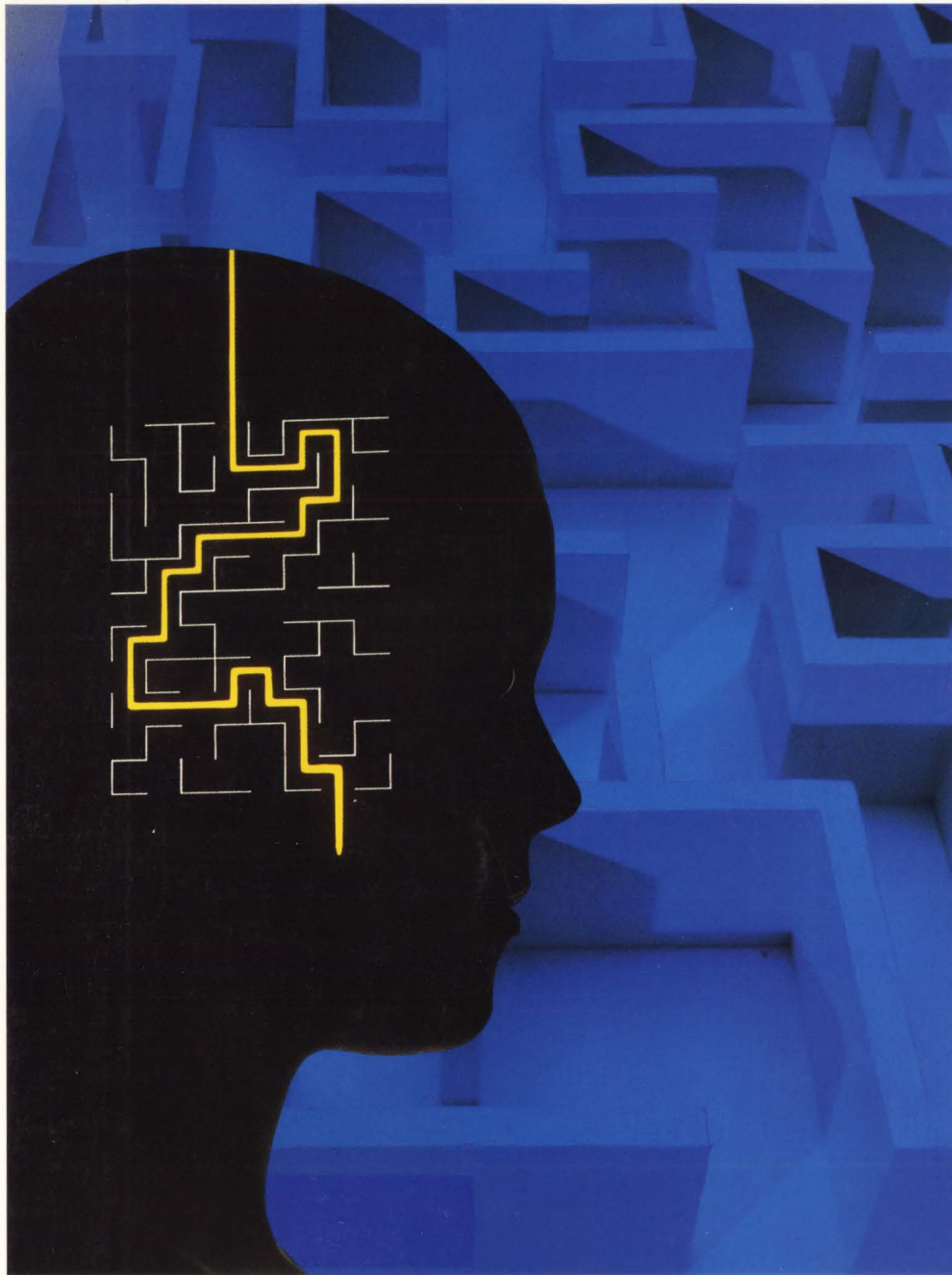


# TURBO TECHNIX

THE BORLAND LANGUAGE JOURNAL • MARCH/APRIL 1988 • VOLUME ONE NUMBER THREE • \$10.00



## EXPERT SYSTEM DESIGN WITH TURBO PROLOG

Tools for AI  
applications

Creating Turbo  
Prolog inference  
engines

Writing custom  
exit procedures  
in Turbo Pascal

Turbo Basic  
event handling

BULK RATE  
U.S. POSTAGE  
PAID  
PERMIT NO. 1452  
SEATTLE, WA



NEW DATABASE  
**OS/2 API**  
AVAILABLE NOW!

# Finally. A pro for people who h

Nobody ever said programming PCs was supposed to be easy.

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

Not any more.

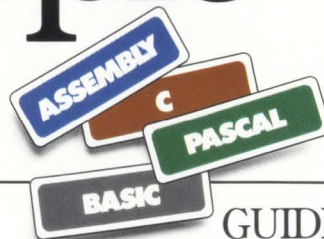
Not since the arrival of the remarkable new program in the lower right-hand corner.

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

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

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

know to program in your favorite language.



## GUIDES DATA

### Instant Access Program

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

### ASSEMBLY (600K of data)

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

### BASIC (270K each database)

- IBM BASIC, Microsoft QuickBASIC and TurboBASIC.
- Statements and Functions: Describes all statements and built-in library functions.

- Tables: Line-drawing characters, ASCII chart, keyboard codes, error codes, operators, etc.

### C (600K each database)

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

### PASCAL—Turbo (360K of data)

- Language: Describes statements, syntax, operators, data types and records.
- Library: Describes the library procedures and functions.
- Tables: ASCII chart, line-drawing characters, keyboard codes, error codes, reserved words, etc.

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

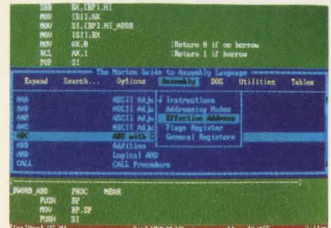
You can, of course, find most of this





# programming tool ate manual labor.

information in the books and manuals on our shelf.  
 But Peter Norton—who's written a few books himself—figured you'd rather have it on your screen.  
 In seconds.  
 In either full-screen or moveable half-



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

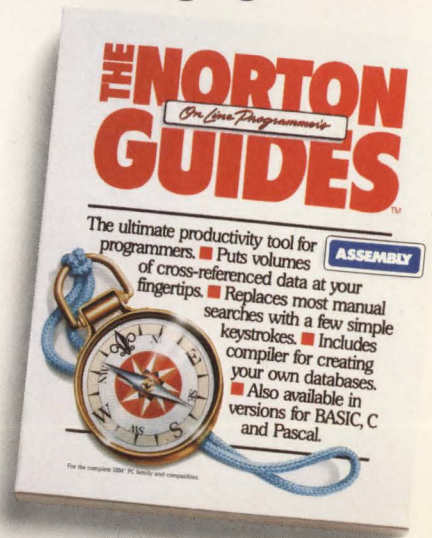


Summary data expands on command into extensive detail. And you can select from a wide variety of information.

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



# TURBO TECHNIX

The Borland Language Journal

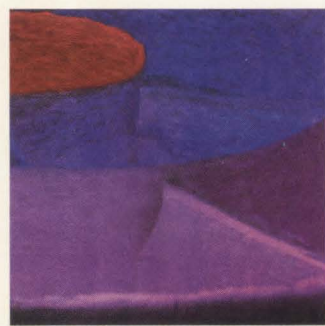
March/April 1988

Volume 1 Number 3

## FEATURES

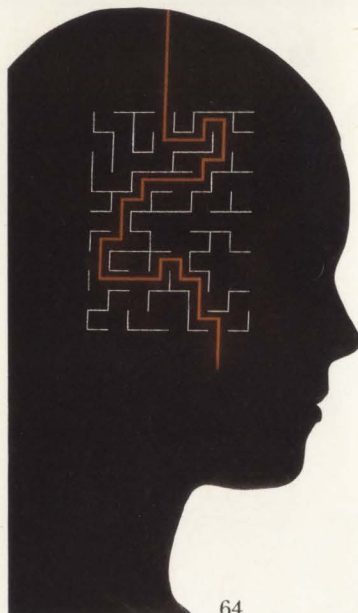
### TURBO PASCAL

- 12 Custom Exit Procedures  
*Tom Swan*
- 23 Rounded Rectangles with  
the BGI  
*Jeff Duntemann*
- 24 Just In **CASE**  
*Jeff Duntemann*
- 28 Filling Regions with the Turbo  
Pascal Graphix Toolbox  
*Fred Robinson*
- 34 Curves, Bezier-Style  
*Kent Porter*



42

The SideKick Plus Kernel acts as both supervisor and resource for resident tasks that may be supplied by Borland or written by third-party developers.



64

The first marketable concept to rise from decades of research in AI is the expert system. By making the knowledge of many experts available to all, expert systems make the power of one mind the equal of many. Building expert systems is not difficult with a little knowledge—and Turbo Prolog.

### TURBO C

- 42 The SideKick Plus API:  
Introduction  
*Jeffrey Goldberg and Steven Boye*
- 52 Making the **switch()**  
*Kent Porter*
- 56 Maintaining Programs  
with **MAKE**  
*Reid Collins*

- 61 Building Far Pointers with  
**MK\_FP**  
*Michael Abrash*

- 62 Comment Nesting  
*Roger Schlafly*

### TURBO PROLOG

- 64 Expert System Design from  
a Height  
*Michael Floyd*
- 67 Building an Inference Engine  
With Turbo Prolog  
*Keith Weiskamp*
- 80 Suitable for Framing  
*Michael Floyd*
- 89 Metalogic and Expert Systems  
*Safaa H. Hashim*



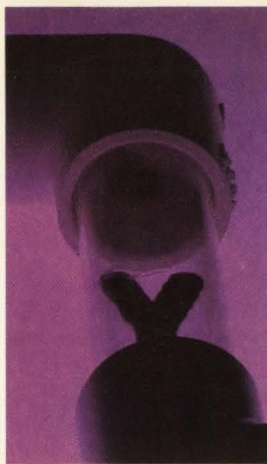
98

The assembly language screen handlers offered with the Turbo Basic Database Toolbox can put plenty of snap into a video-intensive application.

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

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





116  
The richness of a programming language depends heavily on the power of its control structures. PAL has all control structures expected in a structured language, plus one or two more that proceed from its database abilities.

## TURBO BASIC

- 98 Turbo Basic Screens at Assembler Speed  
*David A. Williams*
- 105 SELECT CASE: Choosing One From the Many  
*Ralph Roberts*
- 110 Event Trapping in Turbo Basic  
*Ralph Roberts*

## BUSINESS LANGUAGES

- 116 PAL Control Structures  
*Dan Shafer*

## DEPARTMENTS

- 4 BEGIN: The Mine and the Machine Shop  
*Jeff Duntemann*
- 6 Dialog
- 121 Binary Engineering: Preconditions and Postconditions  
*Bruce Webster*
- 134 Language Connections: Linking Turbo Prolog and Turbo Pascal 4.0  
*Peter Immarco*

- 140 Tales from the Runtime: Memory Models  
*Mark L. Van Name and Bill Catchings*
- 144 Archimedes' Notebook: Designing a Two-Band Vertical Antenna  
*Augie Hansen, KBOYH*
- 149 Critique: Turbo C Tools  
*Peter Aitken*
- 154 BookCase: Using Turbo Prolog  
*Reviewed by Sanjiva Nath*
- 155 BookCase: Advanced Techniques in Turbo Prolog  
*Reviewed by Alex Lane*
- 157 BookCase: Advanced MS-DOS  
*Reviewed by Peter Aitken*
- 158 Turbo Resources
- 159 Coming Up
- 160 Philippe's Turbo Talk

**Cover:** Our special Turbo Prolog section looks into the machinery of expert systems, which model the problem-solving abilities of human experts in many knowledge domains. Inference engines, reference frames, metalogic—we offer you some expert system expertise, and Turbo Prolog can tie it all together. Page 64.

**Cover V2.0:** We've done a little post-optimization on the TURBO TECHNIX cover. Design: Karen Miner. Photography: Bradley Ream.

## TURBO TECHNIX

*Publisher*  
Marcia Blake  
*Editor in Chief*  
Jeff Duntemann

### EDITORIAL

*Managing Editor*  
Michael Tighe  
*Technical Editor*  
Michael A. Floyd  
*Copy Editor*  
Pamela Dillehay  
*Technical Consultants*  
Brad Silverberg  
David Intersimone  
Roger Schlafly  
Dan Kernan

### DESIGN & PRODUCTION

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

### ADMINISTRATION

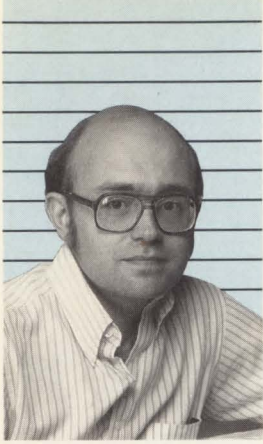
*Purchasing*  
Brad Asmus

### ADVERTISING

*Assistant to the Publisher*  
Sheriann Glass  
*Advertising Sales Manager*  
John Hemsath  
(408) 438-9321  
*Western Region*  
Janet Zamucen  
(714) 858-0408  
*New England/Mid-Atlantic Regions*  
Merrie Lynch  
Nancy Wood  
(617) 848-9306  
*South Region*  
Megan Patti  
(813) 394-4963

TURBO TECHNIX (ISSN-0893-827X) is published bimonthly by Borland Communications, a division of Borland International, Inc., 4585 Scotts Valley Drive, Scotts Valley, CA 95066. TURBO TECHNIX is a trademark of Borland International, Inc. Entire contents Copyright © 1988 Borland International, Inc. All rights reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the publisher. For a statement of our permission policy for use of listings appearing in the magazine, send a self-addressed stamped envelope to Permissions, TURBO TECHNIX, 4585 Scotts Valley Drive, Scotts Valley, CA 95066. Editorial and business offices: TURBO TECHNIX, 4585 Scotts Valley Drive, Scotts Valley, CA 95066. Subscription rate is \$49.95 per year; rate in Canada \$60.00 per year, payable in U.S. funds. Single copy price is \$10.00. For subscription service write to Subscriber Services, TURBO TECHNIX, 4585 Scotts Valley Drive, Scotts Valley, CA 95066. POSTMASTER: Send address changes to TURBO TECHNIX, 4585 Scotts Valley Drive, Scotts Valley, CA 95066.





# BEGIN

## The mine and the machine shop

Jeff Duntemann

Perhaps the question has been put to you: "If these AI systems are so smart, why haven't they cured cancer?" Do what I do: Sip your drink and say, "How 'bout them Giants?" It's true, though: The words "artificial intelligence" have taken some hits for not keeping promises they never really made. Perhaps we could use some new words. Certainly we need some new understanding.

Human intelligence has two facets: logic and reason on one hand; and deeper things that well up from the subconscious—intuition, hunches, creativity, memory—on the other. We've been studying logic and reason since classical times, and by now we know something about them. As for those other things ... not only haven't we a clue as to how they work, we may not even have a *model* for how they work. In other words, it may not be possible to understand them, given the analytical tools we have.

Those tools are logic and reason, and are essentially serial in nature. They operate by breaking down a complex whole into individual components small enough to be understood alone, while keeping track of the whole as a map of relationships and classifications. It's the best we can do, yet the analysis of the whole bears the same resemblance to the whole as a dissected frog bears to a live frog on a lily pad: It's all laid out in clear view, but it doesn't catch flies anymore.

The subconscious mind has its own tools for grasping a complex whole. Every so often, when struggling to analyze a difficult con-

cept, the light breaks through from beneath the fog and for a wondrous moment or two we simply *understand*. All the disparate aspects of the problem stand together in perfect harmony. Then the light goes out—usually as we race to keyboard or notepad to capture what we can of the moment—and what we knew we understood we only remember once understanding. A metaphor here, a relationship there, shards of a structure sketched out on paper, but no matter how much we manage to pin down, well, somehow it just doesn't catch flies anymore. Why? The subconscious mind deals only in the gestalt. Reason and logic begin by dismantling the gestalt. Understanding how we *understand* may be forever beyond us.

All the best that is human bubbles up from the subconscious in this fashion. It's pretty raw stuff, and we never quite get all of it. To make it useful, we need to apply the tools of logic and reason in the guise of philosophy, science, and engineering. By correlating hunches and drawing conclusions from fragmentary evidence, we build our civilization. That isn't to say we know where hunches come from.

Perhaps we should study neural networks. These are electrical or logical models of the interconnectedness of nerve tissue, and we have found that they excel at storing and recognizing patterns in an eerily gestalt-like manner. The problem is that even the human neural network is only the container—and attempting to reverse-engineer the human mind by

studying patterns of brain cells may be like a crew of ants blindly replicating a copy of an IBM PC in the hope that Turbo Pascal will suddenly appear in memory.

My metaphor for intelligence is the mine and the machine shop. Out of the mine comes a jumble of ore—fragments of rock containing traces of metal combined with other substances, plus the bonus of a rich vein or pure nugget from time to time. The ore itself is useless. It must pass through the smelter to become metal, and through the machine shop to become a useful artifact. The processes are distinct yet inextricable: Without the mine, the machine shop can do nothing; and without the machine shop, the ore from the mine is dead weight.

The subconscious is the mine from which we take the raw material of human thought, and logic and reason form the machine shop where our crazy notions are converted into knowledge. Artificial insight or artificial creativity may be forever beyond us, but artificial *reason* has been with us for some time. A Turbo Prolog program is just that—a tool for shaping and correlating knowledge. Later on in this issue, Technical Editor Mike Floyd shows us how such tools can be built.

AI is a partnership. Human insight and machine reason can be a potent combination. Give it a chance. Better still—give it a try. We'll help you as best we can. ■

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



# Interlocking Pieces: Blaise and Turbo Pascal.

Now, for  
Turbo Pascal 4.0!

## THE BLAISE M E N U

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TELEX NUMBER - 338139

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

**BLAISE COMPUTING INC.**

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

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



# DIALOG

## We confront some sticky issues; how could it be OTHERWISE?

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

Address letters to:

DIALOG  
TURBO TECHNIX Magazine  
4585 Scotts Valley Dr.  
Scotts Valley, CA 95066

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

### GLUE GLITCH

To The Editor:

Congratulations on the launch of a wonderful magazine. I found seven or eight items that I could use personally in the first reading. Parallel ports, the world's simplest BASIC communications program, and the like were right up my alley.

My only complaint: The mailing label did not have enough glue and almost fell off. That would have been a tragedy.

Please resist the temptation to move away from short, hard-hitting practical articles. The last thing the world needs is another place to read long, boring pontifications on computing. Keep it crisp and to the point!

A jaded computerist brought back to life.

Erik Westgard  
Itasca, IL

Never fear. We'll stay in your alley as long as we can. (The word "pontification" isn't even in our spell checker.) The glue problem is a sticky situation, however... we adhere to the philosophy that labels should stay put, so we'll apply ourselves and see if we can paste up a solution. When we do, by gum, we'll make it stick.

### THOSE SEMICOLONS AGAIN

To The Editor:

Before I dive into this one, let me say that the first issue of TURBO TECHNIX looked great! Congratulations to Jeff and the rest of the crew for a fantastic job.

In the article "Sense and Semicolons," on page 51, it says: "[Rule] 4. A semicolon immediately before ELSE is always an error."

This is true in standard Pascal. In Turbo Pascal, however, there is a very important exception: the CASE..OF statement.

In Turbo Pascal, it's perfectly legal to write:

```
CASE inputChar OF
  'A' : Aardvark;
  'B' : Balloon;
  'C' : Chimpanzee;
ELSE
  Watermelon
END; { CASE }
```

In fact, it's good practice to always place a semicolon before the ELSE clause, so that the compiler can easily locate errors in nested IF and CASE..OF statements.

But this is a relatively minor nit. All in all, the article (and the entire issue) were superb. Keep up the good work!

Brett Glass  
Palo Alto, CA

Ouch! You got me, Brett. (Thanks also to several others for pointing out this lapse in logic.) The ELSE clause in CASE..OF is an addition to ISO Pascal, and there is a competing (and in my opinion, better) keyword that does the same job: OTHERWISE. There is a definite danger in making the last case in a CASE..OF statement an IF statement with an ELSE clause. Since the reserved word OTHERWISE appears in no other part of the language, the problem of ELSE syntax ambiguity goes away. I take up the subject in more detail in "Just in CASE" in this issue; check it out. There is a movement afoot to add OTHERWISE to Turbo Pascal for the PC as an alias for CASE..OF's ELSE. Turbo Pascal for the Mac uses OTHERWISE, and this change would make two already similar compilers even more compatible. Let's hope it happens.

—Jeff Duntemann

### THROW CAUTION WHERE?

To The Editor:

Volume 1, Number 1, page 79, under the subhead, "CAUTION TO THE WIND...": "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." The IBM Printer Adapter card data outputs come from a 74LS374 IC. Upon shorting to ground, each output line's current is specified to be between 30-130 milliamps. (This is from the TI data sheet.) The data sheet contains a caution: "Not more than one output

continued on page 8



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

**SYSTEM BUILDER™ \$199.<sup>95</sup> & REPORT BUILDER™ \$179.<sup>95</sup>**  
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.)



## REPORT BUILDER CYCLE:

Key in the report parameters on screen

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

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

Press a key, wait 6 secs\*

## SYSTEM BUILDER CYCLE:

Paint the menu screens

Paint the application screens

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

Press a key, wait 6 secs\*

\*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.

"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  
Androsoft Corporation*

**Softland International, Inc.**  
320 Harris Ave., Suite A  
Sacramento, CA 95838

**(800) 654-7766**

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

Please rush me \_\_\_ copies of SYSTEM BUILDER at \$199.95 per copy and \_\_\_ copies of REPORT BUILDER at \$179.95 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. \$20 restocking fee if not returned in original condition.

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

<sup>1</sup>Trademarks of International Business Machines Corp.

<sup>2</sup>Trademark of Microsoft Corp.

<sup>3</sup>Turbo Pascal is a registered trademark of Borland International.

<sup>4</sup>dBASE is a registered trademark of Ashton-Tate.

**SOFTLAND**

International, Inc.

An affiliate company of Royal American Technologies





should be shorted at a time and the duration of the short circuit should not exceed one second."

Your experimenter is likely to ground some of the outputs. The 74LS374 can get quite hot; up to 4 watts worth if all 8 outputs are grounded when all outputs are set to 1. Is your staff prepared to repair irate readers' damaged printer boards?

Also, each data output line is bypassed to ground with a 0.0022 microfarad capacitor, which reduces radio interference, but also slows the output waveform rise time. Slow rise times can cause spurious oscillations in your experimenter's digital circuits connected to the data lines.

Stanley Logue  
San Diego, CA

Well, "caution to the wind" is probably a bit too carefree. Stanley is right—you shouldn't ground more than one of pins 2-9 on the parallel port's DB25. But for that matter you shouldn't ground any of them; there's no logical reason to ground an output pin. The bidirectional pins 1, 14, 16, and 17, on the other hand, have pull-up resistors on their outputs and may be freely shorted to ground indefinitely whether they are acting as inputs or outputs.

I've used the printer port in a number of real-world hardware test situations and it's always performed marvelously. I think it's the quickest, easiest way to control devices in the outside world.

—Bruce Eckel

## AND WHAT OF THE MAC?

To the Editor:

Never before have I felt the urge to write and congratulate a magazine on its quality, until I read *TURBO TECHNIX* from cover to cover. The material is keeping me entertained well enough so that I don't mind waiting another nine weeks until my copy of Turbo Pascal 4.0 arrives ... I figure after memorizing this issue, my 4.0 pro-

grams will be better for the wait. I can't wait for the second issue, because I'm not sure how you can improve on the first.

But what of the Macintosh, a machine whose developers cry out for this caliber of help? I read the articles on PAL, Turbo Prolog, and Turbo C. I loved them, but what I really need right now is the same lucid material covering Turbo Pascal on the Mac. I can't even find Tom Swan's book *Programming with Macintosh Turbo Pascal* in the stores.

Michael Leahy  
Columbus, OH

*Tom Swan's book is excellent—if it isn't on the shelves, order it!*

*Whether to be PC-specific was one of the toughest calls we had to make in designing TURBO TECHNIX. An underlying philosophy here is that you cannot separate a programming language from the machine environment within which it operates. Most people can master a language, but making use of machine services is complex, subtle, and almost always insufficiently documented. We need to cover machine-specific issues in great detail, but if we tried to cover both the PC and the Mac in one magazine we would end up doing justice to neither.*

*Borland is presently working on various means of providing additional support and information to Macintosh users, and we're adding letters such as yours to a query file for corporate planners' use. Mac people, let us know what you need, so we can work toward a future in which the grass grows green on both sides of the 68/86 fence.*

—Jeff Duntemann

## YOU ASKED FOR IT ...

To the Editor:

Thank you so much for *TURBO TECHNIX*. I was really impressed with the quality of the publication ... a truly outstanding job.

The entire publication was filled end to end with well-written, interesting, and useful articles. I especially enjoyed the articles by Jeff Duntemann and Michael Covington. The magazine has wonderful organization—it was really nice to see that you addressed all the major language products. The article called "Language Connections: The Turbo Prolog-Turbo C Connection" was also extremely interesting.

Jeff ... you asked in your editorial what we, as Turbo programmers, really need. Well, listed below are a few of the things that I would like to see:

1. How to interface Turbo Prolog and Turbo Pascal;
2. How to easily perform **Call(Goal)** functions within Turbo Prolog;
3. How to easily perform functor extraction operations in Turbo Prolog;
4. How to easily perform database management of large databases and knowledge bases in Turbo Prolog;
5. How to set up and utilize scripts, frames, demons, and expert system shells in Turbo Prolog;
6. AI and expert system techniques;
7. Handling graphics from within Turbo Prolog, including information on the **LoadPic** command from the Turbo Prolog Toolbox. I have tried to use this command without much luck—some help would be appreciated.

Thank you, once again, for this excellent publication.

Lindsay D. Hiebert  
Kansas City, MO

*Lindsay, we're on your wavelength; many of your wish list items are in your hand.*

*Your first question is an easy one. Peter Immarco shows precisely how to interface Turbo Prolog and Turbo Pascal 4.0 in this month's "Language Connections" column.*

*Implementing a call predicate in Turbo Prolog is not easy but it is possible. In fact, Safaa Hashim demonstrates a call in his article "Metalogic and Expert Systems" in this issue. His example is specific to an expert system design, but the technique of rewriting rules as database facts and interpreting them is a general one.*

*Functor extraction is another difficult area that requires an interpreter to solve. Essentially, you are asking how to implement functor in Turbo Prolog. The trick here is to extract the functor from a structure as it is interpreted. Ergo, first we need an interpreter. Again, "Metalogic and Expert*

*continued on page 10*



# Upgrade Your Technology

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

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

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

If you haven't yet received your copy of the Programmer's Connection Buyer's Guide, act now. Upgrading your programming technology could be one of the wisest and most profitable decisions you'll ever make.

USA ..... 800-336-1166

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

Business Hours: 8:30 AM to 8:00 PM EST Monday through Friday

Prices, Terms and Conditions are subject to change.  
Copyright 1988 Programmer's Connection Incorporated

**Sale Prices effective through 03/31/88**



## ORDERING INFORMATION

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

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

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

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

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

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

**INTERNATIONAL ORDERS.** Shipping charges for International and Canadian orders are based on the shipping carrier's standard rate. Since rates vary between carriers, please call or write for the exact cost. International orders (except Canada), please include an additional \$10 for export preparation. All payments must be made with US funds drawn on a US bank. Please include your telephone number when ordering by mail. Due to government regulations, we cannot ship to all countries.

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

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

### blaise products

	List	Ours
ASYNCH MANAGER <i>Specify C or Pascal</i> .....	175	135
C TOOLS PLUS/5.0 .....	129	99
LIGHT TOOLS <i>for Datalight C</i> ..... Sale	100	55
PASCAL TOOLS/TOOLS 2 .....	175	135
Turbo ASYNCH PLUS/4.0 .....	129	99
Turbo C TOOLS .....	129	99
Turbo POWER TOOLS PLUS/4.0 .....	129	99
VIEW MANAGER <i>Specify C or Pascal</i> .....	275	199

### Peabody Pop-Up Reference Utility by Copia International

List \$100 Ours \$89

Peabody is a fast and flexible on-line reference utility with databases available for Turbo Pascal or Microsoft C. It provides instant, accurate and complete language information in pop-up frames at the touch of a key. With Peabody, you can select general topics from a structured subject menu, or use Peabody's hyperkey to get instant help for the keyword closest to the cursor. Specify database desired. Additional databases are available for \$100 with manual or \$50 without manual.

### database management

Clipper <i>by Nantucket</i> .....	695	379
dBASE III Plus <i>by Ashton-Tate</i> .....	695	389
Fox Base Plus <i>by Fox Software</i> .....	395	249
Genifer <i>by Bytel</i> .....	395	249
R:Base 5000 <i>by Microm</i> .....	495	359
R:Base System V <i>by Microm</i> .....	700	439

### peter norton products

Advanced Norton Utilities .....	150	89
Norton Commander .....	75	55
Norton Guides <i>Specify Language</i> .....	100	65
Norton Utilities .....	100	59

### Flash-Up with FREE Mouse from Software Bottling of NY

List \$89 Ours \$79

Flash-Up is a memory-resident macro, menu and note maker compatible with most languages. Easy-to-use features include a pull-down interface and on-line help. And until 03/31/88, you'll also get a Microsoft compatible mouse FREE.

### borland products

EUREKA <i>Equation Solver</i> .....	167	105
Paradox 1.1 <i>by Ansa/Borland</i> .....	495	359
Paradox 2.0 <i>by Ansa/Borland</i> .....	725	525
Paradox Network Pack <i>by Ansa/Borland</i> .....	995	725
Quattro: The Professional Spreadsheet ..... <i>New</i>	195	125
Reflex: The Analyst .....	150	99
Sidekick .....	85	57
Superkey .....	100	64
Turbo Basic Compiler .....	100	64
Turbo Basic Database Toolbox .....	100	64
Turbo Basic Editor Toolbox .....	100	64
Turbo Basic Telecom Toolbox .....	100	64
Turbo C Compiler ( <i>Call for support products</i> ) .....	100	64
Turbo Lightning .....	100	64
Turbo Lightning Word Wizard .....	70	47
Turbo Pascal ..... <i>Sale</i>	100	59
Turbo Pascal Database Toolbox .....	100	64
Turbo Pascal Developer's Toolkit .....	395	259
Turbo Pascal Editor Toolbox .....	100	64
Turbo Pascal Gameworks Toolbox .....	100	64
Turbo Pascal Graphix Toolbox .....	100	64
Turbo Pascal Numerical Methods Toolbox .....	100	64
Turbo Pascal Tutor .....	70	41
Turbo Prolog Compiler .....	100	64
Turbo Prolog Toolbox .....	100	64

### c language

CBTREE <i>by Peacock Systems</i> ..... <i>New, Sale</i>	159	119
Essential Software Products <i>All Varieties</i> .....	CALL	CALL
Greenleaf Products <i>All Varieties</i> .....	CALL	CALL
Vitamin C <i>by Creative Programming</i> .....	225	149
VC Screen Forms Designer .....	100	79

### microsoft products

Microsoft C Compiler 5 w/CodeView .....	450	285
Microsoft COBOL Compiler with COBOL Tools .....	700	439
Microsoft Excel .....	495	319
Microsoft FORTRAN Optimizing Compiler .....	450	285
Microsoft FORTRAN for XENIX .....	695	439
Microsoft Learning DOS .....	50	38
Microsoft MACH 20 ..... <i>New</i>	495	329
Microsoft Macro Assembler .....	150	99
Microsoft Mouse <i>Specify Serial or Bus</i> .....		
<i>with Paint &amp; Mouse Menus</i> .....	150	99
<i>with Microsoft Windows &amp; Paint</i> .....	200	139
<i>with EasyCAD</i> .....	175	119
Microsoft Pascal Compiler .....	300	189
Microsoft QuickBASIC .....	99	66
Microsoft QuickC .....	99	66
Microsoft Windows .....	99	66
Microsoft Windows 386 .....	195	129
Microsoft Windows Development Kit .....	500	299
Microsoft Word .....	450	285
Microsoft Works .....	195	129

### periscope products

Periscope I <i>with Board</i> .....	345	275
Periscope II <i>with NMI Breakout Switch</i> .....	175	139
Periscope II-X <i>Software Only</i> .....	145	105
Periscope III <i>8MHz Version</i> .....	995	795
Periscope III <i>10MHz Version</i> .....	1095	875

### turbo pascal utilities

AZATAR DOS Toolkit <i>by AZATAR</i> .....	95	85
Btrieve ISAM File Mgr <i>by Novell</i> .....	245	184
DOS/BIOS & Mouse Tools <i>by Quinn-Curtis</i> .....	75	67

Flash-up <i>by Software Bottling</i> .....	89	79
Flash-up Developer's Toolbox .....	49	45
MACH 2 <i>for Turbo Pascal by MicroHelp</i> .....	69	55
MetaByte D/A Tools <i>by Quinn-Curtis</i> .....	100	89
Overlay Manager <i>by TurboPower Software</i> ..... <i>New</i>	45	39
Science & Engrg Tools <i>by Quinn-Curtis</i> .....	75	67
Screen Sculptor <i>by Software Bottling</i> .....	125	89
Speed Screen <i>by Software Bottling</i> .....	35	32
System Builder <i>by Royal American</i> .....	150	129
IMPEX Query Utility .....	100	89
Report Builder .....	130	115
TDEBUG 4.0 <i>by TurboPower Software</i> .....	45	39
Tmark <i>by Tangent Designs</i> .....	80	69
Turbo Analyst <i>by TurboPower Software</i> .....	75	59
Turbo Plus <i>by Nostradamus</i> .....	100	89
Turbo Professional 4.0 <i>TurboPower</i> ..... <i>New Version</i>	99	79
TurboHALO <i>from IMSI</i> .....	95	69
TurboPower Utilities <i>by TurboPower</i> .....	95	78
TurboRef <i>by Gracon Services</i> .....	50	35
Universal Graphics Library <i>by Quinn-Curtis</i> .....	130	119

### TurboGeometry Library by Disk Software

List \$100 Ours \$89

TurboGeometry contains over 150 routines that perform geometric calculations. Topics include: intersection of lines, arcs, circles and planes; finding coefficients of line equations, planes and circles; distance between points, lines, circles, arcs, and planes; decomposition of polygons; 2D/3D transformations; 2D/3D curve generation, vector computations; convex hull computations; and much, much more.

### other products

Brief <i>by Solution Systems</i> .....	195	CALL
Dan Bricklin's Demo II <i>by Software Garden</i> .....	195	179
Dan Bricklin's Demo Pgm <i>by Software Garden</i> .....	75	57
Dan Bricklin's Demo Tutorial <i>by Software Garden</i> .....	50	45
Instant Assistant <i>by Nostradamus</i> .....	100	89
Instant Replay III <i>by Nostradamus</i> .....	150	129
OPT-Tech Sort <i>by Opt-Tech Data Proc</i> .....	149	99
Peabody <i>by Copia Intl, Specify Language</i> ..... <i>New</i>	100	89
QBase <i>Relational Database by Crescent</i> .....	99	89
QuickPak <i>by Crescent Software</i> .....	69	59
Resident Expert <i>Specify lang by Santa Rita</i> .....	CALL	CALL

**CALL for Products Not Listed Here**



```

; CHECKPASS: PAL 2.0 procedure to check passwords and user names
; modified by Alan Zenreich -- 11/13/87 212-691-0170
PROC CheckPass()
PRIVATE
z,
zusername,
zuserpass,
znameok,
zpassword
; header contains name of proc
; variables private to this proc
; used as counter for loops
; name accepted from user
; password accepted from user
; name accepted or not
; password found in table

CURSOR OFF ; turned cursor off
PASSWORD "dontshowit" ; present password for protected table
VIEW "secrets" ; places secrets table on workspace
MOVETO [name] ; makes "name" field current
FOR z FROM 1 TO 3 ; top of FOR loop to check name
  @1,0 CLEAR EOL ; clear a space for prompt
  ?? "or press [Esc] to quit"
  @0,0 CLEAR EOL
  ?? "Enter your name: " ; prompt user for mane
  CURSOR NORMAL
  ACCEPT "A15" TO zusername ; get input
  CURSOR OFF
  CLEAR ; clear the screen
  IF NOT RETVAL THEN ; user pressed escape
    znameok=FALSE
    QUITLOOP
  ENDIF

  LOCATE zusername ; is name in the table?
  IF retval THEN ; Yes, so
    znameok=TRUE
    zpassword=[password] ; get the password
    QUITLOOP ; go on to the next step
  ELSE ; No, so
    znameok=FALSE
    BEEP ; tell the user about it
    MESSAGE "That name can't be found"
  ENDIF
ENDFOR

UNPASSWORD "dontshowit"
RESET ; clears workspace and assures that unpassword takes affect

IF znameok THEN ; name was valid
  FOR z FROM 1 TO 3 ; check for valid password
    @1,0 CLEAR EOL
    ?? "or press [Esc] to quit"
    @0,0 CLEAR EOL
    ?? "Enter your password: " ; prompt user for name
    STYLE ATTRIBUTE 0 ; black on black
    ACCEPT "A15" TO zuserpass ; get input
    STYLE ; restore normal attributes
  SWITCH
    CASE NOT RETVAL:
      RETURN FALSE ; pressed esc
    CASE zuserpass = zpassword:
      RETURN TRUE ; password is good
    OTHERWISE:
      BEEP ; password no good
      MESSAGE "Invalid password, try again"
  ENDSWITCH
ENDFOR
ENDIF
RETURN FALSE
ENDPROC ; end of procedure definition

z=Checkpass() ; allows for procedure swapping
IF NOT z THEN
  EXIT ; if not TRUE, then exit
ENDIF

```

## DIALOG

continued from page 8

*Systems*" points in the right direction. By starting with Hashim's principles, one could implement **functor** by matching with and parsing a given structure.

Management of the dynamic database is key to using Turbo Prolog and is a subject TURBO TECHNIX plans to cover well. So stay tuned!

You would like to see more on AI techniques, expert system, and knowledge representation? We couldn't be more on track in this issue. The theme is *Expert Systems and Turbo Prolog*. Knowledge representation, frames, scripts, and demons are discussed in detail in my article, "Suitable for Framing." In addition, Keith Weiskamp implements a complete expert system shell in "Building an Inference Engine in Turbo Prolog."

Handling graphics, parsing, context-sensitive help, and the like are all topics addressed in the Turbo Prolog Toolbox. We plan to cover the toolbox in great detail in coming issues.

—Mike Floyd

## CHECKPASS, TAKE TWO

Alan Zenreich was one of several people who pointed out some bugs in the PAL listing PASSWD.SC (Volume 1, Number 1, page 131). That particular version of the listing was printed in error, having been halfway through a conversion from Paradox 1.0 to Paradox 2.0. Alan was nice enough to "polish up" the **CheckPass** procedure for us, and we're printing it here. This version does require Paradox 2.0, and it improves security by clearing the workspace with a **RESET** and assuring that the password withdrawal does, in fact, take effect. In our phone conversation, Alan made a valid point: As PAL evolves, new reserved words will be added to its parser, and if you have PAL code containing identifiers identical to those new reserved words, the code will need a lot of search-and-replace work to upgrade to the new Paradox version. Alan's own solution is to preface his own identifiers with a "z" as he has done here, under the (valid) assumption that few if any reserved words are ever likely to start with "z." The improved source code may be downloaded from CompuServe as PASS2.ARC.

—Jeff Duntemann



# TURBO PASCAL AND TURBO C... MEET

## TURBOHALO

"Ideal! TurboHALO does the job comparable to packages costing \$3000 to \$4000."

**Jim Bromley**  
Superintendent of  
Spectrum Management

"TurboHALO is so fun...  
I use it to design  
programs as a hobby...It's got  
lots of ability."

**William Porter**  
Control Systems Manager

"I like the speed of  
TurboHALO...it's ten times faster  
than the competition."

**Deniz Terry**  
Doctoral Candidate

"We evaluated all of the graphic  
development packages for Turbo  
Pascal, and TurboHALO was the  
hands down winner!"

**Quinn Curtis**  
Largest New England Distributor

### It's time to put graphics into your programming.

A picture's worth a thousand words. So your programming isn't complete until you have graphics. TurboHALO brings your screen and printer to life with subroutines that draw, chart, map, and display. All with color, shape, clarity, perspective and motion. With TurboHALO, create any picture you can imagine.

### TurboHALO gives you graphics power.

TurboHALO gives you everything you need for Turbo C and Turbo Pascal graphics programming. A library of over 150 graphics subroutines. Drivers for over 42 graphics hardware devices for the IBM PC family and compatibles.

You can create the images you want, on the hardware you have!

### Fast, proven and reliable.

TurboHALO is up to ten times faster than other graphics toolkits. And with TurboHALO you get proven, reliable programming tools used by professionals for years.

### You'll like TurboHALO or your money back!

TurboHALO is available for only \$95.

You get an unconditional 45-day money-back guarantee on TurboHALO.

To order, call your dealer or IMSI at (415) 454-7101 or (800) 222-4723; in CA (800) 562-4723; in Washington DC (202) 363-9340 or in NC (919) 854-4674.

**YES,** I want to see the difference TurboHALO makes in my graphics programming! Rush me the following TurboHALO Graphics Toolkit(s) @ \$95 each plus \$3 shipping. California residents add 6% sales tax.

- TurboHALO for Turbo Pascal  
 TurboHALO for Turbo C  
 Check enclosed for \$ \_\_\_\_\_  
(made payable to IMSI)  
 Charge my credit card  
for \$ \_\_\_\_\_  VISA  MasterCard

Signature \_\_\_\_\_

Card Number \_\_\_\_\_ Expiration Date \_\_\_\_\_

Name \_\_\_\_\_

Title \_\_\_\_\_

Company \_\_\_\_\_

Street \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

**ImSi**<sup>TM</sup>

Mail to:  
IMSI, 1299 Fourth Street, San Rafael, CA 94901

# FOR GRAPHICS PROGRAMMING

TurboHALO requires 256K memory (min); memory resident drivers require 2K (Turbo Pascal only); DOS version 2.0 or higher; Borland language/compiler required. TurboHALO is a trademark of Media Cybernetics and IMSI. Turbo C and Turbo Pascal are trademarks of Borland.



# CUSTOM EXIT PROCEDURES

A Turbo Pascal program isn't over until it's over, and it isn't over until after the **END**.

Tom Swan



WIZARD

*Be kind to our web-footed friends,  
For a duck might be somebody's mother,  
who lives all alone in the swamp,  
where it's always cold and damp (dahmp).  
Now, you may think that this is the end.  
Well it is.*

—Children's parody of John Philip Sousa's *Stars and Stripes Forever*.

When a Turbo Pascal 4.0 program ends, several invisible events occur. The standard input and output files are closed. A message is displayed if a runtime error caused the program to end prematurely. A return code is passed back to DOS (or to a parent program from a child process). Changed interrupt vectors are restored to their original values, that were saved earlier by Turbo Pascal runtime routines when the program started.

In other words, to paraphrase the famous "duck" song, you may think that a Turbo Pascal program's **END**. is the end. Well, it isn't.

## LINKING INTO THE EXIT CHAIN

By following a few simple rules, you can weld your own links to the chain of events following the **END**. of a Turbo Pascal program. A custom exit procedure runs immediately before Turbo Pascal's normal exit events occur, gaining control when one of the following happens:

- The program ends normally
- A **Halt** statement is executed anywhere in the program
- An **Exit** statement is executed in the program's outer block
- A runtime error occurs

Listing 1, **ExitShell**, demonstrates how to write a custom exit procedure. When you run the program, it displays:

```
Welcome to ExitShell
Press <Enter> to end program...
Inside CustomExit procedure
```

The message, "Inside CustomExit procedure," appears after you press Enter to end the program, proving that procedure **CustomExit** runs even though the program never calls it directly. To make this happen, **ExitShell** performs these two assignments at the beginning of the program's main body:

```
savedExitProc := exitProc;
exitProc := @CustomExit;
```

The first assignment saves the value of **exitProc**, a generic **Pointer** variable defined in the **System** unit, which Turbo Pascal automatically links to every compiled program. **ExitShell**'s global variable, **savedExitProc**, holds the original **exitProc** value for the duration of the program. In your own programs, always save **exitProc** in a similar global variable. Never assign **exitProc** to a variable declared local to a procedure or function, or to a dynamic variable on the heap. The saved **exitProc** must be available after the program ends and, therefore, only a global variable will do.

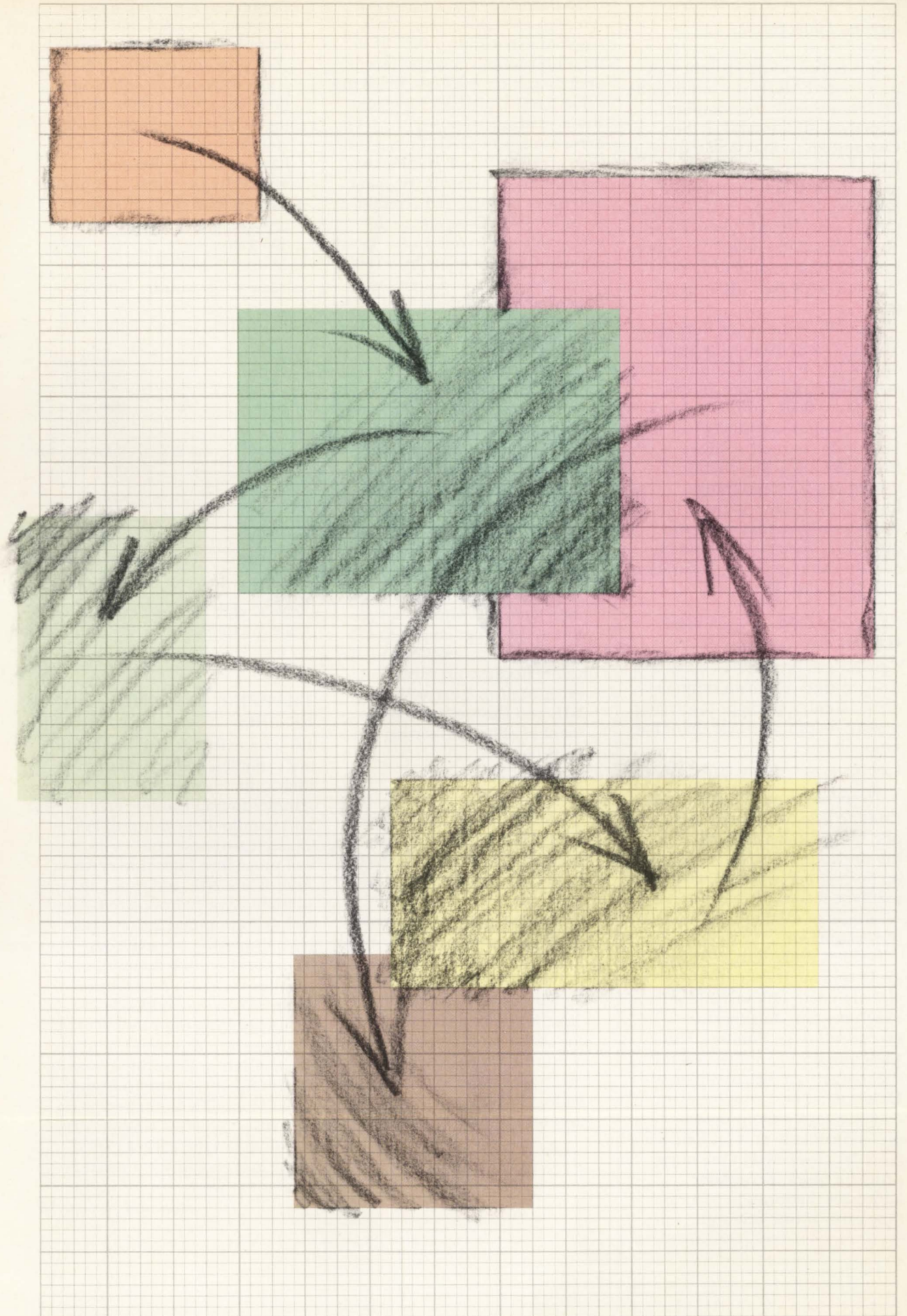
The second assignment sets **exitProc** to the address of the custom exit procedure—**CustomExit** in **ExitShell**. The **@** operator returns the address of **CustomExit**. Because **exitProc** points to the custom exit procedure, Turbo Pascal will call this procedure when the program ends.

