other Prolog aficionados.



The book is divided into five parts: modular programming with Turbo Prolog, interfacing Turbo Prolog with the outside world, natural language processing, math and logic, and constructing useful utilities. Readers can obtain the code presented in the book by sending the author $10 or by downloading the code from CompuServe (a nice touch). In general, each part is well written and the code is thoroughly explained, even though I have a few complaints. More on that later.

WHAT CAN YOU DO WITH IT?

If you have been using Turbo Prolog and you're still asking this question, then this book might be just what you're looking for. The first part of the book is dedicated to answering the question: How can Turbo Prolog be used to develop modular programs? To answer this, the author develops a sample program, **Micro_Parse**, which performs simple analysis of a sentence. Modular programming is an important topic that most Turbo Prolog books, including the user's manual, do not cover in depth. Turbo Prolog provides unique and powerful features for modular programming such as global variables, projects, the module list, and global definitions for predicates and domains. The author presents these topics clearly and illustrates how they are used with the sample program. Also included is a section on writing and testing modules, which provides some unique insights into developing programs.

I found the second part, which presents techniques for interfacing Turbo Prolog with other languages and applications, to be the most interesting and useful. The topics covered include interfacing Turbo Prolog with C, Pascal, assembly language, and DOS; memory management; using the Turbo Prolog Toolbox serial port predicates; working with external data files; and techniques for accessing database and spreadsheet files. The basic concepts involved in interfacing Turbo Prolog with other languages are generally well presented.

However, I found the section on interfacing with C somewhat disappointing. Here the author presents a very short example of a character counting program to illustrate how Turbo Prolog can be combined with Microsoft C 4.0. Although the material is accurate, (this is the first correct published explanation that I am aware of) I think the example is too simplistic. Combining languages is always difficult and I think the reader would greatly benefit if more examples were provided.

The author does do the reader a great service by accurately showing how Turbo Prolog interfaces with assembly language. Early versions of the user's manual explained the interface incorrectly, and I'm sure any programmer who has tried to make the two languages work together will treasure this material.

Another notable chapter in the second part is the one dedicated to interfacing with DOS. This chapter explains the **bios** predicate in detail and presents the **portbyte** predicate. As a bonus, RS-232 interfaces using the Turbo Prolog Toolbox are discussed. I think the Toolbox is one of the most useful sets of programming tools that I have ever used, and I'm glad the author included some discussion about the tools.

In Part 3, the author presents an important AI application: natural language processing. This section begins with a general discussion about the fundamental issues involved in natural language processing. Topics such as conceptual dependency theory, pattern matching (ELIZA), grammars, and general parsing techniques are covered. But don't expect to find any code in this section. Chapters 13 and 14 in Part 3 do, however, present a few sample programs. Most noteworthy is the augmented transition network (ATN) example presented in chapter 14. ATNs are powerful devices for representing grammars and are used in many commercial and theoretical natural language processing programs. (The ATN program was written by a high school student.)

I have only one complaint with the section on ATNs. I think the author should have provided a more in-depth discussion about how the ATN example program works and how it can be expanded. I think the "Oh by the way, here is the code" approach is far below the quality of the rest of this book.

The last two parts, 4 and 5, are devoted to math, logic, and techniques for developing useful utilities. Some interesting logic puzzle programs are presented, including the well-known Master Mind game. To illustrate the math capabilities of Turbo Prolog, a program for performing mortgage amortization is developed. The intent of these programs is to show the reader how to solve problems with Turbo Prolog. In this respect I think the author is successful. But the chapter that impressed me the most was the last one, which illustrates how to simulate calls and pass predicates in Turbo Prolog. This is the type of material I expect to find in an advanced programming book.

CONCLUSION

In general, *Advanced Turbo Prolog Programming* is a well-written text that should appeal to all programmers who want to both improve their Turbo Prolog programming skills and learn techniques for developing useful applications. It is refreshing to read a book that uses practical examples to explain important programming issues. Most Turbo Prolog books (and even standard Prolog books) rely too heavily on the "john loves mary" examples, which needless to say, aren't helping to advance the popularity of the Prolog language. *Advanced Turbo Prolog Programming* stands out from the pack because it takes a much more serious and practical approach to the Turbo Prolog language.

I recommend that you buy two copies of this book: one to keep for your own library, and one to send to your favorite Pascal or C programmer to get the message out: Turbo Prolog is more than just a toy AI language. ■

—Keith Weiskamp

COMING UP

Floating point in Turbo C . . .

Expressing analog quantities within your program's digital universe is subtle, as anyone who has read the IEEE floating point specification can attest. Discover what's going on behind the numbers: Turbo C's floating point smarts allow your programs to use the math coprocessor when you have one, and emulate it when you don't.