Together, these assignments link a procedure (which can have no parameters) into the exit chain. You can name the exit procedure anything you like.

Inside the exit procedure, assign the saved pointer back to **exitProc**. This preserves the exit chain, letting other processes execute their own exit procedures after yours finishes. Except for this step, there is no limit to what you can do inside a custom exit procedure. You can read and write files, display values, use DOS functions, call other procedures and functions, and perform any other actions as part of your program's shutdown sequence.

*continued on page 14*







*continued from page 12*

It is critical to declare the exit procedure **FAR** by surrounding its declaration with the compiler directives **{\$F}** and **{\$F-}**. (See **CustomExit** in **ExitShell**.) Turbo Pascal calls the exit procedure with a **FAR CALL** instruction (technically called an *Inter-Segment Call*). Using the **{\$F}** directive tells Turbo Pascal to end the procedure with a complementary *Inter-Segment FAR RETURN* instruction.

From my experience in preparing the examples for this article, one of the most common mistakes is forgetting to declare an exit procedure **FAR**. Turbo Pascal does not prevent this mistake (which, as I painfully learned, causes a crash from which you'll probably have to reboot). If screwy things happen, or if the computer hangs when your program ends, you probably forgot to surround the custom exit procedure declaration with **{\$F}** and **{\$F-}**.

## CUSTOMIZING A RUNTIME HANDLER

Custom exit procedures make it easy to write your own runtime error handler, perhaps displaying a more helpful message than that same ol' line:

```
Runtime error 106 at 0000:001E
```

To see an example of a custom runtime handler, type in and compile Listing 2, **ErrorShell**. As in the previous program, two assignments begin **ErrorShell**, saving **exitProc** in **savedExitProc** and assigning to **exitProc** the address of **CustomExit**.

Inside **CustomExit** in Listing 2, an **IF** statement examines two global System unit variables, **exitCode** (type **Integer**) and **errorAddr** (type **Pointer**). **ExitCode** holds one of three values: the integer number passed to a **Halt** statement, a runtime error code, or zero if the program ended normally or if an **Exit** statement was executed in the outer program block. **ErrorAddr** addresses the location of a runtime error if one occurred.

To determine the meaning of a nonzero **exitCode** value, check whether **errorAddr** is **NIL**. If so, then no runtime error occurred and, therefore, **exitCode** holds the value passed to a **Halt** statement. But if **exitCode** is nonzero, and if **errorAddr** is not **NIL**, then a runtime error occurred. In this case, **errorAddr** specifies the segment and offset address of the runtime error, and **exitCode** equals the error code. (See your Turbo Pascal manual for a complete list of runtime error codes and their meanings.) To better understand how to use **exitCode** and **errorAddr**, try the following three experiments.

1. Run **ErrorShell** (Listing 2). When the program asks for an integer value, type 0 and press Enter. The zero value passed to **Halt** is assigned to **exitCode**, which **CustomExit** then examines. Consequently, **CustomExit**'s **IF** statement does not execute and, therefore, the program silently ends as though it had no custom exit procedure.
2. Run **ErrorShell** again, but this time type 100. When you press Enter, the program passes 100 to **Halt**, setting **exitCode** to that value. Because no runtime error occurred, **errorAddr** is **NIL** and **CustomExit**'s **IF** statement executes, displaying the messages:  

```
Program halted!  
Exit code = 100
```
3. Run **ErrorShell** a third time. Type ABC and press Enter. Assigning alphabetic characters to integer variable **num** causes a runtime error during the call to **Readln**, assigning to **errorAddr** the address of the instruction that caused the error and setting **exitCode** to 106, Turbo Pascal's error code for an "Invalid numeric format." Sensing that a runtime error has occurred, **CustomExit**'s **IF** statement does not execute, instead letting Turbo Pascal display its familiar runtime error message.

As you can see from these experiments, **exitCode** and **errorAddr** tell you the reason your program is ending. Table 1 lists the possible combinations of the two values. By testing **exitCode** and **errorAddr**, you can write a custom error handler to take different actions before passing control back to DOS or to the parent process that activated the program.

<b>exitCode</b>	<b>errorAddr</b>	MEANING
= 0	= NIL	normal program end
<> 0	= NIL	<b>Halt(n)</b> executed;
<> 0	<> NIL	Runtime error occurred;
		<b>exitCode</b> =error code;
		<b>errorAddr</b> =address

Table 1. **exitCode** and **errorAddr** combinations.

## TRAPPING RUNTIME ERRORS

Because **exitCode** and **errorAddr** are variables, you can change their values inside a custom error handler to trap runtime errors and handle them yourself. For example, suppose your custom exit procedure finds that **errorAddr** is not **NIL**, indicating that a runtime error has occurred. After taking appropriate action—perhaps displaying the **exitCode** value, deallocating memory, closing files, and so on—set **errorAddr** to **NIL** and **exitCode** to zero, canceling the runtime error.

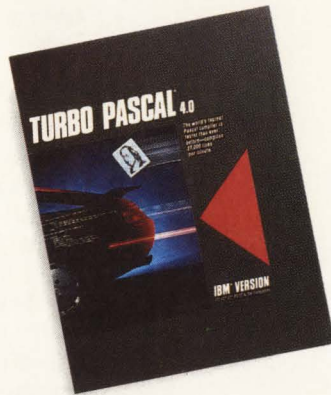
To see how this works, replace **CustomExit** in **ErrorShell** (Listing 2) with the procedure in Listing 3. Run the modified program and type ABC to force a runtime error as you did in the earlier experiment. You should see a message similar to this:

*continued on page 16*



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

**O**ur 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

## 4.0 breaks the code barrier

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

## 4.0 uses logical units for separate compilation

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

## 4.0's cursor automatically lands on any trouble spot

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

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

### Sieve (25 iterations)

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

Sieve of Eratosthenes, run on an 8MHz IBM AT

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

### Compilation of CHESS.PAS (5469 lines)

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

CHESS.PAS compiled on an 8 MHz IBM AT

**Only \$99.95**

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

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

\*Run on an 8MHz IBM AT.

\*\*If within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

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

BORLAND

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

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

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

Name \_\_\_\_\_

Ship Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_

Zip \_\_\_\_\_ Telephone ( ) \_\_\_\_\_

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

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

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

Please check box(es)

- Turbo Pascal 4.0 Compiler
- Turbo Pascal Tutor
- Turbo Pascal Database Toolbox
- Turbo Pascal Graphics Toolbox
- Turbo Pascal Editor Toolbox
- Turbo Pascal Numerical Methods Toolbox
- Turbo Pascal Gameworks

**Suggested  
Retail**

**Upgrade  
Price†**

**Serial No.**

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

Please specify diskette size  5¼"  3½"

Payment:  VISA  MC  Check  Bank Draft

Credit card expiration date: \_\_\_\_\_/\_\_\_\_\_/\_\_\_\_\_

Card # \_\_\_\_\_



```

PROGRAM ExitShell;

{ Demonstrate how to write a custom exit procedure }

VAR   savedExitProc : Pointer; { Old ExitProc value }

{$F+} PROCEDURE CustomExit; {$F-}

{ Custom exit procedure }

BEGIN
  Writeln( 'Inside CustomExit procedure' );
  exitProc := savedExitProc   { Restore saved exitProc pointer }
END; { CustomExit }

BEGIN

  savedExitProc := exitProc;   { Save ExitProc pointer }
  exitProc := @CustomExit;     { Install custom error procedure }

  Writeln;
  Writeln( 'Welcome to ExitShell' );
  Write( 'Press <Enter> to end program...' );
  Readln

END.

```

## LISTING 2: ERRSHEL.PAS

```

PROGRAM ErrorShell;

{ Demonstrate how to write a custom halt and
  runtime error handler. }

VAR   savedExitProc : Pointer; { Old ExitProc value }
      num : integer;          { Test number }

{$F+} PROCEDURE CustomExit; {$F-}

{ Custom exit and runtime error handler }

BEGIN

  IF ( exitCode <> 0 ) AND ( errorAddr = NIL ) THEN
  BEGIN
    Writeln;
    Writeln( 'Program halted!' );
    Writeln( 'Exit code = ', exitCode ) { Display halt code }
  END; { if }

  exitProc := savedExitProc   { Restore saved exitProc pointer }

END; { CustomExit }

```

## EXIT PROCEDURES

*continued from page 14*

A small problem has developed. Please jot down the following numbers and call the programmer at 555-1212.

Address = 0:749  
Code = 106

Thank you for your support!

This certainly is more friendly than Turbo Pascal's usual runtime error message, even if the last line sounds like that hick in the wine cooler commercial. The new exit procedure sets **errorAddr** to **NIL** and **exitCode** to zero, canceling the runtime error. This way, Turbo Pascal is unaware that an error occurred. If you remove these two assignments, Turbo Pascal displays its own runtime error message in addition to your custom note.

## UNITS AND EXIT PROCEDURES

Another use for custom exit procedures is to add automatic shutdown code to units. (For a tutorial on Turbo Pascal 4.0 units, see my article, "Getting To Know Units," *TURBO TECHNIQ*, November/December 1987.) Each unit that a program uses can insert its own exit procedure into the chain of events that occurs when the host program ends.

This technique opens countless doors for programmers. A memory management unit might deallocate a list of master pointers stored on the heap. A database unit might close temporary files, dumping buffered data to disk. A telecommunications unit could hang up the phone. These actions are guaranteed to occur even if the program halts prematurely due to a runtime error.

Units install custom exit procedures in the same way as Listings 1 and 2 demonstrate. In this case, though, because multiple units might install several procedures into the exit chain, it's important to understand the order in which the program and various components in the units run.

A unit can have a main body, called the *initialization section*. The statements in this section run

*continued on page 18*



# Sophisticated User Interfaces in Minutes!

Put magic in your programs with *turbo* *magic*<sup>™</sup>

*New!*  
*Version 2.0*



## The World's Best Code Generator!

---

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

---

Users and critics say it all!...

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

Sally Stott, Software Developer

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

George Kwascha, TUG Lines, Nov/Dec 87

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

Jim Powell, Computer Language, Jun 87

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

Neil Rubenking, PC Magazine, 24 Feb 87

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

Kathleen Williams, Turbo Tech Report, May/Jun 87

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

Terry Lovegrove, Library Hi Tech News, Oct 87

---

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

---

**sophisticated  
software**



6586 Old Shell Road, Mobile, AL 36608

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



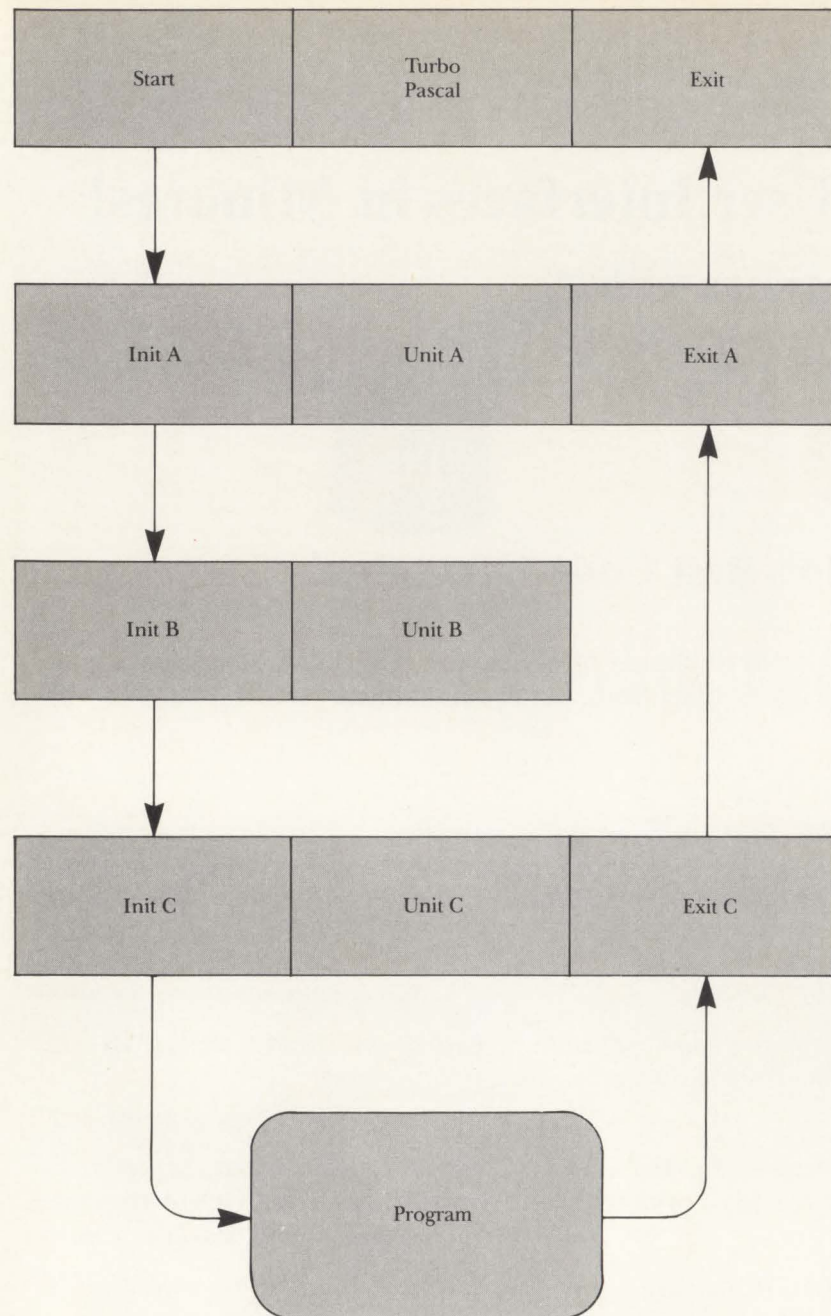


Figure 1. Multiple exit procedures in units run in the opposite order from the order the initialization sections run.

## EXIT PROCEDURES

*continued from page 17*

before the first statement in a program that uses the unit. When a program uses multiple units, the initialization sections in all units run in the same order as the unit names appear in the program's **USES** declaration. After all initialization sections finish executing, the program body statements begin running. Then, when the program ends, any exit procedures installed by the units run in the opposite order of the unit declarations in the **USES** statement. Consider a program that begins like this:

```
PROGRAM DemoExit;
```

```
USES UnitA, UnitB, UnitC;
```

The program uses three units, **UnitA**, **UnitB**, and **UnitC**. Figure 1 illustrates the order in which the unit initialization sections and exit procedures run. After Turbo Pascal completes its own startup chores, the initialization sections in **UnitA**, **UnitB**, and **UnitC** run in that order. Then, when the program ends—either normally, through **Halt** or **Exit**, or due to a runtime error—Turbo Pascal calls the custom exit procedures one by one, this time in the opposite order of the unit declarations. In the example shown in Figure 1, units A and C attach procedures to the exit chain. Unit B does not have an exit procedure and thus performs no actions when the program ends.

Listings 4 through 7 correspond with units A, B, and C in Figure 1. To run the complete example, compile **UNITA.PAS**, **UNITB.PAS**, and **UNITC.PAS** each to a TPU (Turbo Pascal Unit) disk file. Next, compile **DEMOEXIT.PAS** (Listing 7), and then run it. You should see these lines on display:

```
Inside Unit A initialization
Inside Unit B initialization
Inside Unit C initialization
Welcome to DemoExit
Inside Unit C exit procedure
Inside Unit A exit procedure
```

Several important details contribute to making this multiple-



unit example work correctly. Units A and C save **exitProc** in global **Pointer** variables declared inside each unit's **IMPLEMENTATION** section. You could declare **savedExitProc** in the unit **INTERFACES**, but because there is no reason for statements outside of the unit to use the saved pointers, it's probably best to hide **savedExitProc** variables in the **IMPLEMENTATIONS** where items are visible only to statements inside the units.

The custom exit procedures—**ExitA** and **ExitC** in Listings 4 and 6—are declared **FAR**, but are not listed in the **INTERFACE**. This makes the procedures private to the unit, and prevents the host program (or another unit) from calling exit procedures directly, a poor and possibly dangerous practice. As in Listings 1, 2, and 3, the exit procedures restore the saved **exitProc** pointers before ending, thus preserving the exit chain.

A simple experiment demonstrates what happens if you fail to preserve the exit chain when using multiple exit procedures. Remove the following statement from the **ExitC** procedure of **UnitC**:

```
exitProc := savedExitProc
```

When you run the buggy program, **UnitC**'s exit procedure runs but **UnitA**'s does not. This happens because **UnitC** breaks the exit chain by failing to restore the saved **exitProc** pointer. Therefore, the pointer to **UnitA**'s exit procedure is lost. Careful readers may realize that if **exitProc** addresses **ExitC** in **UnitC**, and if **ExitC** fails to restore the previous **exitProc** pointer, an infinite loop is the logical result. **ExitC** would end, and Turbo Pascal would repeatedly call **ExitC** as the next exit procedure in the chain.

Turbo Pascal prevents this runaway condition by setting **exitProc** to **NIL** before calling each exit procedure in the chain. When a program ends for one of the rea-

*continued on page 20*

```
BEGIN
```

```
  savedExitProc := exitProc;  { Save ExitProc pointer }
  exitProc := @CustomExit;   { Install custom error procedure }

  Writeln;
  Writeln( 'Welcome to ErrorShell' );
  Writeln;
  Write( 'Enter an integer value: ' );
  Readln( num );
  Halt( num )
```

```
END.
```

#### LISTING 3: CUSTOMEX.SRC

```
($F+) PROCEDURE CustomExit; {$F-}
```

```
BEGIN
```

```
  IF errorAddr <> NIL THEN
```

```
    BEGIN
```

```
      Writeln( '-----' );
      Writeln( 'A small problem has developed.' );
      Writeln( 'Please jot down the following numbers' );
      Writeln( 'and call the programmer at 555-1212.' );
      Writeln;
      Writeln( 'Address = ',
                seg(errorAddr^), ':', ofs(errorAddr^));
      Writeln( 'Code   = ', exitCode );
      Writeln;
      Writeln( 'Thank you for your support!' );
      Writeln( '-----' );
```

```
      errorAddr := NIL;  { Cancel runtime error }
      exitCode := 0
```

```
    END; { if }
```

```
    exitProc := savedExitProc  { Restore saved exitProc pointer }
```

```
  END; { CustomExit }
```



## LISTING 4: UNITA.PAS

```

UNIT UnitA;

INTERFACE

IMPLEMENTATION

VAR savedExitProc : Pointer; { Old exitProc pointer }

{$F+} PROCEDURE ExitA; {$F-}
BEGIN
  Writeln( 'Inside Unit A exit procedure' );
  exitProc := savedExitProc
END; { ExitA }

BEGIN
  savedExitProc := exitProc;
  exitProc := @ExitA;
  Writeln( 'Inside Unit A initialization' )
END.

```

## LISTING 5: UNITB.PAS

```

UNIT UnitB;

INTERFACE

IMPLEMENTATION

BEGIN
  Writeln( 'Inside Unit B initialization' )
END.

```

## LISTING 6: UNITC.PAS

```

UNIT UnitC;

INTERFACE

IMPLEMENTATION

VAR savedExitProc : Pointer; { Old exitProc pointer }

{$F+} PROCEDURE ExitC; {$F-}
BEGIN
  Writeln( 'Inside Unit C exit procedure' );
  exitProc := savedExitProc
END; { ExitC }

BEGIN
  savedExitProc := exitProc;
  exitProc := @ExitC;
  Writeln( 'Inside Unit C initialization' )
END.

```

## EXIT PROCEDURES

*continued from page 19*

sons listed earlier, a portion of the runtime code linked to the program executes the following loop, expressed here in Pascal-like pseudo code:

```

WHILE exitProc <> NIL DO
BEGIN
  exitProc := NIL;
  Call procedure at exitProc
END; { while }
Perform Turbo Pascal's exit chores

```

Because **exitProc** is **NIL** when **UnitC**'s modified **ExitC** procedure begins running, failing to restore **exitProc** to its saved value ends the runtime loop, causing Turbo Pascal to immediately perform its own exit chores. Knowing this technical detail about how Turbo Pascal works through the exit chain suggests a way to explicitly break the chain—just leave **exitProc** unchanged. (There's no need to set **exitProc** to **NIL**, although doing so is harmless. **ExitProc** is already **NIL** when a custom exit procedure begins.)

Breaking the exit chain is an advanced technique and you should employ it only after careful thought. There are times when the method might come in handy, though. Suppose the first of several exit procedures discovers a fatal disk error in a database system of many related units. To prevent subsequent exit procedures from writing to disk, and therefore, displaying multiple disk error messages for the identical condition, the exit procedure could break the exit chain. The program might use an **IF** statement similar to this:

```

{$I-} Close(f); {$I+}
errnum := IOResult;
IF Errnum <> 0 THEN
Writeln
  ('Fatal disk error #',errnum);
ELSE exitProc := savedExitProc;

```

If the **Close** on file **f** fails, the program displays an error message, leaving **exitProc** unchanged (equal to **NIL**) and, therefore, breaking the exit chain. Otherwise, it restores **exitProc** to its saved value, continuing with the next exit procedure in line. This way, disk errors abort the program, preventing other exit procedures from gaining control.



## DEBUGGING WITH EXIT PROCEDURES

Listing 8, **SysDebug**, shows how to use an exit procedure as a debugging device during program development. Adding **SysDebug** to a program's **USES** declaration causes several internal Turbo Pascal variables to display when the program ends. After debugging, remove **SysDebug** from the **USES** declaration and recompile to create the finished code file.

To use **SysDebug**, compile Listing 8 to disk. Next, compile Listing 9, **SDTest**. When you run **SDTest**, type 0, 100, or ABC—just as you did in the earlier experiments. In each case, the program ends by displaying a list of variables similar to the sample in Figure 2.

```
Memavail = 411440 bytes
Maxavail = 411440 bytes
PrefixSeg = $3628
CSeg = $3643
DSeg = $3767
SSeg = $378D
SPtr = $3FF6
HeapOrg = $3B8D:0000
HeapPtr = $3B8D:0000
FreePtr = $9000:0000
FreeMin = 0
HeapError = $3694:00D2
ExitProc = NIL
ExitCode = 106
ErrorAddr = $0000:009D
RandSeed = 0
FileMode = 2
Runtime error 106 at 0000:009D.
```

Figure 2. Adding the unit **SysDebug** to a program's **USES** statement provides an exit procedure that displays several variables when the program ends. The values above are from a sample run of Listing 9, **SDTest**, after typing **ABC** into a numeric variable and causing a runtime error.

There isn't room here to describe the meaning of each variable that **SysDebug** displays, although most values should be self-explanatory. Notice that because procedure **CustomExit** does not reset **errorAddr**, Turbo Pascal displays its usual message if a runtime error occurs.

When writing your own units and programs, you can use similar debugging techniques. For example, suppose you have a telecom-

*continued on page 22*

### LISTING 7: DEMOEXIT.PAS

```
PROGRAM DemoExit;
USES UnitA, UnitB, UnitC;
BEGIN
  Writeln( 'Welcome to DemoExit' )
END.
```

### LISTING 8: SYSDEBUG.PAS

```
UNIT SysDebug;

{ System globals debugging unit }

INTERFACE

IMPLEMENTATION

TYPE String15 = String[15]; { WPointer string parameter }
VAR savedExitProc : Pointer; { Old ExitProc value }

PROCEDURE WHex( v : word );
{ Display v as a 4-digit hex string }
CONST
  digits : ARRAY[ 0 .. 15 ] OF char = '0123456789ABCDEF';
BEGIN
  Write( digits[ hi( v ) div 16 ],
         digits[ hi( v ) mod 16 ],
         digits[ lo( v ) div 16 ],
         digits[ lo( v ) mod 16 ] )
END; { WHex }

PROCEDURE WPointer( description : String15; p : Pointer );
{ Display pointer value in 0000:0000 format }
BEGIN
  Write( description );
  IF p = NIL THEN Write( 'NIL' ) ELSE
  BEGIN
    Write( '$' );
    WHex( Seg(p^ ) );
    Write( ':' );
    WHex( Ofs(p^ ) )
  END; { if }
  Writeln
END; { WPointer }
```



```

PROCEDURE WWord( description : String15; w : Word );
{ Display word value in 0000 hex format }
BEGIN
  Write( description, '$' );
  WHex( w );
  Writeln
END; { WWord }

{$F+} PROCEDURE CustomExit; {$F-}

{ Display system global variables at exit }

BEGIN
  Writeln;
  Writeln( 'System unit global variables' );
  Writeln( '-----' );
  Writeln( 'Memavail = ', memavail, ' bytes' );
  Writeln( 'Maxavail = ', maxavail, ' bytes' );
  WWord( 'PrefixSeg = ', prefixSeg );
  WWord( 'CSeg = ', cSeg );
  WWord( 'DSeg = ', dSeg );
  WWord( 'SSeg = ', sSeg );
  WWord( 'SPtr = ', sPtr );
  WPointer( 'HeapOrg = ', heapOrg );
  WPointer( 'HeapPtr = ', heapPtr );
  WPointer( 'FreePtr = ', freePtr );
  Writeln( 'FreeMin = ', freeMin );
  WPointer( 'HeapError = ', heapError );
  WPointer( 'ExitProc = ', exitProc );
  Writeln( 'ExitCode = ', exitCode );
  WPointer( 'ErrorAddr = ', errorAddr );
  Writeln( 'RandSeed = ', randSeed );
  Writeln( 'FileMode = ', fileMode );

  exitProc := savedExitProc { Restore saved exitProc pointer }

END; { CustomExit }

BEGIN
  savedExitProc := exitProc; { Save ExitProc pointer }
  exitProc := @CustomExit { Install custom exit procedure }
END.

```

## LISTING 9: SDTEST.PAS

```

PROGRAM SDTest;

{ Test SysDebug unit }

USES SysDebug;

VAR num : integer;

BEGIN
  Writeln( 'Welcome to SysDebug unit test' );
  Writeln;
  Write( 'Type an exit code : ' );
  Readln( num );
  Halt( num )
END.

```

munications unit named **Modem**. You could write a separate unit, **ModemDebug**, to dump **Modem**'s global variables after a test program run. Isolating the debugging statements in a separate unit's exit procedure makes it easy to remove the debugging when you're ready to compile the production program. Later, if problems develop, you can quickly add the debugging statements by compiling after reinserting **ModemDebug** in the program's **USES** declaration.

## SUMMING UP

Custom exit procedures give Turbo Pascal programmers the ability to forge new links in the normal chain of events that take place when programs end. When programming your own custom exit procedures, remember to follow these four critical steps:

1. Save **exitProc** in a global **Pointer** variable.
2. Assign to **exitProc** the address of your custom exit procedure.
3. Declare the custom exit procedure **FAR** by surrounding its declaration with **{\$F}** and **{\$F-}** compiler directives.
4. Restore the saved **exitProc** pointer from the value saved in step 1, and do it before leaving the custom exit procedure.

As the examples in this article demonstrate, exit procedures can inspect **Halt** statement codes, handle runtime errors, perform unit deinitialization sequences, and help with debugging. And if you think that *this* is the end—well, this time, it is. ■

*Tom Swan is the author of several books, including Mastering Turbo Pascal, Mastering Turbo Pascal Files (Howard W. Sams), and Programming With Macintosh Turbo Pascal (John Wiley & Sons). Tom is currently revising Mastering Turbo Pascal for version 4.0, scheduled for publication early in 1988.*

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



# ROUNDED RECTANGLES WITH THE BGI

Take the edge off your graphics windows with a little BGI cleverness.

Jeff Duntemann



PROGRAMMER

Corners can be crass. In a highly polished graphics windowing application, a softer touch around the edges can add much to the professional feel of the user interface. Using the Borland Graphics Interface (BGI) to create rectangles with rounded corners is relatively easy: Draw four 90-degree circular arcs for the corners, and then connect the endpoints of the arcs with straight lines.

This would be trivial, except ... BGI arcs are specified by an arc center and a radius, not by endpoints. Calculating the endpoints of an arc from its center and radius involves considerable floating point math, and would take more time than you might want to spend within a graphics drawing routine.

Fortunately, the BGI calculates the coordinates of the endpoints as part of its assembly language arc-drawing algorithm, and these coordinates may be returned to your programs through a procedure called **GetArcCoords**. **GetArcCoords** returns the arc's endpoint coordinates in a record of type **ArcCoordsType** (Listing 1), defined in the interface section of the BGI's **Graph** unit.

The procedure **RoundedRectangle** (Listing 2) requires these parameters—the coordinates of the upper left corner, the width and height of the rectangle, and the radius of the corners. **RoundedRectangle** first draws the four arcs at the corners, returning the arc coordinates each time through a call to **GetArcCoords**. Then it draws the four connecting lines using those coordinates. No "absurdity-checking" is done within **RoundedRectangle**. If you specify an arc radius of 100 for the corners when your width and height are 25 pixels each, the BGI will draw something, but it might not resemble anything close to a rectangle. The value you use for the corner radius will depend on your needs and on the resolution of your graphics device. Try a few values (10 often works well) and use what looks good.

Looking good is, after all, what graphics are for. ■

Listings may be downloaded from CompuServe as RRECT.ARC.

## LISTING 1: ARCTYPE.SRC

```
ArcCoordsType = RECORD
    X, Y           : Integer;
    Xstart, Ystart : Integer;
    Xend, Yend     : Integer
END;
```

## LISTING 2: ROUNDRECT.SRC

```
(->>>RoundedRectangle<<<-----)
( )
( Filename : ROUNDRECT.SRC -- Last Modified 1/23/88 )
( )
(           by Jeff Duntemann )
(           Turbo Pascal V4.0 )
( )
( This routine draws a rectangle at X,Y; Width pixels wide and )
( Height pixels high; with rounded corners of radius R. )
( )
( The Graph unit must be USED for this procedure to compile. )
( )
( From COMPLETE TURBO PASCAL, Third Edition, by Jeff Duntemann )
( Scott, Foresman & Co. 1988 ISBN 0-673-38355-5 )
(-----)

PROCEDURE RoundedRectangle(X,Y,Width,Height,R : Word);
VAR
    ULData,LLData,LRData,URData : ArcCoordsType;
BEGIN
    ( First we draw each corner arc and save its coordinates: )
    Arc(X+R,Y+R,90,180,R);
    GetArcCoords(ULData);
    Arc(X+R,Y+Height-R,180,270,R);
    GetArcCoords(LLData);
    Arc(X+Width-R,Y+Height-R,270,360,R);
    GetArcCoords(LRData);
    Arc(X+Width-R,Y+R,0,90,R);
    GetArcCoords(URData);
    ( Next we draw the four connecting lines: )
    Line(ULData.XEnd,ULData.YEnd,LLData.XStart,LLData.YStart);
    Line(LLData.XEnd,LLData.YEnd,LRData.XStart,LRData.YStart);
    Line(LRData.XEnd,LRData.YEnd,URData.XStart,URData.YStart);
    Line(URData.XEnd,URData.YEnd,ULData.XStart,ULData.YStart);
END;
```



# JUST IN CASE

Choose one of many (or None of the Above) with Turbo Pascal's CASE..OF statement.

Jeff Duntemann

Far too many years ago there was a movie called *If It's Tuesday, This Must Be Belgium*. The movie was about the sort of American tourist who collects European countries like baseball cards, and to whom a seven-countries-in-seven-days tour is the only way to travel. The slightly twisted logic of the movie's title always brings to mind Pascal's CASE..OF statement. Let's take a look at CASE..OF, and let's start by playing tourist with the IF statement.

## IF..ELSE..IF

The star of *If It's Tuesday, This Must Be Belgium* had absolute faith in his itinerary. If lapheld portables had been available, he could have written a program to tell him where he was based on the system clock. The program would query the system clock and return a value in an enumerated type giving the day of the week:

```
TYPE
  Day = (Monday, Tuesday, Wednesday,
         Thursday, Friday, Saturday, Sunday);
```

```
VAR
  Today : Day;
```

A function named `GetDayOfWeek` would return a value of type `Day` depending on the day as known by the system clock:

```
Today := GetDayOfWeek;
```

Once the day of the week was known, a series of IF statements would decide which country the tourist was in:

```
IF Today = Monday
  THEN Country := 'Luxembourg';
IF Today = Tuesday
  THEN Country := 'Belgium';
IF Today = Wednesday
  THEN Country := 'Netherlands';
IF Today = Thursday
  THEN Country := 'France';
IF Today = Friday
  THEN Country := 'Austria';
```

```
IF Today = Saturday
  THEN Country := 'West Germany';
IF Today = Sunday
  THEN Country := 'Switzerland';
```

Here, we have a separate IF statement for each day of the week. This works, but it's far from the best way to go. Why? Every one of the seven IF statements would *always* have to be evaluated, even if by chance the day happened to be **Monday**.

Pascal offers a much better way, by allowing IF statements to be nested. This is done by making each of the seven IF statements (except the first, of course) part of the ELSE clause of the previous IF statement. The whole structure is shown in Figure 1. It reads much the way it works out logically: If it's Monday, this must be Luxembourg; else if it's Tuesday, this must be Belgium; else if it's Wednesday, this must be the Netherlands, and so on.

```
IF Today = Monday THEN
  Country := 'Luxembourg'
ELSE IF Today = Tuesday THEN
  Country := 'Belgium'
ELSE IF Today = Wednesday THEN
  Country := 'Netherlands'
ELSE IF Today = Thursday THEN
  Country := 'France'
ELSE IF Today = Friday THEN
  Country := 'Austria'
ELSE IF Today = Saturday THEN
  Country := 'West Germany'
ELSE IF Today = Sunday THEN
  Country := 'Switzerland';
```

Figure 1. Nested IF statements.



From a program execution standpoint, control leaves the nested **IF** statement as soon as one of the Boolean expressions becomes **True**. In other words, if it's Monday, the assignment statement

```
Country := 'Luxembourg'
```

is executed, and the **IF** statement terminates. Only if the day is Sunday are all seven tests actually performed.

### ENTER CASE..OF

The big **IF** statement shown in Figure 1 is perfectly good Pascal. However, you don't need anything that convoluted. Pascal provides an entirely separate control structure to deal with situations exactly like this: the **CASE..OF** statement.

A **CASE..OF** statement is like an *n*-way switch for program flow. A single value, called a *case selector*, is tested for one of several values. For each one of those values, program flow can take a different path.

The result is very much like the nested **IF** statement, but the **IF..THEN..ELSE** work is handled beneath the surface. All of the equivalent logic to the nested **IF** statement is contained in the **CASE..OF** statement shown in Figure 2. This is both smaller and simpler than the **IF** statement. In Figure 2, the variable **Today** acts as the case selector. Turbo Pascal looks at **Today's** value and compares it to the enumerated constants to the left of the colons. These constants are called *case labels*, and there is one for each possible value of **Today**. As an example, if **Today** is tested and found to be equal to the enumerated constant **Tuesday**, the statement

```
Country := 'Belgium';
```

is executed. Then, the **CASE..OF** statement is finished, and it passes control on to the next Pascal statement in the program.

There may be as many as 256 case labels in a single **CASE..OF** statement. Case labels must be *constants*. They may not be variables. The enumerated constants acting as case labels in Figure 2 are in their declaration order for clarity's sake. They can be in any order, however. You may hear

other programmers say that the most frequently occurring labels should be near the top of the statement to maximize performance, but I recommend against that, if it obscures the meaning of the **CASE..OF** statement. The increase in performance is so slight as to be unnoticeable, unless the **CASE..OF** statement is in the middle of a tight loop. Tight loops are a poor place for something as elaborate as a **CASE..OF** statement for performance reasons.

In Figure 2, there is only one enumerated constant in each case label. That is not a requirement. There may be many constants in each case label, separated by commas. Also, a *closed interval* may act as a case label. In other words, a sequence like '**A'..'D**' is equivalent to the list of constants '**A**', '**B**', '**C**', '**D**'. If any one of those constants in a given case label is found to

be equal to the value of the case selector, the statement associated with that case label is executed.

Consider Figure 3. Here, an enumerated type called **State** contains a constant representing each of the United States. (Note that the customary abbreviations **IN**, for Indiana, and **OR**, for Oregon, cannot be legal constants because both are reserved words in Turbo Pascal.) The **CASE..OF** statement below the enumerated type definition is from an imaginary mail-management program that tallies incoming pieces of mail by several geographic regions of the United States. Because there are several states in each geographic region, there are several enumerated constants in each case label. Each time a piece of mail in one of the regions is detected, a counter for that region is incremented.

In both of our examples, enumerated constants have been used

*continued on page 26*

#### CASE Today OF

```
Monday   : Country := 'Luxembourg';
Tuesday  : Country := 'Belgium';
Wednesday : Country := 'Netherlands';
Thursday : Country := 'France';
Friday   : Country := 'Austria';
Saturday : Country := 'West Germany';
Sunday   : Country := 'Switzerland';
```

```
END; { CASE }
```

Figure 2. A simple **CASE..OF** statement.

#### TYPE

```
{ II = Indiana; OG = Oregon to avoid reserved word conflict }
State = (AK,AL,AR,AZ,CA,CO,CT,DE,DC,FL,GA,HI,IA,ID,IL,II,KS,
        KY,LA,MA,MD,ME,MI,MN,MO,MS,MT,NE,NV,NH,NJ,NM,NY,NC,
        ND,OH,OK,OG,PA,RI,SC,SD,TN,TX,UT,VA,VT,WA,WI,WV,WY);
```

#### VAR

```
FromState : State;
```

#### CASE FromState OF

```
CT,MA,ME,NH,
RI,VT           : CountNewEngland := CountNewEngland + 1;
DC,DE,MD,NJ,
NY,PA           : CountMidAtlantic := CountMidAtlantic + 1;
FL,GA,NC,SC     : CountSoutheast := CountSoutheast + 1;
IA,IL,II,MI,
MN,OH,WI,WV     : CountMidwest := CountMidwest + 1;
AL,AR,KY,LA,
MO,MS,TN,VA     : CountSouth := CountSouth + 1;
KS,ND,NE,SD,
WY              : CountPlains := CountPlains + 1;
AK,CA,CO,HI,
ID,MT,OG,UT,
WA              : CountWest := CountWest + 1;
AZ,NM,NV,OK,
TX              : CountSouthwest := CountSouthwest + 1;
```

```
END; { CASE }
```

Figure 3. Multiple constants in case labels.



as case labels. Actually, constants of any ordinal type may be used as a case label, including **Char**, all integer numeric types (**Integer**, **Word**, **LongInt**, **Byte**, and **ShortInt**), **Boolean**, and any enumerated type. Real number constants, set constants, record constants, and array constants may *not* act as case labels.

## NONE OF THE ABOVE

In Standard Pascal, that is about all there is to know about **CASE..OF**. Most of you are probably aware of the truck-sized hole left in Standard Pascal's **CASE..OF** structure: the case in which the case selector matches *none* of the case labels.

Standard Pascal calls this situation a runtime error. The programmer is supposed to test the variable acting as the case selector *before* the **CASE..OF** statement is executed, to be sure that all possible values of the case selector are covered by a case label. To me this is silly; the **CASE..OF** statement is a testing statement, and it is perfectly capable of acting responsibly if one of the case selector values does not have a corresponding case label. The correct thing to do is simply to "fall through"; in other words, the **CASE..OF** statement takes no action in such a "none of the above" case. Most Standard Pascal implementations work that way.

Turbo Pascal takes it even further. In Turbo Pascal, a **CASE..OF** statement may take an optional **ELSE** clause that executes whenever the current value of the case selector does not match any of the case labels. Figure 4 is a simple example. Here, a hypothetical engine-monitoring system from a slightly futuristic vehicle detects a problem in the engine and displays a message on a status panel corresponding to the detected problem code. If a problem code is generated for which

```

VAR
  ProblemCode : Byte;

Beep;
Writeln('*****WARNING!*****');
CASE ProblemCode OF
  1 : Writeln(' [001] Fuel Supply has fallen below 10%');
  2 : Writeln(' [002] Oil pressure is below min spec');
  3 : Writeln(' [003] Engine temperature is too high');
  4 : Writeln(' [004] Battery voltage is below min spec');
  5 : Writeln(' [005] Brake fluid level is below min spec');
  6 : Writeln(' [006] Coolant level is below min spec');
  7 : Writeln(' [007] Transmission fluid level is below min spec');
ELSE
  BEGIN
    Writeln(' [***] Logic failure in problem reporting system. ');
    Writeln('      Please contact Studebaker as soon as possible!')
  END
END; { CASE }

```

Figure 4. The **ELSE** clause in a **CASE..OF** statement.

there is no known message, the system politely suggests that the driver contact the manufacturer immediately before the engine melts down. (One would hope the vehicle is equipped with a cellular phone.)

This is a good place to point out that the statement executed for any of the case labels or for the **ELSE** clause may be *any* legal Pascal statement, including (for example), a compound statement bracketed by **BEGIN** and **END**, another **CASE..OF** statement, a **FOR** loop, or an **IF** statement. This being true, another problem arises—ownership of **ELSE** clauses.

## WHOSE ELSE IS THIS, ANYWAY?

In my article "Sense and Semicolons" (*TURBO TECHNIX* November/December, 1987), I failed to point out an ugly trap that can be triggered by not paying attention to semicolon placement. The **CASE..OF** statement in Figure 5 has, for the last case label, an **IF** statement with an **ELSE** clause. The **CASE..OF** statement itself, however, has no **ELSE** clause. Think about what would happen if you placed a semicolon after the procedure invocation:

```
ExecuteCommand(IncomingCharacter)
```

In effect, this cuts loose the **IF** statement's **ELSE** clause. Normally, a freestanding **ELSE** clause

would be caught by the compiler as an error. However, because the **CASE..OF** statement does not have an **ELSE** clause of its own, the **CASE..OF** statement gladly appropriates the freestanding **ELSE** clause for itself. The code in Figure 5 will still compile, but now, the statement

```
PunctuationComing := True
```

has become the **CASE..OF** statement's **ELSE** clause, and will be executed every time a character comes in for which the **CASE..OF** statement has no other label. This situation is *not* equivalent to the original logic of Figure 5's **CASE..OF** statement, which simply falls through when it encounters unrecognized characters. A bug has been created that could take some considerable headscratching to resolve.

The problem comes down to this: Depending on the presence or absence of that single semicolon, an **ELSE** clause may belong to the **CASE..OF** statement or to an **IF** statement associated with the last case label in the list of case labels. The **ELSE** clause may legally belong to either, but the logic of the **CASE..OF** statement is radically different in either case.

Some implementations of Pascal avoid this whole conflict by using the reserved word **OTHERWISE** for the **CASE..OF** statement's **ELSE** clause. For Turbo



Pascal, we must work smart and make sure that all semicolons are wanted and are placed where they belong.

### NOT A HARD CASE

A formal summary of the **CASE..OF** statement's syntax is given in Figure 6. Here are some things to remember:

- The case selector must be a variable of some ordinal type or of a subrange of an ordinal type. It may be an array element, pointer referent, or record field as long as it is still of some ordinal type.
- The case label may be a single constant, a list of constants separated by commas, or a closed interval of constants. These constants, of course, must be of an ordinal type or subrange type identical to that of the case selector.
- There can be no duplication of values among the case labels. In other words, if the constant 6 is present in one case label, it cannot be present in any other case label.
- The case labels can be in any order, but it is customary to place them in their own collating order; i.e., 'A' before 'B' before 'C', and so on.
- Statements attached to case labels or to the **ELSE** clause may be any legal Pascal statement.
- There may be only 256 case labels in any single **CASE..OF** statement.
- Make sure that any semicolon immediately ahead of the **CASE..OF** statement's **ELSE** clause really belongs there.

Quite apart from remembering these technical points, keep the *logic* of the **CASE..OF** statement in mind as you code. The idea is to choose one path among many, based on a selector value. If you know your Turbo Pascal, and if your itinerary is sound, you'll always know where you are. ■

```

CONST
CR = ^M;
LF = ^J;

CASE IncomingCharacter OF
'A'..'Z' : CapComing := True;
'a'..'z' : LowerCaseComing := True;
'1'..'0' : DigitComing := True;
CR      : EndOfLineComing := True;
LF      : EndOfLine := True;
':!'..'@' : IF CommandMode THEN ExecuteCommand(IncomingCharacter);
          ELSE PunctuationComing := True
END; { CASE }

```

Figure 5. One semicolon makes all the difference in the world.

```

CASE <case selector> OF
<constant list 1> : <statement 1>;
<constant list 2> : <statement 2>;
<constant list 3> : <statement 3>;
<constant list 4> : <statement 4>;

. . . .

<constant list n> : <statement n>;
ELSE <statement>
END;

```

Figure 6. Formal definition of the **CASE..OF** statement.

**New!** Hire a Pro for  
Your New Turbo 4.0

Turn on the power of Turbo PROFESSIONAL 4.0, a library of more than 300 state-of-the-art routines optimized for Turbo Pascal 4.0. You'll have professional quality programs finished faster and easier.

Turbo PROFESSIONAL 4.0 includes complete source code, comprehensive documentation and demo programs that are powerful and useful. The routines include:

- Pop-up resident routines
- BCD arithmetic
- Virtual windows and menus
- EMS and extended memory access
- Long strings, large arrays, macros, and much more.

**Turbo PROFESSIONAL is only \$99.**  
Call toll free for credit card orders.  
**1-800-538-8157** extension 830  
1-800-672-3470 extension 830 in CA

**Satisfaction Guaranteed** or your money back within 30 days.



Turbo Pascal 4.0 is required. Registered owners of Turbo Professional by Sunny Hill Software may upgrade for \$30. Include your serial number.

For other information call 408-438-8608, 9 AM to 5 PM PST. Shipping & taxes prepaid for US and Canadian customers, others please add \$6 per item.

**TURBO**  
*Power*

TurboPower Software 3109 Scotts Valley Dr., Suite 122 Scotts Valley, CA 95066



# FILLING REGIONS WITH THE TURBO PASCAL GRAPHIX TOOLBOX

Pump pixels into the empty spaces on your screen, no matter what shape they are.

Fred Robinson

Filling rectangular regions on a graphics screen is so simple that it can be done purely by calculation. In other words, given the coordinates of two diagonal corners of a rectangular region, the addresses of all interior points can be quickly and accurately derived mathematically. Calculating the interior points of irregular regions is almost impossible. The best way to fill nonrectangular regions is to examine the screen buffer point by point to see what is filled and what is not, stopping when boundaries are encountered.

A generalized region-filling routine is therefore a handy thing to have. With very little code, such a routine can be added to the Turbo Pascal Graphix Toolbox 4.0. I have written two implementations of a region filler, both of which are described in this article.

## A FEW DEFINITIONS

What we need is a means of setting or resetting a continuous area of pixels, bounded by pixels of the same color or by the edge of the display. By continuous, I mean pixels that are touching at their edges, not merely at their corners, as shown in Figure 1.

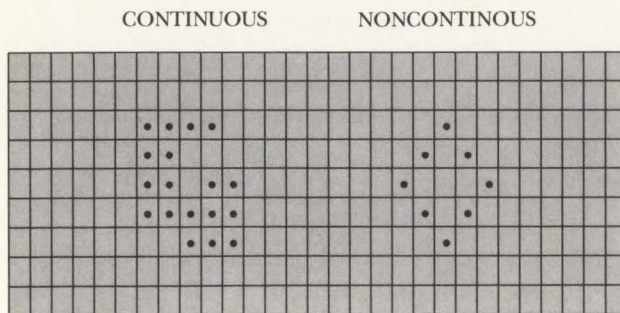


Figure 1. Continuous and noncontinuous pixels.

In contrast to the area to be filled, the *border* that encloses the area may be like the noncontinuous image in Figure 1. In other words, its pixels may be connected either at the edges or at the corners. However, if even a one-pixel gap exists, the filling will spill out of its intended border and fill adjacent areas.

In the discussion that follows, a *background* pixel is a pixel that is part of the region to be filled (for instance, an unlit pixel), and is represented in the figures by a blank space. A *foreground* pixel is the same color as the border and is represented by a small black circle. In the Turbo Pascal Graphix Toolbox, you are limited to only two different colors on the screen at one time, so the border color must always be the same as the color that fills the region.

## FILL 'ER UP

One way to implement a region filler is to use recursion. **Flood\_Fill**, in Listing 1 (FLOOD.INC), is a recursive region filler. Its algorithm is simple yet elegant in the way recursion tends to be: If the pixel to be filled is not yet set, it is set and **Flood\_Fill** proceeds to check its four orthogonal neighbors. If any of the neighbors are not set, **Flood\_Fill** calls itself to fill them. For a procedure written entirely in a high-level language, **Flood\_Fill** is very fast. It is also a memory hog of the first order. Since each call to **Flood\_Fill** can generate up to four additional calls (each of which may further generate up to four additional calls, and so on geometrically), your stack will disappear in a tremendous hurry. By default, only 16,384 bytes of memory are allocated to the stack, and even with the **\$M** compiler command, the stack may only be given 65,520 bytes. If you exhaust the memory allocated to the stack, runtime error 202 (stack overflow) occurs and your program crashes. Therefore, **Flood\_Fill** is good only for relatively small areas.



POINT STACK  
X

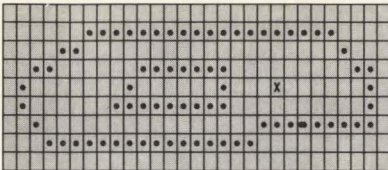


Figure 2. Example region to be filled, starting at x.

POINT STACK  
3  
2

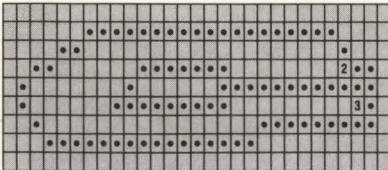


Figure 3. Fill the first line containing x, and locate candidate points 2 and 3 in the process.

POINT STACK  
4  
2

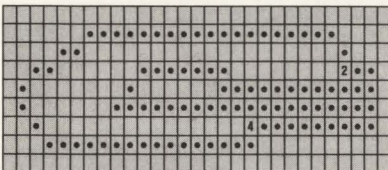


Figure 4. Fill the line containing 3, and find point 4.

POINT STACK  
5  
2

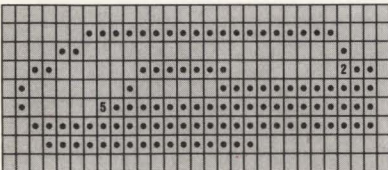


Figure 5. Fill the line containing 4, and find point 5.

POINT STACK  
6  
2

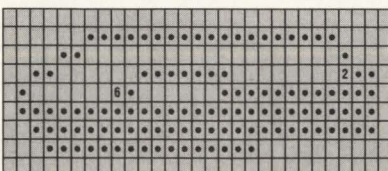


Figure 6. Fill the line containing 5, and find point 6.

continued on page 30

LISTING 1: FLOOD.INC

```

PROCEDURE Flood_Fill (X, Y: Real);
{ Example of a flood-fill algorithm }

{ By Fred Robinson
  Monotreme Software      Copyright (c) 1987 Monotreme Software
  29766 Everett
  Southfield, MI  48076
  USA }

VAR
  Start_X, Start_Y, X1Loc, X2Loc: Integer;

(*****)

PROCEDURE Do_The_Flood_Fill (X, Y: Integer);

{ The actual filling routine. Taken from Fundamentals of Inter-
  active Computer Graphics, Foley & Van Dam 1982, p. 448. }

BEGIN
  IF (X>=X1Loc) AND (X<=X2Loc) AND (Y>=Y1RefGlb) AND
    (Y<=Y2RefGlb) THEN
    IF NOT PD (X, Y) THEN
      BEGIN
        DP (X, Y);
        Do_The_Flood_Fill (X+1, Y);
        Do_The_Flood_Fill (X-1, Y);
        Do_The_Flood_Fill (X, Y+1);
        Do_The_Flood_Fill (X, Y-1)
      END (* THEN *)
    END (* Do_The_Flood_Fill *);

(*****)

BEGIN (* Flood_Fill *)
{ Get pixel coordinates of (X, Y) }

IF DirectModeGlb THEN
  BEGIN
    Start_X := Round (X);
    Start_Y := Round (Y)
  END (* THEN *)

ELSE
  BEGIN
    Start_X := WindowX (X);
    Start_Y := WindowY (Y)
  END (* ELSE *);

{ Set the proper X-bounds }

IF HatchGlb THEN
  BEGIN
    X1Loc := X1RefGlb;
    X2Loc := X2RefGlb;
  END (* THEN *)

ELSE
  BEGIN
    X1Loc := X1RefGlb SHL 3;
    X2Loc := X2RefGlb SHL 3 + 7
  END (* ELSE *);

Do_The_Flood_Fill (Start_X, Start_Y)
END (* Flood_Fill *);

```



```
PROCEDURE Fill_Region (X, Y: Real);
```

```
{ Region-filling code developed from a description of the algorithm
in Fundamentals of Interactive Computer Graphics, Foley & Van Dam
1982, p. 450-451.
```

```
No provision has been made for differences between the FILL color
and the BOUNDARY color; both are assumed to be the current set color.
This is because the Graphix Toolbox is not set up to cope with multi-
color high-resolution display images. }
```

```
{ By Fred Robinson
Monotreme Software Copyright (c) 1987 Monotreme Software
29766 Everett
Southfield, MI 48076
USA }
```

```
TYPE
{ Stack for storing potential starting points }
Pair_Ptr = ^Pair;
Pair = RECORD
    X, Y: Integer;
    Next: Pair_Ptr
END;
```

```
VAR
Top_Pair, This_Pair: Pair_Ptr;
Start_X, Start_Y, X1Loc, X2Loc: Integer;
```

```
(*****)
```

```
PROCEDURE Fill_Line (X, Y: Integer);
```

```
{ This procedure fills in the pixel line Y, starting at point X,
first moving to the right to the rightmost unfilled pixel, then
from (X, Y) to the left to the leftmost unfilled pixel. On both
passes, the lines above & below are checked for candidate starting
points. That is, if (X, Y) is not already filled in. }
```

```
VAR
X1, X2, Y_Above, Y_Below: Integer;
```

```
(*****)
```

```
PROCEDURE Check_Point (X, Y: Integer);
```

```
{ This procedure checks (X, Y) and, if it is a point to start
filling at, adds it to the stack. }
```

```
VAR
This_Pair: Pair_Ptr;
```

```
BEGIN
IF (X>=X1Loc) AND (X<=X2Loc) AND (Y>=Y1RefGlb) AND
(Y<=Y2RefGlb) THEN
{ Making sure that (X, Y) is within legal limits }
IF NOT PD (X, Y) THEN
IF (X=X2Loc) OR PD (X+1, Y) THEN
{ Believe it or not, this double-IF construct
is faster than ANDing the two conditions }
BEGIN
New (This_Pair);
This_Pair^.X := X;
This_Pair^.Y := Y;
This_Pair^.Next := Top_Pair;
Top_Pair := This_Pair
END (* THEN, THEN, THEN *)
END (* Check_Point *);
```

```
(*****)
```

## FILLING REGIONS

continued from page 29

POINT STACK

7  
2

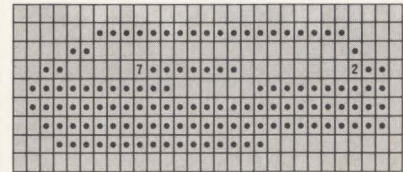


Figure 7. Fill the line containing 6, and find point 7.

POINT STACK

8  
2

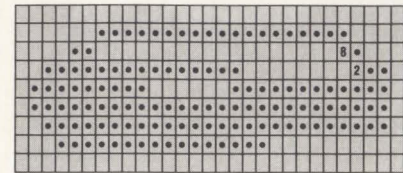


Figure 8. The search for a candidate point must be extended beyond the end of point 7's line until point 8 is found.

POINT STACK

9  
2

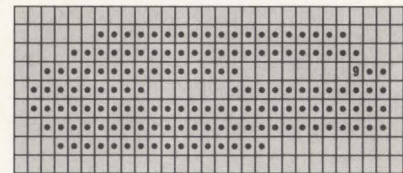


Figure 9. Note how point 2 has been marked a second time as point 9.

POINT STACK

2

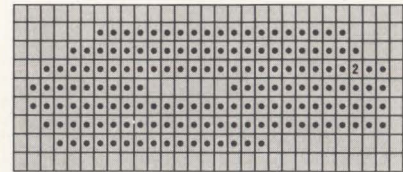


Figure 10. Point 2 is ignored, since it is within a filled line, and the filling operation is completed.



Filling large areas without massive use of the stack requires a more complicated algorithm. You may notice that a region to be filled consists of a number of horizontal lines of pixels, and each line must be filled in with a color. This suggests a better way. The sequence of Figures 2 through 10 shows this filling process, step by step.

### x MARKS THE SPOT

Given the task of filling the irregular region shown in Figure 2, we can start at an arbitrary pixel marked with an "x." The first step is to fill the line containing the starting pixel, as shown in Figure 3. This is done by first filling to the right of the "x," and then to the left. While filling the line, the routine checks above and below the line being filled for a background pixel immediately to the left of a foreground pixel. This background pixel is called a *candidate point*. The location of each of these points is stored in a stack-like First-In, First-Out data structure created on the heap as a linked list. In Figure 3, the points detected are marked 2 and 3.

Now there are two more points where filling must begin. Starting with the most recent candidate point found, the fill process is repeated, as shown in Figure 4. While filling the line starting with point 3, point 4 is found.

The process continues with the line containing point 4. Note that the line containing point 2 is not yet filled because it is not at the top of the stack. Point 5 is found while filling point 4's line, and once point 4's line is filled, the line containing point 5 is filled, and so on until the stack of candidate points is empty. You can follow the process to completion in Figures 5 through 10.

Notice in Figure 8 that while filling the line containing point 7, checking for candidate points above the line is extended to the right *beyond* the end of point 7's line until a candidate point (point

*continued on page 32*

```

BEGIN (* Fill_Line *)
IF NOT PD (X, Y) THEN
  BEGIN
    { Fill in to the right of (X, Y) }

    X1 := X;
    Y_Above := Y - 1;
    Y_Below := Y + 1;

    WHILE (X1<=X2Loc) AND NOT PD (X1, Y) DO
      BEGIN
        Check_Point (X1, Y_Below);
        Check_Point (X1, Y_Above);
        DP (X1, Y);
        X1 := X1 + 1
      END (* WHILE *);

    { Check above and below beyond the right end of the line just
      filled in }

    X2 := X1 - 1;

    WHILE (X2<=X2Loc) AND NOT PD (X2, Y_Below) DO
      BEGIN
        Check_Point (X2, Y_Below);
        X2 := X2 + 1
      END (* WHILE *);

    X2 := X1 - 1;

    WHILE (X2<=X2Loc) AND NOT PD (X2, Y_Above) DO
      BEGIN
        Check_Point (X2, Y_Above);
        X2 := X2 + 1
      END (* WHILE *);

    { Fill in to the left of (X, Y) }

    X1 := X - 1;

    WHILE (X1>=X1Loc) AND NOT PD (X1, Y) DO
      BEGIN
        Check_Point (X1, Y_Below);
        Check_Point (X1, Y_Above);
        DP (X1, Y);
        X1 := X1 - 1
      END (* WHILE *)
    END (* THEN *)
  END (* Fill_Line *);

  (*****)

BEGIN (* Fill_Region *)

{ Get pixel coordinates of (X, Y) }

IF DirectModeGlb THEN
  BEGIN
    Start_X := Round (X);
    Start_Y := Round (Y)
  END (* THEN *)
ELSE
  BEGIN
    Start_X := WindowX (X);
    Start_Y := WindowY (Y)
  END (* ELSE *);

```



```

PROGRAM Fill_Circle;

{$M 65520,0,655360} { Set stack allocation to maximum }

uses
  Crt,GDriver,GKernel,Turbo3;

{$I FILL.INC}
{$I FLOOD.INC}

{ A demonstrator for Fill_Region and Flood_Fill. }

{ By Fred Robinson
  Monotreme Software      Copyright (c) 1987 Monotreme Software
  29766 Everett
  Southfield, MI 48076
  USA }

VAR
  I: Integer;
  X, Y: Real;
  White: Boolean;
  Ch: Char;

BEGIN
  Randomize;
  White := True;
  InitGraphic;
  DefineWorld (1, -1, -0.75, 1, 0.75);
  SelectWorld (1);
  SelectWindow (1);
  SetLineStyle (0);

  REPEAT
    IF White THEN
      SetColorWhite
    ELSE
      SetColorBlack;

    White := NOT White;

    FOR I := 1 TO 25 DO
      BEGIN
        SetAspect (2.25*Random);
        DrawCircle (2*Random-1, 1.5*Random-0.75, 1.5*Random)
      END (* FOR *);
  
```

## FILLING REGIONS

*continued from page 31*

8) is found. Naturally, this extension is not done if the line above or below the line being filled has already been filled.

Also notice that in Figure 9, point 2 has been marked a second time as point 9. (Both points still exist as separate records on the stack, however.) This is a consequence of the algorithm and is perfectly legitimate. When the line-filling routine is handed a point that has already been filled, it ignores the point. This is what happens once the line containing point 9 is filled: Point 2 is now within a filled line, and the line-filler routine ignores it.

Candidate points for starting the filling are background points to the immediate left of a foreground point, and background points along the right edge of the display screen. This second group is included because the edges of the display are necessarily considered boundaries.

The algorithm we've just stepped through is summarized in three pieces of pseudo-code in Figure 11.

The complete Turbo Pascal routine **Fill\_Region** is shown in Listing 2, FILL.INC.

Listing 3 (FILLCIRC.PAS) is the program **Fill\_Circle**, which acts as a demonstration program for both of the region fill routines. **Fill\_Circle** draws a number of circles on the screen, and then fills in randomly chosen regions among the intersecting circles. The drawing color alternates from black to white on each cycle of circles and filling; pressing a key ends the program. Near the end of the main program are calls to both **Fill\_Region** and **Flood\_Fill**. One of the two must always be commented out. Choose the one you wish to demonstrate and comment out the other. Keep in mind that **Flood\_Fill** is included only to demonstrate its unsuitability for filling large areas; as soon as it attempts to fill a sizeable region, it exhausts stack space and ends the program with error 202.



**Fill\_Region** is very effective, and will fill a region of any size or shape. It is not especially fast, but it doesn't require assembly language, nor does it demand huge amounts of stack or heap memory. It also provides a nice complement to the rectangle-fill routine in the Turbo Pascal Graphix Toolbox. ■

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

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

```

routine fill_region (x,y)
  push (x,y) onto stack

  while the stack is not empty do:
    pop a point (x,y) from stack
    call fill_line with (x,y)

exit fill_region

routine fill_line (x,y)
  if (x,y) is not filled then
    from (x,y) to the right until
      a lit point is reached do:
        check_point above
        check_point below
        set next point in the line

    continuing along the line
      above, to the right until
        a lit point is reached:
          check_point

    continuing along the line
      below, to the right until
        a lit point is reached:
          check_point

    from (x,y) to the left until
      a lit point is reached do:
        check_point above
        check_point below
        set next point in the line

exit fill_line

routine check_point (x,y)
  if (x,y) is on the display then
    if (x,y) is not lit then
      if (x+1,y) is lit,
      or (x+1,y) is not
      on the display then:
        push (x,y) onto the stack

exit check_point

```

*Figure 11. Fill\_Region summarized in pseudo-code.*

```

FOR I := 1 TO 10 DO
  BEGIN
    REPEAT
      X := Random * 2.0 - 1.0;
      Y := Random * 1.5 - 0.75
    UNTIL NOT PointDrawn (X, Y);

    Fill_Region (X, Y)
  { Flood_Fill (X, Y) } { Using Flood_Fill exhausts stack space }
  END (* FOR *)
  UNTIL KeyPressed;

  Read (Kbd, Ch);
  LeaveGraphic
  END.
  { Push the given starting point onto the stack }

  New (Top_Pair);
  Top_Pair^.Next := NIL;
  Top_Pair^.X := Start_X;
  Top_Pair^.Y := Start_Y;

  { Set the proper X-bounds }

  IF HatchGlb THEN
    BEGIN
      X1Loc := X1RefGlb;
      X2Loc := X2RefGlb;
    END (* THEN *)

  ELSE
    BEGIN
      X1Loc := X1RefGlb SHL 3;
      X2Loc := X2RefGlb SHL 3 + 7
    END (* ELSE *);

  { Fill in until there are no more starting points on the stack }

  WHILE Top_Pair<>NIL DO
    BEGIN
      This_Pair := Top_Pair;
      Top_Pair := Top_Pair^.Next;
      Start_X := This_Pair^.X;
      Start_Y := This_Pair^.Y;
      Dispose (This_Pair);
      Fill_Line (Start_X, Start_Y);
    END (* WHILE *)
  END (* Fill_Region *);

```



# CURVES, BEZIER STYLE

Take some dangerous curves in complete safety with Turbo Pascal in the driver's seat.

Kent Porter



PROGRAMMER

One of the classic difficulties in high-level graphics programming is the representation of complex, irregular curves. Examples are decorative curlicues, handwriting, and the shapes of real-world objects with curved surfaces. Because of the importance of these curves in CAD (computer-aided design), this problem has attracted a great deal of energy, especially in the aerospace and automotive industries. In this article, we'll discuss one of the most practical solutions—the Bezier method—which enables you to express a complex curve in terms of a few points; we'll also give you a Turbo Pascal unit that you can apply to your own curve-drawing needs.

The method takes its name from its inventor, P.E. Bezier (pronounced "bay-zee-AY"), a French mathematician working for Renault. Developed in the early 1970s, the Bezier method is one of a group of mathematical curios called *cubic splines*, all of which describe curves. Bezier's fundamental proposition is that a polygon can approximate a curve, which wends its way among the vertices (the "corners," or *control points* as we'll call them in this discussion). If the polygon closes back on its origin, the curve is also closed, such as in a circle or an ellipse. An open polygon, on the other hand, describes a curve with two end points. The curve touches only the end points, with the other vertices exerting an influence over its path. Figure 1 shows a Bezier curve drawn in CGA medium resolution, along with the polygon containing the curve's control points.

A science fiction analogy helps explain how a Bezier curve works. Suppose a starship is moving from one star to another. As it travels, it encounters other celestial bodies whose gravity pulls it away from a straight line. All the objects in the universe influence its path to some extent, but the nearest ones have the most effect. When it finally arrives at its destination, the starship's track will have described a complex, constantly varying curve.

In the course of the journey, the craft's position at any instant is represented by two kinds of information. One is the position coordinates represented by  $x$ ,  $y$ , and  $z$  (for horizontal, vertical, and depth, respectively). These coordinates indicate the starship's absolute location. The second piece of information is the percentage of total distance the starship has traveled from start to finish. This is a relative indicator ranging from 0 to 1 that is represented by a value called  $u$  in the calculations in Listing 1. When  $u = 0.0$ , the craft hasn't departed yet; when  $u = 1.0$ , it has arrived; and when  $u = 0.5$ , the starship is halfway to its destination. Thus, if you were the navigator, you could determine your position at any time by deriving the coordinates from  $u$ , which is, in effect, the controlling variable of the voyage. Bezier furnishes a formula for doing this.

## CALCULUS 101

Unfortunately, no one has yet come up with a way to do more than the most primitive graphics without resorting to heavy math, and Bezier curves are no exception. The Bezier equation for a curve is:

$$P(u) = \sum_{i=0}^n p_i B_{i,n}(u)$$

In other words, for any point  $u$ , the location is the sum of  $n + 1$  control-point factors proceeding from point 0. This factor is a blending function

$$B_{i,n}(u) = C(n,i) u^i (1-u)^{n-i}$$

which in turn is derived from the binomial coefficient:

$$C(n,i) = n! / (i!(n-i)!)$$

So much for the equations. Now let's talk about what they mean.



The blending function is where the real work of the Bezier method gets done. This function calculates a "gravity factor" for each control point relative to the current **u**. The closer a control point is to the metaphorical starship, the more "gravity" it has and the more it influences the craft's course, which is represented by the coordinates. When **u = 0**, the spacecraft is still at the point of origin, or **p[0]**, and no other points have any influence on its position. **p[1]** begins to pull the craft as it leaves the point of origin. Halfway between **p[0]** and **p[1]**, both points have the same influence on the starship. However, **p[2]** also attracts the craft enough to pull it away from a straight line between **p[0]** and **p[1]**; other points beyond **p[2]** also attract the spaceship in diminishing proportion to their distance from it. As the craft approaches the end point, there is nothing beyond to draw it away, and so the craft arrives.

The blending function is merely a mathematical statement of this effect. For each control point, it returns a fractional value by which the coordinates of that control point are multiplied. The sums of these **x**, **y**, and **z** products are the coordinates of the position at **u**.

By drawing lines between successive **u**'s, we make a map of the route, which is the Bezier curve based on the control point layout.

## OF HULLS AND SUCH

The power of Bezier curves lies in their ability to represent a complex curve with a few points expressed in coordinates. The simplest curve is described by three points: the origin, the destination, and one intermediate vertex. The outcome in this case is a smooth arc tending toward the intermediate point.

*continued on page 36*

### LISTING 1: BEZIER.PAS

```

UNIT Bezier;

  { Functions and procedures for Bezier curves }

INTERFACE

USES   graph;

CONST  maxPoints = 10;    { max control points for a curve }
       segments  = 40;    { nbr of segments in a curve }

TYPE   vectorArray = ARRAY [0..maxPoints, 0..2] OF INTEGER;

PROCEDURE BezierFcn (VAR x, y, z : REAL;
                    u       : REAL;
                    n       : INTEGER;
                    VAR p   : vectorArray);
PROCEDURE DrawBezier2D (VAR p : vectorArray;
                      npts : INTEGER);
{ ----- }
IMPLEMENTATION

FUNCTION c (n, i : INTEGER) : INTEGER;

  { Binomial coefficient used in blending function }

VAR   j : INTEGER;

FUNCTION fact (q : INTEGER) : INTEGER;
VAR   f, c : INTEGER;
BEGIN   { Compute factorial of q }
  f := 1;
  FOR c := q DOWNTO 2 DO
    f := f * c;
  fact := f;
END;

BEGIN
  c := fact (n) div (fact (i) * fact (n - i));
END;
{ ----- }

FUNCTION blend (i, n : INTEGER; u : REAL) : REAL;

  { Bernstein blending function used in Bezier formulation }

VAR   partial : REAL;
       j       : INTEGER;

BEGIN
  Partial := c (n, i);
  FOR j := 1 TO i DO
    Partial := partial * u;
  FOR j := 1 TO (n - i) DO
    Partial := partial * (1 - u);
  Blend := partial;
End;
{ ----- }

PROCEDURE BezierFcn;

  { Returns 3D coordinates for current 'u' on the curve }

VAR   i : INTEGER;
       b : REAL;

BEGIN
  x := 0;
  y := 0;
  z := 0;

```



## CURVES

continued from page 35

```
FOR i := 0 TO n DO
BEGIN
  b := blend (i, n, u);
  x := x + p [i, 0] * b;
  y := y + p [i, 1] * b;
  z := z + p [i, 2] * b;
END;
END;
{ ----- }

PROCEDURE drawBezier2D;

  { Draws a 2D Bezier curve. Graphics mode assumed. }

VAR
  i          : INTEGER;
  oldX, oldY,
  u, x, y, z : REAL;

BEGIN
  FOR i := 0 TO segments DO
  BEGIN
    u := i / segments;
    BezierFcn (x, y, z, u, npts, p);
    IF i = 0 THEN
      MoveTo (ROUND (x), ROUND (y))
    ELSE
      LineTo (ROUND (x), ROUND (y));
  END;
END;
{ ----- }

END.
```

### LISTING 2: CURVE.PAS

```
Program curve;

  { Draw a Bezier curve on the CGA }

USES graph, Bezier, crt;

CONST lastPoint = 6;    { highest subscript used }

VAR pt                : vectorArray;
    graphDriver, graphMode, errorCode, n : INTEGER;
    wait              : CHAR;
{ ----- }

PROCEDURE initPoints;    { Define control points }

VAR n, j : INTEGER;

BEGIN
  FOR n := 0 TO maxPoints DO    { Zero the array }
  FOR j := 0 TO 2 DO
    pt [n, j] := 0;
  pt [0, 0] := 160;  pt [0, 1] := 110;
  pt [1, 0] := 160;  pt [1, 1] := 160;
  pt [2, 0] := 55;   pt [2, 1] := 160;
  pt [3, 0] := 55;   pt [3, 1] := 20;
  pt [4, 0] := 205;  pt [4, 1] := 40;
  pt [5, 0] := 190;  pt [5, 1] := 110;
  pt [6, 0] := 280;  pt [6, 1] := 110;
END;
{ ----- }
```

More interesting curves result from several points laid out in an order that very roughly approximates the desired trajectory, as in Figure 1. That curve derives from seven control points. The straight lines join the points in order, forming an open polygon, or *hull* as it is often called. Normally, of course, you wouldn't draw the hull itself, since the object of the game is the curve. However, we show the hull here so that you can see how it provides an intuitive notion of how the curve will be drawn.

Bezier curves are insensitive to orientation. In Figure 1, the origin is located at the center of the curl and the destination is located at the far right. However, if we had specified the control points in the opposite order, exactly the same curve would have resulted.

Figure 2 illustrates a closed Bezier curve. The hull has six control points. The four corner control points are obvious. Less obvious are the origin and destination control points, which are coincident at bottom center where the circle and hull meet. If you pulled one of the corners of the hull out or pushed it in, the Bezier curve would form a misshapen circle.

One restriction on Bezier curves is that they cannot cross their own path. This is easy to avoid—don't let one hull line cross another. There's no easy way for software to check for this conflict, so it's your responsibility. If you try to make it happen, your program will either lock up in an endless loop, or it will crash with some bizarre error message. Later in this article, we'll show a way to overcome the problem by joining two separately calculated curves that may safely cross.

### IMPLEMENTING A BEZIER UNIT

Listing 1 contains BEZIER.PAS, a Turbo Pascal 4.0 unit that partially implements a set of Bezier routines. We say "partially" because,



although the Bezier calculations return three-dimensional coordinate sets, the unit doesn't furnish a 3-D drawing routine. There's a good reason for this: 3-D graphics is a subject for a very thick book, and we haven't room to cover it here. If you want to learn more, see *Principles of Interactive Computer Graphics*, W. Newman and R. Sproull, New York: McGraw-Hill, 1979. The Bezier algorithm in Listing 1 is based in part on Newman and Sproull, pp. 315-319.

The **maxPoints** constant is set to 10, and the **vectorArray** type uses this value as the upward limit. In fact, because subscripting starts with 0, the array actually contains a maximum of eleven control points. This is consistent with the summarization function of the Bezier equation, which specifies

$$\sum_{i=0}^n$$

or in other words, **n + 1** control points. (Note that the term *vector* means a set of coordinates—these are the columns [0..2] in **vectorArray** that correspond to **x**, **y**, and **z**.) Eleven control points for a curve seems a reasonable limit; if you need more, consider joining curves as discussed later.

The **segments** constant is the basis for **u** in the calculations. This constant defines how many line segments make up the complete curve. The **drawBezier2D** procedure controls curve-drawing with the loop

FOR i := 0 TO segments DO. . .  
and calculates **u** as **i/segments**. Therefore  $0.0 \leq u \leq 1.0$  represents a percentage of the complete transit from origin to destination.

Two of the five subroutines in **BEZIER.PAS** are externally visible. The **drawBezier2D** procedure is probably the only one you'll call, but you might want to call **BezierFcn** for applications such as marking points on the curve—say at every 10 percent of its length, or at its midpoint. Given the desired

*continued on page 38*

```
BEGIN
{ Set up the screen in CGA 320 x 200 mode, palette 1 }
GraphDriver := CGA;
GraphMode   := CGAC1;
InitGraph (graphDriver, graphMode, 'C:\DRIVERS');

{ Check to make sure it happened }
ErrorCode := graphResult;
IF errorCode <> grOK THEN BEGIN
  Writeln ('Graphics error ', errorCode);
  Writeln ('Program cannot run!');
  HALT (1);
END;

{ Draw the hull outline }
InitPoints; { First initialize control points }
SetLineStyle (dottedLn, 0, normWidth);
SetColor (1);
MoveTo (pt [0, 0], pt [0, 1]);
FOR n := 1 TO lastPoint DO
  LineTo (pt [n, 0], pt [n, 1]);

{ Now draw the curve itself }
SetLineStyle (solidLn, 0, normWidth);
SetColor (2);
DrawBezier2D (pt, lastPoint);

{ Clean up after a keypress }
Wait := readkey;
CloseGraph;
END.
```

#### LISTING 3: TWOCURVS.PAS

```
Program twoCurvs;

{ Draw joined Bezier curves on the CGA }

USES graph, Bezier, crt;

CONST lastPoint = 4; { highest subscript used }

VAR a, b : vectorArray;
    graphDriver, graphMode, errorCode, n : INTEGER;
    wait : CHAR;
{ ----- }

PROCEDURE initPoints; { Define control points }
VAR n, j : INTEGER;

BEGIN
  FOR n := 0 TO maxPoints DO { Zero the array }
    FOR j := 0 TO 2 DO BEGIN
      a [n, j] := 0;
      b [n, j] := 0;
    END;
  a [0, 0] := 10; a [0, 1] := 110; { first hull }
  a [1, 0] := 10; a [1, 1] := 0;
  a [2, 0] := 120; a [2, 1] := 0;
  a [3, 0] := 180; a [3, 1] := 110;
  a [4, 0] := 240; a [4, 1] := 110;

  b [0, 0] := 240; b [0, 1] := 110; { second hull }
  b [1, 0] := 310; b [1, 1] := 110;
  b [2, 0] := 310; b [2, 1] := 0;
  b [3, 0] := 180; b [3, 1] := 0;
  b [4, 0] := 120; b [4, 1] := 199;
END;
{ ----- }
```



```

BEGIN
{ Set up the screen in CGA 320 x 200 mode, palette 1 }
  GraphDriver := CGA;
  GraphMode   := CGAC1;
  InitGraph (graphDriver, graphMode, 'C:\DRIVERS');

{ Check to make sure it happened }
  ErrorCode := graphResult;
  IF errorCode <> grOK THEN BEGIN
    Writeln ('Graphics error ', errorCode);
    Writeln ('Program cannot run');
    HALT (1);
  END;

{ Draw the first hull outline }
  InitPoints;      { First initialize control points }
  SetLineStyle (dottedLn, 0, normWidth);
  SetColor (1);
  MoveTo (a [0, 0], a [0, 1]);
  FOR n := 1 TO lastPoint DO
    LineTo (a [n, 0], a [n, 1]);

{ Draw the second hull }
  MoveTo (b [0, 0], b [0, 1]);
  FOR n := 1 TO lastPoint DO
    LineTo (b [n, 0], b [n, 1]);

{ Mark the joint with a vertical line }
  MoveTo (240, 100);
  LineTo (240, 120);

{ Now draw the first curve }
  SetLineStyle (solidLn, 0, normWidth);
  SetColor (2);
  DrawBezier2D (a, lastPoint);

{ And second curve }
  SetColor (3);
  DrawBezier2D (b, lastPoint);

{ Clean up after a keypress }
  Wait := readkey;
  CloseGraph;
END.

```

## CURVES

*continued from page 37*

point in terms of **u**, **BezierFcn** returns the coordinates. The functions **C** and **Blend** are useful only to **BezierFcn**, and therefore are not callable from outside the unit.

### PUTTING IT TO WORK

The output from the demonstration program CURVE.PAS in Listing 2 is the curlicue shown in Figure 1. This program uses the **Bezier** unit and also the Turbo Pascal 4.0 **Graph** unit.

The program itself is simple and self-explanatory. It puts the display adaptor into CGA four-color (320 × 200) graphics mode (which also works on the EGA and VGA). After outlining the hull, the program draws the Bezier curve in a contrasting color, then waits for a keypress and terminates.

### JOINING CURVES

You might need to join two or more curves when the path of the desired curve must cross over itself, or when the curve is so complex that it requires more control points than the **Bezier** unit supports.

In order to achieve a smooth joint, you must follow two rules. First, make two end points of the adjacent curves coincident so that one curve ends at exactly the same point where the next curve begins. The second rule is to create continuity in the adjoining hulls across the joint.

For example, assume that you have two hulls called **A** and **B**. **A**, which flows into **B**, has six control points. To achieve continuity, **A[4]**, **A[5]** (coincident with **B[0]**), and **B[1]** must all lie on the same plane, so that a straight line passes through all of them. The span from **A[4]** through **B[1]** should be fairly long, so that the curves have room to sweep gracefully to their coincident end points. A long span eliminates a sudden kink or angle that disrupts the curve's flow.



Figure 3 illustrates this. Both hulls are outlined in dotted lines. A vertical line marks the joint, which is in the bottom center of the right-hand polygon. Curve **A** sweeps up from the left side of the screen and down to the end point, where curve **B** picks up and crosses back over **A**. Run the TWOCURVS.PAS program in Listing 3, and note that curve **A** and curve **B** are drawn in different colors. This color scheme clearly identifies the two curves and pin-points where one curve ends and the next curve begins.

The Bezier method provides a powerful means for describing complex shapes using just a few data points. An interesting exercise in the Bezier method would be to write a mouse-knowledgeable program that lets you click on points on the screen, draw a curve, move the points around, and print the screen when you like the curve.

Because the Bezier method was developed in response to the need for CAD tools in automotive design, they're eminently well-suited to computer art, and offer a fascinating object of study as well. ■

---

*Kent Porter is the author of Stretching Turbo Pascal (Prentice Hall Press), and writes for TURBO TECHNIX and other computer magazines.*

---

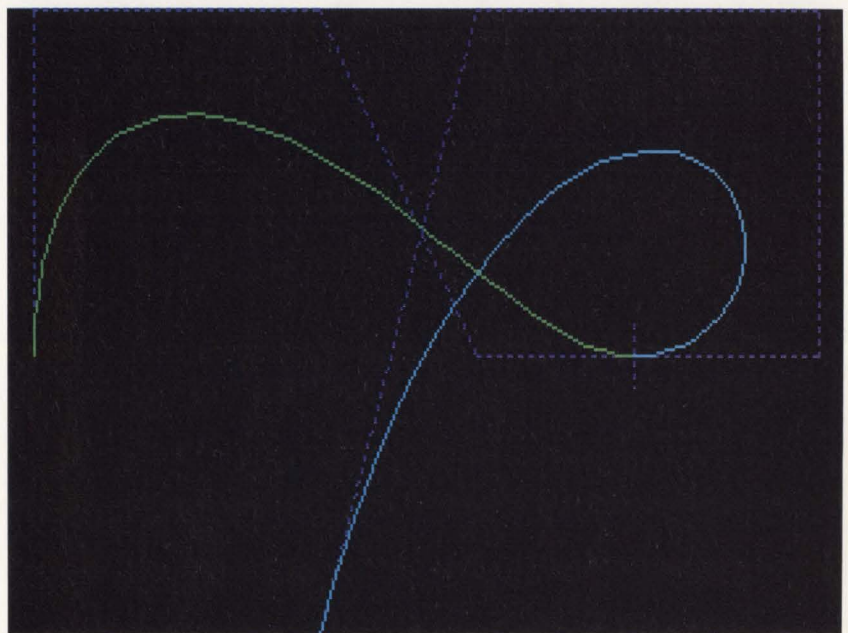
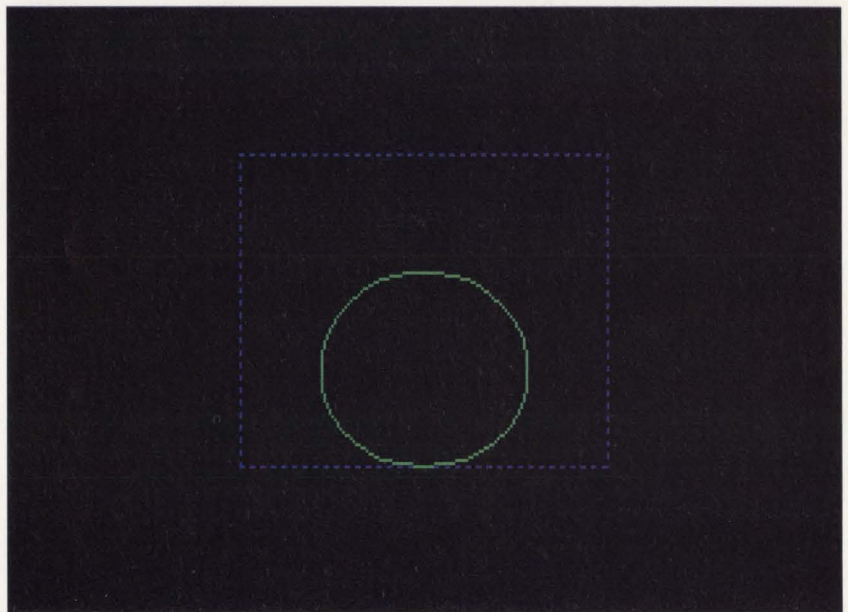
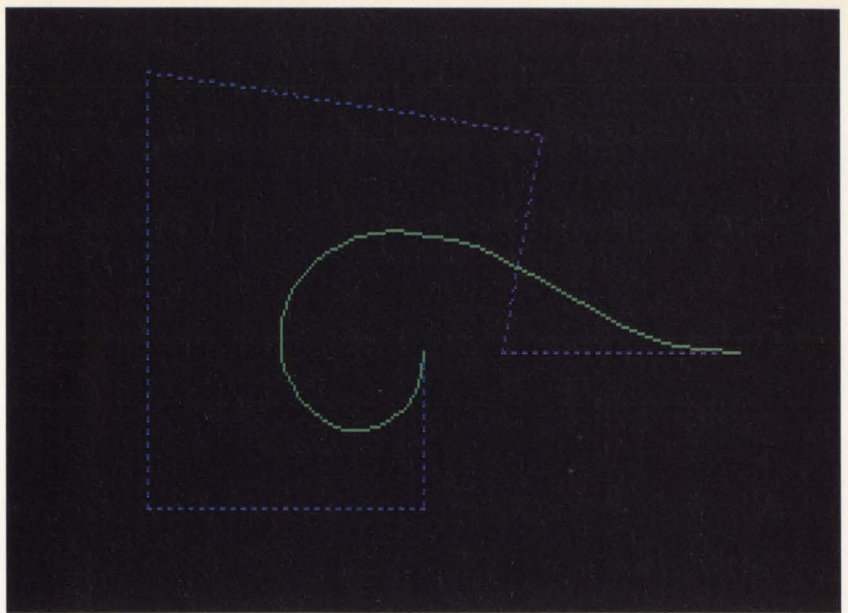
*Listings may be downloaded from CompuServe as BEZIER.ARC.*

---

*Figure 1. (Top) A Bezier curve and its hull. Each vertex of the hull, including the endpoints, is a control point to the curve itself.*

*Figure 2. (Middle) A closed Bezier curve. The circle's two coincident endpoints lie at the point where the curve contacts the hull.*

*Figure 3. (Bottom) Joining two Bezier curves. The vertical line shows the point where the curves join. Note that the two hulls pass through this point and subtend a straight line for some distance to either side of the join point.*

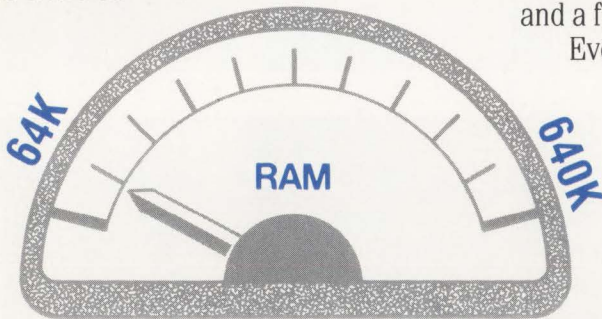




# Lots of software packages help you work,

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

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



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

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

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

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

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

## *Here's What You Get!*

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

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

- **Outlook:**

**The Outline Processor:** nine

Outliners with automatic numbering, tree charts, and table of contents

- **The File Manager:** extended DOS file and directory management

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

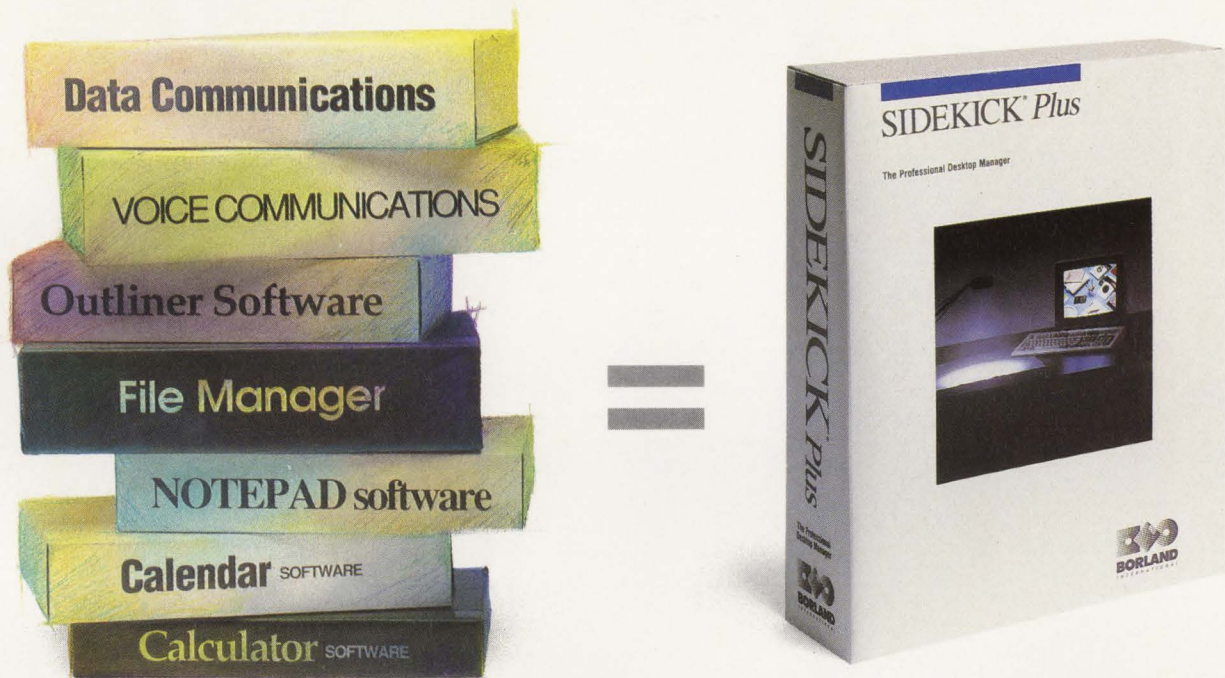
**Only \$199.95**  
(not copy protected)

**60-day money-back guarantee\***

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



# Only one helps you work smarter...



## **NEW!** Announcing SideKick Plus!

**\$25  
Rebate**

**SideKick Owners: Get a great deal from  
your dealer and a \$25 Rebate from Borland!**

**\$25  
Rebate**

Go to your favorite retailer for a great deal on new SideKick Plus. And, because you're a SideKick owner, we'll make it an even better deal—with a \$25 rebate from Borland! To receive your rebate, you must return your completed SideKick Plus registration form from your manual, a copy of your dated SideKick Plus sales receipt, and this completed coupon to: *Borland International, Dept. PSPR, 4585 Scotts Valley Drive, P.O. Box 660001, Scotts Valley, CA 95066*

\_\_\_\_\_  
Name  
\_\_\_\_\_  
Street  
\_\_\_\_\_  
City State Zip  
\_\_\_\_\_  
Phone

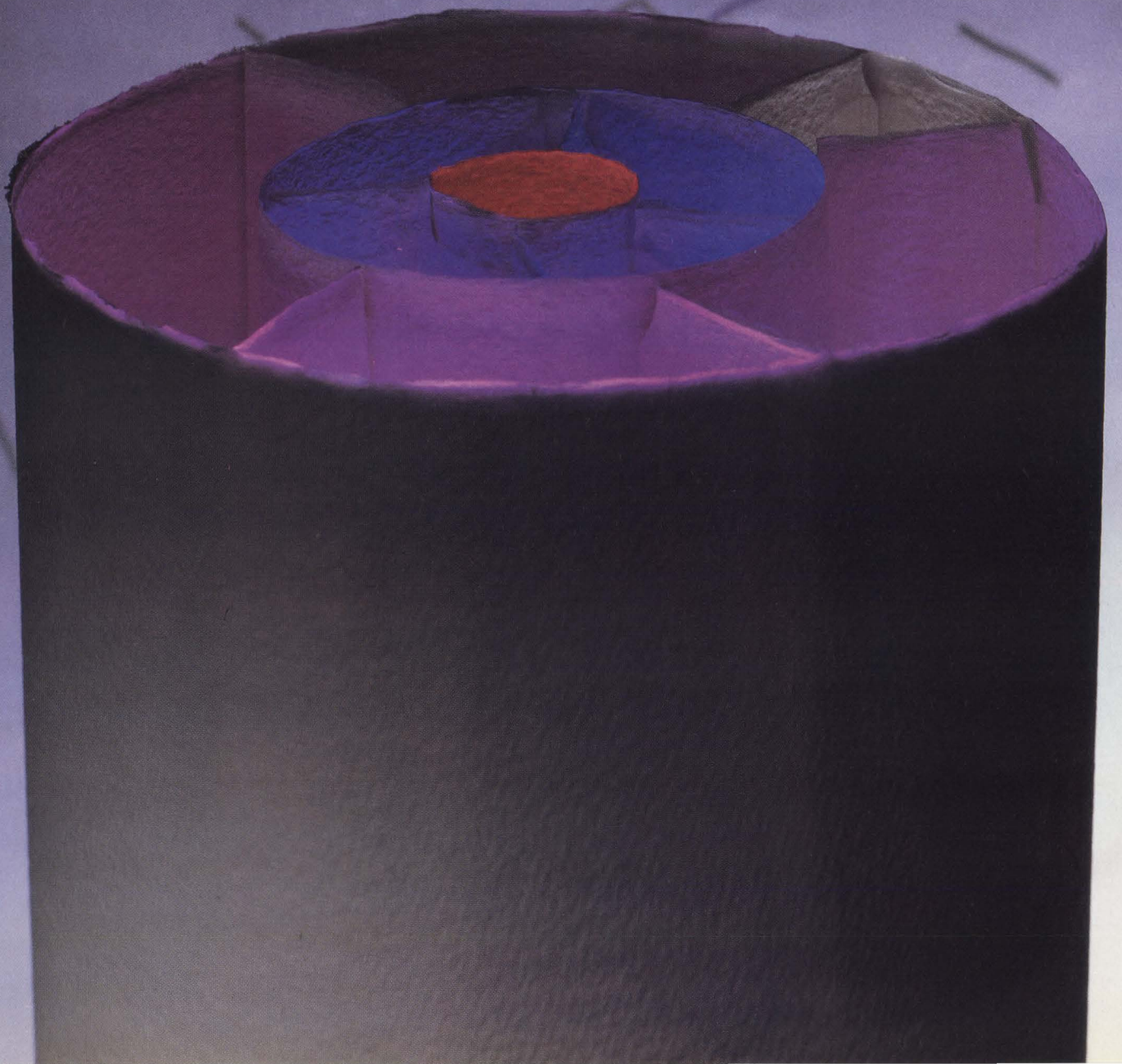
*Available at your dealer after  
March 20, 1988*



SideKick Serial No. (must be included to process rebate)\*

\*If your copy of SideKick does not have a serial number, remove and return the front cover of your SideKick manual.  
To take advantage of this rebate you must purchase SideKick Plus by May 31, 1988 and return the rebate request to Borland by July 31, 1988. This offer is good for one rebate per registered copy of SideKick. Not good with any other offer from Borland. Please allow 6 to 8 weeks for delivery of rebate. Offer good in the U.S., and Canada only.