Turbo C for Turbo Pascal, Turbo Prolog, and Turbo Basic people . . .

Bruce Webster explains the mindset behind the C language, and Reid Collins provides a guided tour of the Turbo C development environment and the MAKE utility.

and the TECHNIX keep coming . . .

Need some speed in your Turbo Prolog projects? Take hold of tail recursion. Michael Covington is your guide, and with his help you can call yourself and *not* be late for dinner.

Far too many people simply treat the 80286 and 80386 as fast 8088s, and make no use at all of these processors' more advanced features. Juan Jimenez explains how to let your programs determine which processor lurks at the heart of the machines they're running on. With a little help from Bruce Tonkin and Gary Entsminger, Juan provides routines for all four Turbo languages to detect the CPU type (8088/188, 8086/186, 80286 or 80386) using fully supported CPU features documented by Intel.

Bruce Tonkin provides a utility that converts assembly language .COM files to Turbo Basic's INLINE format. All our columnists will return with more practical advice.

TURBO RESOURCES

You're looking for Borland language information. Where to go? Well, for starters, right here. A 12-month free subscription to *TURBO TECHNIX* is yours for the asking when you register any of the Borland languages (including Quattro, Paradox, Eureka, and Sprint) or language toolboxes. A subscription request card is packaged with each of those products—do fill it out and return it to be sure you get every issue. If your copy of a Borland language product was shipped without the subscription request card, just write, "I would like to subscribe to *TURBO TECHNIX*," in the bottom margin of the license statement.

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* (Bantam Computer Books, New York: 1986). Howard Benner's ZAPCIS shareware utility can help you minimize connect time and automate sessions. It is available for downloading on CompuServe from DL 3 of the IBM Communications Forum (**GO IBMCOM**).

How to access the Borland Forums on CompuServe: All *TURBO TECHNIX* listings are available in all Borland forums, in DL 1 (short for Data Library 1). From the initial CompuServe prompt, type

```
GO <forum name>
```

or follow the menus. If you are not already a member of a forum, you must join by following the menus before you can download the listing files. There is no additional charge for joining a forum.

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 ARC51.EXE archiving scheme. You will need this program or one compatible with it to extract listing files from downloaded archives. The ARC51.EXE shareware program is available from the DL—please contribute to the authors.

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 NOVDEC87.ARC contains all listing archives from the November/December 1987 issue, JANFEB88.ARC for the January/February 1988 issue, and so on. The name of an article's individual listings archive file is given in the magazine at the end of the article.

To download an archive file, type

```
DOW <filename>/PROTO: XMO
```

at the DL 1 prompt. After pressing Enter, start the XMODEM receive function of your own communications program. 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 ARC51.EXE at the DOS prompt this way:

```
C>ARC51 E <filename>
```

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 in Canada and Mexico; $38.00 overseas.

TUG
P.O. Box 1510
Poulsbo, WA 98370

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. The following is a list of 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 (D.C.)
Information: (301) 652-7791
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
P.O. Box 8681
Chicago, IL 60680
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/Prolog SIG: 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 La Salle University
Philadelphia, PA 19141

SAN FRANCISCO PC USERS GROUP
Information: (415) 221-9166
3145 Geary Blvd., Suite 155
San Francisco, CA 94118

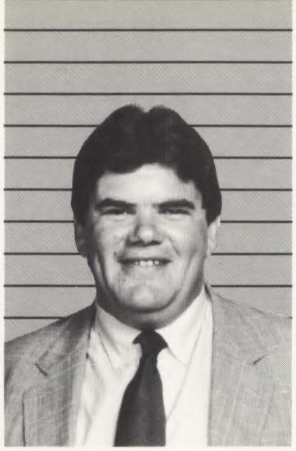
ST. LOUIS USERS GROUP
Information: (314) 968-0992
BBS: (314) 361-8662
Pascal SIG: Jeffrey Watson (314) 481-4239
C/Assembler SIG: David Rogers (314) 968-8012
BASIC SIG: Dennis Dohner (314) 351-5371

TWIN CITIES PC USER GROUP
Information: (612) 888-0557
BBS: (612) 888-0468
P.O. Box 3163
Minneapolis, MN 55403

Independent CBBS systems with programming orientation

QUESTOR PROJECT	Washington, D.C.
(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



PHILIPPE'S TURBO TALK

Fast is fun and we all hate to wait!

Philippe Kahn

In the Wild West, gunfighters were either quick on the draw or they were history. Back then, there were obvious health benefits in being faster than anyone else.

Unfortunately, in today's computer industry it's not fatally dumb to be slow, but it is a waste of your own good time when someone or something slows you down.