# THE SIDEKICK PLUS API: INTRODUCTION

Extend SideKick Plus with your own custom tasks—and let SideKick Plus do the hard work.

Jeffrey Goldberg and Steven Boye



WIZARD

How would you like your programming environment to include a TSR kernel, a pop-up menu system, a memory manager, and a communications manager?

Interested? Well, welcome to the SideKick Plus programming environment, embodied in the SideKick Plus Application Program Interface (API). The SideKick Plus API is a new Borland product containing all the tools necessary to design and implement your own SideKick Plus tasks. In this article, we'll give you a brief introduction to the SideKick Plus API architecture and programming techniques. Future issues of *TURBO TECHNIX* will explain the process more fully, and provide further working examples of actual applications.

## THE TSR PROBLEM

Borland's original SideKick was the first widely used Terminate and Stay Resident (TSR) application program. Other vendors quickly adopted the concept and a host of TSR applications appeared. Each worked well on its own, but when more than one TSR was resident in memory at one time, conflicts began. TSR battles over control of the keyboard interrupt have become legendary. The larger problem of multiple TSRs occupying the lion's share of the PC's 640K of memory grew worse as TSRs grew in complexity and hence in size. Dropping one of several resident TSR applications from memory (except for the last TSR) did not reclaim that memory for use by other applications.

A standard was needed for a higher-level TSR manager—a single TSR with the ability to share machine resources equitably, while claiming a minimal amount of the precious 640K memory. To support today's and tomorrow's technology, this TSR also needed to be fairly device-independent from an applications standpoint. With all this in mind, Borland designed SideKick Plus to unify TSR programs and eliminate troublesome TSR conflicts.

## THE KERNEL AND THE TROOPS

Consensus is a uniquely human concept; among co-resident computer programs, one of them must be the boss. SideKick Plus assumes this supervisory role by always placing its central core—called the *kernel*—into memory. The kernel takes control of many things, including peripherals and memory management. What the user sees as SideKick Plus applications are separate program modules called *tasks*. To do their work, these tasks call routines that are part of the kernel; they may also call other subprograms known as *services*. We'll say more about these services later. Right now, let's take a look at a SideKick Plus task.

## A SIDEKICK PLUS TASK—THE NOTEPAD

Anyone familiar with the original SideKick, or with any Borland language, will recognize the SideKick Plus Notepad. Now, however, up to nine Notepads may exist, each able to read a file of 54,000 characters. Even with all nine Notepads full—a showstopping 474K of notes—SideKick Plus occupies only 64K of your precious memory. This feat is due to the *dynamic memory system* in SideKick Plus, which swaps both data and code to the hard disk, EMS memory, or RAM disk. SideKick Plus handles this swapping process for you whenever you request memory to be allocated for your program.

After bringing up the Notepad from the main SideKick Plus menu, you'll also notice a new multi-level menu system. Pressing F10 brings up a menu on the right side of the window. Some of the options read like English sentences, such as Insert Time and Date at Cursor. For many of the options, you can still use the familiar SideKick Ctrl-key shortcuts; a status line in the lower left corner of the screen even indicates the shortcut, if one is available.

*continued on page 44*



## LISTING 1: MINICOM.PRO

```

/*****
**
** Profile for MINICOM -- TURBO TECHNIX Example
**
*****/

MODULE TASK("MiniCom", ALTM, 100);

VERSION(0, 0, 1);          /* Version 0, req. kernel version 1 */
SERVICE(ServFile, swap, 0); /* Req. version 0 of File I/O service */
SERVICE(ServEd, swap, 3); /* Req. version 3 of Editor service */
SERVICE(ServTTY, noswap, 0); /* TTY driver is used */
SERVICE(ServMM, swap, 2); /* Modem Manager service */

TEXT SECTION;
TXTonLine      = "Online           \Hesc\N-Exit";
TXToffLine     = "Offline          \Hesc\N-Exit";
TXTerror       = " ERROR ";
TXTdiscon      = "Disconnecting";
TXTillegalno   = "Illegal telephone number";
TXTphoneErr    = "No carrier";
TXTedLine      = "Note           \Hesc\N-Exit";

MENU SECTION;
FUNCTIONKEYS( MainFkeys, EdFkeys);
CLASSES( Online, Offline );

MENU(Main, " MiniCom "):
"Dial"      menu(M_dial), class(Offline);
"Hangup"    procleaf(hangup), class(OnLine), fkey("Hangup", F8, MainFkeys);
"Note"      menu(M_note);

MENU(M_dial, " Number "):
"          " strleaf(phoneno, ,, dial);

MENU(M_note, " Note "):
"Edit"      procleaf(edit), fkey("Note", F9, MainFkeys);
"Send"      procleaf(send), class(OnLine);

```

## LISTING 2: MINICOM.C

```

/*****
**
** MiniCom
**
** This is a sample communications program which
** demonstrates the SideKick Plus API. It uses
** four services.
** ServFile (File I/O)
** ServTTY (COM port driver)
** ServMM (Modem Manager)
** ServEd (Editor service)
**
*****/

/* include API header files (API.LIB) */
/* ===== */
#include <apihead.p>
#include <apifunc.p>
#include <skkeys.p>

/* include APIP generated header file */
/* ===== */
#include "techmenu.h"

/* Important application constants */
/* ===== */
#define MAX_EDSIZE ((10 * 1024 + 905) / 16) /* About 10k */
#define COLOR_PALETTE 41 /* Colors the same as Phonebook */
#define SETUP_VERSION 0 /* Setup version required */

```

## SIDEKICK PLUS

*continued from page 42*

A user may easily modify the Borland shortcut conventions in SideKick Plus. One keystroke pops up a menu that allows the user to change the shortcut or the menu title, or even to move the menu item. Once the programmer defines the default menus and saves them to a menu file, the application isn't affected by changes that the user makes in the menu.

### A SIDEKICK PLUS SERVICE—THE EDITOR

SideKick Plus's editor is an example of a SideKick Plus service. A *service* is a piece of code that several applications share, and can be used by many applications at the same time. SideKick Plus includes many services, ranging from reading in a line of text to performing all DOS file operations. The example program in Listings 1 and 2 uses several of these services. To illustrate the use of SideKick Plus's editor as a service, let's look at the Phonebook.

Although its roots are in the original SideKick Dialer, the SideKick Plus Phonebook is now more than just a dialer—it's a fully automated communications package with a block-structured script language. Unlike the Dialer, the Phonebook lets the user open a new entry with a single keystroke, fill in the form, and press Esc. Because the SideKick Plus editor is available as a service to the Phonebook, the user can now append a note to any Phonebook entry by pressing F9, which opens an editor window attached to the entry. The Phonebook's editor is the same as the Notepad's editor. The Phonebook's editor window, however, saves its text to the Phonebook file rather than to a normal text file.

### TASK AND KERNEL ORGANIZATION

Services, tasks, and the kernel make up a three-part puzzle. Their relationships are summarized in Figure 1.



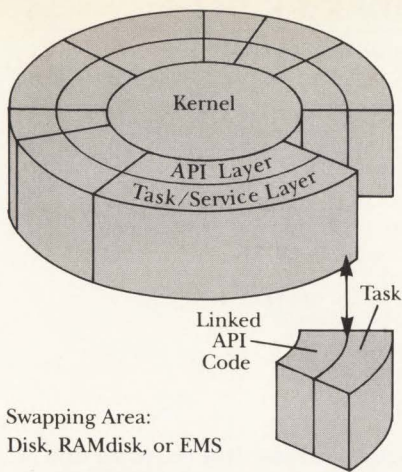


Figure 1. SideKick Plus brings tasks and/or services in from the swapping area as needed to keep its memory utilization as low as possible.

As the heart of SideKick Plus, the kernel has many jobs:

- Memory management of both the underlying application and of SideKick Plus code and data
- Coordination of tasks and services
- Keyboard input
- Video support, including window management and graphics screen restoration
- Coordination with DOS, which includes defining SideKick Plus as a TSR or as a normal, non-resident application
- Task scheduling

## TASKS

Tasks are the main building blocks of SideKick Plus applications. A task, such as the Calculator or Notepad, will usually appear on the SideKick Plus main menu and can be selected by using the key-stroke shortcuts, such as Alt-C or Alt-N, respectively.

A task consists of a set of procedures called in pairs (see Figure 2) by the kernel at special times. The first pair is **initproc** and **killproc**. As the name suggests, **initproc** is the initialization procedure that is called when you load SideKick Plus, and typically initializes windows and opens databases. Similarly, **killproc** is called when SideKick Plus is unloaded from memory.

*continued on page 46*

```

/* Window Variables used in application */
/* ===== */

int    mainwindow;          /* Comm. window that opens first */
int    edwindow;           /* Editor Window */

int    mainposx             = 1; /* Initial position/size of mainwindow */
int    mainposy             = 1;
int    mainsizex            = 70;
int    mainsizey            = 20;
int    edposx               = 10; /* Initial position/size of edwindow */
int    edposy               = 5;
int    edsizex              = 60;
int    edsizey              = 17;

/* Declarations for Window resizing and recoloring */

void    far    mainredraw( void );
WRSCB  mainwrscb = { /* Window ReSize Control Block
                    for the main window */
    MODULEID, /* Module id, #define in the PRO file */
    (PROC)mainredraw, /* Procedure called
                    after resizing the window */
    5, 5, /* Minimum size allowed */
    NIL, NIL, /* Maximum size. NIL means Kernel decides */
    0, 0, /* Zoom position */
    NIL, NIL /* Zoom maximum size.
            NIL means Kernel decides */
};

WRSCB  edwrscb = { /* Window ReSize Control Block
                  for editor window */
    MODULEID, /* Module id, #define by APIP */
    (PROC)NIL, /* Task redraw function */
    5, 5, /* Minimal size */
    NIL, NIL, /* Maximum size */
    0, 0, /* Zoom position */
    NIL, NIL /* Zoom size */
};

/* End of Window Declarations */

char    phoneno[24] = ""; /* Phone number variable
                          as read from the menu */

BOOL    online = FALSE; /* On-line or Off-line variable */
BOOL    newstat = FALSE; /* Need to redraw status line (line 25)? */

/* Editor Control Block and Editor INformation Block */
/* ===== */

EDCB    edcb;
EDINFB  edinfb = {
    0, /* Size in bytes. 0 = Use MAX_EDSIZE */
    0, /* Cursor at first character in text */
    0, /* Block start at 1st character */
    0, /* Block end at 1st character so no
        marked and displayed block. */
    ED_INSERT | ED_TAB, /* Insert = ON and Hard Tabs = ON */
    80 /* Right Margin at 80 */
};

/* Dynamic memory block with text */
/* ===== */

int    TextHandle = 0; /* Handle to block with text */
int    TextPara = 1; /* # of paragraphs allocated in initproc */

/* Empty command table */
/* ===== */

COMTABLE maincomtable [] = {
    0, 0, 0 /* Key, Procedure called, Parameter */
};

```



```

/* SETUP Control Block */
/* ===== */
/* Save Setup saves these items */
SETUPCB setuplist[] = {
    &MainFkeys,    sizeof(FKCB), /* Main set of function keys */
    &EdFkeys,      sizeof(FKCB), /* Editor function keys */
    phoneno,       sizeof(phoneno), /* The Phone number */
    &edinfb,       sizeof(edinfb), /* The editor settings */
    &mainposx,     8 * sizeof(int) /* The size and position */
                                     /* of the windows. */
};
#define SETUP_LEN    5

/*****
**                                     **
**          UTILITY FUNCTIONS          **
**                                     **
*****/

/*
** Called after a resize or recolor. Updates mainpos and
** mainsize with their new values and resets the viewport.
** Does NOT replace the text after a resize/recolor.
*/

void far mainredraw( void )
{
    getwinpos( &mainposx, &mainposy);
    getwinsize( &mainsizex, &mainsizey);
    setscooperel(1, 1, 1, 1);
}

/*
** Disconnects, updates the menu system, and sets newstat so
** that the status line reflects the change of state.
*/
void disconnect( void )
{
    message(NIL, (char *)TXTdiscon); /* TXTdiscon in PRO file */
    modemhangup();
    modemexit();
    winkill(); /* Kill the message window */
    hideclass(Online); /* Remove menu items with
                        class(Online) in the PRO file */
    unhideclass(Offline); /* Show menu items with
                           class(offline) in the PRO file */
    newstat = TRUE;
    online = FALSE;
}

/*****
**                                     **
**          MENU ACTIVATED FUNCTIONS          **
**                                     **
*****/

void hangup( void )
{
    disconnect();
}

/*
** Called when you use F10 Send.
** Transmits all the data in the editor and echoes it to the screen.
*/
void send( void )
{
    char *p;
    char *pend;

```

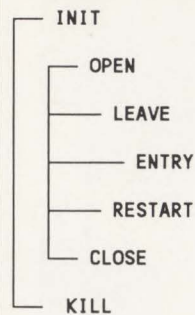


Figure 2. A task may be viewed as a set of procedures grouped in pairs and called by the kernel at appropriate times.

The next pair of procedures is **openproc** and **closeproc**. The kernel calls **openproc** when the user first opens the task from the main menu. **openproc** usually fills in the windows initialized by **initproc**. **closeproc** is called when the user presses Esc to exit the application, and allows the program to save the user's files and to close databases (**closeproc** may be left blank).

The last pair of procedures is **leaveproc** and **restartproc**, which are called when the user presses Ctrl-Alt or Esc to leave SideKick Plus, or Ctrl-Alt to restart SideKick Plus. Although you often leave these procedures empty, you can use them to open and close databases, or to automatically save data.

The heart of the task is **entryproc**. This procedure is a continuous loop that reads a character from the keyboard and stops when the Esc key is pressed. **entryproc** is often a very simple procedure, such as:

```

while (TRUE)
{
    c = readchar((PROC)NIL);
    if (c == ESCAPE)
        break;
}

```

Let's look briefly at task scheduling, which is handled by the kernel. One of the kernel's responsibilities is to manage the keyboard. Whenever the program asks the kernel for input from the keyboard, the kernel presumes that the user wants to activate a

continued on page 48



# Quattro lets 1-2-3 users do more without having to learn more

If you know how to use 1-2-3<sup>®</sup>, switching to Quattro<sup>™</sup> is a snap. There's no back-to-school. It's not that 1-2-3 and Quattro are the same; they're not. But you can tell Quattro to behave in familiar ways, and the sense of familiarity you'll enjoy is more like the one you feel while driving home, but in a brand new sports car.



## *Quattro goes faster and does things better*

1-2-3 moves along pretty well, but Quattro flies right by because it does things differently and it does them sooner rather than later. Quattro's recalcs for example, are intelligent, which means it only recounts the numbers that count and doesn't slowly and unnecessarily recount the whole spreadsheet. And SQZ!<sup>®</sup> Plus for Quattro automatically compacts and expands your spreadsheets by up to 95%.

Quattro gets things done faster including inserting or deleting rows and columns, paging, or painting presentation-quality graphics.

Quattro lets you print your presentation-quality graphs without having to buy a separate graphics package. It gives your work the completely professional look.

## *Quattro, the new Professional Spreadsheet for only \$199.95*

Quattro is a dramatic improvement in spreadsheet technology, graphics, speed, power, and price. It's only \$199.95 and comes with our unique 60-day money-back guarantee and free technical support. To make both you and your work look good, get Quattro today.

### 60-Day Money-back Guarantee\*

For the dealer nearest you, a brochure, or to order,

**Call (800) 543-7543**

\*Customer satisfaction is our main concern; if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.  
All Borland products are trademarks or registered trademarks of Borland International, Inc. 1-2-3 is a registered trademark of Lotus Corp. SQZ! is a registered trademark of Symantec Corp./Turner Hall Publishing Division.  
Copyright ©1988 Borland International, Inc. BI 1181A

QUATTRO



## SIDEKICK PLUS

continued from page 46

new task. This is a major presumption. It means that any task you write must allow keyboard input most of the time and be ready to switch tasks anytime the user requests a new task from the keyboard.

As you can see, the kernel handles many important jobs in any SideKick Plus application. If the kernel is the heart of the program, where is the action?

### THE ACTION

The action of the SideKick Plus task lies in procedures that are called by the kernel in response to input from the keyboard. For example, if F10 is pressed, the kernel pops up the menu for the task. The programmer uses the command table and the .PRO file to define which procedures the kernel calls, the contents of the menu, and the definition of the task's shortcuts.

**Command table.** The command table is the simpler method to implement, and does not let the user redefine the task's shortcuts. A command table consists of a number of entries, each containing three fields:

```
COMTABLE      DemoComTab [] = (
  CTRLA,  spdmp,  1,
  CTRLB,  test,   55,
  CTRLC,  demo,   42,
  CTRLD,  dyndmp, 1,
  0,      0,      0
);
```

We'll look at the first entry in this command table as an example. The first field (containing a constant, **CTRLA**) is the key that activates the procedure defined by the second field (containing a procedure name, **spdmp**), with the parameter defined by the contents of the third field (the value 1). The constant **CTRLA** was defined earlier in a header file as being equivalent to ASCII Ctrl-A. When the user presses Ctrl-A, the procedure **spdmp** is invoked with a parameter value of 1. Each key that is pressed is checked for a match against each entry in the command table. The last entry in the command table is zero-filled to indicate the end of the table.



Figure 3. MiniCom accepts a phone number from the user through a prompt defined in the .PRO file.

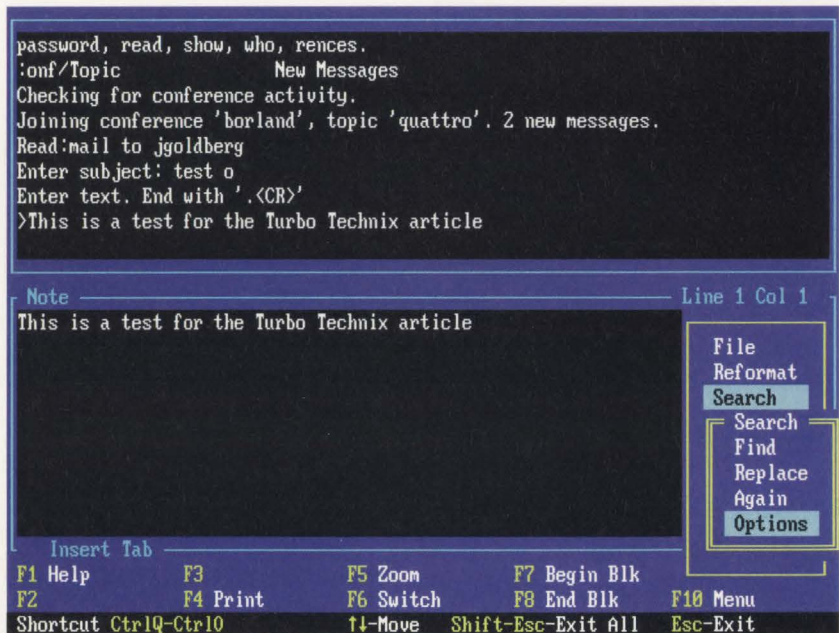


Figure 4. MiniCom in action, connected to an online service. The open edit window contains a short message that has been transmitted to the service.

**The .PRO file.** The .PRO file contains all of the menus for the task, together with the task's shortcuts and the procedures it calls. For easy translation into foreign languages, put any user messages into the .PRO file.

The .PRO file makes designing a menu system easy. You don't even need to write that **read-integer** or **readstring** routine for the nth time. For example, to read

in a filename, use the command **strleaf**. The following portion of the .PRO file in Listing 1

```
MENU(M_dial, " Number "):
" _____ "

    strleaf(phoneno,,,dial);
puts Number in the main menu
with a single line below it that
```



```

p = reclaimmem( TextHandle ); /* Get memory with Texthandle */
pend = p + edinfb.bytes;      /* How big is it? */
for ( ; p < pend; )          /* Send it.... */
{
    if (modemwrite(*p))       /* .... To the modem */
        writechar(*p++);     /* .... To the window */
}
releasemem( TextHandle );    /* Don't need memory any more, */
/* allow Kernel to swap it out. */
}

/*
** Called when you press F9 or F10 Note Edit.
** Opens an editor window with the text stored
** in the block referenced by TextHandle.
*/

void edit( void )
{
    int rc;

    winopen(edwindow);
    rc = eddynedit(&edcb, &edinfb, "Note", TextHandle, TextPara);
    if (rc)
        TextPara = rc;
    getwinpos( &edposx, &edposy);
    getwinsize( &edsizeX, &edsizeY);
    winclose();
}

/*
** Called when you press F10 Dial and enter a number.
** The number is in variable phoneno, if it is valid
** If the user pressed Enter dial the telephone number that
** he has entered in the input field. This has been stored in
** phoneno. If a connection is made the menu system and status
** lines are updated so they reflect the status change.
*/
PROC dial( int NotEnter )
{
    int rc;

    if (NotEnter)
        return POP_QUIT; /* Close menu if not the Enter key */

    /* The following sets Modem parameters;
       change for different configurations */

    modeminit( BITS7|STOP1|PAREVEN|BAUD1200|COM1|
               PXONXOFF|MODENA|DUPFULL|PDATA|PTONE,
               0, 0, 0, 0 );

    rc = modemdial(phoneno);
    if (rc == OK) /* Dialed OK so update */
    { /* status line and menus. */
        newstat = online = TRUE;
        hideclass(Offline); /* Remove class(offline) menu items */
        unhideclass(Online); /* Add class(online) menu items */
        return POP_QUIT; /* Close the menus */
    }
    error(TXTErr,
          (char *)((rc == ILLEGALNO) ? TXTillegalno : TXTphoneErr) );
    beeperror(); /* Put up error msg and beep */
    wait(); /* Wait for any keystroke */
    winkill(); /* Kill the message window */
    modemexit();
    return POP_STAY; /* Keep the menus open */
}

```

prompts the user for a text string. (See Figure 3, which shows a screen display of this process in action.) When the user presses Enter in the string, the kernel places the string into **phoneno** and calls procedure **dial** to check the string for validity.

The .PRO file also specifies such things as whether the module is a task or a service, and the task's number. Listing 1 contains the .PRO file that accompanies MINICOM.C.

### ADDITIONAL INGREDIENTS THAT MAKE UP A TASK

A task must be able to do two other things before winning its SideKick Plus spurs: resizing windows and saving setups. Although both features are optional, it is not difficult to make all the windows resizable, or to save setup information to the SideKick Plus .EXE file.

Let's look at window resizing first. Each time the user changes the size of the window, the kernel calls a far procedure specified by the Window Resize Control Block or WRSCB. This procedure adjusts the window contents to match the new size of the window, and also performs any other adjustments, as shown below:

```

getwinpos(&mainposx,&mainposy);
getwinsize(&mainsizeX,&mainsizeY);
setscoperel(1, 1, 1, 1);

```

This code updates the global variables containing the position and size of the window, and makes text output sent to the window conform to the new size.

Setup saving is equally easy to implement, and is similar to window resizing. You create a SETUP Control Block, or SETUPCB, containing the addresses and sizes of whatever you wish to save. Then, when the user decides to save a setup, the kernel calls the procedure **setupproc**, which saves the SETUPCB and the setup data to disk as part of the SKPLUS.EXE file.

### THE MECHANICS

Now that we've looked at different parts of the programming side of a SideKick Plus module, let's put them together.

*continued on page 50*



1. Compile the .PRO file with the APIP (Application Program Interface Preprocessor) compiler, to produce header files for inclusion into your C program; and assembler or C files that you compile separately from the C task.
2. Compile the C task, including the header files.
3. Link the C task, the API library, and the compiled APIP file.
4. Make the resulting .EXE file into a .BIN file by using EXE2BIN.
5. Use the Install program to make your newly created .BIN file part of the SideKick Plus executable file, SKPLUS.EXE. The .BIN file no longer needs to be present for execution, and can be erased or archived.

#### A SAMPLE APPLICATION

MiniCom is a real SideKick Plus application, written in less than 400 lines of Turbo C code. It is a simple communications program with a 10K notepad that lets you write messages offline, and then transmits them to your online service. Like any SideKick Plus communications application, MiniCom is independent of the type of modem connected to the system, and supports the 9600 bps Telebit Trailblazer just as easily as a 110 bps Teletype.

You can see MiniCom in action in Figure 4. The source code is given in Listing 2, MINICOM.C.

A fully compiled and linked MiniCom module is available on CompuServe. To make MiniCom part of your SideKick Plus executable file, do the following:

- Download MINCOM.ARC from CompuServe.
- Use ARCX to expand the file into its component parts: MINICOM.BIN, MINICOM.C, MINICOM.PRO, and MINICOM.SYS.

```

/*****
**
**      APPLICATIONS ENTRY POINTS CALLED BY KERNEL
**
*****/

/*
** The Kernel calls initproc when you load SKPLUS.EXE.
** It initializes the main and editor windows, the editor,
** and finally allocates a dummy memory block of one
** paragraph to store the text from the editor.
*/
void  initproc()
{
    mainwindow = wininit(          /* Main window initialization */
        mainposx, mainposy,       /* Initial position and size */
        mainsizex, mainsizey,
        COLOR_PALETTE,          /* Initial colour */
        WIN_FRAME_SINGLE, WIN_OPT_ESC,
        &mainwrscb,              /* Name of the WRSCB */
        (char *)TXTOffLine,      /* Initial PRO file status line */
        maincomtable,           /* Name of command table */
        &MainFkeys );           /* Name of function key table */

    edwindow = wininit(          /* Editor window initialization */
        edposx, edposy,          /* Initial position and size */
        edsizex, edsizy,
        COLOR_PALETTE,          /* Initial colour */
        WIN_FRAME_SINGLE, WIN_OPT_ESC,
        &edwrscb,                /* Name of WRSCB */
        (char *)TXTedLine,      /* Initial status line
                                     in PRO file */
        maincomtable,           /* Name of command table */
        &EdFkeys );            /* Name of function key table */

    edinit( &edcb, edwindow, &EdFkeys, MAX_EDSIZE);

    allocatemem( &TextHandle, 1); /* Alloc some memory for later use */
    releasemem( TextHandle);      /* Allow it to be swapped out */
    edinfb.bytes = 0;
}

/*
The Kernel calls Killproc when you unload SideKick Plus from memory.
*/
void  killproc()
{
    if (online)
        disconnect();
}

/*
** Called by the Kernel when you first open the application.
*/
void  openproc()
{
    winopen(mainwindow);
    if (!online)
    {
        hideclass(Online);
        setscope( 1, 1, 1, 1);
        clearscope();          /* Clears the window */
    }
    else
        hideclass(Offline);
}

/*
** Called by the Kernel when you Esc out of the application.
*/
void  closeproc()
{
}

```



```

/*
** Called by the Kernel when you press Ctrl-Alt to
** leave SideKick Plus.
*/
void leaveproc()
{
}

/*
** Called by the Kernel when you press Ctrl-Alt to
** activate SideKick Plus.
*/
void restartproc()
{
}

/*
** Called by the Kernel when you save the setup.
*/
void setupproc()
{
    savesetup( setuplist, SETUP_LEN, SETUP_VERSION);
    savepalette( COLOR_PALETTE);
}

/*
** Handler called by the kernel while the application is
** waiting in readchar for more characters. It will first
** update the status line, if necessary, after which it copies
** all characters from the modem until the user presses a key.
** The handler then returns to the kernel so this in turn can
** return to the application through readchar with the key.
*/
int readfrommodem( void )
{
    char mc;

    if (newstat) /* New Status line required? */
    {
        newstat = FALSE;
        winnewstatus((char *)
                    (online ? TXTonLine : TXToffLine));
    }
    for (;;)
    {
        if (online && modemread(&mc, 0))
            writechar(mc);
        if (keypressed())
            return FALSE;
    }
}

/*
** While the application is open this function is executing.
** The Kernel calls entryproc after openproc completes the
** open operation. It must contain a continuous loop that
** calls readchar.
*/
void entryproc()
{
    int c;

    while (TRUE)
    {
        readfrommodem();
        c = readchar( readfrommodem );
        if (c == ESCAPE)
            break;
        else if (online)
            modemwrite((char)c);
    }
}

```

- Be sure the .BIN and .SYS files are in the SideKick Plus directory.
- Unload SideKick Plus from memory.
- At the DOS prompt, type: INSTALL MINICOM.SYS.

This procedure makes MiniCom part of SKPLUS.EXE, and rebuilds SideKick Plus into its default configuration. MINICOM.SYS is the batch file that automatically installs the new module by merging the .BIN file into memory, rebuilding the library file SKMAIN.BIN to include MiniCom, and finally rebuilding your SKPLUS.EXE. If you later redesign SKPLUS.EXE by running Install again, you'll find MiniCom on the Design New SKPLUS menu.

## CONCLUSION

This article is by no means a complete treatment of the SideKick Plus API. We have only touched on the SideKick Plus architecture and the process by which a SideKick Plus task is created. The list of services available through the API is long and rich, and describing them is more fitting for a book than for a single magazine article. In future issues of *TURBO TECH-NIX*, we will present more information about SideKick Plus's services and operation, plus further details of task development and more sample applications. ■

---

*Jeffrey Goldberg is Product Manager for Borland's European Research and Development, and is a member of the SideKick Plus Research and Development team. He can be reached on CompuServe at 76127,355, and BIX as JGOLDBERG. Steven Boye is a member of Borland's Research and Development staff dedicated to SideKick Plus.*

---

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



# MAKING THE `switch()`

Pick a path, any path, with Turbo C's `switch()` statement.

Kent Porter

What makes computer programs smart is their ability to make decisions and act upon them. The simplest kind of decision-making is either/or: if a condition exists, do X; otherwise, do Y. In C, we might write this as

SQUARE ONE

```
if (condition)
    x();
else
    y();
```

where `x()` and `y()` are functions elsewhere in the program. But not all decisions have either/or outcomes. Sometimes, as in a user's menu selection, there are many alternative actions based on a single condition. For those instances, C provides a special kind of multiple-choice statement called `switch()`.

If you've programmed before in BASIC or Pascal, it is helpful to think of `switch()` as a cousin to BASIC's `ON..GOTO..` or `ON..GOSUB..`, to Turbo Basic's `SELECT CASE` statement, or to Pascal's `CASE` statement. They're all the same idea, but C—more flexible than other languages—allows certain things with its `switch()` statement that the others do not.

## THE CONCEPT OF `switch()`

The concept behind `switch()` is that of an enumerated set of actions based on some controlling variable. Since that sounds dreadfully theoretical, let's tie it to a simple real world example:

```
if today is
    Saturday:    sleep in;
    Sunday:      go to church;
    Any other day: go to work;
```

Here, `today` is the controlling variable. Following its evaluation are alternatives to be done depending on what day of the week `today` is.

The controlling variable is called the *selector*, and the list of its possible values are the *cases*. Note that the above example has two specific cases (`Saturday` and `Sunday`) and a catchall case (`any other day`). Later we'll see how to deal with catchall cases using `switch()`.

Let's look at an actual `switch()` written in C. Suppose `n` is always a digit from 0 to 3, and we want to print its spelled-out equivalent on the screen. That is, when `n = 0`, print "zero" and if `n = 1`, print "one," etc. The `switch()` construction is:

```
switch(n) {
    case 0: cputs ("zero");
            break;
    case 1: cputs ("one");
            break;
    case 2: cputs ("two");
            break;
    case 3: cputs ("three");
            break;
}
```

The opening statement, `switch(n)`, says that you want to evaluate `n` and jump to one of the cases following based on the outcome of `n`. The curly braces are always required (even if there's only one case) to identify the extent of the `switch()` structure.

The keyword `case` signifies the start of each alternative. It's followed by one potential value of the selector, terminated by a colon. Then comes the code that you want executed for that case; a call to `cputs()` in this example.

Note the keyword `break`. It means, "This is the end of this case." When execution reaches a `break` statement, it jumps to the closing curly brace, bypassing all the intervening cases and exiting from the `switch()`. The last case doesn't need to have a `break` statement, since it merely transfers control to the curly brace immediately following, but it's a good idea to include it. You'll see why later.

Listing 1 is a simple program that expands on the example given above. It asks you to type some numbers, or the letter `q` to quit. Every time you type a number, the program responds by printing its English equivalent on the screen. There are some subtle



differences between the original example and Listing 1, as we'll see.

### THE `getch()` FUNCTION

The case variables are enclosed in single quotes. Why? Because `getch()` captures your keystrokes not as numbers, but as ASCII characters, which it assigns to the `char` variable `d`. The case specifiers must be of the same data type as the selector, so we specify them as the characters '0', '1', etc., instead of specifying numerics.

The Turbo C function `getch()`, by the way, accepts one character from the keyboard and does not echo it back. That's why only the processed result of your input (the English equivalent) appears on the screen, instead of your actual input.

Note also that the `break` statements appear on the same line as the `cputs()` statements. The Turbo C compiler doesn't care if multiple statements are on the same line, and since these are all simple cases that do essentially the same thing, putting `break` on the same line with `cputs()` reduces the program length without decreasing readability.

When you run the program, try typing something besides numbers. If you press any non-numeric key except `q`, nothing happens. This is because the `switch()` cases don't anticipate any values for `d` except numbers. Conclusion: The `switch()` statement only acts on the cases it explicitly expects; it ignores any case that it doesn't recognize.

That's why nothing happens if you press some alphabetic key such as `p` or `v`. Failing to find a case for dealing with those values of `d`, the `switch()` statement merely jumps to the closing curly brace.

But what if you want `switch()` to recognize unanticipated cases? For that situation you can use the catchall case discussed next.

### LAST RESORT

A *catchall* case is the case of last resort: "Since I can't do anything

else with the selector, I'll . . ." The catchall case is like `else` in an `if` statement.

Catchalls can detect errors. In the context of Listing 1, pressing any key except one of the numbers is an error. In that event, instead of doing nothing, you might want to sound a beep to signal the user that an error has occurred.

The `switch()` statement makes a special provision for catchalls. It is the keyword `default`, which must always be the last selection in the case list. Its format is:

```
default: <statements>;
```

Listing 2 (NUMS2.C) adds the catchall case to the `switch()` and sounds a beep at the console when the user presses any non-numeric key.

Before we proceed further, let's pause a moment to consider some peculiarities of the `switch()` statement.

### ABOUT SCALARS

*Scalars* are sets of values that have a regular, predictable sequence. An example is the unsigned integers (0, 1, 2, 3, . . ., *n*). Another is the signed integers, which in Turbo C encompass the set of numbers -32768, . . ., -1, 0, 1, . . ., 32767. Yet another example is the ASCII characters, ranging in value from 0 through 255. You can also regard the members of an *enumerated type* as scalars. For example

```
typedef enum {sun, mon, tue, wed,  
             thu, fri, sat} days;
```

in which `sun=0`, `mon=1`, and so on.

In contrast to scalars, there are values with fractional parts (types `float` and `double`), and aggregate types such as structures, unions, bitfields, and arrays. None of these has a regular, incremental series of values.

The selector in a `switch()` statement must be a scalar. It cannot resolve to some wishy-washy value like a floating point number or the combination of structure elements, because that's too complex for any compiler to deal with. Therefore the data types of `switch()` selectors in Turbo C are limited to:

- int (signed or unsigned),
- char (signed or unsigned),
- long (signed or unsigned), or
- an enumerated type

Trying to use any nonscalar type as a `switch()` selector causes the Turbo C compiler to dig in its heels and refuse to cooperate.

Although scalars are series, their cases need not be handled in any particular order within a `switch()`. For example, in Listings 1 and 2, we could have coded the order as:

```
switch (d) {  
  case '9': cputs ("nine"); break;  
  case '0': cputs ("zero"); break;  
  case '5': cputs ("five"); break;  
  etc.  
}
```

In general, this kind of haphazard order isn't a good idea because it makes programs hard to read and understand. However, when a very large `switch()` is inside a loop and you're concerned about program efficiency, you achieve a slight improvement by placing the most frequently occurring cases near the top of the list. This is because the `switch()` statement hunts through the cases seeking a match for the current selector value. The fewer cases it has to search, the quicker the code will run.

Next we'll consider an instance in which you have to specify the cases out of natural sequence.

### FLOW-THROUGH CONTROL

Suppose you're writing a financial accounting system for a small business. In determining its requirements, you find that this company needs to run a routine general ledger job every day. Additionally, the first and the fifteenth of the month are paydays, and the company bills on the tenth. You can specify this monthly agenda in pseudo-code as follows (for more information on pseudo-code, see "Binary Engineering," *TURBO TECHNIX*, November/December, 1987):

*continued on page 54*



```

/* NUMS.C: Spells out keyed numbers to illustrate switch() */
#include <stdio.h>

main ()
{
char   d;

puts ("Type some numbers, then q to quit\n");
do {
d = getch ();          /* get a keystroke */
switch (d) {          /* act on it as follows */
case '0': cputs ("zero "); break;
case '1': cputs ("one "); break;
case '2': cputs ("two "); break;
case '3': cputs ("three "); break;
case '4': cputs ("four "); break;
case '5': cputs ("five "); break;
case '6': cputs ("six "); break;
case '7': cputs ("seven "); break;
case '8': cputs ("eight "); break;
case '9': cputs ("nine "); break;
}
} while (d != 'q');    /* repeat until getting 'q' */
}                      /* then quit */

```

## LISTING 2: NUMS2.C

```

/* NUMS2.C: Enhancement of NUMS.C to illustrate catch-all case */
#include <stdio.h>
#define BEEP 7          /* ASCII beep character */

main ()
{
char   d;

puts ("Type some numbers, then q to quit\n");
do {
d = getch ();          /* get a keystroke */
switch (d) {          /* act on it as follows */
case '0': cputs ("zero "); break;
case '1': cputs ("one "); break;
case '2': cputs ("two "); break;
case '3': cputs ("three "); break;
case '4': cputs ("four "); break;
case '5': cputs ("five "); break;
case '6': cputs ("six "); break;
case '7': cputs ("seven "); break;
case '8': cputs ("eight "); break;
case '9': cputs ("nine "); break;
default: putchar (BEEP); break; /* catch-all */
}
} while (d != 'q');    /* repeat until getting 'q' */
}                      /* then quit */

```

## MAKING THE switch

continued from page 53

```

if date is
1 or 15:  do payroll;
          do daily job;
          end of case;
10:      do billing;
          do daily job;
          end of case;
Any other: do daily job;
end of cases;

```

The translation into C presents a couple of surprises:

```

switch (date) {
case 1:
case 15: payroll ();
         daily ();
         break;
case 10: billing ();
default: daily ();
}

```

This is an example of *flow-through control*, the ability of the `switch()` statement to proceed from one case into the next. See what happens: **case 1** does nothing, but since it lacks a `break` statement, control flows through to **case 15**. Therefore **case 1** and **case 15** are multiple entry points into the same process.

The **default** case and **case 10** are a variation on this theme. On the tenth of the month, **case 10** is selected and the billing is done. After that, control flows to the **default** case, which is the normal daily job. We know that the daily job will get executed on the tenth because there's no `break` statement in **case 10**, and thus nothing to stop control from flowing through to the next case. This makes the **default** case a subset of **case 10**, even though it's the one that gets selected most often.

We could have written **case 10** the same as **case 15**, substituting `billing()` for `payroll()`. The way it's written is a shortcut that saves a little code.

Note that there's no "flow-around" control; you can't jump from one case to another, bypassing intervening steps. That's why **case 15** calls the daily job; there's no way for it to leapfrog over **case 10** into the default.

Listing 3 is a simple program that puts this `switch()` structure to



work. For each date you type in, the program shows which activities it does on that date. You can stop it by typing 0.

Using the multiple flow-throughs lets you construct action hierarchies. For example, if the company using this program decides to move its billing date to the fifteenth, you can rearrange the **switch()** cases as follows:

```
switch (date) {
  case 15: billing ();
  case 1: payroll ();
  default: daily ();
}
```

Now on the fifteenth, you get billing, payroll, and the daily job; on the first you get payroll and the daily; on all other days you get the daily job only. This is because no matter which case you select, you always flow through to all the subsequent cases.

The use of flow-through control can get you into trouble if you're not careful. You might insert a new case between two that are logically sequential (e.g., between **case 10** and **default**) without realizing that the one above lacks a **break** statement. Suddenly your program goes crazy. The best prevention against this is to heavily comment flow-through cases.

As your C programs increase in complexity, you'll confront more and more situations where a choice cannot be answered plainly by a simple **if..then** decision. The **switch()** statement handles those multiple-choice situations simply and elegantly. Learn to use it, and your programs will never find themselves standing in the crossroads, wondering which way to go. ■

---

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

---

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

### LISTING 3: ACCTG.C

```
/* ACCTG.C: Demo of flow-thru in switch() statement */
#include <stdio.h>

/* LOCAL FUNCTIONS */
void daily (void)
{
  puts ("Doing daily job");
} /* ----- */
void payroll (void)
{
  puts ("Doing payroll");
} /* ----- */
void billing (void)
{
  puts ("Doing billing");
} /*----- */

/* MAIN PROGRAM */

main ()
{
  int  date = 0;

  do {
    printf ("\n\nEnter a date (or 0 to quit)... ");
    scanf ("%d", &date);
    if (date != 0)
      switch (date) {
        case 1:          /* flow to case 15 */
        case 15: payroll (); /* flow from case 1 */
                    daily ();
                    break;
        case 10: billing (); /* flow to default case */
        default: daily (); /* flow from case 10 */
      } /* end of switch () */
    } while (date != 0);
  } /* ----- */
```



# MAINTAINING PROGRAMS WITH MAKE

Set up an "instruction manual" using MAKE, and large programs assemble themselves.

Reid Collins

Turbo C offers a completely integrated development environment that satisfies the needs of most amateur and professional C programmers. After you have written enough small- and medium-size C programs to feel comfortable with the language, you may wander into the arena of developing large programs, possibly as a way of making a living. When you do, look closely at the command-line environment: it's designed to help you handle the subtle and not-so-subtle problems associated with large projects.

An important component of the command-line environment is the MAKE utility program. MAKE.EXE is a program builder/maintainer that automates the process of constructing a program from its component parts. Large programs are usually divided into a number of program modules that are linked together to produce executable programs. Individual modules can all be written by the same programmer, but it is more likely that a number of programmers will each contribute different pieces to the whole, and that the set of programmers on the project may change over time.

If you have commercial aspirations, note that the successful programs in today's highly competitive marketplace must both provide the solutions needed by their users *and* be easy to understand and use. Even programs for your own personal use should be designed with the goals of utility and ease of use in mind. Why not do it right?