WHY WE ALL HATE TO WAIT

We're used to speed. We don't ride horseback, we fly. We don't wait for letters to make it past hungry dogs, we use electronic mail or faxes. We don't wait in lines at the bank, we use ATMs. We don't walk, we run. We don't like waiting—and there's no good reason today why anyone has to wait. Waiting is a waste of time.

Actually, Hemingway said something like, "Time is the thing we have the least of," and none of us wants to waste it or let others waste it for us by making us do things the hard way. But I guess not everyone gets the Hemingway message.

We design cars to do a comfortable 85 miles an hour on billion-dollar freeway systems built to handle that speed safely, then slow everyone down to 55. In this case, the technology is there, but the intelligence is not. We all love the German autobahn!

FLAWED TECHNOLOGY SLOWS YOU DOWN JUST AS SURELY AS FLAWED LOGIC

The compiler that can't compile fast enough is wasting your time. The database that can't find a record or sort your data fast enough is eating up the clock. The program that demands you take six different steps when it's possible to do them all in one keystroke is stealing time. The word processor that is so "user friendly" that it forces you to use templates in order to remember its "ergonomic design" certainly is not letting you make the best use of your time. The spreadsheet that laboriously recalculates every number it knows, instead of just the numbers that matter, is about as helpful as leg-irons to a sprinter trying to beat the 100-meter record. And, of course, the same holds for the operating system that is so "state of the art" that its promoters will go to great lengths explaining why there have to be "performance tradeoffs."

Programs like these are all time bandits. They take their time and, in the process, rob yours.

YOU CAN'T AFFORD TO OBEY PARKINSON'S LAW

Professor Northcote Parkinson's laws are funny. His "Work expands to fill the time available" law means that if people have all day to do something, they'll take all day to do it. Someone may have all day to write a letter, so they can take all day and it's no big deal.

But most of us don't have that luxury. Deadlines are stalking us, the clock is ticking and the meter is running. Time is "of the essence" and we can't afford to dawdle. We can't allow ourselves to be cast in the same mold as the woodsman with a blunt axe. He takes half a day instead of half an hour to trim a tree but then says he "hasn't got time to stop."

Slow software is the modern-day blunt axe.

WE'RE NOT HERE TO WASTE YOUR TIME

The serious (some say slavish) commitment we have to speed and to not wasting anyone's time should be apparent by now. We want to champion the concept of fast, compact and efficient programs in a world where people are saying, "It's okay if it's slow, because soon all the machines will be 24MHz 80386s; it's okay if it's big, because memory is cheap; and in any case, it'll all be solved by the new multitasking operating system."

Well, if you think about it, if it's twice as fast on a machine today, it'll be twice as fast on a future machine, too, and no matter how you look at it, memory can always be used in a smarter way than to store sloppy programs. Even if it's free!

We all want to have fun, and fast is fun. So let's not waste precious time with software that's a little slow on the draw! ■

Borland's Quattro: The New Generation Spreadsheet–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. 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.

Quattro: The New Generation Spreadsheet

	FEATURE	*QUATTRO*	*LOTUS 2.01*
SPEED	ReCalc Cash Flow Model (5K cells)	*.27*	*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.

Using macros—electronic shortcuts—is easy with Quattro

Quattro offers a complete macro debugging environment and puts you in control as you "single-step" through your macros and record them as you work.

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 new generation 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 makes your work go faster.

*60-day money-back guarantee**

QUATTRO

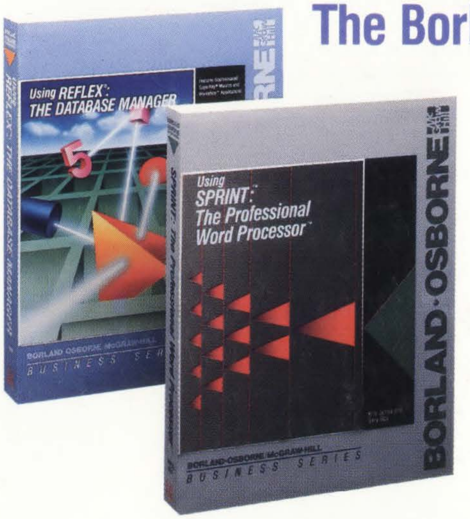
BORLAND
INTERNATIONAL

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

*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.

Announcing

Two Dynamic New Imprints


Using REFLEX: THE DATABASE MANAGER
BORLAND-OSBORNE/McGRAW-HILL BUSINESS SERIES
Using SPRINT: The Professional Word Processor
BORLAND-OSBORNE/McGRAW-HILL BUSINESS SERIES
BORLAND·OSBORNE

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
Advanced TURBO PROLOG Version 1.1
TURBO PASCAL Programmer's LIBRARY
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