Ease of use is obtained through simple external design. To achieve this goal, programs are usually very complex internally. Our task as programmers is to keep the internal complexity under control. That's where MAKE comes into play. In essence, MAKE is a tool for managing complexity.

## FEATURES OF MAKE

The MAKE utility is a program that reads a set of instructions and interprets them in light of implicit

and explicit rules in order to build and maintain a program or a set of programs.

Figure 1 shows a general view of a program-maintenance process for a program written in C. A set of source files, FILE1.C through FILEN.C, and some header files are the sources from which the object files are built. The PROG.EXE file is created by linking the object files with standard runtime library modules and possibly some custom library modules.

MAKE controls the running of the compiler and the linker. The instructions used by MAKE are provided in a *makefile*, which consists of several required and optional elements. The next section describes these elements in detail. In brief, the makefile contains lists of dependencies that specify the relationships between source files and target files, and rules that describe how source files are translated and combined to produce the target files.

MAKE can also build and maintain documents in operating environments that use separate programs for editing and formatting document files. The UNIX operating system, for example, offers the **vi** screen editor and the **nroff** and **troff** text formatters. Under DOS, the combination of Emerging Technology's EDIX screen editor and WORDIX text formatter permits documents to be assembled automatically by the Turbo C MAKE program.

## MAKEFILES

The default name for the MAKE instruction file is MAKEFILE. If MAKE is invoked without a file argument, it looks for MAKEFILE in the current directory. You can use other names if you wish. This is necessary if you have more than one makefile in the same directory. I usually name the makefile for the program it builds and give it a .MK or .MAK extension. A makefile for the program FRAMUS, for example, would be FRAMUS.MAK.



As Figure 2 shows, a makefile consists of some required and optional elements. The only required element is at least one *dependency statement*, which is an *explicit rule*. All other elements, such as *comments*, *implicit rules*, *macro definitions*, and *directives*, are optional.

**Comments.** Although comments are optional, it's a good idea to include descriptive comments in makefiles just as you should in source code files. Descriptive comments help you and others who read your code to understand its purpose and the details of its implementation.

Each comment is preceded by a pound sign (#) on the same line. The comment can start anywhere on the line and terminates automatically at the end of the line. Here are some examples of comments in makefiles:

```
#
# The makefile for program FRAMUS
#
```

```
# default rules
```

```
MODEL = s      # memory model
```

Comments cannot be split across lines. Use a separate comment-start delimiter for each comment line.

**Rules.** As noted earlier, there are two kinds of rules: explicit and implicit. *Explicit* rules specify the relationship between a target and the source or sources upon which it depends. For example, a linkable object file typically depends upon a C source file and possibly some header files. The line

```
file1.obj:    file1.c header.h
```

indicates that FILE1.OBJ is built from FILE1.C and a local header file called HEADER.H. There can also be a dependency upon some of the standard header files, too, but these files normally do not change within the lifetime of a version of the compiler, so standard header files are not usually listed in dependency lists.

MAKE decides which objects should be remade according to the date and time stamps of

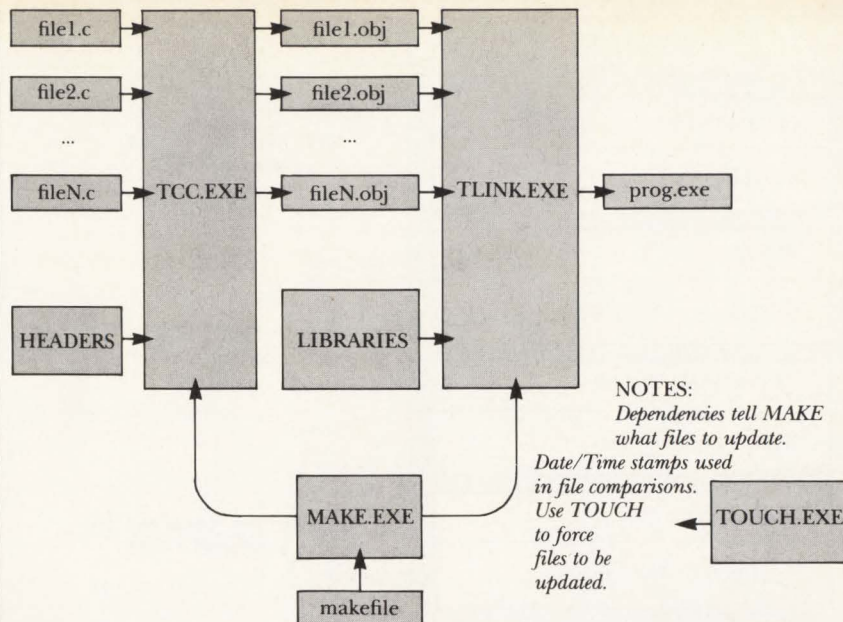


Figure 1. Building and maintaining programs with MAKE.

related files. DOS maintains the date and time a file was last modified as a pair of two-byte integers in the directory entry for the file. As shown in Figure 3, when MAKE sees a dependency, it gets the date/time stamps of the target and source files and compares them. If any source file is newer (carries a more recent date/time stamp) than the target file, the target is remade according to the specified rule.

*Implicit* rules are generalizations of specific (explicit) rules. An implicit rule describes how one type of file relates to another type.

A prime example is the relationship between an object file (.OBJ extension) and the source file from which it is created (.ASM, .C, etc.). The implicit rule

```
.c.obj:
    tcc -c -ms $<
```

tells MAKE that an object file is created from a C source file by the TCC compiler program. The `-c` option limits the process to a simple compilation step. Without it, TCC would call the linker to create an .EXE file in addition to the .OBJ file.

A comparable rule for the creation and maintenance of an object file from an assembly language source file looks like this:

```
.asm.obj:
    masm $<;
```

The exact form of the implicit rule depends on the programming tool that performs the required translation and the operation you want to perform. The mysterious looking `$<` token in these implicit rules is one of the predefined macros supported by Turbo C MAKE. This one yields the full pathname of the specified target file.

**Macros.** A macro is essentially a short-hand notation for programmers. Something small, such as a short name or even a single keystroke, is defined to have a meaning that usually would involve considerably more typing. MAKE accepts macros that are defined by the user in addition to several predefined macros.

To define a macro in a makefile, simply type the name, an equal sign, and the definition. Here, for example, is a macro that identifies the `\TCINCLUDE` directory for my Turbo C setup:

```
INCLUDE = c:\tc\include
```

The string on the left side of the equal sign is the *macro name*. The string on the right is called the *expansion text*. After a macro has been defined, you can use it in MAKE instructions by using the following symbolic notation. The

*continued on page 58*



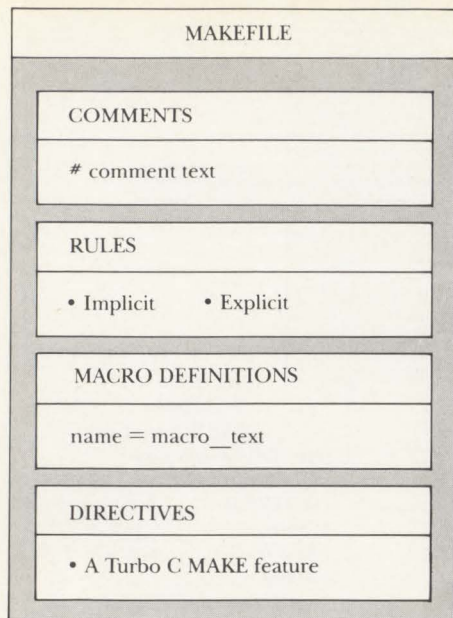


Figure 2. Elements of a makefile.

Macro Name	Description
\$*	Base filename. Removes the extension part of the filename and returns the full pathname up to the base filename.
\$<	Full filename. Yields the full pathname including any leading path information, the filename, and extension.
\$:	Filename path. Removes the filename and extension from a full pathname and yields the path to the directory that contains the file.
\$.	Filename and extension. Strips off any leading path information and yields only the filename and extension parts.
\$&	Filename only. Yields only the filename part of a pathname by stripping off the leading path information and the extension.
\$d(<name>)	Defined test. If the macro <i>name</i> is defined, this macro yields a value of 1; otherwise the value is 0.

The first five of the predefined macros take a target filename and yield all or a portion of it when evaluated by MAKE. The sixth macro is an extension that determines whether a name is currently defined.

Table 1. Predefined macros in Turbo C MAKE.

## MAKE

continued from page 57

macro name is surrounded by parentheses and preceded by a dollar sign:

`$(INCLUDE)`

When MAKE runs, the macro invocation shown above is replaced by its expansion text. MAKE would operate as if you had directly typed `c:\tc\include`.

I encourage the use of macros because the practice tends to

improve makefile readability and simplifies the task of changing multiple occurrences of the same text—just edit the macro definition. Then any and all invocations of the macro will automatically get the new text when MAKE runs. The technique used in makefiles is analogous to using `#define` directives in C source files.

Several predefined macros are supported by MAKE. Table 1 shows each of the predefined macros and its purpose. All but one macro are filename macros that are used in rules to return full

or partial names from associated dependency lines. Given the path name `c:\tc\project\source.c`, here is what the first five predefined macros yield:

```
$* -> c:\tc\project\source
$< -> c:\tc\project\source.c
$: -> c:\tc\project\
$. -> source.c
$& -> source
```

(We'll discuss the sixth predefined macro after we define MAKE directives.)

**Directives.** The Turbo C MAKE utility offers something that other MAKE programs do not: *directives*. Turbo C MAKE directives are similar in purpose and implementation to C preprocessor directives. The most obvious difference is that MAKE directives are signaled by an exclamation point as the first character on a line. The C preprocessor uses the pound sign, which MAKE already uses as a start-comment delimiter.

The array of MAKE directives is impressive. There is a directive for file inclusion (`!include <filename>`) that permits the text of other makefiles to be brought into the including makefile. Conditional directives (`!if..!elif..!else..!endif`) permit a wide range of decision-making opportunities. The `!if` and `!elif` directives take expressions that are formed by using C-like expression syntax. Each expression is evaluated as a signed long integer. The standard C unary (`~`, etc.), binary (`<=`, `&&`, etc.), and ternary (`?:`) operators may be used to form compound expressions.

The sixth predefined macro, `$d`, takes a macro name invocation after it, and yields 1 if the macro is defined or 0 if it is not. The `$d` can be used in an expression as part of an `!if` or other MAKE directive to control some aspect of the program-building process.

The following MAKE lines determine whether the macro `LASER` has been defined. If it has, a special version of a printer program is created for a laser printer. Otherwise, a generic version of the print program is created. The



macro **OBJS** is defined as a list of object files and **LIB** is the path of the standard library directory.

```

!if $d(LASER)
# make the laser version
lpr.exe: lpr.obj $(OBJS)
    tlink $(LIB)\c0s lpr $(OBJS), ...
!else
# make the generic version
pr.exe: pr.obj $(OBJS)
    tlink $(LIB)\c0s pr $(OBJS), ...
!endif

```

The directives accepted by the Turbo C MAKE utility provide more power and flexibility than the MAKE programs provided as standalone programs or with other C compilers.

### THE MAKE COMMAND LINE

The easiest way to use MAKE is to put the instructions in a file called MAKEFILE and type **MAKE** at the DOS prompt.

**Options.** If the directory in which MAKE is being invoked contains more than one makefile, you can use the **-f** option to specify the makefile to be used. The command

```
make -flpr.mak
```

tells MAKE to use the file LPR.MAK as the makefile. This version of MAKE does not permit any space between the option letter (**f**) and the filename.

Other MAKE command-line options include:

- **-s** Run silently—when used, MAKE doesn't print commands, it just executes them
- **-D** Define named identifiers, such as **-DDEBUG**, to turn on debugging
- **-U** Undefine a previously defined identifier
- **-n** Display commands but do not execute them
- **-I <directory>** Search <directory> in addition to the current directory for include files
- **-?** or **-h** Print a help message

**Termination.** While any command executed by MAKE is running, you can press Ctrl-Break (or Ctrl-C) to terminate processing. If a command spawned by MAKE fails, MAKE quits so that you can see the error and warning messages and take appropriate action.

Sample makefile

```

# implicit rule
.c.obj:
    tcc -c -ms $<
...
# dependencies
file1.obj: file1.c header.h

```

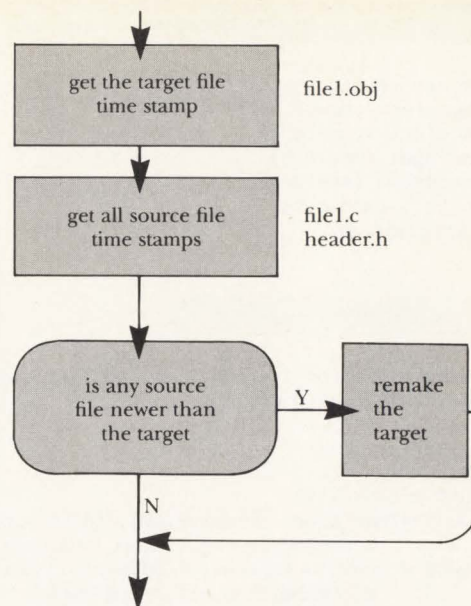


Figure 3. Interpreting rules in makefiles.

### A SAMPLE MAKEFILE

The Turbo C package includes a version of MicroCalc, which is a simple but effective spreadsheet program. The entire source is provided along with a file called MCALC.PRJ (Listing 1), a *project-make* file that is designed for use by the Turbo C integrated environment. Our purpose is to write a makefile that does the same job as MCALC.PRJ, only in the Turbo C command-line environment. We will use the MicroCalc sources because they are available to all Turbo C users and afford us a representative project-control situation.

The seven lines in MCALC.PRJ identify the modules that comprise the MicroCalc program. The first six lines identify object modules that are to be recreated from their respective sources. Each depends upon the header file MCALC.H in addition to the associated C source file.

The seventh item in the project module list is an explicit object filename that tells Turbo C to simply use an existing object module. This was done because the C source file for the MCMVSMEM module contains inline assembler statements. When TC.EXE compiles the source file, it has to create a temporary assembler source file and then call an assembler, such as

MASM, to produce the required object file. Because Borland cannot be certain that a Turbo C user has a copy of MASM, the company has provided the pre-assembled MCMVSMEM.OBJ file.

To use the project-make file, we must be operating in the integrated environment (TC.EXE). It might be necessary or desirable to operate in a batch environment instead, primarily for the purpose of automating the entire process of preparing a product for testing or distribution. For example, I have worked on projects that contain hundreds of thousands of source-code lines that take up to five hours to "rebuild." Ideally, I need to be able to start such a lengthy rebuilding process at the end of a work day and come in the next day to find the job completed. Running MAKE (which calls TCC) from a batch file provides the needed flexibility.

Listing 2 shows the text of a makefile that is suitable for use from the command line and from within batch files. It contains many of the features supported by the Turbo C MAKE utility. The *definitions* section defines macros for the memory model, the path to the library directory, and lists of objects and sources.

*continued on page 60*



## LISTING 1: MCALC.PRJ

```
mcalc (mcalc.h)
mcpaser (mcalc.h)
mcdisplay (mcalc.h)
mcinput (mcalc.h)
mcommand (mcalc.h)
mcutil (mcalc.h)
mcmvsmem.obj
```

## LISTING 2: MAKEFILE

```
# makefile for the MCALC spreadsheet program
# 9/15/87, Reid Collins

# definitions
MDL = s
LIB = c:\tc\lib
OBJS = mcalc.obj mcpaser.obj mcdisplay.obj mcinput.obj \
      mcommand.obj mcutil.obj mcmvsmem.obj
SRCS = mcalc.c mcpaser.c mcdisplay.c mcinput.c \
      mcommand.c mcutil.c mcmvsmem.c

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

# dependencies
mcalc.exe: $(OBJS)
    tlink @linklist

$(OBJS): $(SRCS) mcalc.h

# end makefile
```

## LISTING 3: LINKLIST

```
c:\tc\lib\c0s mcalc mcpaser mcdisplay mcinput mcommand mcutil mcmvsmem
mcalc
nul
c:\tc\lib\maths c:\tc\lib\emu c:\tc\lib\cs
```

## MAKE

*continued from page 59*

The *default rules* section provides the implicit rule that tells MAKE how to produce an object file from a C source file. The lines in the *dependencies* section are the explicit rules that identify the relationships among various files. MAKE reads the lines, checks time stamps of the files, and remakes any target file that is out-of-date with respect to its sources.

Consider the following rule:

```
$(OBJS): $(SRCS) mcalc.h
```

This says that an object in the list of objects representing the **OBJS** macro depends on a related source file in the list. The source file is represented by the **SRCS** macro and the header file is **MCALC.H**. If either the source file or the header file carries a more recent date than the object file, the object will be remade. We use the default rule because there is no explicit rule in the makefile. Notice that it is not necessary to specify each object separately.

One last detail: The number of filenames and other parameters that must be passed to the **TLINK** program exceeds the length supported by the DOS command line. To get around this problem, use a separate linker-response file, **LINKLIST** (Listing 3), which you invoke by adding the line

```
tlink @linklist
```

in the dependency for **MCALC.EXE**.

## NOW IT'S YOUR TURN

The MAKE utility and the software project-management philosophy that it was designed to support both have their roots in the UNIX operating system. A natural extension of MAKE is to use it as the primary building block in a complete software generation system. This system can maintain software and also manage version and access control for large, multi-programmer projects. But that's a topic for another time. ■

---

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

---

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



# BUILDING FAR POINTERS WITH MK\_FP

Touch any point in memory using a pointer generated by this simple macro.

Michael Abrash

Turbo C provides a set of macros for getting at any part of your PC's memory without resorting to assembly language or complex C code. The **MK\_FP** macro, contained in the DOS.H header file, is particularly interesting because it lets you build far pointers for accessing any address in the PC's one megabyte address space. *Far pointers* consist of paired segment:offset values, where the address pointed to is  $((\text{segment} \times 16) + \text{offset})$ . *Near pointers* consist of offsets only, and can only address 64K within Turbo C's default data segment.

The syntax of **MK\_FP** is

```
<farptr> = MK_FP(<segment>, <offset>);
```

where **farptr** is a far pointer of any type, **segment** is an unsigned integer specifying the segment portion of the pointer, and **offset** is an unsigned integer specifying the offset portion of the pointer.

Listing 1, MAKEFAR.C, builds a far pointer to the start of the color text display memory at B800:0000. Every byte at an even address is set to the asterisk character. Every byte at an odd address is set to the value 2. These odd bytes are attribute bytes, which describe how the characters at the preceding even bytes will be displayed, and the value 2 causes the previous characters to be displayed in green on a black background. Note that to work with a monochrome text mode display buffer you must change **MK\_FP's** segment parameter 0xB800 to 0xB000.

The value generated by **MK\_FP** can only be assigned to a far pointer. Function **FillScreen** in Listing 1 explicitly declares **DisplayMemoryPtr** to be a far pointer able to point anywhere in the PC's memory—no matter if a near or far data memory model is being used for the program as a whole. ■

Michael Abrash is an Engineering Fellow working on advanced graphics projects for Video 7, Inc., of Fremont, California.

Listings may be downloaded from CompuServe as FARPTR.ARC.

LISTING 1: MAKEFAR.C

```
/*
 * Program to demonstrate the use of the MK_FP macro by building
 * a far pointer and using it to fill color text display memory
 * with green asterisks.
 */
#include <dos.h>          /* contains MK_FP macro */
#define FILL_LENGTH      2000 /* # of characters in an 80x25
                             screen */
#define COLOR_TEXT_SEGMENT 0xB800 /* segment at which color text
                             mode display memory begins */

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

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

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

/*
 * Sample program to call FillScreen.
 */
main()
{
    FillScreen('*', 2);
}
```



# COMMENT NESTING

Keep in mind the nesting habits of comments when hatching your latest hack.

Roger Schlafly

Standard C and Pascal do not allow comments to be nested. As soon as the compiler sees the begin comment indicator, it ignores all characters until it reaches the end comment indicator, as shown here:

PROGRAMMER

```
/* In C, this whole sentence is
a comment, even though an extra
'/*' appears in it. */
```

This is a serious nuisance if you are in a habit of "commenting out" large sections of code. For example, if you have this bit of code

```
if (!++*s++) /* cryptic C code */
    longjmp(buf)
```

and you comment it out, you get:

```
/*
if (!++*s++) /* cryptic C code */
    longjmp(buf)
*/
```

Unfortunately, the compiler sees the first `/*` as the start of the comment, and thinks that the comment ends with the `*/` that follows **C code**. It compiles the `longjmp(buf)` statement and then gets a syntax error at the next `*/`. The portable way to temporarily remove the code is to use the preprocessor:

```
#if 0
    if (!++*s++) /* cryptic C code */
        longjmp(buf)
#endif
```

This gets the job done, even if there are preprocessor statements within the section of code. Lattice C popularized another method, by supplying an option (through a compiler switch) allowing comments to be nested. With this option, beginning and ending comment indicators must balance one another, just as parentheses must do within an

expression. Turbo C also has such a switch, `-C`, for Lattice compatibility. The default is to disallow nested comments, but with `-C` in force, comments may be nested, Lattice-style.

Turbo Pascal users don't have an option to nest comments, but the compiler does support two logically distinct kinds of comment delimiters:

```
{ This is a comment. }
(* This is a comment too. *)
```

A Turbo Pascal comment must be terminated with the same kind of delimiter character that started the comment. In other words, `}` must close a comment begun with `{`, and `*` must close a comment begun with `(*`. So if you consistently use one style of comments for documenting your code, and another style for disabling portions of your code, everything will work fine:

```
(*
REPEAT { quit when Check fails }
    AllDone := Check(Arg1,Arg2);
    Arg2 := Arg1
UNTIL AllDone;
*)
```

Turbo Prolog uses the C comment delimiters and C comment conventions, but unlike Turbo C, there is no Turbo Prolog compiler option for allowing comments to be nested. Turbo Basic has broken away from total dependence on line numbers, but to a great extent it is still a line-oriented language. A comment in Turbo Basic begins anywhere on a line at the first occurrence of a single-quote character and continues to the end of the current line. Other single-quote characters may be used in a comment line, so it is possible to comment out a line of code that includes a comment by placing the single quote at the beginning of the line:

```
'PRINT I,J,K 'Display the values on the screen
```



Unfortunately, to comment out a block of lines, each line in the block must be commented out separately, with its own single quote in column 1.

### A C PUZZLE

Now for a C puzzle. Suppose you want a C program that will compile and run whether or not you select the option to nest comments; and suppose you want the program to behave differently depending on whether the option was selected. It is tricky to write such a program because `/*` and `*/` will almost always be syntax errors unless the compiler recognizes these delimiters as the beginning or end of a comment.

## **Turbo C has a switch, -C, that allows comment nesting.**

The solution is to use a macro that pastes tokens together. The official ANSI C way to do this is, of course, supported by Turbo C. I've demonstrated the method in Listing 1. When you compile and run the code, the message it prints will depend on whether or not you used the `-C` compiler switch (or the equivalent environment menu option) to allow nested comments. Because it's a puzzle, I won't explain how the solution works in detail. Once you unravel it, however, you will know just about all there is to know about nesting comments in Turbo C. ■

---

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

---

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

#### LISTING 1: COMMENT.C

```
#define CONCAT(x,y) x ## y
/**/
#define NoNesting          1
CONCAT(/,*)
*/

/* This is a comment. */
#if !NoNesting
/* And this is a /* nested */ comment. */
#endif

#include <stdio.h>

int cdecl main(int argc, char **argv)
{
    #if NoNesting
        puts("Comments do not nest.");
    #else
        puts("Comments nest.");
    #endif
    return 0;
}
```



# EXPERT SYSTEM DESIGN FROM A HEIGHT

The next expert you consult may already be sitting on your desk!

Michael Floyd

English Lit 101—down the hall and to the left. Chem 1A—next building over, second floor. Expert Systems and Turbo Prolog—you've come to the right place, and the front row is available.

**SQUARE ONE** If you're a procedural programmer who has been looking for a reason to get Turbo Prolog, look no more. If you already own Turbo Prolog and have been waiting for the right application to use it, the wait is over. *Expert Systems and Turbo Prolog* is the theme in this special issue, and class is now in session. The curriculum is a complete tutorial on expert systems and how they can be developed in Turbo Prolog. And you have the opportunity to learn from some of the best.

In this article, you'll get a bird's-eye view of expert system design as we cover what expert systems are, how they're used, and what features are common to most knowledge-based systems. Once you've gotten the basics here, you need only turn the page to learn about the heart of an expert system—the inference engine—from Keith Weiskamp. Another flip of the page takes you on a journey through the world of knowledge representation as I discuss object-oriented programming with frames. Finally, Safaa Hashim takes you where no Turbo Prolog programmer has gone before, in a unique discussion of metalogic and expert systems.

## WHAT IS AN EXPERT?

Before we talk about expert systems, we should define what an expert is. An expert possesses a skill or knowledge set that represents mastery of a given field. We call upon experts to solve problems or to perform tasks for which we lack expertise. For instance, we may call a plumber to fix a clogged drain, a doctor to diagnose an illness, or a stock broker to advise us on stocks.

When an expert approaches a problem, he or she obtains as much information about the problem as possible. Then the expert quickly rules out extraneous information and boils the problem down to a class of problems that he or she is familiar with. Finally, the expert applies rules of thumb, or *heuristics*, to solve the problem.

## EXPERT SYSTEMS

Like their human counterparts, expert systems possess a domain of knowledge and a set of rules for using that knowledge to resolve a given situation.

An expert system typically consists of a knowledge base containing all of the information about a specific problem, an inference mechanism that performs reasoning and problem-solving, and a user interface that acquires new information and reports solutions. Figure 1 shows the relationship between the three parts of a typical expert system.

We group expert systems into two major areas in terms of the problems they solve—*synthesis* and *analysis*. A synthesis system generally combines knowledge with procedures to perform actions. An example of a synthesis system is XCON (also known as R1), which configures VAX mainframe computers.

An analysis system, on the other hand, breaks down a problem in order to study or analyze a given situation. MYCIN, developed at Stanford University and used to diagnose bacterial infections, is a classic example of an analysis system.

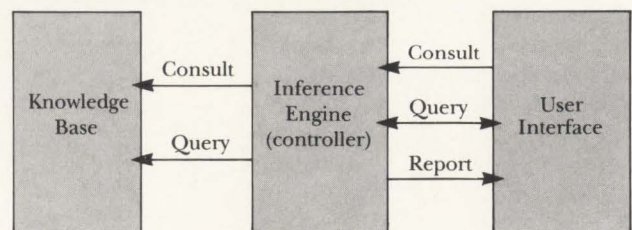


Figure 1. The relationship between the three parts of a typical expert system.



Some expert systems incorporate both synthesis and analysis. VM (also developed at Stanford University) monitors patients in an intensive care unit, identifies possible alarm conditions, reports on those conditions, and recommends treatment for them.

Expert systems can also be grouped into applications categories. Most AI researchers categorize these applications as interpretation, prediction, diagnosis, design, planning, monitoring, debugging, repair, instruction, and control. However, applications do cross boundaries, so this categorization method can fall short in many cases.

### EXPERT SHELLS

In addition to expert systems that solve specific problems, other systems, called *expert shells*, solve classes of problems. Expert shells load and consult different knowledge bases, and apply the same rules of inference to solve a variety of problems.

As an example, EMYCIN is an expert shell that was developed from the original MYCIN expert system mentioned earlier.

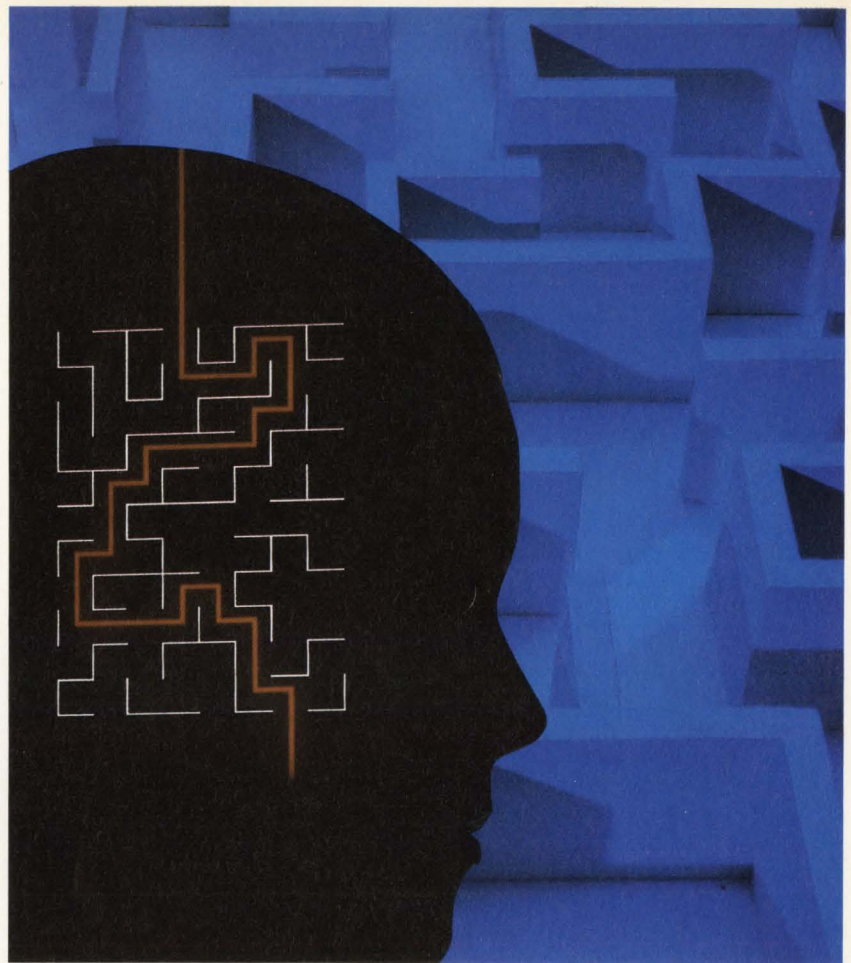
EMYCIN takes the inference engine from MYCIN and applies it to other problem domains. EMYCIN is particularly good at deductive problems involving a large amount of data.

### RULE-BASED SYSTEMS

By far, the most popular problem-solving strategy used in expert system design is rule-based. A rule-based system uses *production rules* to specify a set of conditions and a conclusion or action that follows from those conditions. The following *if-then* rule represents a production rule:

```
Rule N  IF condition 1 is true
        AND condition 2 is true
        AND ...
        AND condition n is true
        THEN some conclusion or action
           follows
```

Production rules are used in either *forward-chaining* or *backward-chaining* systems. A forward-chaining system starts



with a set of conditions and works its way toward the conclusion. In other words, once the rule is selected, the system attempts to prove the first condition, then the second condition, and so on. If all the conditions prove true, then the conclusion follows. When all conditions are satisfied, the rule is *triggered*. When the conclusion is initiated (i.e., when the action in the THEN part of the rule is taken), the rule is *fired*. Because the conclusion follows naturally from the set of conditions, this approach has an inductive quality.

A backward-chaining system starts with the conclusion and works its way back to the condition set. We can see this more clearly by rewriting the if-then rule:

```
Rule N  Conclusion is true IF
        Condition 1 is true AND
        Condition 2 is true AND
        ... AND
        Condition n is true.
```

This approach assumes that the conclusion is true and sets out to prove the conditions. Thus, back-

ward chaining lends a deductive quality to the system.

The Prolog language provides a natural way to write production rules that chain backward from the conclusion (or goal) to the conditions (or antecedents). For instance, in Turbo Prolog we can write a rule to define the state of good health:

```
health(good) if
    temperature(normal) and
    blood_pressure(normal) and
    tests(negative).
```

In attempting to prove that the patient is in good health, Turbo Prolog assumes that the conclusion **health(good)** is true. Turbo Prolog then tries to satisfy each of the conditions to prove the goal.

There are advantages and disadvantages to each chaining method. For instance, a forward-chaining system selects relevant

*continued on page 66*



rules by triggering them. To collect all relevant rules may require looking at every condition in the rule base.

A backward-chaining system chooses only matching rules, so the possibilities are immediately narrowed. However, attempting to satisfy a rule's conditions may lead to a dead end (i.e., a condition in the rule fails).

Some systems do both forward and backward chaining. In some cases, an algorithm selects the best method of chaining. In other cases, the user selects the chaining method to be used.

### GENERATE AND TEST

Another problem-solving strategy we should mention is known as *generate and test*. In this approach, all possible solutions are first generated, then each possibility is tested. Generate-and-test systems are useful only when there is a small solution set. Larger systems must use advanced techniques to limit the solution set. In the interest of efficiency, larger systems must be nonredundant—they should never propose the same solution twice. Since Turbo Prolog provides a natural way to represent production rules, and because rule-based systems have offered greater success in expert system development, we will not cover generate-and-test systems in this issue. We mention them here only for completeness.

### COMMON FEATURES

Despite numerous approaches to expert system design, some features are common to most or all expert systems. One such feature is a query facility, which is invoked when the user consults the computer expert. Generally, the problem domain dictates the query's approach. However, the approach also depends on the

amount of information the system requires from the user in order to solve the problem. For instance, when the user has little information to offer initially, the query system may employ a question-and-answer approach. When the user has a large amount of information to offer initially, the query system may use a natural language processor. If the type of data to be acquired from the user is known, the query facility may display a form that the user must fill in.

Another feature common to all expert systems is a report facility. An expert system must be able to display its findings, make recommendations, and so forth. This report may be a simple answer to a question, or may involve complex screen displays. For instance, a system that deals with large amounts of data may graph the data so that the user can visualize trends.

Although not absolutely necessary, most analysis systems have the ability to explain their logic. An explanation facility is usually broken down into a *how processor* and a *why processor*. The how processor explains how a particular conclusion is reached, acting as a window into the logic of the knowledge base.

In contrast, the why processor explains why a particular action (such as asking a question) is taken, acting as a window into the current state of the consultation.

A *rule language* is another feature of many expert shells. In an expert shell such as EMYCIN, the user can create new knowledge domains by using a rule language very much like our if-then rules. These rules are presented in an English-like style, so that new knowledge domains can be created and maintained more easily.

### ADVANTAGES OF USING PROLOG

The Prolog language provides a natural way to represent production rules. In addition, Prolog's built-in pattern matching (unification) and backtracking facilities make it easy to implement a

backward-chaining inference mechanism.

Turbo Prolog also adds other capabilities. For instance, the implementation of a production rule language similar to the one in EMYCIN would require a parser. The Parser Generator provided in the Turbo Prolog Toolbox could be invaluable in creating such a parser.

Turbo Prolog and the Turbo Prolog Toolbox also allow you to develop a sophisticated user interface quickly and easily. Although I've dealt mostly with theory here, the importance of the user interface should not be underestimated. If you can't interact with the system, or can't understand the system's output, then the system becomes unusable.

### CONCLUSION

When designing an expert system, there are a great many things to consider. First, what kind of expert do you want to create? Will it be an adviser, a troubleshooter, a monitoring and control system, or some other type of system? Second, will the system deal with a specific problem, or a class of problems? These questions in turn affect the way knowledge is represented (see "Suitable for Framing"), how metaknowledge is structured (see "Metalogic and Expert Systems"), and how solutions are inferred (see "Building an Inference Engine with Turbo Prolog"). So, follow us on a journey from human reasoning to computer reasoning as *TURBO TECHNIQ* explores the anatomy of a computer expert. ■

### REFERENCES

- Hayes-Roth, Frederick & D. A. Waterman. *Building Expert Systems*, Reading, MA: Addison-Wesley Publishing Company, Inc., 1983.
- Winston, Patrick Henry. *Artificial Intelligence*, second edition, Reading, MA: Addison-Wesley Publishing Company, Inc., 1984.



# BUILDING AN INFERENCE ENGINE WITH TURBO PROLOG

Data, data everywhere and not a thought to think.

—Jeff Armstrong

Keith Weiskamp



PROGRAMMER

If you've always wanted to unravel the mystery of how expert systems work, you've come to the right place—the inference engine. Fortunately, building an inference engine with Turbo Prolog is much easier than you might think. Turbo Prolog's built-in pattern matching and backtracking features provide a solid foundation for implementing an engine. In fact, such an application really shows off the intrinsic power and flexibility of Turbo Prolog. In this article, we'll focus on the inference engine component of the expert system; however, we'll look at the engine in the context of a complete expert system shell. This approach has one major benefit—it illustrates how the inference engine interacts with the other components of the expert system.

The classic textbook expert system is composed of an inference engine, a knowledge base, working memory, and a user interface. The hierarchy of these components is shown in Figure 1.

The user interface sits on top to communicate between the engine and the user. The knowledge base contains the rules and logic to guide the engine in its reasoning process. Finally, the working memory serves as the warehouse to store the knowledge (facts) that the inference engine gathers as it goes about its reasoning process.

## ORIGINS IN LOGIC

When we infer something, we come to a conclusion by either guessing, speculating, or surmising. For instance, we may infer that it is cold outside because there's snow on the ground. An inference, then, is the process of inferring. In logic, an inference is the process of deriving a strict logical consequence from a given premise. In cases where there is a degree of uncertainty, we can derive the likely consequences based on some degree of probability.

You might now be wondering where the origins of inferential computation come from. Actually, the

father of this technology is Plato—the first true logic programmer. The fundamental techniques used by the inference engine were developed by the ancient Greek philosophers. An inference is nothing more than a component of formal logic. In such a logical system, we can divide the universe into two parts:

1. Relationships (what can be said about the world)
2. Deductions (what can be proven from the known information)

Essentially, the inference process consists of starting with something that is known and deducing something that is unknown. The most popular technique for this transformation is called *modus ponens*. This technique states that if we have two objects, and we are given the following relationships

1. Object 1 is true
2. If Object 1 Then Object 2

we can prove that Object 2 is true. This inference can be expressed in logical symbols as:

$$(\text{Object 1} \wedge (\text{Object 1} \rightarrow \text{Object 2})) \rightarrow \text{Object 2}$$

Believe it or not, this simple principle is the driving force behind the inference engine. Of course, in order to simulate the expertise of a person, the inference engine must also be able to deal with missing information and information that might not be true in all cases. We'll touch on this point again shortly. But first, let's take a closer look at the inner workings of an inference mechanism.

## WHAT'S IN AN ENGINE

When it comes to expert systems, the inference engine runs the show. In fact, an expert system without an inference engine is like an orchestra without a conductor. The engine serves as the link between the knowledge base and the user by performing two important tasks: reasoning and control.

The inference engine simulates the human reasoning process by using a predetermined algorithm for inferring facts from knowledge, usually in the form of rules. In fact, in many cases the inference

*continued on page 68*



## INFERENCE ENGINES

continued from page 67

engine is nothing more than a fancy rule interpreter. The reasoning algorithm is based on a control strategy for selecting rules from the knowledge base. Two of the popular strategies that you've probably come across are *forward chaining* and *backward chaining*. Forward chaining means working toward a conclusion by starting with a set of facts and appropriate assumptions. If a solution can't be reached, the inference engine backtracks and uses other facts and assumptions to try and reach a conclusion. Backward chaining, on the other hand, is the reverse of forward chaining. With this strategy, the inference engine starts with a conclusion or hypothesis and works backward in order to prove it.

The system we will show you incorporates backward chaining. Because of Turbo Prolog's backtracking mechanism, implementing a backward chaining inference engine is really quite easy.

To better understand the function of the inference engine in an expert system, let's consider an example. Assume we have the following rules in a knowledge base

1. IF person has a good employment record  
AND IF person's income is over \$40,000  
AND IF person owns their home  
THEN person qualifies for a home equity loan
2. IF person has been steadily employed for over three years  
AND IF person has good job outlook  
THEN the person has a good employment record

and the following facts are known:

1. John owns his home.
2. John's income is \$50,000.
3. John has worked for the software factory for 10 years.
4. John has a good job outlook.

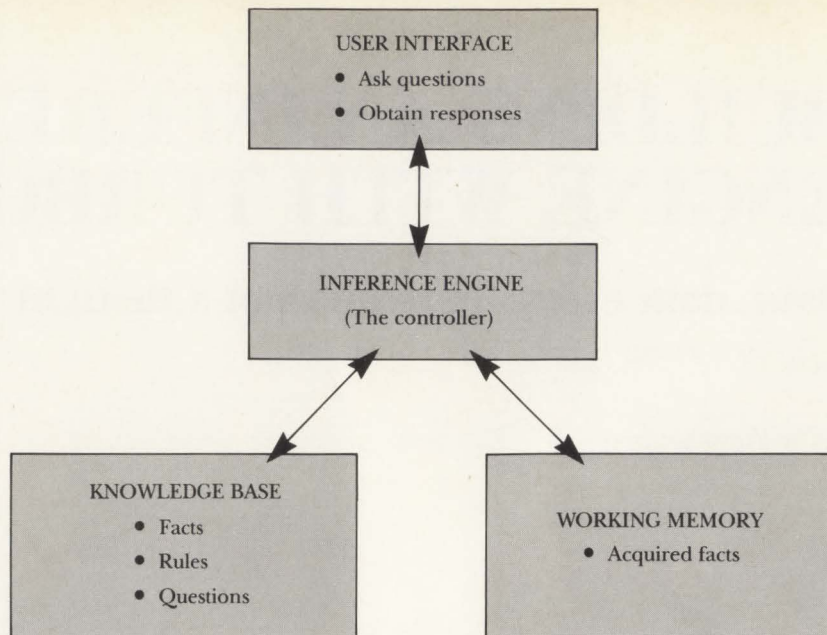


Figure 1. Components of an expert system.

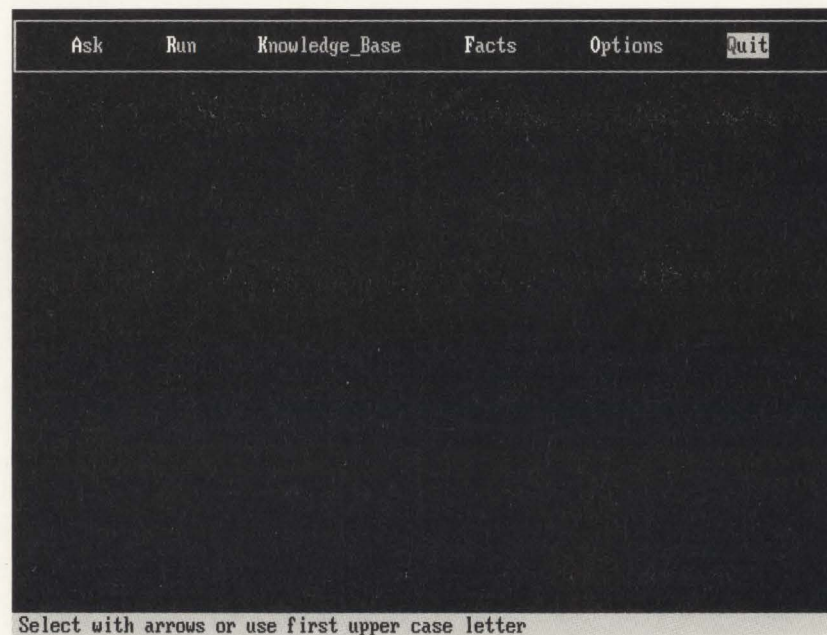


Figure 2. User interface for TPENGINE expert system shell using the Turbo Prolog Toolbox.

With this information, the inference engine can determine if John qualifies for a loan. Essentially, the inference engine's job is to either start with a conclusion and find the facts necessary to support the conclusion, or to start with facts and deduce a conclusion from the known information. In our example, the inference process starts with the conclusion "John can qualify for a loan," and attempts to verify it by using the

rules in the knowledge base. Rule 1 is examined first; however, to solve this rule, rule 2 must also be used. The process of using a rule is called *rule firing*. Note that to solve a rule, the inference engine can either verify facts or fire other rules.

### CONTROLLING THE INFERENCE PROCESS

We've examined the inference component of an inference



engine in detail; however, we've only casually discussed the other component—control. Remember that the inference engine not only deduces unknown facts from known facts, but it also must decide how to select and process rules and facts (knowledge). To make these decisions, the control component performs the following functions:

**Pattern Matching.** This function matches a given rule with the facts stored in the system. Since pattern matching is built into the Prolog language, we don't have to worry about writing a pattern matcher when developing an inference engine. This is one of the strong points of using Prolog to implement expert systems.

**Selecting Rules.** The inference engine uses this strategic algorithm to find the best rules for a given inference. The selection algorithm can be designed in many different ways, and in general is related to the way knowledge is stored in the knowledge base.

**Executing (firing) a rule.** Once a rule is selected, it must be executed (fired). This process involves verifying each component of the rule by asking the user questions, or searching the knowledge base or working memory for known facts.

**Performing Actions.** After processing an inference, the engine usually performs a set of actions, such as updating the knowledge base and working memory. In most expert systems, the facts acquired during an inference session are saved so that the inference engine can use them at a later time.

Selecting rules is the main task of the control component. A wide variety of techniques have been used by expert system developers, including the backward chaining and forward chaining search strategies discussed earlier. In our example of the inference process using the two loan qualification rules, the backward chaining method was employed to verify the conclusion or goal: "John can qualify for a loan." This statement

*continued on page 70*

#### LISTING 1: BDESIGN.KB

```
rule("Is book designed well?",is("well_designed"),
    [are_correct("page_numbers"),are_correct("margins"),
    are_correct("type_fonts"),are_correct("art_work")]).
rule("Are chapter fonts ok?",are_correct("chapter_fonts"),
    [is_set("chapter_header"),is_set("chapter_body"),
    are_set("paragraph_heads")]).
rule("Are page numbers ok?",are_correct("page_numbers"),
    [are_on("even_pages","left"),are_on("odd_pages","right"),
    are_in("pages","order")]).
rule("Are page numbers ok?",are_correct("page_numbers"),
    [are_on("even_pages","center"),are_on("odd_pages","center"),
    are_in("pages","order")]).
rule("Are margins ok?",are_correct("margins"),
    [are_consistent("left_right_margins"),
    are_consistent("top_bot_margins")]).
rule("Are code listings ok?",are_correct("code_listings"),
    [are_in("listings","small_font"),is_set("code_type")]).
rule("Are code listings ok?",are_correct("code_listings"),
    [are_in("listings","bold_face"),is_set("code_type")]).
rule("Is code set ok?",is_set("code_type"),
    [has("listings","header"),check("syntax")]).
rule("Is code syntak ok?",check("syntax"),[is_language("pascal"),
    terminate("semicolon","statements")]).
rule("Is code syntak ok?",check("syntax"),[is_language("prolog"),
    terminate("period","clause")]).
rule("Are type fonts ok?",are_correct("type_fonts"),
    [has_code("book"),are_correct("code_listings"),
    are_correct("chapter_fonts")]).
rule("Are type fonts ok?",are_correct("type_fonts"),[no_code("book"),
    are_correct("chapter_fonts")]).
question(are_on("even_pages","left"),
    "Are even page numbers on left-hand side?").
question(are_on("odd_pages","right"),
    "Are odd page numbers on right-hand side?").
question(are_on("even_pages","center"),
    "Are even page numbers centered ?").
question(are_on("odd_pages","center"),
    "Are odd page numbers centered?").
question(are_in("pages","order"),
    "Are pages in correct numeric order?").
question(are_consistent("left_right_margins"),
    "Are the left and right margins consistent?").
question(are_consistent("top_bot_margins"),
    "Are the top and bottom margins consistent?").
question(has_code("book"),"Does the book have code?").
question(no_code("book"),"The book does not have code?").
question(are_in("listings","small_font"),
    "Are the code listings in a smaller font the the text?").
question(are_in("listings","bold_face"),
    "Are the code listings in bold face?").
question(has("listings","header"),
    "Do the code listings have the correct header?").
question(is_language("pascal"),
    "Are the code listings in Pascal?").
question(is_language("prolog"),
    "Are the code listings in Prolog?").
question(terminate("semicolon","statements"),
    "Does each statement terminate with a semicolon?").
```



```
question(terminate("period","clause"),
        "Does each clause terminate with a period?").
question(is_set("chapter_header"),
        "Does each chapter have the correct heading?").
question(is_set("chapter_body"),
        "Is the type font consistent for each chapter?").
question(are_set("paragraph_heads"),
        "Does each paragraph have the correct heading?").
question(are_correct("art_work"),
        "Does each chapter have the correct figures?").
```

## LISTING 2: TPENGINE.PRO

```

/*****
File: tpengine.pro
Author: Keith Weiskamp 1987

        tpengine is basic inference engine written in Turbo prolog.
        The inference engine supports multiple knowledge bases (rule
        sets).

Uses: The following tools from the Turbo Prolog Toolbox are needed:

tdoms.pro
tpreds.pro
status.pro
pulldown.pro
boxmenu.pro

*****/

code=8000
include "tdoms.pro"

DOMAINS
        /* names of rule heads and facts */
rule_list = rule_type*
rule_type = is(symbol);
           are_correct(symbol);
           are_consistent(symbol);
           is_set(symbol);
           are_set(symbol);
           has(symbol,symbol);
           has_correct(symbol,symbol);
           are_on(symbol,symbol);
           are_in(symbol,symbol);
           check(symbol);
           has_code(symbol);
           no_code(symbol);
           is_language(symbol);
           terminate(symbol,symbol)

rlst = string*
file = dest

DATABASE

pdwstate(ROW,COL,SYMBOL,ROW,COL) /* pull-down menu state */
querylist(STRINGLIST) /* the active list of queries */
currentquery(rule_type) /* the user selected query */
storedfacts(rule_type) /* facts saved in working memory */
storedfalsefacts(rule_type)
tempfacts(rule_type) /* temporary facts */
tempfalsefacts(rule_type)
```

matches the conclusion of rule 1; therefore, this rule is fired and the inference engine must then verify the conditions:

Does John have a good employment record?

Is John's income over \$40,000?

Does John own his home?

Now the first condition matches the conclusion of rule 2. In this case, rule 2 becomes our new subgoal that must be solved before our original goal can be verified. The conditions for the subgoal are:

Has John been steadily employed for over three years?

Does John have a good job outlook?

Of course, now we're at the end of the line and can verify the subgoal by using the facts in the knowledge base. Once this subgoal is solved, we can return to the first goal.

## GETTING STARTED

The program we'll create consists of an inference engine that can access multiple knowledge bases. The knowledge base we'll use is composed of a set of production rules and questions. The inference engine is wrapped around an interface constructed with some of the tools provided with the Turbo Prolog Toolbox.

Figure 2 shows a screen dump of the program interface. Note that six options are provided: **A**sk, **R**un, **K**nowledge\_\_**B**ase, **F**acts, **O**ptions, and **Q**uit. The first option, **A**sk, allows us to select a query for the inference engine to process.

The queries are taken from the production rules defined in the knowledge base. When we select a query, we are essentially asking the inference engine to help us verify that a given statement or situation is valid. **R**un starts the inference engine, and the **K**nowledge\_\_**B**ase option loads or clears a knowledge base. **F**acts provides access to the working memory of facts that the inference engine uses to process queries. The system that we'll build temporarily stores the facts that it



gathers while processing a query. These facts can be saved so that they are available to the engine when processing other queries. Finally, Options is included to support a trace feature that will allow us to review the reasoning process of the inference engine. To construct the program, let's start with the knowledge base and then work our way up to the inference engine.

## BUILDING THE KNOWLEDGE BASE

A variety of representation schemes are available for building a knowledge base, including rules, frames, scripts, and semantic networks. We'll use production rules in our system because they are easy to process with the inference engine. (For more on knowledge representation and frames, see "Suitable for Framing" elsewhere in this issue. For more on rules in the Turbo Prolog database, refer to "Metalogic and Expert Systems," also in this issue.)

The knowledge base that we'll build to test the inference engine contains some of the rules and facts used by book designers. In fact, these rules are specific to the design of programming books. Figure 3 shows the rules.

Note that each rule in Figure 3 consists of either facts that must be verified or other rules. For example, the first premise of the rule "The type fonts are correct" contains the fact "The book does not have code" and the rule "The code listings are correct." In addition, some of the rules use the **OR** operator to join different premises. The Turbo Prolog implementation of this knowledge base is shown in Listing 1. Study this listing to see how the English-like production rules are represented as database clauses. Note that each **rule** clause represents one of the defined rules. In the cases where rules have more than one set of premises (i.e., rules that use the **OR** operator), a **rule** clause is defined to represent each set of premises.

The knowledge base also contains a series of questions to support the defined rules. Each question is implemented as a database clause of the form:

*continued on page 72*

```
rule(string,rule_type,rule_list) /* rules loaded from a KB file */
question(rule_type,string)      /* questions from the KB */
how(rule_type,symbol,symbol) /* store rules and facts for trace */

/* these files are provided with the Turbo Prolog Toolbox */
include "tpreds.pro"           /* support predicates for toolbox */
include "status.pro"          /* status bar message tools */
include "pull-down.pro"       /* pull-down menu tools */
include "boxmenu.pro"         /* box menu tools */
```

### PREDICATES

```
msg(ROW,COL,STRING)
getquery(StringList)
unique(StringList, StringList)
member(string, StringList, integer)
process_file
inference(rule_type,rlist)
process_rule(rlist,rule_type,rule_list)
process_facts(rlist,rule_list)
getresponse(symbol)
validresponse(symbol,rule_type,rlist)
get_first(rule_list,rule_type)
delete(rule_type,rule_list,rule_list)
process(rule_type)
rmv_rule(rule_type,rule_list,rule_list)
check_ans(symbol,symbol)
add_fact(rule_type)
check_fact(rlist,rule_type)
process_why(rlist)
display_rule(string)
clearfacts
clear_facts
clear_false_facts
clear_temp_facts
clear_temp_false_facts
cleartempfacts
clear_trace
clear_rules
clear_ques
showfacts
showtrace
add(string,rlist,rlist)
convert(symbol,rule_type)
remembertempfacts
remember_facts
remember_false_facts
```

### CLAUSES

```
/* The Inference Engine Clauses *****/
process(Query) :- inference(Query,_),
    rule(Str, Query,_),!,
    write("Your query:\n ",Str,"\nis solved.").
process(Query) :-
    rule(Str, Query,_), !,
    write("Your query:\n ",Str,"\ncannot be solved.).
```



```

RULE: A book is designed well
  IF
    The pages are numbered correctly
  AND
    The type fonts are correct
  AND
    The margins are set correctly
  AND
    The art work matches the text

RULE: The chapter fonts are correct
  IF
    The chapter header is set correctly
  AND
    The text in the chapter body is set correctly
  AND
    The paragraph heads are set correctly

RULE: The pages are numbered correctly
  IF
    (Even page numbers are on the left corner
  AND
    Odd page numbers are on the right corner
  AND
    Each page is in the correct numerical order)
  OR
    (Even page numbers are centered
  AND
    Odd page numbers are centered
  AND
    Each page is in the correct numerical order)

RULE: The margins are set correctly
  IF
    All left margins are consistent
  AND
    ALL right margins are consistent

RULE: The code listings are correct
  IF
    (The listings are in a smaller font than the
    chapter text
  AND
    The code type is set correctly)
  OR
    (The listings are in a bold face style
  AND
    The code type is set correctly)

RULE: The code type is set correctly
  IF
    Each listing has a header
  AND
    The code syntax is correct

RULE: The code syntax is correct
  IF
    (The code is in Pascal
  AND
    Each line of code is terminated by a semicolon)
  OR
    (The code is in Prolog
  AND
    Each clause is terminated by a period)

RULE: The type fonts are correct
  IF
    (The book does not have code
  AND
    The code listings are correct
  AND
    The chapter fonts are correct)
  OR
    (The book does not have code
  AND
    The chapter fonts are correct)

```

Figure 3. Rules to be used in the book designer knowledge base.

## INFERENCE ENGINES

*continued from page 71*

### database

`question(rule_type, string)`

Here the term **rule\_type** refers to one of the facts defined by the rule clauses. The second argument consists of the question the inference engine asks when attempting to verify the fact. As an example, the question

```
question(has_code(book),
        "Does the book have code?")
```

is used to process the fact, "The book has code."

### THE EXPERT SHELL PROGRAM

The complete expert system shell is shown in Listing 2. The program is divided into three components: the inference engine clauses, support clauses, and clauses to process pull-down menu selections. The clauses that make up the core of the inference engine are **process\_inference**, **process\_rule**, **rmv\_rule**, and **process\_facts**. We'll look at these clauses in more detail in a moment, but first let's see how the program is initialized.

Before the user can do anything useful with the system, he must load the knowledge base. Once the knowledge base is loaded, **getquery** is called to construct a list of all of the rules in the knowledge base. This list is further processed by **unique**, which removes all rules with duplicate rule heads. The final result is a list of queries that is stored; the program can later construct a menu of queries from this list. In this sense, the program is dynamic; selecting **Ask** displays queries that reflect the current state of the knowledge base.

After loading a knowledge base, you can select a query with the **Ask** option, and start the inference engine by selecting **Run**.

### INSIDE THE INFERENCE ENGINE

Now we're ready to tackle the heart of the expert system. Starting the inference engine calls **process**, with the user's query as the argument. **process** in turn calls **inference**, which controls the evaluation of the query. The infer-

*continued on page 74*



# New! Introducing Turbo C 1.5— the best optimizing compiler gets even better!

*The professional  
optimizing compiler  
for less than \$100*

Turbo C® is a technically superior production-quality compiler. (Borland's equation solver, Eureka™, is written in Turbo C.) And our Turbo C 1.5 offers a new library of the highest presentation-quality graphics in the industry—the kind you'll see in Quattro,™ our new professional spreadsheet.

And spectacular graphics are just part of the brand-new features. Turbo C 1.5 enhancements also include:

- A professional-quality graphics library of over 70 functions
- A librarian that allows you to build your own object module libraries
- Context-sensitive help for the language and the library routines



Actual photograph of Turbo C graphics displayed on IBM 8514 screen.\*

- Text/video functions, including windows
- 43- and 50-line mode support
- VGA, CGA, EGA, Hercules, and IBM 8514 support
- File search utility (GREP)
- Sample graphics applications
- More than 100 new functions

For professional-quality C at an affordable price, no one else comes close to Turbo C. Because no one can deliver technical superiority like Borland.

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

For the dealer nearest you or to order, call  
**(800) 543-7543**



Minimum system requirements: For the IBM PS/2™ and the IBM® and Compaq® families of personal computers and all 100% compatibles. PC-DOS (MS-DOS®) 2.0 or later. 384K.

\*Artwork metatitle courtesy of Genographics™ Corporation

\*\*Customer satisfaction is our main concern; if within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department, and we will arrange a refund.

All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders. Copyright © 1987 Borland International, Inc.

BI 11658

## It's easy to upgrade to Turbo C 1.5!

Just complete this coupon and mail it with payment before June 30, 1988. Or, call us at (800) 543-7543 and be ready to give our operators your name, credit card number, and the serial number on your Turbo C master disk.

**Turbo C 1.5 Upgrade Price** **\$ 33.50**

CA and MA residents add sales tax \_\_\_\_\_

Shipping and handling \_\_\_\_\_  
in US \$5.00 (Outside US add \$10)

Total amount enclosed \$ \_\_\_\_\_

**Must include your Turbo C serial #** \_\_\_\_\_

Return this coupon and the Turbo C RTL source code registration form from your Turbo C manual along with your payment by March 31, 1988 and receive your Turbo C 1.5 upgrade for free! (No phone orders please.)

**Turbo C 1.5 Runtime Library  
Source Code** **\$ 150.00**

CA & MA residents add sales tax \_\_\_\_\_

Price includes shipping to all US cities. \_\_\_\_\_

(Outside US add \$10) \_\_\_\_\_

Total amount enclosed \$ \_\_\_\_\_

Please specify diskette size  5¼"  3½"

Method of Payment:  VISA  MC  Check  Bank Draft

Credit card expiration date: \_\_\_\_\_ / \_\_\_\_\_

Card # | | | | | | | | | | | | | | | | | | | | | |

Name \_\_\_\_\_

Ship Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_

Zip \_\_\_\_\_ Phone (\_\_\_\_) \_\_\_\_\_

**Mail coupon to:** Turbo C 1.5 Upgrade Dept., Borland International  
4585 Scotts Valley Drive, Scotts Valley, CA 95066

This offer is limited to one upgrade per valid product serial number. Not good with any other offer from Borland. Outside US make payments by bank draft payable in US dollars drawn on a US bank. CODs and purchase orders will not be accepted by Borland.





*continued from page 72*

```

inference(Query,_) :- not(rule(_,Query,_)). /* inference engine */
inference(Query,Rlst) :- rule(No,Query,Cond_list),
    addl(No,Rlst,Nrlst),
    assert(how(Query, rule,query)),
    get_first(Cond_list,Cond),
    inference(Cond,Nrlst),
    process_rule(Nrlst,Cond,Cond_list),!.

process_rule(Rlst,Query,Cond_list):-
    rmv_rule(Query,Cond_list,N1),!,
    process_facts(Rlst,N1).

rmv_rule(Query,Cond_list,N1) :- rule(_,Query,_),
    delete(Query,Cond_list,N1).
rmv_rule(_,Cond_list,Cond_list). /* only facts in list */

process_facts(_, []).
process_facts(Rlst,Cond_list) :- get_first(Cond_list,Prop),
    check_fact(Rlst,Prop), !,
    delete(Prop,Cond_list,New_list),
    process_facts(Rlst,New_list).
process_facts(Rlst,[Query|Cond_list]) :- rule(_,Query,_),
    inference(Query,Rlst),
    process_facts(Rlst,Cond_list).

getresponse(R) :- readln(Ask),
    check_ans(Ask,Rep), !,
    R=Rep;
    nl, write("Try another answer please."),nl,
    getresponse(R).

check_ans(yes,y).
check_ans(y,y).
check_ans(n,n).
check_ans(no,n).
check_ans(why,w).
check_ans(w,w).

add_fact(Fact) :- not(storedfacts(Fact)),
    assert(storedfacts(Fact)).
add_fact(Fact) :- nl, write(Fact," is already stored."),nl.

check_fact(_,Fact) :- storedfacts(Fact),
    assert(how(Fact, fact, memory)).
check_fact(_,Fact) :- tempfacts(Fact),
    assert(how(Fact, fact, memory)).
check_fact(_,Fact) :- storedfalsefacts(Fact), !, fail.
check_fact(_,Fact) :- tempfalsefacts(Fact), !, fail.
check_fact(Rlst,Fact) :- question(Fact,Str),
    write(Str),
    getresponse(Response),
    validresponse(Response,Fact,Rlst).

/* Store all facts (true and false) in temporary memory. */

validresponse(y,Fact,_) :- assert(tempfacts(Fact)),
    assert(how(Fact, fact, user)).
validresponse(n,Fact,_) :- assert(tempfalsefacts(Fact)), !, fail.
validresponse(w,Fact,Rlst) :-
    makewindow(1,7,7,"Why Window",10,20,10,40),
    makestatus(112,"Press any key to continue"),
    process_why(Rlst),
    readkey(_),
    removestatus,
    removewindow, !,
    check_fact(Rlst,Fact).

```

ence engine first locates the rule in the knowledge base that matches the query and then processes the rule by examining each premise of the rule. The algorithm for proving each premise is:

1. If the premise matches one of the rules in the knowledge base, then locate the first premise of that rule and start over with step 1.
2. If the premise is stored as a fact in memory, then the premise is proven. Go to step 4.
3. If the premise is an "askable" fact, then ask the user to verify the fact. If the fact is verified go to step 4.
4. Locate the next premise for the rule and continue with step 1.

This translates into Turbo Prolog as:

```

inference(Query,_) :-
    not(rule(_,Query,_)).
inference(Query,Rlst) :-
    rule(No,Query,Cond_list),
    addl(No,Rlst,Nrlst),
    assert(how(Query, rule,query)),
    get_first(Cond_list,Cond),
    inference(Cond,Nrlst),
    process_rule(Nrlst,Cond,
        Cond_list),!.

```

The first inference clause terminates the recursive inferencing process when a rule's premise does not reference another rule. The second inference clause, on the other hand, contains the core of the inference engine. Here, the first step is to obtain a rule from the knowledge base with a call to **rule**. The argument returns the list of premises, **Cond\_list**. The next step is to add the name of the rule to a rule list that processes "why" responses. This feature, which we'll discuss in the next section, lets us ask the inference engine why it is asking us a particular question. The clause **get\_first** returns the first premise from the list of premises (**Cond\_list**); the inferencing process then continues with the new premise. Note that **inference** keeps calling itself as long as a rule's premise refers to another rule.

You're probably wondering how the inference engine knows if it is



processing a rule or a fact. If a premise matches one of the rules stored in the knowledge base, the inference engine knows it has found a rule and continues to step through the rule list. On the other hand, if a match is not found, the first **inference** clause succeeds. This suspends the recursion built into **inference** long enough to evaluate the facts defined in a rule with a call to **process\_rule**. Let's look at an example.

Assume that we have the following two rules stored in a knowledge base:

```
rule("rule 1",a,[b,c,d]).
rule("rule 2",c,[f,g]).
```

The terms **a** and **c** refer to the rule heads (conclusions) and the terms **b**, **c**, **d**, **f**, and **g** represent their premises (or conditions for satisfying the rule). If we start the inference engine with

```
inference(a,_).
```

the first call to **rule** produces the list of premises **[b,c,d]**. In this case, the first premise, **b**, is interpreted as a fact since there are no other rules in the knowledge base that match this premise. The recursive inference clause then terminates, and **process\_rule** is called with the arguments:

```
process_rule(["rule 1"],b,[b,c,d]).
```

This clause removes the first element from the rule list if it is a rule, and calls **process\_facts** with the rest of the list to verify the premises.

Our expert system shell contains an internal knowledge base, called the *working memory*, which stores the facts that the inference engine validates. The first step in evaluating a fact is to examine this working memory to determine if the fact is already known. If the fact is not stored, the inference engine asks the user to verify the fact. This is handled by the clause:

```
check_fact(Rlst,Fact):-
    question(Fact,Str),
    write(Str),
    getresponse(Response),
    validresponse(Response,
        Fact,Rlst).
```

Here, **question** extracts the appropriate question from the knowledge base and the user is asked to respond to the question by answering "yes" or "no." If our

*continued on page 76*

```
process_why([]).
process_why([Head|Tail]) :- process_why(Tail),
    display_rule(Head).

display_rule(H) :- rule(H,Rprop,_),
    write("\nProcessing rule ",H," ",Rprop),nl.

cleartempfacts :- clear_temp_false_facts, clear_temp_facts.
clear_temp_false_facts :- retract(tempfalsefacts(_)), fail.
clear_temp_false_facts.
clear_temp_facts :- retract(tempfacts(_)), fail.
clear_temp_facts.

clearfacts :- clear_false_facts, clear_facts.
clear_false_facts :- retract(storedfalsefacts(_)), fail.
clear_false_facts.
clear_facts :- retract(storedfacts(_)), fail.
clear_facts.

remembertempfacts :- remember_facts, remember_false_facts.
remember_facts :- retract(tempfacts(Fact)),
    assert(storedfacts(Fact)), remember_facts.
remember_facts.
remember_false_facts :- retract(tempfalsefacts(Fact)),
    assert(storedfalsefacts(Fact)), remember_false_facts.
remember_false_facts.

showfacts :-
    storedfacts(Fact),
    write("\nFact: ",Fact),
    fail.
showfacts.

showtrace :-
    how(Term, Type, Spec),
    write("\nTrace: ",Term, " ", Type, " ", Spec),
    fail.
showtrace.

clear_trace :- retract(how(_,_,_)),
    fail.
clear_trace.

clear_rules :- retract(rule(_,_,_)),
    fail.
clear_rules.

clear_ques :-
    retract(question(_,_)),
    fail.
clear_ques.
```



```

/* Support Clauses *****/
convert(Sym,Term) :- openwrite(dest,"convert.dat"),
    writedev(device(dest),
    write(Sym),
    closefile(dest),
    openread(dest,"convert.dat"),
    readdevice(device(dest),
    readterm(rule_type,Term),
    closefile(dest),
    readdevice(keyboard).

addl(Mem,L,[Mem|L]).

get_first([H|_],H).

delete(_,[],[]).
delete(Head,[Head|Tail],Tail) :- !.
delete(Token,[Head|Tail],[Head|Result]) :- !,
    delete(Token,Tail,Result).

unique([], []).
unique([H|T], Result):-
    member(H, T, _), !,
    unique(T, Result).
unique([H|T], [H|Result]):-
    unique(T, Result).

member(H, [H|_], 1).
member(H, [_|T], Pos):-
    member(H, T, Cpos),
    Pos = Cpos + 1.

getquery(Lst):-
    findall(Str, rule(Str,_,_), Lst).

/* Process pull-down menu options *****/

pdwaction(1,0):-
    querylist(Ql), /* build the query list */
    makestatus(112,"Use arrow keys to select a query"),
    boxmenu(3,5,10,30,7,7,Ql,"Select a query",1,Pos),
    member(Item,Ql,Pos),
    rule(Item,Query,_),
    assert(currentquery(Query)),
    removestatus.

pdwaction(1,0):-
    querylist(_).

pdwaction(1,0):-
    msg(3,5,"Knowledge base not loaded").

```

fact **b** is verified, the inference engine continues to process the rest of the premises (**c** and **d**). When **c** is evaluated, the inference engine discovers that this premise references a fact, and therefore, produces the premise list [**f,g**]. The next step is to evaluate the facts. Once they are proven, the engine backtracks and evaluates the premise **d**. The decision tree for this process is shown in Figure 4. If any of the facts fail, the query is proven false. Keep in mind that the inference engine still supports compound rules. That is, we can define a rule of the form

**c** IF (**f** AND **g**) OR (**h** AND **i**)

which is expressed as:

```

rule("rule 2",c,[f,g]).
rule("rule 2",c,[h,i]).

```

In this case, if premise **f** or **g** fails, the inference engine evaluates the rule by attempting to verify premises **h** and **i**. Since Turbo Prolog automatically backtracks and searches for alternative solutions, we get this feature for free!

**THE WHY AND TRACE FEATURES**

To make our inference engine more sophisticated, we can add a why processor and a tracing feature. The why processor, implemented as

```

process_why([]).
process_why([Head|Tail]):-
    process_why(Tail),
    display_rule(Head).

```

displays the name of the rules in the order they are processed. The rules are displayed in a window as shown in Figure 5. Note that the rule name as well as the rule head are displayed. In this case, the "Why Window" tells us that the inference engine is asking us the question, "Are the left and right margins consistent?" This is to evaluate the rule "Are margins ok?", which is part of the rule "Is book designed well?" In this respect, the "Why Window" helps us to see the hierarchy of the rules being processed by the engine.

To support this feature, we only need to keep a list of the rule names that are being investigated by the inference engine. If the



user selects this option by answering a question with the response "why" or "w," the rules currently stored in the list are displayed in the order they are being processed. Of course, our display of the rules is somewhat primitive; however, you can jazz it up by modifying the `process_why` clause so that the why explanation reads more like real English. (At least we have windows!)

The second feature we'll need is a trace. You can also call trace a "how" processor, since it tells us "how" the inference engine does its reasoning. After a query is evaluated, we can select the trace listed under the `Options` command; we'll then be presented with a window containing the reasoning steps used by the inference engine. Figure 6 shows a sample trace that was generated to solve the top-level query "Is book designed well?"

Note that each entry in the trace contains three components: the head of a rule or fact, a tag to tell us if the trace item is a rule or fact, and a tag to tell us the context of the trace item. For example, the trace statements

```
are_correct("margins") rule query
are_consistent("left_right_margins") fact user
are_consistent("top_bot_margins") fact user
```

are interpreted as:

```
To solve the rule:
The margins are correct
The fact:
Left/right margins are consistent
was verified by the user
The fact:
Top/bottom margins are consistent
was verified by the user
```

Since facts can also be verified by examining the working memory, the trace also displays facts of the form

```
no_code("book") fact memory
```

to indicate that the fact was stored in the working memory.

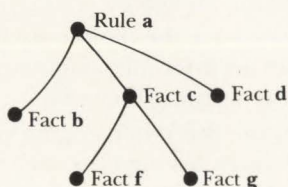


Figure 4. Decision tree showing how rules and facts are checked.

continued on page 78

```
pdwaction(2,0):- /* Solve a query */
    retract(currentquery(Query)),
    cleartempfacts, /* initialize the environment */
    clear_trace,
    makestatus(112,
        "Answer questions with (y)es, (n)o or (w)hy"),
    makewindow(1,7,7,"Query Window",3,5,15,60),
    process(Query),! /* start the inference engine */
    changestatus("Press any key to continue"),
    readkey(_),
    removewindow,
    removestatus.

pdwaction(2,0):-
    msg(3,16,"Query not selected").

pdwaction(3,1):- /* load the knowledge base */
    makewindow(1,7,4,"",3,26,5,30),
    window_str("Enter Knowledge Base:"),
    process_file.

pdwaction(3,2):- /* clear the knowledge base */
    clear_rules,
    clear_ques,
    retract(querylist(_)),
    msg(3,26,"Knowledge Base is cleared").

pdwaction(4,1):- /* display facts stored in memory */
    makewindow(1,7,7,"Facts in memory",3,20,20,50),
    showfacts,
    makestatus(112,"Press any key to continue"),
    readkey(_),
    removestatus,
    removewindow.

pdwaction(4,2):- /* Add a fact to working memory */
    makewindow(1,7,7,"Add a fact",3,20,5,30),
    write("Enter fact to add to memory:\n"),
    readln(Resp),
    convert(Resp,Fact),
    add_fact(Fact),
    removewindow,!.

pdwaction(4,2):-
    removewindow,
    msg(3,43,"Fact cannot be added").

pdwaction(4,3):- /* clear facts */
    clearfacts,
    msg(3,26,"Facts are cleared").

pdwaction(4,4):- /* Store temporary facts */
    remembertempfacts,
    msg(3,26,"Temporary facts are saved").

pdwaction(5,1):- /* About this program */
    makestatus(112,"Press any key"),
    makewindow(1,7,7,"About this program",3,30,5,40),
    write("This program is a inference engine\n"),
    write("Written by Keith Weiskamp....\n"),
    readkey(_),
    removewindow,
    removestatus.
```



To implement the trace, we use the **how** database clause, which is defined as:

```
database
  how(rule_type,symbol,symbol)
```

Every time a rule is processed or a fact is verified, it is asserted into the database. These actions occur in the clauses **inference**, **check\_fact**, and **validresponse**. The trace is displayed by opening a window and calling **showtrace**, which steps through the **how** database:

```
showtrace:-
  how(Term,Type,Spec),
  write("\nTrace: ",Term, " ",
        Type, " ",Spec),
  fail.
showtrace.
```

Again, this code could easily be modified to display the trace in a more user-friendly format.

**USING THE PROGRAM**

To run the program, simply compile and execute Listing 2. However, before you can do anything with the program, you must load a knowledge base. If you attempt to select a query or run the program before loading a knowledge base, the program displays an error message. To test the engine, load the knowledge base BDESIGN.KB shown in Listing 1. Once this knowledge base is loaded, select the Ask option and you'll be presented with a set of queries as shown in Figure 7.

You can also use the **Facts** option to **Show**, **Add**, **Clear**, and **sTore** facts. These options perform the following tasks:

- **Show**: Display all of the facts currently stored in working memory.
- **Add**: Add a fact to working memory.
- **Clear**: Remove all the facts from working memory.
- **sTore**: Save the temporary facts from an inference engine run to the working memory.

When the inference engine verifies facts by asking the user questions, it stores the facts in a temporary database. This database is also examined by the engine as it

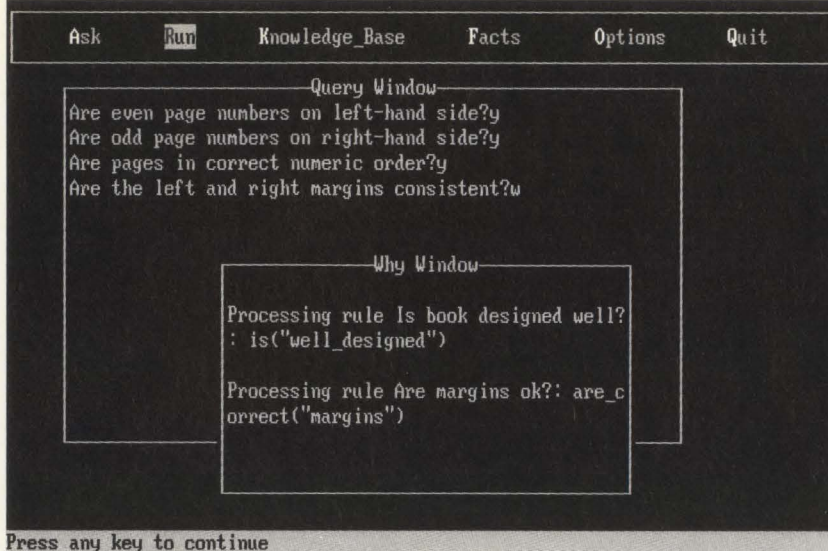


Figure 5. Screen shot showing the output of the "why" processor.

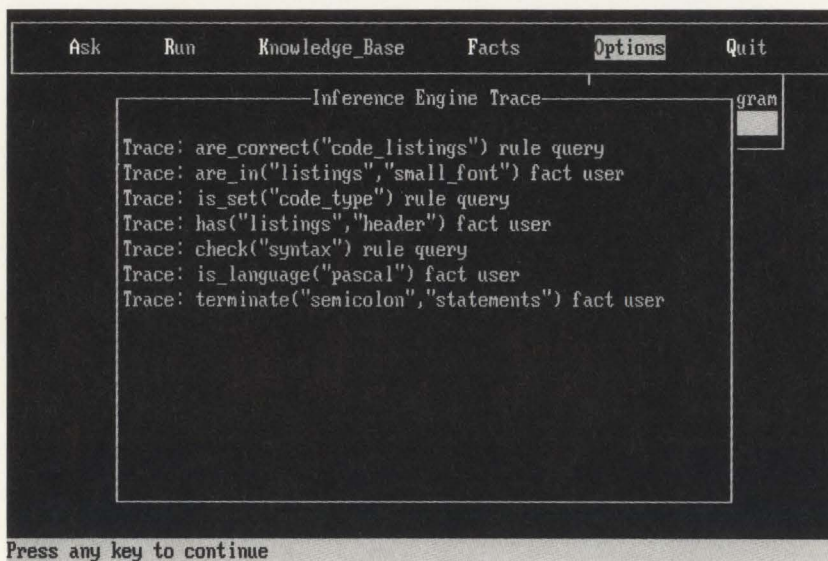


Figure 6. Screen shot showing the output of the trace facility.

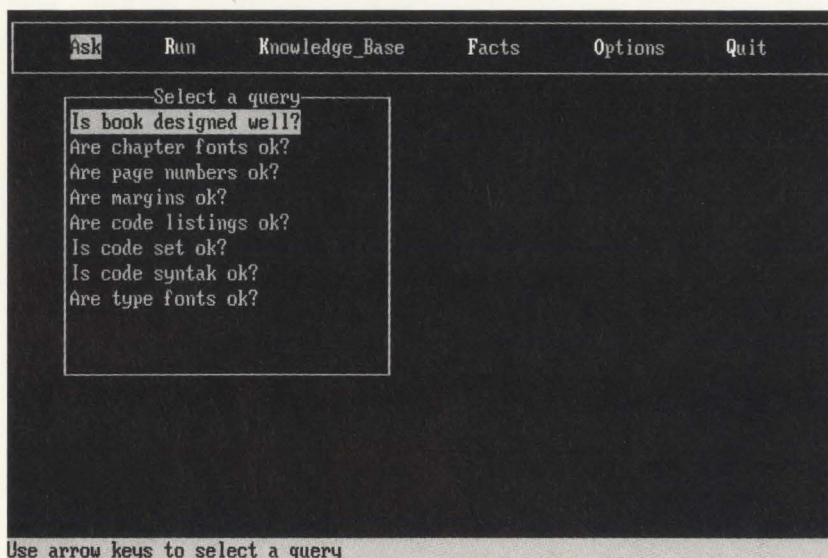


Figure 7. Ask pull-down menu showing sample queries.



processes other facts. Whenever a new query is selected for processing by the inference engine, this temporary database is cleared; if you want to save the facts gathered from a particular query, you should use the `sTore` option to save them.

## CONCLUSION

We have not only constructed a basic inference engine, but we've also put together a working expert system, including the user interface and knowledge base to illustrate how the inference engine operates. The program can be loosely defined as an expert system shell since it is not tied to one particular knowledge domain.

The inference engine is designed to process production rules; however, it could be modified to support other knowledge representation schemes, such as frames. The important thing to realize is that the knowledge base serves as a road map to direct the reasoning process of the engine. In fact, the order of the rules defined in the knowledge base determines the order of the inferencing process. Therefore, if you define your own rule sets, make sure you consider the effect that the order of your rules has on the outcome of the inferencing process. ■

## SUGGESTED READING

- Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. Reading, MA: Addison-Wesley, 1986.
- Frenzel, Louis E., Jr. *Understanding Expert Systems*. Indianapolis, IN: Howard W. Sams & Co., 1987.
- Marcus, Claudia. *Prolog Programming*. Reading, MA: Addison-Wesley, 1986.
- Raphael, Bertram. *The Thinking Computer*. San Francisco, CA: W. H. Freeman and Co., 1976.
- Weiskamp, Keith and Hengl, Terry. *Artificial Intelligence Programming With Turbo Prolog*. New York, NY: John Wiley & Sons, 1987.

*Keith Weiskamp is the editor-in-chief of PC AI magazine, and is co-author of Artificial Intelligence Programming with Turbo Prolog.*

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

```
pdwaction(5,2):- /* Show the trace */
    makewindow(1,7,7,"Inference Engine Trace",3,10,20,60),
    showtrace,
    makestatus(112,"Press any key to continue"),
    readkey(_),
    removestatus,
    removewindow.

pdwaction(6,0):-/* msg(15,10,"Please press the space bar."), */
    shiftwindow(1), removewindow, removestatus, exit.

process_file:- /* read in the knowledge base using consult */
    readln(Str),
    consult(Str),
    getquery(QryLst),
    unique(QryLst, Mlst),
    assert(querylist(Mlst)),
    removewindow.

process_file:-
    nl, window_str("Knowledge Base Can't be loaded"),
    readkey(_),
    removewindow.

msg(R,C,S):-
    makestatus(112,"Press any key"),
    makewindow(1,7,4,"",R,C,5,30),
    window_str(S),
    readkey(_),
    removewindow,
    removestatus.

GOAL

makewindow(1,7,0,"",0,0,24,80),
makestatus(112,
    "Select with arrows or use first upper case letter"),

pulldown(7,
    [ curtain(5,"Ask", []),
      curtain(14,"Run", []),
      curtain(23,"Knowledge_Base",["Load", "Clear"]),
      curtain(43,"Facts", ["Show","Add", "Clear", "sTore"]),
      curtain(55,"Options", ["About This Program",
        "Show trace"]),
      curtain(68,"Quit" ,[])
    ],
    _,- ),
write("\n Exit Turbo Inference Engine "), nl.
```



# SUITABLE FOR FRAMING

An object-oriented approach using frames can make an unruly knowledge base more manageable.

Michael Floyd

**PROGRAMMER** In designing an expert system, one immediately thinks about the knowledge base and how to represent its knowledge. For instance, if we are designing a simple backward chaining diagnostic system, we would probably think in terms of production (IF/THEN) rules. But in larger systems, production rules tend to be unmanageable after a certain point. To define a particular object thoroughly may require twenty, thirty, or forty rules, or maybe more.

Describing a single object clearly and completely is no problem in itself. But when we are describing hundreds of objects, the problems grow exponentially.

It is no wonder that a great deal of time has been spent on ways to better represent knowledge. One popular knowledge representation scheme is the *frame*. Frames are a means of organizing knowledge in an efficient manner. In this article, I will explain what frames are, why they're useful, and how you can implement them in Turbo Prolog. In the process, I'll cover related topics such as *value inheritance*, *defaults*, and *if-needed procedures*. I'll also show how to create and modify frames on-the-fly.

## THE FRAME CONCEPT

Frames are data structures that provide a way to organize data about objects. We use frames to describe an object, action, or event. Frames provide a way to group common concepts and situations into a hierarchy for easier and more efficient manipulation. They are particularly useful in the context of knowledge representation where there is a pattern to the knowledge that can be characterized or *stereotyped*.

You can visualize a frame by thinking of it as a box. This box is itself an object that contains other objects. On the side of the box, we may provide a description—what it's made of, what's in it, what it should be used for, and so forth. In addition, the box may have other smaller boxes in it.

A frame, like the box, is an object with a set of

Frame concept

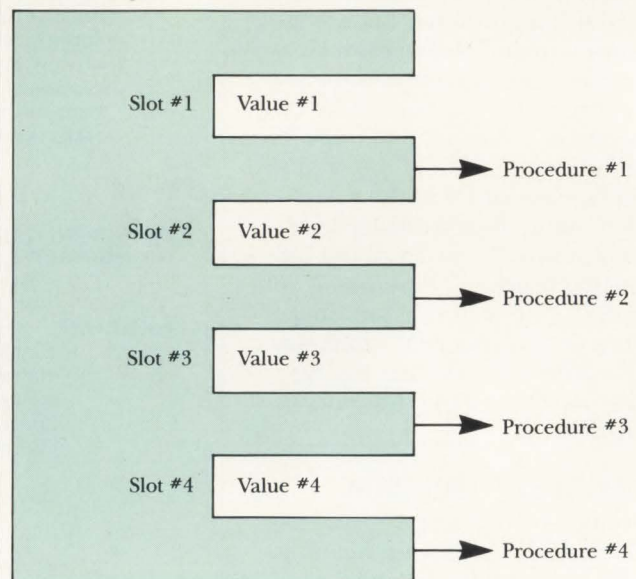


Figure 1. Elements of a single frame.

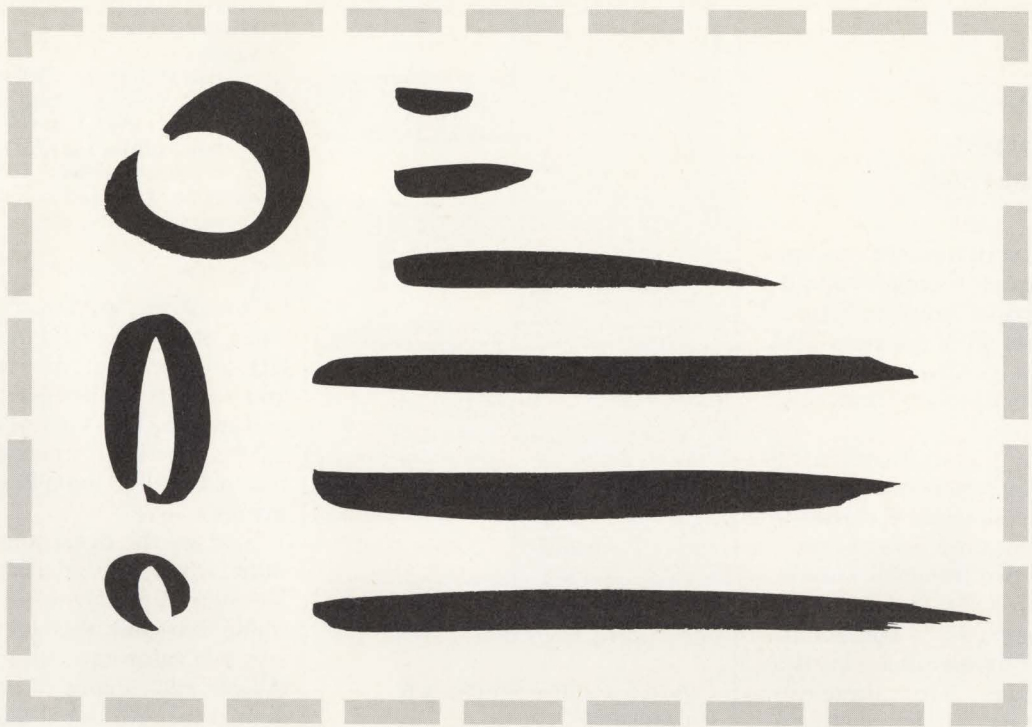
attributes that describe the frame. These attributes are known as *slots*. Figure 1 shows a graphic representation of the frame concept.

Notice in Figure 1 that the name of the frame represents the general topic or concept that the frame refers to. Also note that the slots (which are numbered) contain an attribute and a value for that attribute. We can fill the slots with either data or objects. In addition, we can attach procedures that perform some action to specific slots.

To put this in context, consider this example in which we describe a microcomputer. First, let's describe our prototype for a general class with the following characteristics:

*continued on page 82*







## FRAMING

continued from page 80

1. CPU
2. monitor
3. keyboard
4. one serial port
5. one parallel port
6. printer

With this description, we can construct a frame for a microcomputer, like the one shown in Figure 2. Note that not all of the slots have values in them. Also note that the slot for ports has more than one value.

By linking frames together with other frames, we can develop a hierarchy from general classes to specific cases. One type of link that relates two frames is known as an *IS-A* link. An *IS-A* link lets us say things such as, "a microcomputer is a computer *and* a PC is a microcomputer." From these two statements, we can deduce that a PC is a computer.

Another type of link, known as an *AKO* (A-Kind-Of) link, allows us to link classes to subclasses. With an *AKO* link, we could say that a computer is a kind of machine. Figure 3 shows the relationship of a particular PC to the general class of machines using *IS-A* and *AKO* links. This process of obtaining instances from classes is known as *inheritance*. We will have more to say about inheritance in a moment.

Now that we have a general understanding of the frame concept, let's take a look at how it can be implemented in Turbo Prolog.

### FRAMES IN TURBO PROLOG

In Turbo Prolog, we will use structures to construct a frame describing a microcomputer. Since the slots that fill the frame are known, domain declarations can be used to describe the frame. Our first attempt might look something like this:

```
domains
  frame = frame(object,slots)

  slot = cpu(symbol);
        monitor(symbol);
        keyboard(symbol);
        serial_ports(integer);
        parallel_ports(integer);
        printer(symbol)

  slots = slot*

  object = symbol
```

Microcomputer

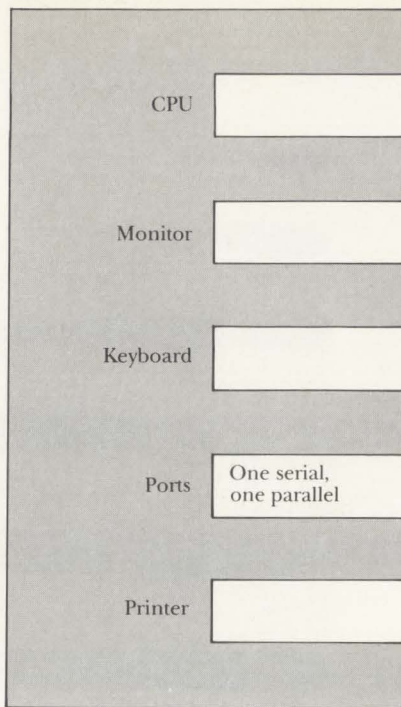


Figure 2. A frame describing a microcomputer

Just as our general definition states, the frame consists of an object and a list of slots that describe the object. The **slot** domain defines the possible slots that our frame system can have. Our frame for a microcomputer would look like

```
frame(microcomputer,
      [cpu(80286),
       monitor(high_res),
       keyboard(enhanced),
       serial_ports(1),
       parallel_ports(1),
       printer(yes)]).
```

which defines the prototype microcomputer as an 80286-based machine with a high-resolution monitor, an enhanced keyboard, one serial port, one parallel port, and a printer.

Not bad for a first try, but there are a few problems. First, a slot may itself be a frame. For instance, we may want to describe a class of microcomputers rather than an individual system. Also, since the slots are hard coded, we have to change the program each time we add a new slot.

A better approach is to write the frame in general terms. That way we can construct new frames or slots on-the-fly.

domains

```
frame = frame(object,slots)
frames = frame*
slot = slot(object,value)
slots = slot*
value =
  int(integer); ints(integers);
  real_(real); reals(reals);
  str(string); strs(strings);
  frame(object,slots);
  frames(frames)
object = symbol
integers = integer*
reals = real*
strings = string*
```

Let's work our way, from the bottom group of the declarations, up. First are the basic types for **object**, and our lists for **integers**, **reals**, and **strings**. The symbol domain has been omitted since symbols can be handled equally well as strings.

Next are the declarations for **value**, which define all the possible values for a given slot in a frame. Basically, this declaration says that **value** can either be an **integer**, **real**, **string**, or one of their respective lists. In addition, a value can be a frame, or a list of frames (we'll have more about the **frame** declaration in a moment).

Now that the possible values for the frame system have been defined, we can describe a slot for a frame. A slot is simply an object and a value (a description of the object). Notice that in this declaration **slot** appears both as the domain type (on the left side of the equals sign) and as the functor name (on the right side). This is allowed in Turbo Prolog and makes it clear as to which structure the domain refers. The next step is to declare a list of slots.

Now we're ready to construct the frame. Consistent with our original definition, a frame is an object and a list of slots that describe the attributes of the frame. Again, notice that **frame** appears as both the domain type and the functor name. Also, recall that the **frame** structure was declared as a **value**. Turbo Prolog lets us assign structures to different domains, just as it does with basic types. You've probably done this many times with a declaration such as

```
domains
  age = integer
  iq = integer
```

which says that a person's age and I.Q. are both integers.



## INHERITANCE

As we mentioned earlier, frames have certain properties that help us organize and represent knowledge. One of these properties is called inheritance. *Inheritance* is how particular objects inherit attributes from a template for that class of objects.

As an example, recall our prototype for a microcomputer; we said that it would have at least a CPU, monitor, keyboard, one serial port, one parallel port, and a printer. This prototype is our template for the general class of microcomputers. Now we can describe a particular microcomputer (a PC) that inherits a property (it has a CPU) from our template. To see how this works, let's construct a frame for the microcomputer using our new representation for a frame. We'll use the predicate **has** to describe the relation:

```
has(microcomputer,
    [slot(cpu,str("yes")),
     slot(monitor,str("yes")),
     slot(serial_port,int(1)),
     slot(parallel_port,int(1)),
     slot(printer,str("yes"))]).
```

Again, this says that a microcomputer has a CPU, a monitor, one serial port, one parallel port, and a printer. This is similar to our first frame definition of a microcomputer. The difference is that each attribute is in a slot structure rather than being hard coded. We can now describe specific types of microcomputers:

```
has(ibm_pc,
    [slot(cpu_type,str("8086")),
     slot(monitor,str("monochrome")),
     slot(drive,str("360K"))]).
```

```
has(pc_xt,
    [slot(cpu_type,str("8088")),
     slot(monitor,str("rgb")),
     slot(graphics_card,str("CGA"))
    ]).
```

```
has(pc_at,
    [slot(cpu_type,str("80286")),
     slot(monitor,str("Hi-Res")),
     slot(graphics_card,str("EGA"))
    ]).
```

These three frames describe the attributes of a PC, a PC-XT, and a PC-AT respectively. We can further refine the categories by describing specific configurations of particular machines:

```
has(dans_pc,
    [slot(modem,str("internal")),
     slot(hard_disk,str("20 MB"))]).
```

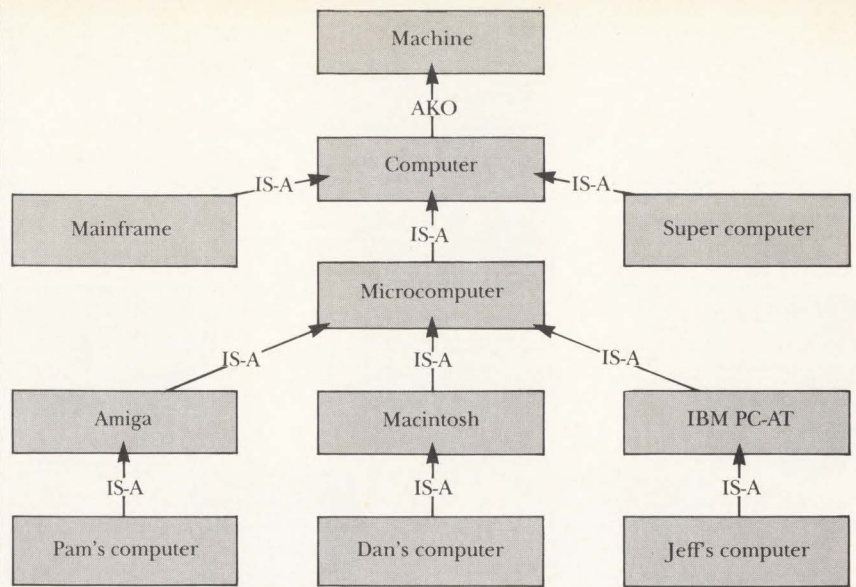


Figure 3. Network of links between frames.

```
has(jeffs_pc,
    [slot(coprocessor,str("80287")),
     slot(ext_memory,str("2 MB")),
     slot(hard_disk,str("30 MB"))
    ]).
```

To relate the particular machines to the class of microcomputers, we can say that Dan's machine is a PC-XT, and that a PC-XT is a microcomputer. Similarly, we can relate Jeff's PC as a PC-AT:

```
is_a(pc_xt,microcomputer).
is_a(pc_at,microcomputer).
```

```
is_a(dans_pc,pc_xt).
is_a(jeffs_pc,pc_at).
```

Figure 4 shows the relationships between the frames we have just created.

So far, we have merely stated facts about computers. In order for the two specific machines to inherit the values of a microcomputer, we need some *rules of inheritance*. Since these rules describe how facts are used in the system (as opposed to rules about the knowledge domain), we refer to them as *meta rules* (see "Metalogic and Expert Systems" elsewhere in this issue):

```
attr(Object,Value):-
    description(Object,Value).
```

```
attr(Object,Value):-
    is_a(Object,Object1),
    attr(Object1,Value),
    description(Object,
        SomeValue).
```

```
description(Object,Value):-
    has(Object,Description),
    member(Value,Description).
```

The first rule says that the attribute of an object is a value if the description of the object has that value. This rule looks to see if there is an explicit fact that satisfies the premise. For instance, we know that a PC-XT has a CGA graphics card because it is stated explicitly in the frame for the PC-XT.

If there's no explicit fact about the object, the second rule relates our rule of inheritance. It says that the attribute of an object is a value if we can relate that object to some other object with that value. For instance, we know that Jeff's computer has a CPU because he has a PC-AT, which is a microcomputer, and a microcomputer has a CPU.

The **description** rule does the lookups. Using the call to **has**, it looks up the given object and instantiates its associated list of slots to the description. The call to **member** then goes through the description list to see if the particular value we are looking for is in the list. The complete program is given in Listing 1.

In running Listing 1, you can enter goals such as:

```
attr(jeffs_pc,
    slot(coprocessor,str("80287")))
```

```
attr(dans_pc,
    slot(serial_port,int(1)))
```

*continued on page 84*



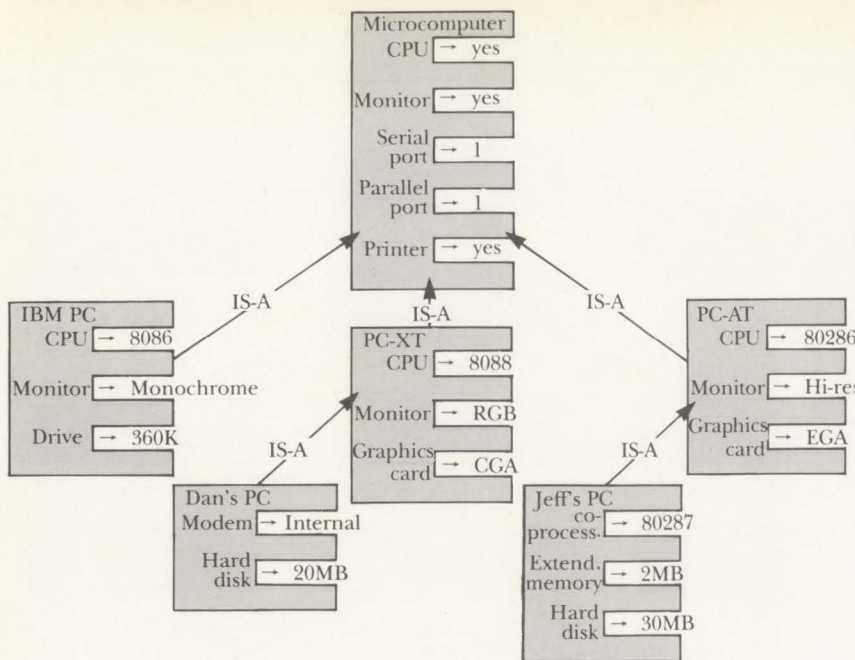


Figure 4. Relationship between class and instances of microcomputers.

## FRAMING

continued from page 83

These two goals simply return **True**, verifying that Jeff's PC has an 80287 math coprocessor, and that Dan's PC has one serial port.

By the way, **attr** as written only handles IS-A links. To handle AKO links, we should add a third **attr** rule.

```
attr(Object,Value):-
  ako(Object,Object1),
  attr(Object1,Value),
  description(Object,
    SomeValue).
```

This rule treats AKO links exactly the same as IS-A links. This implies that the difference between AKO and IS-A links is purely semantic. There may, however, be reason to report that a value was inherited through an AKO link. Our example does not consider AKO links since they are treated the same. If you wish to include them, be sure you have at least one **ako** fact such as `ako(computer,machine).`

in your database.

## DEFAULTS

So far, we have been filling slots with values that are always true. However, we know that the world is not so black and white. There are times when we can assume that something is probably true, but must allow for some degree of

uncertainty. We need to be able to fill slots with default values that can be overridden if our assumption falls through.

To do this, we must slightly modify our definition of a slot to take into account the type of value that the slot contains. This reference to a value's type is known as the *facet* of the slot. Our new definition handles both value and default facets. The change to the domain declaration of **slot** consists of adding a third argument:

```
domains
  slot = slot(id,value,facet)
```

Of course, we need to define what a **facet** is.

```
facet = symbol
```

There are no changes to the inheritance procedure since the algorithm for locating a default facet is the same as for locating a value facet. We have to change the call to **attr** to specify whether the procedure should find value facets (*value inheritance*) or default facets (*default inheritance*). Obviously, the calling procedure has to deal with default inheritance in a slightly different manner.

The only other change is to the actual frames in order to specify whether a slot is a value or a default slot. Here are the changes to the microcomputer frame for handling facets:

```
has(microcomputer,
  [slot(cpu,str("80286"),default),
   slot(monitor,
     str("hi-res"),default),
   slot(serial_port,int(1),value),
   slot(parallel_port,
     int(1),value),
   slot(printer,str("yes"),default)
  ]).
```

These statements say that our template for a microcomputer has three default values (an 80286 CPU, a high-resolution monitor, and a printer) that can be changed. Our two absolute values maintain that our template always has one serial port and one parallel port.

## IF-NEEDED PROCEDURES

In some instances, such as when a slot contains no explicit values, we can calculate a value based on existing information. For example, perhaps we want to find out the amount of available memory on a 640K machine, and we know that RAM-resident programs take up 128K. By associating a procedure with an **avail\_mem** slot, we can calculate the amount of available memory if it is not explicitly stated. Hence, we call it an *if-needed* procedure. Here's an algorithm for an if-needed procedure to calculate the amount of available memory.

1. Get the amount of total memory from the **total\_memory** slot.
2. Get the amount of memory used by RAM-resident programs in the **memory\_used** slot.
3. Calculate the amount of available memory and report the result.

In Turbo Prolog this translates as:

```
calc_mem:-
  find(slot(total_mem,
    int(Total),value)),
  find(slot(mem_used,
    int(Used),value)),
  Avail_mem = Total - Used,
  write(Avail_mem).
```

We can, in theory, inherit if-needed procedures. One method is to establish a set of procedures that can be referenced by an indexed value, such as:

```
procedure(1):-
  ... /* Do some stuff */
procedure(2):-
  ... /* Do some other stuff */
```



This does not allow us to create if-needed procedures on-the-fly, however. To handle this, we need to be able to create and call procedures at will. However, the techniques required are beyond the scope of this article (see "Meta-programming and Expert Systems" elsewhere in this issue). Our next example simply incorporates the if-needed procedure directly into the search algorithm.

## PUTTING IT TOGETHER

Now that we have facilities for value and default inheritance, along with the ability to attach if-needed procedures, we can put together a controlling clause that combines procedures. Most likely, we will want to inherit values first since they are absolute truths and do not involve a calculation. If there is no value facet, we will look for an if-needed procedure to calculate the value. Failing that, we will finally look for a default value that carries some degree of uncertainty:

```
n_inheritance:-
  attr(jeffs_pc,
    slot(avail_mem,X,value)),
  write(X).
n_inheritance:-
  attr(jeffs_pc,
    slot(total_mem,
      int(Total),value)),
  attr(jeffs_pc,
    slot(mem_used,
      int(Used),value)),
  Avail_mem = Total - Used,
  write(Avail_mem).
n_inheritance:-
  attr(jeffs_pc,
    slot(avail_mem,X,value)),
  write(X,"\n some uncertainty").
n_inheritance:-
  write("No values available").
```

## CREATING AND MODIFYING FRAMES

Now that we have established the inheritance between frames, we are ready to start manipulating them. In general, this means adding the ability to create new frames, modify existing frames, and delete frames that have become obsolete in our knowledge system. In order to assert and retract frames to and from the database, the **has** predicate must be defined as a database predicate.

```
database
  has(object,slots)
```

*continued on page 86*

### LISTING 1: FRAME1.PRO

```
/* Simple Frame Example */

domains
  frame      = frame(object,slots)
  frames     = frame*

  slot       = slot(object,value)
  slots      = slot*
  value      =
      int(integer) ; ints(integers) ;
      real(real) ; reals(reals) ;
      str(string) ; strs(strings) ;
      frame(object,slots) ; frames(frames)

  object     = symbol
  integers   = integer*
  reals      = real*
  strings    = string*

predicates
  attr(object,slot)
  description(object,slot)
  is_a(object,object)
  member(slot,slots)
  has(object,slots)

clauses

/* These 3 clauses describe the way in which frames inherit
properties from their parent */

attr(Object,Value):-
  description(Object,Value).
attr(Object,Value):-
  is_a(Object,Object1),
  attr(Object1,Value),
  description(Object,_).

description(Object,Value):-
  has(Object,Description),
  member(Value,Description).

/* This is the Actual frame to represent the attributes of a typical
microcomputer. */

/* Prototype house */

has(microcomputer,[slot(cpu,str("yes")),
  slot(monitor,str("yes")),
  slot(serial_port,int(1)),
  slot(parallel_port,int(1)),
  slot(printer,str("yes"))]).

/* The rest are particular type houses which carry the specific
attributes, plus those inherited from the parent frame */

has(ibm_pc,[slot(cpu_type,str("8086")),
  slot(monitor,str("monochrome")),
  slot(drive,str("360K"))]).

has(pc_xt,[slot(cpu_type,str("8088")),
  slot(monitor,str("rgb")),
  slot(graphics_card,str("CGA"))]).
```



```

has(pc_at, [slot(cpu_type, str("80286")),
           slot(monitor, str("Hi-Res")),
           slot(graphics_card, str("EGA"))]).

has(dans_pc, [slot(modem, str("internal")),
             slot(drive, str("360K"))]).
has(jeffs_pc, [slot(coprocessor, str("80287")),
              slot(monitor, str("rgb")),
              slot(graphics_card, str("color"))]).

/* is_a relates children to its parent */

is_a(pc_xt, microcomputer).
is_a(pc_at, microcomputer).

is_a(dans_pc, pc_xt).
is_a(jeffs_pc, pc_at).

/* member simply checks to see if a particular value is in the
   slot list */

member(Value, [Value|_]) :- !.
member(Value, [_|Rest]) :- member(Value, Rest).

```

## LISTING 2: FRAME2.PRO

```

/*****
This example demonstrates the use of frames in Turbo Prolog.
The goal shows how to combine value and default inheritance with
if-needed procedures to create an N inheritance scheme. A facility
is also provided to allow a user to create frames on-the-fly.

Mike Floyd      12/16/87
Turbo Technix
Borland International
*****/

domains
  frame      = frame(id, slots, parents)
  frames     = frame*

  slot      = slot(id, value, facet)
  slots     = slot*

  value     = int(integer) ; ints(integers) ;
             real(real) ; reals(reals) ;
             str(string) ; strs(strings) ;
             frame(id, slots, parents) ; frames(frames)

  id        = symbol
  facet     = symbol
  parents   = id*
  integers  = integer*
  reals     = real*
  strings   = string*

database
  has(id, slots)      /* database relation to store frames */
  is_a(id, id)        /* is-a link for objects */
  ako(id, id)         /* ako link for objects */
  dbslot(slot)        /* database storage for slot values */

```

```

create_frame:-
  write("Enter Frame Name: "),
  readln(Frame),
  get_slots(SlotList),
  get_relations,
  insert_frame(Frame, SlotList).

get_slots(Slots):-
  write("Enter Attribute Name: "),
  readln(Id),
  ID <> "",
  write("Enter attribute: "),
  readln(Value),
  assertz(tslot(slot(Id,
                    str(Value)))),
  get_slots(Slots).
get_slots(Slots):-
  findall(Slot, tslot(Slot), Slots).

get_relations:-
  findall(Id, has(Id, _), Idlist),
  show(Idlist),
  write("Enter object1: "),
  readln(Object1),
  write("Enter object2: "),
  readln(Object2),
  assert(is_a(Object1, Object2)).

show([]) :- !.
show([H|List]):-
  write(H), nl,
  show(List).

insert_frame(Id, Slots):-
  asserta(has(Id, Slots)).

```

Figure 5. Turbo Prolog code to create and modify frames.

We had to do this anyway since we would like to be able to **consult** our knowledge base of frames.

To create new frames, we need a clause that constructs the frame and inserts it into the database. In this case, we'll get our input from the user, then construct the frame. Our basic algorithm is to:

1. Get the name of the frame.
2. Get the slot information.
3. Display a list of the current frames and allow the user to define any **is\_a** relationships between objects.
4. Construct the frame and assert it into the database.

The **create\_frame** clause in Figure 5 details this algorithm in Turbo Prolog. By adding this code to the earlier listing, the user can now add their own frames to the system without programmer intervention. If the appropriate **is\_a** relationships are defined, the new frames can inherit values from parent frames just as our original frames do.



There are a couple of things that have not been handled in Figure 5 due to space limitations. First of all, the interface could better insulate the user from the frame syntax by being a little more friendly. Also, when entering attribute values for slots, the **get\_slots** clause assumes that everything is of type **string**. To make this general for any data type requires an additional question to the user about the data type. The clause would then have to lookup the appropriate structure in a table and convert the input. Finally, **get\_slots** assumes that the information input by the user is absolutely true. So, the information is always put in the value facet of the slot.

In order to modify existing frames, we first need to add the ability to remove old frames from the knowledge base. Deleting just the frame serves the purpose, but it leaves extraneous **is\_a** relationships in the database. Not only does this take up unnecessary space in the knowledge base, but it slows down processing time; it takes the system longer to realize that a frame has been removed and cause the goal to fail.

```
delete_frame(FrameId):-
    retract(has(FrameId,_)),
    retract(is_a(FrameId,_)),
    retract(is_a(_,FrameId)),
    fail.
delete_frame(_).
```

The **fail** guarantees that we have retracted all occurrences of **has** and **is\_a** database facts containing **FrameId**. The second **delete\_frame** clause allows us to return from the call successfully.

With the **create\_frame** and **delete\_frame** tools implemented, adding a facility to modify existing frames is a simple matter of combining tools. Listing 2 shows the complete code for the features we've implemented so far.

### TRACING PROGRAM LOGIC

There is one other feature we should mention, but won't implement due to space—keeping track of parent nodes to easily trace through our logic. This is important if we wish to provide an explanation facility in an expert system. This requires some small changes in the declaration of our

*continued on page 88*

```
predicates
    attr(id,slot) /* get or verify the attribute of an object*/
    description(id,slot) /* does frame lookups and instantiation */
    member(slot,slots) /* look for an object in a list of objects */
    create_frame /* Create a frame from user input */
    insert_frame(id,slots) /* insert frame into the database */
    delete_frame(id,slots) /* delete a frame from the database */
    get_slots(slots) /* get all slots related to a frame */
    get_relations /* get all relations and display to user */
    show(parents) /* show a list of parent frames */
    n_inheritance /* implements an N inheritance scheme */

goal
    n_inheritance.

clauses
    n_inheritance:-
        attr(jeffs_pc,slot(avail_mem,X,value)),
        write(X).
    n_inheritance:-
        attr(jeffs_pc,slot(total_mem,int(Total),value)),
        attr(jeffs_pc,slot(mem_used,int(Used),value)),
        Avail_mem = Total - Used,
        write(Avail_mem).
    n_inheritance:-
        attr(jeffs_pc,slot(avail_mem,X,value)),
        write(X,"\\nsome uncertainty related to answer").
    n_inheritance:-
        write("No values available").

/* These 3 clauses describe the way in which frames inherit
properties from their parent */

attr(Object,Value):-
    description(Object,Value).
attr(Object,Value):-
    is_a(Object,Object1),
    attr(Object1,Value),
    description(Object,_).

/* attr() clause to search AKO links */
attr(Object,Value):-
    ako(Object,Object1),
    attr(Object1,Value),
    description(Object,_).*/

description(Object,Value):-
    has(Object,Description),
    member(Value,Description).

member(X,[X|_]):-!.
member(X,[_|L]):-member(X,L).

/* Create and modify frames */
create_frame:-
    write("Enter Frame Name: "),
    readln(Frame),
    get_slots(SlotList),
    get_relations,
    insert_frame(Frame,SlotList).

get_slots(Slots):-
    write("Enter Attribute Name: "),
    readln(Id),
    ID <> "",
    write("Enter Value of Attribute: "),
    readln(Value),
    assertz(dbslot(slot(Id,str(Value),value))),
    get_slots(Slots).
get_slots(Slots):-
    findall(Slot,dbslot(Slot),Slots).
```



```

get_relations:-
    findall(Id,has(Id,_,Idlist),
    show(Idlist),
    write("Enter object1: "),
    readln(Object1),
    write("Enter object2: "),
    readln(Object2),
    assert(is_a(Object1,Object2)).

show([]):-!.
show([H|List]):-
    write(H),nl,
    show(List).

insert_frame(Id,Slots):-
    asserta(has(Id,Slots)).

delete_frame(Id,Slots):-
    retract(has(Id,Slots)).

/* Prototype microcomputer */
has(microcomputer,[slot(cpu,str("80286"),default),
    slot(monitor,str("hi-res"),default),
    slot(serial_port,int(1),value),
    slot(parallel_port,int(1),value),
    slot(printer,str("yes"),default)]).

/* The rest are particular type microcomputers which carry the
specific attributes, plus those inherited from the parent
frame. */

has(ibm_pc,[slot(cpu_type,str("8086"),value),
    slot(monitor,str("monochrome"),default),
    slot(drive,str("360K"),value)]).

has(pc_xt,[slot(cpu_type,str("8088"),value),
    slot(monitor,str("rgb"),default),
    slot(graphics_card,str("CGA"),default)]).

has(pc_at,[slot(cpu_type,str("80286"),value),
    slot(monitor,str("Hi-Res"),default),
    slot(graphics_card,str("EGA"),default)]).

has(dans_pc,[slot(modem,str("internal"),value),
    slot(drive,str("360K"),value)]).

has(jeffs_pc,[slot(coprocessor,str("80287"),value),
    slot(total_mem,int(640),value),
    slot(mem_used,int(128),value)]).

/* is_a relates a child object or frame to its parent */

is_a(pc_xt,microcomputer).
is_a(pc_at,microcomputer).

is_a(dans_pc,pc_xt).
is_a(jeffs_pc,pc_at).

```

## FRAMING

*continued from page 87*

frame structure. First, we need to add a third argument—which is a list of objects—to the frame structure. We call this a list of *parents*. We also need to declare the **parent** and its associated list:

```

domains
    frame(object,slots,parents)
    parent = object
    parent = parent*

```

The next step is to add a statement to the meta rules that adds the name of the parent frame as it is instantiated. I'll leave this and the other improvements I've mentioned, as a homework assignment.

As you can see, frames are most useful when we can categorize an object into a class of objects. This makes frames ideal in a classification-type expert system, such as GENI (on the Turbo Prolog distribution disk). But expert systems are not the only applications well-suited to frames. Since frames can represent events and situations as objects, you can use them in areas such as robotics, where the frame could represent an action to be carried out. Other areas include sensor-controlled systems, image and voice recognition systems, and so forth.

In this article, we've explored the concept of frames, as well as value inheritance, defaults, and if-needed procedures. We've also seen how these concepts can be implemented in Turbo Prolog. In addition, we've created clauses that create, modify, and remove frames on-the-fly. Finally, we've discussed some features, such as tracing program logic, that can enhance a frame-based expert system. By mastering this object-oriented approach to knowledge representation, you can add flexibility as well as efficiency to your expert system.

## REFERENCES

- Winston, Patrick Henry. *Artificial Intelligence*, Addison Wesley, 1984.
- McCord, Michael, et. al.. *Knowledge Systems and Prolog*, Addison Wesley, 1987.


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



# METALOGIC AND EXPERT SYSTEMS

The golden rule of metalogic is: "... do unto thyself."

Safaa H. Hashim



Metalogical programming is a powerful technique that, among other things, allows a system to learn. This is especially important in the design of an expert system. In this article I will discuss what metalogical programming is, why it is important, where you can use it, and how you can implement it.

## METALOGIC AND METALANGUAGE

The term *metalogical programming* comes from the word *metalogic*. According to W. L. Reese, metalogic is "... the study of the structure, signs, connectives, formulae, theorems, and rules of inference of logic from a syntactic, semantic, and pragmatic point of view. Metalogic is the metalanguage of logic."<sup>1</sup>

Therefore, on one hand, logic (as a language) has primitives for representing knowledge about the world; on the other hand, it also has primitives for representing knowledge about itself.

In philosophy this is known as the *Use-Mention Distinction* of a language. This enables us to tell the difference between a language and its metalanguage. The metalanguage of a language allows us to talk about and study the language itself. Metalanguage refers to the primitives of the language.

Bertrand Russell, the well-known philosopher, first drew this distinction. He suggested that to study a language, you need to go outside of the language itself; you need to construct another language on top of the language under consideration—a *metalanguage*.

## METALOGICAL PROGRAMMING IN PROLOG

In Prolog we can achieve a metalogical effect by executing a metalogical predicate. But what is a metalogical predicate? Authorities on Prolog differ slightly in defining a metalogical predicate. Leon Sterling and

Ehud Shapiro offer a global view: "[metalogical predicates] are outside the scope of first-order logic, as they query the state of the proof, treat variables (rather than the terms they denote) as objects of the language, and allow the conversion of data structures to goals."<sup>2</sup>

Sterling and Shapiro further categorize the metalogical predicate into four types:

1. Predicates that determine whether or not a term is a variable (i.e., **var(TERM)**).
2. Predicates that handle term comparison. That is, they check whether two variables are identical (not just unifiable, as are any two variables).
3. Predicates that treat variables as manipulatable objects.
4. Predicates that allow data to be executed as goals. For example, **call(color(red))** is a metalogical predicate that executes its argument, the compound object **color(red)**, as a goal.

The first three types of metalogical predicates are beyond the scope of this article; we will focus on the fourth type. Turbo Prolog has at least one built-in metalogical predicate of the fourth type: **findall**. It is of the fourth type because it takes a predicate as its second argument and executes it as a goal. Then it collects all the answers related to its first argument and puts them in a list. But in addition to **findall**, we need a general system for handling data as goals. We need the metalogical predicate, **call**, which takes a term as its sole parameter and executes it as a goal.

## WHEN IS METALOGIC PROGRAMMING NEEDED?

As stated at the beginning of this article, metaprogramming techniques are useful when writing an AI application that can learn. In learning, the program

*continued on page 90*

<sup>1</sup>Reese, W.L., *The Dictionary of Philosophy and Religion, Eastern and Western Thought*, Atlantic Highlands, New Jersey: Humanities Press, 1980.

<sup>2</sup>Sterling, Leon and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*, Boston, MA: M.I.T. Press, 1986.



```

/* A simple clause processor. */

domains
  PARAMETER = reference SYMBOL
  PARMlist = PARAMETER*
  Llist = PARMlist*

database
  cl(PARMlist,Llist)

predicates
  call(PARMlist)
  calls(Llist)
  feature(PARAMETER,PARAMETER,PARAMETER)

clauses

/* ***** */
/* ***** The knowledge base expressed using the "cl" facts ***** */
/* ***** */

/* "cl" facts which represents Turbo Prolog rules */

cl([reacts,wo,violently],
  [[feels,wo,lonely]]).
cl([turns,wo,green],
  [[color,green],
  [placed_besides,wo,"better looking species"]]).

/* "cl" facts which represent Turbo Prolog facts */

cl([color,green], []).
cl([placed_besides,wo,"better looking species"], []).
cl([feels,wo,lonely], []).
cl([has,wo,"affinity for gold"], []).
cl([has,wo,"affinity for silver"], []).
cl([has,wo,"affinity for precious stones"], []).

/* ***** */
/* A rule that uses the Metalogical predicate "call" */
/* ***** */

feature(Name,Topic,Feature) if
  call([Topic,Name,Feature]),!.

/* ***** */
/* A simple clause-processor. */
/* ***** */

/* processing a single clause: a rule */

call(B) if
  cl(B,[H|T]),
  call(H),
  calls(T),
  !.

/* processing a single clause: a fact */

call(P) if
  cl(P, []).

/* processing a list of clauses */

calls([]).
calls([H|T]) if
  call(H),
  calls(T).

/* ***** END ***** */

```

## METALOGIC

continued from page 89

can modify its behavior; it can acquire new knowledge, that is, new facts and rules. It can also delete old knowledge that has become obsolete or inconsistent with the current state of the knowledge.

In addition, metaprogramming is useful when we want to extend the power of Turbo Prolog. For instance, we might need to match a variable with a built-in Turbo Prolog predicate. Writing interpreters is a good application for this purpose. In an interpreter, there are times when we would like to instantiate a variable to a built-in Turbo Prolog predicate. In other words, we transform a built-in predicate into data and vice versa. This is particularly important in implementing expert system shells. In fact, Listing 2 (which we will look at shortly) shows a portion of the inference engine for TESS (The Expert System Shell)<sup>3</sup>.

Metaprogramming can also extend Turbo Prolog's power in object-oriented programming. An object has procedures as part of its definition in object-oriented programming. In Prolog, objects are represented as facts. Metalogical programming techniques allow us to do object-oriented programming by passing rules as parameters to objects (facts). Object-oriented programming is a powerful tool for many applications, including frames (see "Suitable for Framing" elsewhere in this issue) and in developing user interface systems.

On the flip side, metalogical programming is not needed when a task can be implemented without it—if you can write it directly in Turbo Prolog, do it. Why program at a metalogical level when it can be done on a basic level? If you stick to basic Turbo Prolog, you can keep things simple, save on memory, and cut processing time.

<sup>3</sup>Hashim, Safaa and Philip Seyer. *Turbo Prolog: Advanced Programming Techniques*, Philadelphia, PA: TAB Books, Inc., 1988.



## THE call PREDICATE AND TURBO PROLOG

With that said, let's examine the metapredicate **call**. Using **call**, variables can match whole predicates, facts or rules, as well as other types of domains. For example, consider the following program fragment:

```
database
  caused_by(string,string)

clauses
  noticed(Name,Situation) if
    call(Situation).

  cause_by("no light","no power").
```

Given this program, if we enter the goal

```
noticed(james,caused_by(X,Y))
the system would respond:
X = "no light"
Y = "no power"
True
```

Now, if we enter a new goal

```
noticed(james,
  caused_by("car not working",
    "dead battery"))
the subgoal
call(caused_by("car not working",
  "dead battery"))
```

is executed. Notice that the **call** predicate executes the term **caused\_by**, which is a compound object. Therefore, **call** is interpreting the data as a goal. The goal **caused\_by("car not working", "dead battery")**

fails because there is no fact like this. An extension to this program could be a predicate that asks the user to confirm if a term is true or not. If the term is true, we add the term as a subprogram.

Unfortunately, things are not so easy in Turbo Prolog. The difficulty lies in matching terms with clauses. In Turbo Prolog you can match a term with a term, but you cannot match a term with a clause. To get around the problem, we need a different representation of clauses.

## REPRESENTING RULES IN THE DATABASE

There are two issues behind the design of a uniform representa-

tion of facts and rules. The issues are:

1. How to assert, retract and consult rules as well as facts.
2. How to match a predicate with a term. In other words, pass a predicate as a parameter for other predicates.

In Turbo Prolog, to assert, retract, and consult clauses, we have to declare the clauses as **database**. Only facts are allowed in a database domain type—no rules. To get rules into the picture, we have to write them as facts. To do this, let's use a fact called **cl** (short for clause). For example, in our system, this rule

```
turns(wo,green) if
  color(green)
  placed_besides(wo,
    "better looking species").
```

could be written as a **cl** fact in this way:

```
cl([turns,wo,green],
  [[color,green],
  [placed_besides,wo,
    "better looking species"]
  ]).
```

Notice that writing rules as **cl** facts allows us to assert and retract *rules* (as well as facts) to and from the

knowledge base.

To see how Turbo Prolog processes this representation, let's consider the clause processor in Listing 1. In running this program, we can enter the goal:

```
feature(X,Y,Z).
```

Turbo Prolog matches this goal with the **feature** rule in the program. Then the subgoal

```
call([turns,wo,green])
```

is executed. The program responds:

```
"X=wo, Y=turns, Z=green".
```

## WRITING THE CLAUSE PROCESSOR

Listing 1 contains a single rule and a knowledge base of **cl** facts. The **feature** rule has one subgoal: the predicate **call**. Notice that **call** takes a predicate (which is a list of terms). It tries to prove the predicate true by matching it with the first parameter of a **cl** fact.

There are two **call** clauses. The first clause tries to match its parameter with a **cl** fact—which really represents a rule. The second does the same by

*continued on page 92*

# The Stonehaven LEXICON

*Natural Language Power for your  
Turbo Prolog Programs*

Turbo Lightning spelling checks — with routines to build custom dictionaries.

Synonyms and alternative spellings.

Grammar sidecar to Borland's Lightning provides parts of speech, tense, root words, and derivation for a working vocabulary of 40,000 words.

Color management, parsing, menus, and user correction of strings and files.

Extensive examples, including parsing of natural language commands.

**No Royalties — No Copy Protection**

*Requires Turbo Prolog & Turbo Lightning*

**\$74.95 + \$5.00 UPS Shipping**

**800-356-6875**

**Stonehaven  
Laboratory**



47925 Eightieth St. West  
Lancaster, California 93536

VISA  
M/C



matching the parameter with a **cl** fact representing a fact. Figure 1 illustrates how **call** works.

**DEALING WITH VARIABLES**

Now suppose we have a variable in our rules. Our rules are represented as **cl** facts. The problem is that Turbo Prolog does not allow "free variables" in the database.

So how can we assert and retrieve rules that have variables?

That is, how can we use variables in **cl** facts? The solution is to treat variables as strings by writing them as capitalized words between quotes. To implement this, we need a mechanism to interpret strings that start with an uppercase letter as variables. We also need the ability to match these variables with other terms (constant or variable terms).

To deal with these two issues we need some kind of a *unification algorithm*—an algorithm that recognizes and matches variables

with terms and predicates. We can do this with a binding table.

**THE BINDING TABLE**

A *binding table* is a list of pairs. We call a variable and its value a *binding pair*. For example, the fact

```
cl([share,evelyn,foster,office], []).
```

would match the goal

```
call([share,"X","Y","Z"]).
```

and we would get this binding table (list of pairs):

```
[pair("X",evelyn),
 pair("Y",foster),
 pair("Z",office)]
```

The situation gets complex when we use rules with variables, and even more complex if there are alternative facts and/or rules. In this case we would have multiple bindings for the same variable. For example, if we have these three facts

```
cl([share,evelyn,foster,office], []).
cl([share,maria,leah,apartment], []).
cl([share,philip,kumiko,love], []).
```

we get a binding table with three bindings for each of "X", "Y", and "Z" (this will become apparent in a moment). Table 1 shows the binding pairs resulting from our previous goal.

The process of binding variables is more complex than what we have just stated. Through the execution process (implemented by **call**), variables may start out as free variables. Then later they may be bound; even later they may be bound, and so on. This means variables may have different bindings at different levels. For purposes of Prolog backtracking, we also need to keep a record or a history of each variable used in the *unification process*.

In Table 1, the numbers at the top represent the binding level. That is, if a predicate **share(X,Y,Z)** is called three times, there can be

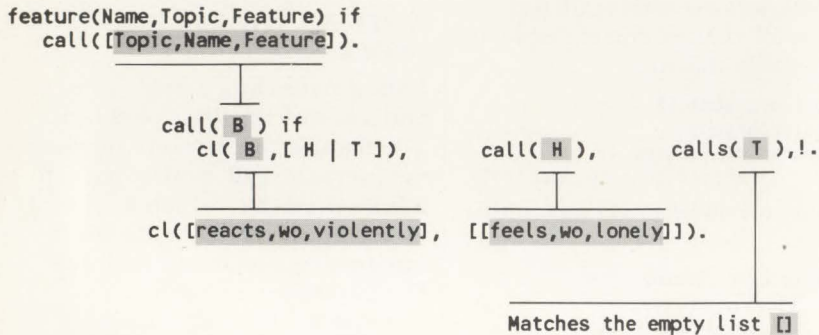


Figure 1. Diagram of how **call** matches a **cl** fact.

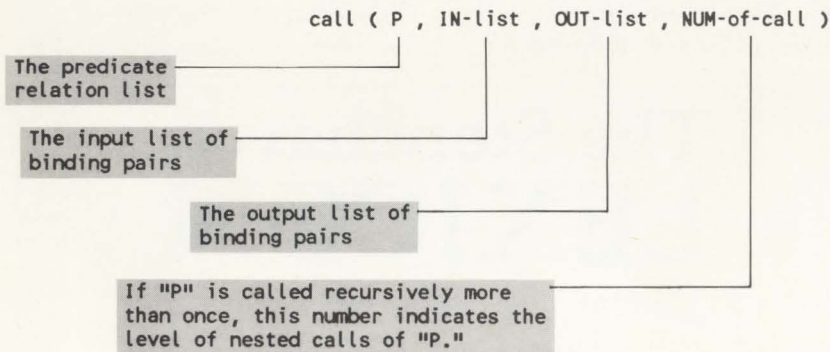


Figure 2. Structure of the **call** predicate.

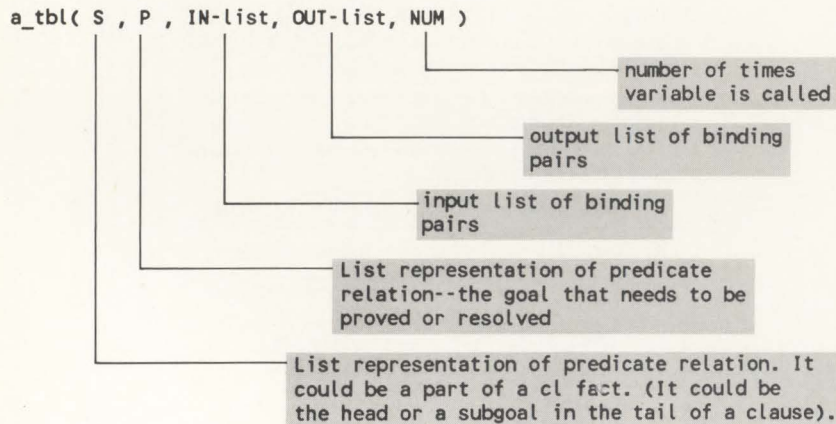


Figure 3. Structure of the **a\_tbl** predicate.



Binding levels

Variable	1	2	3
"X"	evelyn	maria	philip
"Y"	foster	leah	kumiko
"Z"	office	apartment	love

Table 1. A binding table for three variables.

three possible bindings for X, Y, and Z. Let's use our list of pairs to represent the binding table. But first, we need to make one change to the compound object **pair**. We need to add a third (integer) argument that shows the level at which the variable is bound to a value. We refer to it as the binding *number*. As a result the list of binding pairs will look like this:

```
[pair("X",evelyn,0),
pair("Y",foster,0),
pair("Z",office,0),
pair("X",maria,1),
pair("Y",leah,1),
pair("Z",apartment,1),
pair("X",philip,2),
pair("Y",kumiko,2),
pair("Z",love,2) ]
```

**MATCHING UNBOUND VARIABLES**

Remember that in Prolog, variables can also act as place holders. In other words, an unbound variable X can match any other variable. It could match, say, Y or Z (whether they are bound or free). So how can we keep track of which variable is bound and at what level? The **pair** representation we discussed earlier handles this. The structure of **pair** is:

```
pair(VARIABLE, VALUE, INDEX-NUMBER)
```

This representation assigns a unique pair to each variable in the process of executing the **cl** facts. As an example of how free variables can match other free or bound variables, append the following **cl** facts to the program in Listing 2 and run this example:

```
cl([married,philip,kumiko], []).
cl([share,"X","Y",love],
  [ [married,"X","Y"] ]).
```

*continued on page 94*

LISTING 2: TBL-MOD.PRO

```
/* The Binding Tables Algorithm program. */

domains
  INT = reference INTEGER
  PARAMETER = reference SYMBOL
  PAIR = pair(PARAMETER,PARAMETER,INT)
  PAIRlist = PAIR*
  PARMList = PARAMETER*

predicates
  a_tbl(PARMList,PARMList,PAIRlist,PAIRlist,INT)
  bTBL(PARMList,PARMList,PAIRlist,PAIRlist,INT)
  cTBL(PARAMETER,PARAMETER,INT,PAIRlist,PAIRlist)
  dTBL(PARAMETER,PARAMETER,INT,PAIRlist,PAIRlist)
  eTBL(PARAMETER,PARAMETER,INT,PAIRlist,PAIRlist)
  fTBL(PARAMETER,PARAMETER,INT,PAIRlist,PAIRlist)
  gTBL(PARAMETER,PARAMETER,INT,PAIRlist,PAIRlist)
  hTBL(PARAMETER,PARAMETER,INT,PAIRlist,PAIRlist)
  jTBL(PARAMETER,PARAMETER,INT,PAIRlist,PAIRlist)

  variable(PARAMETER)
  capital(PARAMETER)
  member(PAIR,PAIRlist)

clauses
  a_tbl([NAME|Slist],[NAME|Vlist],IN,OUT,NUM) if
    bTBL(Slist,Vlist,IN,OUT,NUM).

  bTBL([],[],TBL,TBL,_) if !.

  bTBL([S|ST],[V|VT],IN,OUT,NUM) if
    cTBL(S,V,NUM,IN,OUT),
    bTBL(ST,VT,OUTA,OUT,NUM).

  cTBL(XS,V,NUM,IN,OUT) if /* "V" here can be free or bound */
    bound(XS),
    variable(XS), /* "XS" is a string starts with capital letter */
    dTBL(XS,V,NUM,IN,OUT), !.
  cTBL(SS,V,NUM,IN,OUT) if /* "V" here can only be bound */
    bound(V),
    variable(V), /* "SS" is a string acting as a constant */
    gTBL(SS,V,NUM,IN,OUT), !.
  cTBL(SS,FV,NUM,IN,OUT) if
    free(FV),
    hTBL(SS,FV,NUM,IN,OUT).
  cTBL(FS,BV,NUM,IN,OUT) if
    free(FS),
    jTBL(FS,BV,NUM,IN,OUT), !.
  cTBL(S,S,_,TBL,TBL).

  dTBL(XS,V,NUM,IN,OUT) if
    bound(V),
    eTBL(XS,V,NUM,IN,OUT), !.
  dTBL(XS,FV,NUM,IN,OUT) if
    free(FV),
    fTBL(XS,FV,NUM,IN,OUT).

  eTBL(XS,BV,NUM,TBL,TBL) if member(pair(XS,BV,NUM),TBL).
  eTBL(XS,BV,NUM,IN,[pair(XS,BV,NUM)|IN]) if
    not(member(pair(_,BV,NUM),IN)).

  fTBL(XS,_,NUM,[],[pair(XS,XS,NUM)]).
  fTBL(XS,V,NUM,[pair(XS,V,NUM)|R],[pair(XS,V,NUM)|R]).
  fTBL(XS,V,NUM,[PAIR|IN],[PAIR|OUT]) if fTBL(XS,V,NUM,IN,OUT).

  gTBL(,_,_,[],_) if !, fail.
  gTBL(SS,XV,NUM,[pair(XS,XV,NUM)|R],[pair(XV,SS,NUM)|R]).
  gTBL(SS,XV,NUM,[PAIR|IN],[PAIR|OUT]) if gTBL(SS,XV,NUM,IN,OUT).

  hTBL(S,S,_,[],[]).
```



```

hTBL(S,S,NUM,[pair(XS,XS,NUM)|R],[pair(XS,S,NUM)|R]).
hTBL(S,FV,NUM,[PAIR|IN],[PAIR|OUT]) if hTBL(S,FV,NUM,IN,OUT).

jTBL(S,S,_,[],[]).
jTBL(FS,BV,NUM,[pair(FS,BV,NUM)|R],[pair(FS,BV,NUM)|R]).
jTBL(FS,BV,NUM,[A|IN],[A|OUT]) if jTBL(FS,BV,NUM,IN,OUT).

/* ***** check if a letter is a capital letter ***** */

variable(String) if
  frontstr(1,String,CHAR,_) /* take first character */
  capital(CHAR). /* check if CHAR is a capital letter */

capital(Uchar) if
  upper_lower(Uchar,Lchar),
  Uchar <> Lchar.

/* ***** A pair is a member of a list of pairs ***** */

member(X,[X|_]).
member(X,[_|Y]) if member(X,Y).

```

## LISTING 3: CALL.PRO

```

/* Processing clauses with variables using binding tables. */

domains
  Llist = PARMLIST* /* Llist is a list of lists of SYMBOLS */

database
  cl(PARMLIST,Llist)

include "tbl-mod.pro" /* tbl-mod.pro is the program in
                       listing 2 */

predicates
  call(PARMLIST,PAIRLIST,PAIRLIST,INT)
  calls(Llist,PAIRLIST,PAIRLIST,INT)

clauses

/* Executing the head of a clause , the conclusion. */

call(P,IN,OUT,NUM) if
  cl(S,[]),
  a_tbl(S,P,IN,OUT,NUM), !.

call(P,IN,OUT,NUM) if
  cl(S,[H|B]),
  a_tbl(S,P,IN,TBLA,NUM),
  calls([H|B],TBLA,OUT,NUM), !.

call(P,IN,OUT,NUM) if
  cl(S,[H|B]),
  NUMA = NUM + 1,
  a_tbl(S,P,IN,TBLA,NUMA),
  calls([H|B],TBLA,OUT,NUMA), !.

/* Processing the tail of the clause, The subgoals or conditions. */

calls([],TBL,TBL,_).

calls([SH|ST],IN,OUT,NUM) if
  a_tbl(SH,P,IN,TBLA,NUM),
  call(P,TBLA,TBLB,NUM),
  calls(ST,TBLB,OUT,NUM), !.

```

Now enter the following goal at the **Goal:** prompt:

```
call([share,"M","W","T"],[],
     OutTBL,0)
```

Our inference engine (predicates **call** and **a\_tbl**) generates a binding table for all of the variables involved in satisfying the goal. The list of binding pairs will resemble:

```
OutTBL = [pair("M","X",0),
          pair("W","Y",0),
          pair("T",love,0),
          pair("X",philip,1),
          pair("Y",kumiko,1)]
```

In other words, the system is saying:

```
M = philip
W = kumiko
T = love
True
```

In our expert system shell we wrote a translation predicate that automatically translates the final binding table into the usual Turbo Prolog answer.

## THE BINDING PROCESS AS A UNIFICATION ALGORITHM

The Turbo Prolog predicate that performs the unification and binding of variables is called **assign\_table** or **a\_tbl** for short. Listing 2 shows the binding table module.

The main predicate in the binding table module, **a\_tbl**, is called by **call** and **calls** (see Listing 3). **call**, as presented so far, has one parameter: the predicate we want to execute. But as it stands, this predicate does not work for predicates with variables. To resolve this, we need to make some changes. We will add two parameters to **call**; both will be lists. Figure 2 shows the new structure.

Usually when we execute a goal, we start with the **IN-list** as an empty list. This means we start with an empty list of pairs, that is, an empty binding table. If we have the goal

```
Goal: call([likes,"X","Y"],[],
          OUT,0)
```



and have asserted the following **cl** fact in our knowledge base

```
cl([likes,evelyn,art], []).
```

then the **Out-list** would be:

```
OUT = [pair(evelyn,"X",0),
       pair(art,"Y",0)]
```

Things get more complex when we have rules in our knowledge base that match the goal. For example the asserted rule

```
cl([likes,"X","Z"],
   [ [likes,"X","Y"],
     [likes,"Y","Z"]
   ]).
```

causes recursion to take place.

This means that we have nested calls to the predicate **likes**. **a\_\_tbl** takes care of this complexity by matching the goal predicate with rules as well as facts. To see how, consider the abstract form of **a\_\_tbl** in Figure 3.

Note that the **a\_\_tbl** takes a predicate **P** and tries to match it with a predicate **S** in the knowledge base. **S** can either be the head or a member of the body of an asserted **cl** fact.

**a\_\_tbl** takes an input list of binding pairs (possibly empty) and outputs another list of pairs with variables matched to either variables or constants. The number of nested calls, **NUM**, is usually set to 0 in **call**. **call** then increments **NUM** whenever there is a nested call to the same predicate.

In **a\_\_tbl**, the asserted predicate and the **goal** must have the same symbol (name) or they will not match. Remember, the name of a predicate is always the first member of its list representation. Once the predicate names match, **a\_\_tbl** removes the first member of each list and passes the rest of each list to **bTBL**. **bTBL** takes one parameter from the asserted relation and one parameter from the goal and calls **cTBL**, which compares them:

```
a__tbl([NAME|Slist],
       [NAME|Vlist],IN,OUT,NUM) if
  bTBL(Slist,Vlist,IN,OUT,NUM).
```

```
bTBL([],[],TBL,TBL,_) if !.
bTBL([S|ST],[V|VT],IN,OUT,NUM) if
  cTBL(S,V,NUM,IN,OUTA),
  bTBL(ST,VT,OUTA,OUT,NUM).
```

*continued on page 96*

#### LISTING 4: A\_CALL.PRO

```
/* Executing built-in predicates by the clause-processor. */

domains
  INTlist = INT*
  REALlist = REAL*
  Llist = PARMLIST* /* Llist is a list of lists of SYMBOLS */

database
  cl(PARMLIST,Llist)

include "tbl-mod.pro" /* tbl-mod.pro is the program in
                       Listing 2 */

predicates
  call(PARMLIST,PAIRLIST,PAIRLIST,INT)
  calls(Llist,PAIRLIST,PAIRLIST,INT)
  a__call(PARMLIST)

  strint(PARMLIST,INTLIST)
  strreal(PARMLIST,REALLIST)

clauses

/* ***** "call" & "calls", The clause processor ***** */

call(P,IN,OUT,NUM) if
  cl(S,[]),
  a__tbl(S,P,IN,OUT,NUM), !.

call(P,IN,OUT,NUM) if
  cl(S,[H|B]),
  a__tbl(S,P,IN,TBLA,NUM),
  calls([H|B],TBLA,OUT,NUM), !.

call(P,IN,OUT,NUM) if
  cl(S,[H|B]),
  NUMA = NUM + 1,
  a__tbl(S,P,IN,TBLA,NUMA),
  calls([H|B],TBLA,OUT,NUMA), !.

calls([],TBL,TBL,_)

calls([SH|ST],IN,OUT,NUM) if
  a__tbl(SH,P,IN,TBLA,NUM),
  call(P,TBLA,TBLB,NUM),
  calls(ST,TBLB,OUT,NUM), !.
```



```

/* ***** "a_call" calling the built-in Turbo Prolog predicates ***** */

a_call(["+",X,Y,Z]) if
  strreal([X,Y],[A,B]),
  C = A + B,
  str_real(Z,C),
  !.
a_call(["-","_","_","_"]) if !, fail.

a_call(["-",X,Y,Z]) if
  strreal([X,Y],[A,B]),
  C = A - B,
  str_real(Z,C),
  !.
a_call(["-","_","_","_"]) if !, fail.

a_call(["*",X,Y,Z]) if
  strreal([X,Y],[A,B]),
  C = A * B,
  str_real(Z,C),
  !.
a_call(["*","_","_","_"]) if !, fail.

a_call(["/",X,Y,Z]) if
  strreal([X,Y],[A,B]),
  C = A / B,
  str_real(Z,C),
  !.
a_call(["/","_","_","_"]) if !, fail.

a_call(["mod",X,Y,Z]) if
  strint([X,Y],[A,B]),
  C = A mod B,
  str_int(Z,C),
  !.
a_call(["mod","_","_","_"]) if !, fail.

a_call(["=",X,X]) if !.

a_call(["round",X,Y]) if
  bound(X),
  str_real(X,A),
  B = round(A),
  str_real(Y,B).

a_call(["sin",X,Y]) if
  bound(X),
  str_real(X,A),
  B = sin(A),
  str_real(Y,B).

```

So **cTBL** gets two parameters from **bTBL**, namely, a string **S** from the asserted predicate and a string **V** from the goal. **cTBL** does a number of tests on these parameters before it takes action.

### call REVISITED

If the predicate is a fact, then a match with an asserted **cl** fact in the knowledge base will simply return **True**. For example, if we have the following clause in our knowledge base

```
cl([likes,evelyn,art],[ ])
```

and we give the goal

```
Goal: call([likes,evelyn,art])
```

it will succeed.

The situation is different when the called predicate is a rule. The body of a database fact is a list of predicates. Each predicate in the body constitutes a subgoal or a condition. For example, in this rule

```
cl([likes,evelyn,art],
  [ goes_often_to,evelyn,"X"],
  [museum,"X"]
]).
```

the sublists `[goes_often_to,evelyn,X]` and `[museum,X]` are conditions to be satisfied before a conclusion about `[likes,evelyn,art]` is reached. Here, each subgoal in the body must be executed first before any conclusion can be reached. This means two calls are needed to match the two subgoals with facts or rules in the knowledge base. Notice that we have a variable in the first and second subgoal, namely **X**.

### ACCESSING BUILT-IN PREDICATES

In Listing 4 you can see the program for the clause processor with additional **a\_call** rules. These rules allow you to make use of some of Turbo Prolog's built-in predicates. For example, these clauses let you use the plus operator:

```

a_call(["+",X,Y,Z]) if
  strreal([X,Y],[A,B]),
  C = A + B,
  str_real(Z,C),!.
a_call(["+","_","_","_"]) if !, fail.

```



To add 2 and 3, we would give the goal

```
a_call(["+", "2", "3", Sum]).
```

and the result would be:

```
Sum = "5"
```

To keep things simple we did not add these `a_call` clauses to our expert system shell (TESS). However, you may want to add them if you're building your own shell.

By the way, this approach to implementing metapredicates is built in to the inference mechanism, and is therefore specific to an expert system or expert system shell. However, these same techniques can be generalized in a Prolog interpreter, giving you full meta capabilities in other applications. Happy metaprogramming! ■

*I would like to thank Philip Seyer for his help in preparing this article.*

## REFERENCES

Bown, Kenneth A. "Meta-Level Programming and Knowledge Representation." Journal of New Generation Computing, Tokyo: OHMSHA Ltd. and New York: Springer-Verlag, 1985.

Bown, Kenneth A., and T. Weinberg. "A Meta-level Extension of Prolog." Cohen, J. and Conery, J., eds. Proceedings of the 1985 Symposium on Logic Programming, Washington, D.C.: IEEE Computer Society Press, 1985.

Miyachi, T. et al. "A Knowledge Assimilation Method for Logic Database." Journal of New Generation Computing, Tokyo: OHMSHA Ltd. and New York: Springer-Verlag, 1984.

Pereira, Fernando C.N. and Stuart M. Shieber. "Prolog and Natural-Language Analysis." CSLI Lecture Notes No. 10, Stanford, CA: Center for the Study of Language and Information, 1987.

---

*Safaa H. Hashim is a graduate student in the Computer Science Division, University of California, Berkeley.*

---

*Files may be downloaded from CompuServe as META.ARC.*

```
a_call(["cos", X, Y]) if
    bound(X),
    str_real(X, A),
    B = cos(A),
    str_real(Y, B).

a_call(["graphics", Mode, Palette, Background]) if
    strint([Mode, Palette, Background], [M, P, B]),
    graphics(M, P, B).

a_call(["dot", Row, Col, Color]) if
    strint([Row, Col, Color], [IRow, ICol, IColor]),
    dot(IRow, ICol, IColor).

a_call(["line", Row1, Col1, Row2, Col2, Color]) if
    strint([Row1, Col1, Row2, Col2, Color],
           [IRow1, ICol1, IRow2, ICol2, IColor]),
    line(IRow1, ICol1, IRow2, ICol2, IColor).

a_call(["true"]).
a_call(["fail"]) if
    fail.

a_call(["!"]) if
    !.

a_call(["not"|T]) if
    bound(T),
    not(a_call(T)),
    !.

/* ***** streal & strint ***** */

streal([], []).
streal([SH|ST], [RH|RT]) if
    bound(SH),
    str_real(SH, RH),
    streal(ST, RT).

strint([], []).
strint([SH|ST], [IH|IT]) if
    bound(SH),
    str_int(SH, IH),
    strint(ST, IT).

/* ***** */
```



SCRNASM.BOX

SCRNSUBS.BOX

ENTSUBS.BOX



# TURBO BASIC SCREENS AT ASSEMBLER SPEED

The Turbo Basic Database Toolbox provides new, fast tools for writing information to your screen.

David A. Williams



Snappy screen displays are one of the things that separate a professional, well-designed program from the "other" kind. Although compiled Turbo Basic programs will run rings around interpreted BASIC, there is always room for improvement in the screen-handling area. Large amounts of data take a noticeable amount of time to fill the screen, and it is difficult to regenerate a screen if you have overwritten part of it and only wish to restore that part.

There is only one way to get brisk displays that appear to snap into place—through assembly language routines that write directly to video memory. While writing such routines is not difficult, many people prefer not to dabble in assembly language. If you are among the hesitant, help is at hand. The Turbo Basic Database Toolbox has a collection of assembly language routines and Turbo Basic subprograms that require no knowledge of either assembly language programming or of the Turbo Basic/assembly language interface. In this article, I will show you how to use these routines with a program that you can employ to generate pop-up menus.

Before you embrace this technique, you should know that programs that write directly to video memory are sometimes termed "ill-behaved" because of two problems. First, these programs can interfere with windowing in multitasking environments. Since the task manager can't detect and inhibit screen writing, a program running in the background might overwrite the display of another program running in the foreground. Second, there is always the possibility, albeit remote, that your program won't run on some computers if the video display doesn't follow the IBM design. Nevertheless, many highly regarded programs use this method and the risk is minimal, especially if you are writing code for your own use.

## VIDEO BASICS

Before you can read from or write to video memory, you need to know where it is. The video memory address depends on the video adapter you're writing to, and the mode in which it is operating. Monochrome adapters, including the Hercules adapter, use memory starting at segment B000H, while the CGA uses memory starting at segment B800H. The EGA and VGA adapters can use either address depending on which monitor is attached: If a color monitor is installed, video memory starts at B800H; for a monochrome monitor, video memory starts at B000H. The video mode is indicated by a number between 1 and 15 maintained by the ROM BIOS; this number tells you all you need to know. If it's less than seven, you have a color system operating in text mode; if it's seven, you have a monochrome system; if it's greater than seven, you are operating in a graphics mode. The Toolbox routines only work in text mode.

You can read and write to a monochrome adapter's video memory at any time without a problem. On the other hand, some early color adapters generate interference, called *snow*, if you write to their video memory when the video circuitry is also addressing that memory to refresh the display. To avoid snow, we have to read and write during the horizontal or vertical retrace period when the screen is blanked. Don't worry about the details, because the Toolbox routines take care of it automatically.

You also need to know how the data in the video memory is organized. A 25-by-80 screen holds 2000 characters. Each character requires two bytes in video memory. The first holds the ASCII representation of the character, and the second holds the video attribute that determines the appearance of the character on the screen. The video attribute byte is bit-mapped, with different bits carrying different meanings. Figure 1 shows how the attribute byte is

*continued on page 100*



	1=	Background			1= High-	Foreground		
	Blink	bits			light	bits		
Invisible	0	0	0	0	0	0	0	0
Underline	0	0	0	0	0	0	0	1
Normal	0	0	0	0	0	1	1	1
Reverse video	0	1	1	1	0	0	0	0

Figure 1. The monochrome attribute byte.

	1=	Background			Intensity	Foreground		
	Blink	Colors				Colors		
Black	0	0	0	0	0	0	0	0
Blue	0	0	0	1	0	0	0	1
Green	0	0	1	0	0	0	1	0
Cyan	0	0	1	1	0	0	1	1
Red	0	1	0	0	0	1	0	0
Magenta	0	1	0	1	0	1	0	1
Brown	0	1	1	0	0	1	1	0
Light grey	0	1	1	1	0	1	1	1
Dark grey	0				1	0	0	0
Light blue	0				1	0	0	0
Light green	0				1	0	1	0
Light cyan	0				1	0	1	1
Light red	0				1	1	0	1
Light magenta	0				1	1	0	1
Yellow	0				1	1	1	0
White	0				1	1	1	1

Figure 2. The color attribute byte.

interpreted by monochrome displays; Figure 2 shows how the attribute byte is interpreted for color displays. Note that the "light" colors (colors with the intensity bit set to 1) are not legal background colors, since the background color is specified by only three bits.

**Note that the "light" colors (colors with the intensity bit set to 1) are not legal background colors, since the background color is specified by only three bits.**

If you want your program to run automatically on either color or monochrome systems, it must determine what system it is running on and make the necessary adjustments. The Toolbox routines do part of this for you, but you must determine the attribute that will give the desired effect on both color and monochrome monitors. Some attributes intended for a color monitor are visible on a monochrome monitor, but in general, you must use different attributes for each, especially if you are generating highlighted or reverse video text. For example, an attribute byte of 17H gives you white letters on a blue background on a color monitor, but does not produce visible text on a monochrome monitor.

#### THE TOOLBOX ROUTINES

The Turbo Basic Database Toolbox includes a collection of routines for reading and writing to



video memory in monochrome or color text modes, and for performing several other video functions.

The file **SCRNASM.BOX** from the distribution disk contains the following seven assembly language routines:

- CurPos** Sets the cursor position.
- CurSize** Sets the cursor size.
- GetVid** Returns the current video mode.
- SetVid** Sets the current video mode.
- Scroll** Scrolls a screen section up or down one row.
- ReadVid** Reads text and attributes from video memory.
- WriteVid** Writes text and attributes to video memory.

The first five use the **CALL INTERRUPT** statement to call interrupt 10H, the ROM BIOS video service interrupt. The last two are assembly language **INLINE** subprograms. The **INLINE** routines are formatted so that a comment statement beside each opcode string contains the assembly language source code for the opcodes on that line. This also allows you to alter them, but there is little sense (and considerable danger) in doing so.

These low-level routines are versatile, but require some setup work. The file **SCRNSUBS.BOX** contains the following three Turbo Basic subprograms:

**ScrnInit**—Determines the video mode and initializes the shared variables used by the other routines.

**SaveScreenArea**—Saves the text and screen attributes of a specified screen area.

**WriteScreenArea**—Writes text and attributes to a specified area of screen memory.

These three routines perform the necessary setup and then call the assembly language routines to do the real work. For most applications, these three high-level routines make it unnecessary to deal with the assembly language routines themselves.

## A SAMPLE PROGRAM

**MENU.BAS** (Listing 1) illustrates several of the Toolbox screen handling routines. It generates a pop-up menu with a selection bar and a one-line prompt at the top of the screen for each menu selection. You make a selection by moving the menu bar with the up or down arrow keys, and then pressing Enter when the bar highlights your choice. The prompt changes as the selection bar moves over the selections. Alternatively, you can make a selection by pressing a key corresponding to the number of the selection.

The main program calls the three routines from **SCRNSUBS.BOX**; these in turn call several of the assembly routines from **SCRNASM.BOX**. The program begins with two **\$INCLUDE** metastatements that include **SCRNASM.BOX** and **SCRNSUBS.BOX** into the program's logic. Since all numerical values passed to these routines must be integers, **SCRNSUBS.BOX** also executes a **DEFINT a-z** state-

ment, defining variables **a** through **z** as integers.

Next, the program clears the screen and defines a function that we'll use to get input from the keyboard. Then, a call to **ScrnInit** determines the video mode and returns it in the shared variable **Scrn.Mode** (defined in **SCRNSUBS.BOX**). **ScrnInit** also initializes two variables used by the other **SCRNSUBS.BOX** routines. One, **Scrn.Segment**, contains the segment address of video memory. The other, **Scrn.Retrace**, indicates whether it is necessary to restrict video memory access to the retrace periods. Since the main program uses neither **Scrn.Segment** nor **Scrn.Retrace**, their names don't appear in **MENU.BAS**.

The block **IF** statement establishes variables containing the screen attributes according to the screens, 07H is normal white text on a black background; 70H is reverse video. On color screens, 17H produces white text on a blue background; 71H produces the  
*continued on page 102*

**New!!**

YOU WON'T BELIEVE YOU GOT ALONG WITHOUT IT...

## BoosterGraphics

Graphics Subroutine Library for Turbo Basic

BoosterGraphics gives Turbo Basic programmers the power to create sophisticated graphics effects and graphics based user interfaces - easily & quickly.

Features include:

- Multipage graphics
- High speed pixels, lines, circles
- Image blitter routines
- Easy graphics mode windows
- Instant pull-down menus
- Extended ASCII fonts
- Fast & flexible IMAGE TEXT
- Powerful graphics editor
- BoosterPaint included
- Familiar BASIC syntax for all commands

ALL IN GRAPHICS MODE!!

Compatible with CGA, EGA, VGA, MCGA and HERCULES standards

Introductory price \$55.00 • 30 day money back guarantee  
Assembly source code available • Not copy protected, no royalties  
Free technical support • Free updates if purchased before Jan. 31, 1988

To order, please call us:

**Suncloud Software, Inc.**  
101 West Ninth Street  
Durango, Colorado 81301  
(303) 247-0439

add \$3.00 shipping/handling U.S., \$5.00 Canada  
We ship any way you want - all major credit cards accepted.

Turbo Basic is a registered trademark of BORLAND INT'L



```

'This program generates a pop-up menu using assembly language routines
'from the Turbo Basic Database ToolBox. David A. Williams 12/1/87

$include "scrnsubs.box"           'The high-level Basic routines
$include "scrnasm.box"           'The low-level Assembly routines
cls                               'Clear the screen

'Define a function to get keyboard input
def fnink$
ink:
    i$ = ""
    i$ = inkey$
    if i$ = "" then goto ink
    fnink$ = i$
end def

'This call determines the video mode and from that the video memory
'address. Sets shared variables used by the other routines.
call scrninit

'Check video mode and set foreground & background colors accordingly.
if scrn.mode = 7 then
    norm = 7                       '07H   For mono systems
    rev = 112                      '70H
else
    norm = 23                       '17H   For color systems
    rev = 113                       '71H
    color 7,0                       'Sets screen color for main program
    cls
end if

'Establish arrays for the menu and the one-line prompt.
dim text$(10), Prompt$(8)

'This array contains the menu items and graphical characters to
'make a box
Text$(1) = chr$(218)+string$(10,chr$(196))+chr$(191)
Text$(2) = chr$(179)+" Choice 1 "+chr$(179)
Text$(3) = chr$(179)+" Choice 2 "+chr$(179)
Text$(4) = chr$(179)+" Choice 3 "+chr$(179)
Text$(5) = chr$(179)+" Choice 4 "+chr$(179)
Text$(6) = chr$(179)+" Choice 5 "+chr$(179)
Text$(7) = chr$(179)+" Choice 6 "+chr$(179)
Text$(8) = chr$(179)+" Choice 7 "+chr$(179)
Text$(9) = chr$(179)+" Quit Q "+chr$(179)
Text$(10) = chr$(192)+string$(10,chr$(196))+chr$(217)

'This array contains the prompts
Prompt$(1) = "This is the prompt for menu choice 1" + string$(44," ")
Prompt$(2) = "This is the prompt for menu choice 2" + string$(44," ")
Prompt$(3) = "This is the prompt for menu choice 3" + string$(44," ")
Prompt$(4) = "This is the prompt for menu choice 4" + string$(44," ")
Prompt$(5) = "This is the prompt for menu choice 5" + string$(44," ")
Prompt$(6) = "This is the prompt for menu choice 6" + string$(44," ")
Prompt$(7) = "This is the prompt for menu choice 7" + string$(44," ")
Prompt$(8) = "    Press Q to end the program    " + string$(44," ")

'Fill the screen using standard Basic functions to simulate a program.
for i = 1 to 10
print " Now is the time for all men to come to the aid of the party."
print " The quick red fox jumped over the lazy dog's back."
next
locate 23,20
print "Press any key for menu"
i$ = fnink$

'Now save the screen.
'String variables SaveText$ and SaveAttr$ retain the data until
'we're ready to restore the screen.

```

*continued from page 101*

opposite. The **Color** statement sets the overall screen color to give white characters on a black background.

Now we set up two string arrays, one to hold the menu items and another to hold the text prompts associated with those menu items. The **Text\$** array contains graphical characters used to build a box around the menu. The use of arrays also makes it possible to use a **FOR..NEXT** loop to display the menu and to select the particular prompt we want to display, depending on the location of the selection bar.

Next, we fill the screen with lines of text to simulate a program in progress. The program uses Turbo Basic's **PRINT** statement to demonstrate how long it takes ordinary screen-handling code to fill the screen compared to the pop-up menu generated later. If you are using a 12- or 16-MHz processor, you won't notice much difference, but on a slower machine the improvement is dramatic. With the demonstration screen in place, the program waits for user input before displaying the menu.

When the user responds with a keystroke, the program saves the information displayed on the current screen so that the screen can be restored later. For simplicity's sake, we use the **SaveScreenArea** routine to store every character of the first 15 screen lines rather than trying to exactly match the section of screen used by the menu. You can, however, save any rectangular subset of the screen by passing to **SaveScreenArea** the coordinates of the upper left corner of the area to be saved, along with the number of rows and columns to be saved. The text and its video attributes are saved in separate string variables, **SaveText\$** and **SaveAttr\$**. Note that it is not necessary to declare these variables or set them up before calling **SaveScreenArea**, because they are declared along with the **SaveScreenArea** routine in **SCRNSUBS.BOX**.



The **WriteScreenArea** routine is used three times: first to display the menu, then to display the prompt in reverse video at the top of the screen, and finally to restore the screen before the menu subroutine terminates. The text and video attributes are contained in two separate strings of identical length. We could have displayed the entire menu with one call to **WriteScreenArea**, but writing the text and attribute information separately makes it easier to move the selection bar. As the program

**We could have displayed the entire menu with one call to WriteScreenArea, but writing text and attributes separately makes it easier to move the menu bar.**

cycles through the **FOR..NEXT** loop, it uses one of two attribute strings, depending on the position of the bar. The attribute string **Reverse\$** displays its underlying text in reverse video, and **Normal\$** displays its underlying text in normal video. Understand that the bar itself is nothing more than a line of attributes forcing the underlying text to reverse video.

The total number of bytes displayed by each call to **WriteScreenArea** is equal to the number of rows displayed multiplied by the number of columns displayed. The length of the text and attribute strings should each equal the product of rows multiplied by columns. If a string is longer, the last few bytes of the string are not displayed. If a string is shorter, **WriteScreenArea** displays what-

*continued on page 104*

```

UpperRow = 1
LeftCol = 1
NumberOfRows = 15
NumberOfCols = 80
call SaveScreenArea(UpperRow,LeftCol,NumberOfRows, _
                    NumberOfCols,SaveText$,SaveAttr$)

'Display the menu and process user inputs.
j = 0
while j = 0
  BarRow = 1 'Start on menu item one
  gosub menu 'Display the menu
  k=0
  while k = 0
    i$ = fnink$ 'Get user input
    k = asc(i$)
    if k < 49 then
      k = asc(right$(i$,1))
      if k = 13 then 'Enter key
                    'k will equal 1 thru 8
        k = BarRow
        'Up arrow key
        if BarRow > 1 then BarRow = BarRow - 1
        gosub menu
        k = 0
      elseif k = 72 then 'Down arrow key
        if BarRow < 8 then BarRow = BarRow + 1
        gosub menu
        k = 0
      end if
    end if
  wend

'Find the selection
select case k
case 1, 49
  S$ = "You have chosen selection 1"
  gosub action
  exit select
case 2, 50
  S$ = "You have chosen selection 2"
  gosub action
  exit select
case 3, 51
  S$ = "You have chosen selection 3"
  gosub action
  exit select
case 4, 52
  S$ = "You have chosen selection 4"
  gosub action
  exit select
case 5, 53
  S$ = "You have chosen selection 5"
  gosub action
  exit select
case 6, 54
  S$ = "You have chosen selection 6"
  gosub action
  exit select
case 7, 55
  S$ = "You have chosen selection 7"
  gosub action
  exit select
case 8, 81, 113 'Picks up 8, Q, and q
  j = 1 'Quit the program
end select

wend
cls
end

```



```
'Subroutine to display the menu
menu:
'Establish video attributes for normal line and
'reverse video selection bar
Normal$ = string$(12,chr$(norm))
Reverse$ = chr$(norm)+string$(10,chr$(rev))+chr$(norm)

'We'll use literals in the call instruction this time.
'Each pass thru the loop displays one line of the menu.
'The calling sequence is WriteScreenArea(UpperRow, LeftColumn,
'  NumberOfRows, NumberOfColumns, TextString, AttributeString)

for i = 1 to 10
  UpperRow = i + 1
  if i = BarRow + 1 then Attr$=Reverse$ else Attr$=Normal$
  call WriteScreenArea(UpperRow, 30, 1, 12, Text$(i), Attr$)
next

'Display the prompt
attr$ = string$(80,chr$(rev))
  call WriteScreenArea(1, 1, 1, 80, Prompt$(BarRow), Attr$)
return

'This routine simulates program action that would be taken after
'a menu selection.
action:
'Restore the screen
call WriteScreenArea(1, 1, 15, 80, SaveText$, SaveAttr$)

locate 23,20
print S$
locate 24,23
Print "Press any key to continue";
i$ = fnink$
return
```

ever data it finds in memory following the end of the string.

Once the menu has been displayed, the program waits for the user to make a selection. The inner **WHILE** loop accepts the input character from the user and formats it for the **SELECT CASE** statement. It tests for either the up arrow or down arrow keys, and if either key is pressed, it moves the menu bar appropriately. If a number key is pressed, the program uses the **SELECT CASE** statement to execute a section of code corresponding to the number pressed. If Enter is pressed, the program uses the position of the selection bar to derive an equivalent menu item number. The last menu item requires a "Q" or "q" character to quit the menu, and shows that you can use a letter as well as a number to make a selection.

The first seven menu selections each call a routine named **Action** that restores the screen and waits for user input before continuing. Then the program exits the **SELECT CASE** statement, and the outer **WHILE** loop cycles back to the user input section. When you choose **Quit** from the menu, the program exits the loop and ends.

### THE BOTTOM LINE

You can use this basic technique to design many types of menus and screen displays. Creating fast help screens is another application that comes to mind. The Toolbox designers have done the hard work of putting assembly language speed and power at your disposal. All you need is a little ingenuity in using these routines to give your programs a snappy, professional appearance. ■

---

*David A. Williams is a Principal Staff Engineer for a major aerospace company. He can be reached at 2452 Chase Circle, Clearwater, Florida 34624.*

---

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



# SELECT CASE: CHOOSING ONE FROM THE MANY

Like Turbo Pascal and Turbo C, Turbo Basic provides a way to choose one of many program paths based on a single value.

Ralph Roberts



SQUARE ONE

The **SELECT CASE** statement is a general purpose testing command, for selecting one path among several possible paths, depending on the value of a single expression or variable. Why *case*? A case is simply an instance of something. That ubiquitous little red sign found in older buildings, "In Case Of Fire, Break Glass," is a good example. Selecting a case, then, is simply tagging a desired response to a specific condition: If flames and smoke, then access the fire department subroutine. If there are many cases, one from the many must be chosen in a structured fashion without undue complication. **SELECT CASE** does this well, and is thus a first cousin to **CASE..OF** in Turbo Pascal and **switch** in Turbo C.

Those of us who have programmed in older BASICs recognize the use of the case concept in computer parlance. The following statements are all ways of testing a case in older versions of BASIC, as well as in Turbo Basic:

```
IF A = 1 THEN GOSUB 100
ON X GOTO 2000, 3000, 4000
IF (K = 4) AND (Z = 6) THEN END
```

These statements work quite well for less involved cases, but become increasingly complex when you are attempting to program response to, perhaps, a range of numbers. Sure, it can be done with **IF** statements, though awkwardly. To test for the range of 90 to 99 you might use the statement:

```
IF (X > 90) AND (X < 100) THEN
  GOSUB RangeReact
```

This is not too bad, and it kicks in nicely for the range we want. But what if you want this statement to trigger a subroutine call for one of ten different ranges? Sure, just add nine more lines of source code; after all, as the old saw goes, why should things be simple when they can be so beautifully complex?

There is a better way of implementing case selection. Here's how to test for the range of 90 to 99 using the **SELECT CASE** statement (i.e., doing the same thing as the one-liner above):

```
SELECT CASE X
CASE 90 - 99
  GOSUB RangeReact
END SELECT
```

In this simple example of testing for numbers in only one range, we are not saving much in terms of source code. What we *are* doing, however, is adding a measure of structure beyond the conditional test and the **GOSUB**. This helps the readability of the program, especially if the number of conditions to be tested for increases later on.

## THE SELECT CASE STATEMENT

The syntax for **SELECT CASE** is moderately straightforward:

```
SELECT CASE expression
CASE test list
  statements
[CASE test list
  statements]
[CASE ELSE
  statements]
END SELECT
```

Anything enclosed in brackets is optional. The **test list** is one or more tests to be performed for the indicated value of **expression**. The tests are separated by commas, and **expression** can be either string or numeric. However, **expression** and all the test lists within the **SELECT CASE** statement must be of the same type.

Legal tests for the **CASE** clauses include equality, inequality, greater than, less than, and (as we saw above) range. Some examples of **CASE** tests are given in Figure 1.

*continued on page 106*



## LISTING 1: RANDOM1.BAS

```
'RANDOM1  -- Ralph Roberts

CLS: ON KEY (10) GOSUB EndIt : KEY (10) ON : PRINT : PRINT
RANDOMIZE TIMER

GetRandomNumber:

A = INT(RND * 1000) + 1 : COUNT = COUNT + 1

SELECT CASE A
CASE < 11
PRINT A " was the first # in the range 1-10 to be hit,"
PRINT  " and it took " COUNT " tries."

CASE ELSE
GOTO GetRandomNumber

END SELECT

EndIt:
END
```

## LISTING 2: RANDOM2.BAS

```
'RANDOM2  -- Ralph Roberts

CLS: ON KEY (10) GOSUB EndIt : KEY (10) ON : PRINT : PRINT
RANDOMIZE TIMER

GetRandomNumber:

A = INT(RND * 1000) + 1 : COUNT = COUNT + 1

SELECT CASE A
CASE 90 to 99
PRINT A " was the first # in the range 90-99 to be hit,"
PRINT  " and it took " COUNT " tries."

CASE ELSE
GOTO GetRandomNumber

END SELECT

EndIt:
END
```

## SELECT CASE

*continued from page 105*

If a **CASE** clause has multiple tests, then there is an implied **OR** logical operator between each. In other words, the test clause

```
CASE 10, 20, 30, 40
```

means that a value of 10 or 20 or 30 or 40 causes it to trigger and execute the following statements until another **CASE** clause is encountered, or until one of the **END SELECT**, **CASE ELSE**, or **EXIT SELECT** clauses is encountered. **EXIT SELECT**, when encountered within a **SELECT CASE** statement, immediately terminates the statement and passes control to the statement following **SELECT CASE**. Should the program go through the **SELECT** statement without a case being satisfied, control passes to the next statement after the **END SELECT** statement.

## USING SELECT CASE IN PROGRAMS

Again, the **SELECT CASE** statement is very powerful for comparing a variable against a list of possible values. This allows you to write source code that is less awkward than trying to do the same thing with multiple **IF** statements.

Let's start with a simple example program that shows how to test for a specific range. The program, **RANDOM1.BAS**, is given in Listing 1. **RANDOM1** generates a random number in the range of 1 to 1000, then tests to see if that number fits within the range of 1 to 10. If it doesn't, then a new random number is generated and tested again. When one is finally found that falls within the specified range, that number and the number of tries the program took to find a match are printed to the screen.

It is good programming practice to provide an exit from a program in case it gets trapped in an endless loop. Therefore, key F10 is attached to a routine that ends the program. This is done through *event trapping*. Turbo Basic checks between each statement to see if the specified key has been pressed, giving control to the spe-



cified subroutine if it has. (For more on event trapping, see "Event Trapping in Turbo Basic," elsewhere in this issue.)

One more note: The statement **RANDOMIZE TIMER** uses your computer's clock to get a starting "seed" for the random number generator so that it won't always start the program by generating the same number.

The **CASE** test in Listing 1 is simplistic. If the random number is less than 11 (i.e., in the range of 1 to 10, since no negative random numbers are generated here), the two print statements are executed and control is given to the statement following **END SELECT**, ending the program. If the **CASE** is not satisfied, then the **CASE ELSE** statement is executed, and the program goes back and selects another random number. It is true that the **SELECT CASE** statement could have been placed within a **DO UNTIL** loop to avoid the use of the **GOTO**, but in this program it illustrates the logic of a **CASE ELSE** clause.

Let's take this program and make the single **CASE** test slightly more complex. Instead of looking for a range between zero and some positive number (which the single **IF** statement **IF A < 11 THEN...** could have done just as well), let's change the range to 90 to 99. The modified program is given in **RANDOM2.BAS**. (Listing 2).

Now we are starting to see the true power of **SELECT CASE**. Listing 2 works the same as Listing 1, except that it is now triggered by numbers in the range of 90 to 99. For more complex testing, **SELECT CASE** has a definite edge in time and memory over equivalent collections of **IF** statements.

Now, with another modification, we can make **SELECT CASE** really blow **IF** statements away. How about a list of 20 numbers! They can be *any* 20 numbers, in any order. Instead of 20 **IF** statements, we need only one **CASE** test. See *continued on page 108*

- CASE < X : relational (less than)
- CASE > X : relational (greater than)
- CASE 35 : equality (the "=" sign is assumed)
- CASE 90 to 99 : range (or "CASE X to Y")
- CASE 35, X : equality test (if X = 35)
- CASE 400 to 499, 35 : combinations (implicitly ORed, i.e., in the range of 400-499 or equal to 35)

Figure 1. Tests used in **SELECT CASE** clauses.

**TERRIFIC SCREENS IN UNDER THREE MINUTES. GUARANTEED.**

**T**hat's right. Saywhat, the lightning-fast screen generator, lets you build beautiful, elaborate, color-coded screens in minutes! We're talking about *incredible* screens for menus, data entry, data display, and help-panels that can all be displayed with as little as one line of code in *any* language. Even batch files.

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

**THE PERFECT TOOL FOR PROGRAMMERS.**

Whether you're a novice programmer long-ing for simplicity, or a seasoned pro searching for higher productivity, you owe it to yourself to check out the all-new Version 3.6 of Saywhat. It offers full monochrome emulation and lets you build your own elegant, moving bar menus into any screen. (They work like magic in *any* application, with just one line of code!) You can also combine your screens into powerful screen libraries. And Saywhat's remarkable VIDPOP utility gives all languages running under PC/MS-DOS, a whole new set of screen handling commands - languages like dBASE, Pascal, BASIC, C, Modula-2, FORTRAN, and COBOL. (You can make VIDPOP resident and available at all times,

**MONEY-BACK GUARANTEE**  
If you aren't completely delighted with Saywhat, return it within 30 days for a prompt, friendly refund.

or run it non-resident, transparently, with your favorite application).

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

**ORDER NOW. YOU RISK NOTHING.**

Thousands of satisfied users already know that Saywhat can make screen design and programming a pleasure, not a chore. Why not call toll-free, right now and put Saywhat to the test yourself? The next time you sit down to create a screen, we guarantee you'll be glad you did.

*Saywhat?!*  
**\$49.95**  
(Plus \$5 shipping and handling)

*yes!* I want to try your lightning-fast screen generator so send me \_\_\_\_\_ copies of Saywhat (\$49.95 plus \$5 shipping and handling) subject to your money-back guarantee.

Name \_\_\_\_\_

Address \_\_\_\_\_

City \_\_\_\_\_ State \_\_\_\_\_ Zip \_\_\_\_\_

Check enclosed    Ship C.O.D.    Credit card

# \_\_\_\_\_ Exp. date \_\_\_\_\_

Signature \_\_\_\_\_

To order:  
Call toll-free  
**800-468-9273**  
In California:  
**800-231-7849**  
In Canada  
**800-663-9361**  
International:  
**415-571-5019**  
The Research Group  
88 South Linden Ave.  
South San Francisco, CA 94080

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



## LISTING 3: RANDOM3.BAS

```
'RANDOM3.BAS -- Ralph Roberts

CLS: ON KEY (10) GOSUB EndIt : KEY (10) ON : PRINT : PRINT
RANDOMIZE TIMER

GetRandomNumber:

A = INT(RND * 1000) + 1 : COUNT = COUNT + 1

SELECT CASE A
CASE 50,100,150,200,250,300,350,400,450,500,_,
    550,600,650,700,750,800,850,900,950,1000

    PRINT A " was the first # in the TEST LIST to be hit,"
    PRINT " and it took " COUNT " tries."

CASE ELSE
GOTO GetRandomNumber

END SELECT

EndIt:
END
```

## LISTING 4: RANDOM4.BAS

```
'RANDOM4.BAS -- Ralph Roberts

CLS: ON KEY (10) GOSUB EndIt : KEY (10) ON : PRINT : PRINT
RANDOMIZE TIMER

GetRandomNumber:

A$ = CHR$(INT(RND * 26) + 65) : COUNT = COUNT + 1

SELECT CASE A$
CASE "A","E","I","O","U"

    PRINT "`"A$" was the first letter in the TEST LIST to be hit,"
    PRINT " and it took " COUNT " tries."

CASE ELSE
GOTO GetRandomNumber

END SELECT

EndIt:
END
```

**SELECT CASE***continued from page 107*

Listing 3, RANDOM3.BAS, which is our same random-number tester, except that it looks for a disjoint list of individual values rather than a continuous range of values.

It is possible to have much longer lists than this, too. In testing RANDOM3.BAS, I used a list that had 120 different numbers. Remember, however, when building longer lists, use the underscore character at the end of each line so that Turbo Basic will consider all the lines of your list to be part of one single list.

The **SELECT CASE** technique is not limited only to numbers. You may also use single letters or strings of letters with equal facility. Our program, revamped yet again to test for one letter out of a list of letters, is given in Listing 4, RANDOM4.BAS. Furthermore, you could also use whole words or phrases enclosed in quotes as the test cases, rather than single characters. Now, let's get back to numbers and have some fun with **SELECT CASE**.

**TURBO BASIC ELECTION CENTRAL**

Do you ever wonder how random the random numbers in your programs are? Turbo Basic does very well in distributing these numbers and, using **SELECT CASE**, we can prove it. Since 1988 is an election year, let's hold an on-screen primary to find out who our leader is going to be. The program, ELECTION.BAS, is given in Listing 5.

We'll generate a continuous stream of random numbers from 0 to 999 and test to see into which 100-count bracket each random number falls. For each bracket we'll display the percentage of the numbers belonging to that bracket. The distribution of the random numbers is a little ragged at first, but over a few thousand



numbers the distribution becomes flat within one or two percent. You'll find that this is a very close election. Pity the politicians who are not as efficient as Turbo Basic—or those who have some bugs in their closet ...

## CONCLUSION

The single page dedicated to **SELECT CASE** in the *Turbo Basic Owner's Handbook* merely hints at its incredible power. By using this statement in its many variations, you can write programs that deal effectively with multiple choices that would be very confusing otherwise. You can test long lists of numbers or strings for matches far more easily than with multiple **IF** tests.

**By using *SELECT CASE* in its many variations, you can write programs that deal effectively with multiple choices that would be very confusing otherwise.**

**SELECT CASE** is one of the reasons why so many of us have fallen in love with Turbo Basic—and Turbo Basic returns the favor by making us look good when people read and evaluate our programs. ■

*Ralph Roberts, WA4NUO, is a freelance writer who has written books on many topics, including Turbo Basic, Turbo Prolog, Reflex, and autograph collection.*

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

## LISTING 5: ELECTION.BAS

```
' ::::::::::::::: THE RANDOM ELECTION RETURNS :::::::::::::::
' ::::::::::::::: (a demonstration using SELECT CASE) :::::::::::::::
'
' -- Ralph Roberts

CLS : RANDOMIZE TIMER : ON KEY (10) GOSUB EndIt : KEY (10) ON

COLOR 15,0 : LOCATE 3,7 : ? "RANGE"
        LOCATE 3,21 : ? "HITS" : LOCATE 3,30 : ? "PERCENT"

FOR X = 0 TO 9
    LOCATE X + 5, 5
    ? USING "###";X * 100; : ? " -" (X * 100) + 99 : NEXT X : COLOR 7,0

LOCATE 16,5 : COLOR 0,7 : ? " TURBO BASIC Election Central "

LOCATE 17,5 :          ? "      (A demo of SELECT CASE)      "

GetRandomNumber:

A = INT(RND * 1000) : COUNT = COUNT + 1
COLOR 0,7 : LOCATE 1,5 : ? COUNT; " Random Numbers.... F10 Ends "
COLOR 7,0

SELECT CASE A

CASE 0 TO 99 : LOCATE 5,20 : COUNT1 = COUNT1 + 1
? COUNT1 : LOCATE 5,29 : ? USING "###.##";(COUNT1/COUNT) * 100;
? " %"

CASE 100 TO 199 : LOCATE 6,20 : COUNT2 = COUNT2 + 1
? COUNT2 : LOCATE 6,29 : ? USING "###.##";(COUNT2/COUNT) * 100;
? " %"

CASE 200 TO 299 : LOCATE 7,20 : COUNT3 = COUNT3 + 1
? COUNT3 : LOCATE 7,29 : ? USING "###.##";(COUNT3/COUNT) * 100;
? " %"

CASE 300 TO 399 : LOCATE 8,20 : COUNT4 = COUNT4 + 1
? COUNT4 : LOCATE 8,29 : ? USING "###.##";(COUNT4/COUNT) * 100;
? " %"

CASE 400 TO 499 : LOCATE 9,20 : COUNT5 = COUNT5 + 1
? COUNT5 : LOCATE 9,29 : ? USING "###.##";(COUNT5/COUNT) * 100;
? " %"

CASE 500 TO 599 : LOCATE 10,20 : COUNT6 = COUNT6 + 1
? COUNT6 : LOCATE 10,29 : ? USING "###.##";(COUNT6/COUNT) * 100;
? " %"

CASE 600 TO 699 : LOCATE 11,20 : COUNT7 = COUNT7 + 1
? COUNT7 : LOCATE 11,29 : ? USING "###.##";(COUNT7/COUNT) * 100;
? " %"

CASE 700 TO 799 : LOCATE 12,20 : COUNT8 = COUNT8 + 1
? COUNT8 : LOCATE 12,29 : ? USING "###.##";(COUNT8/COUNT) * 100;
? " %"

CASE 800 TO 899 : LOCATE 13,20 : COUNT9 = COUNT9 + 1
? COUNT9 : LOCATE 13,29 : ? USING "###.##";(COUNT9/COUNT) * 100;
? " %"

CASE > 899 : LOCATE 14,20 : COUNT10 = COUNT10 + 1
? COUNT10 : LOCATE 14,29 : ? USING "###.##";(COUNT10/COUNT) * 100;
? " %"

END SELECT

GOTO GetRandomNumber
EndIt:
END
```



# EVENT TRAPPING IN TURBO BASIC

Don't miss a Main Event—set a trap in Turbo Basic and catch them every time.

Ralph Roberts



WIZARD

The Declaration of Independence begins: "When in the course of human events ..." In that case, the body politic detected taxation without representation and a reaction occurred. Putting it into Turbo Basic terms, we might have:

```
IF Taxation = WithoutRepresentation
  THEN GOTO BostonTeaParty
```

Like humans, computer programs can react to events. Turbo Basic gives us a background processing capability called *event trapping*. You may use the techniques described in this article to both simplify and speed up various operations.

Event trapping allows a Turbo Basic program to continually check for certain events while the program as a whole is executing. An event can occur at any point during program execution, and your program will still respond to it.

Event trapping in Turbo Basic may be controlled precisely with the **\$EVENT** metastatement, turning the event checking on or off as desired for various portions of the program, as we'll demonstrate later in this article. First, let's define the various types of events that may be trapped.

Turbo Basic recognizes and reacts to six types of events. The first is a specified keystroke, trapped with **ON KEY**. With keystroke event checking, your program can be asking itself questions such as, "has F10 been pressed?" or "has the space bar been hit?" or "how about the combination of Alt-Shift-A?" between each statement.

Turbo Basic 1.0 allows you to set up as many as 20 of these keystroke event traps, all waiting patiently for keys at the same time. Turbo Basic checks all of the traps between every Turbo Basic program statement; detection of a specified keystroke or keystroke combination triggers a subroutine invocation.

The second type of event, trapping with **ON TIMER**, involves a countdown timer. A subroutine you write is given control when a specified interval has elapsed. For example, control might be given to

a trap subroutine that updates an on-screen clock when 60 seconds have elapsed. Thus, every minute, the clock would be advanced automatically.

The third type of event is the arrival of a character at a communications port. These events are trapped by **ON COM(n)**, where *n* is the number (either 1 or 2) of a serial communications port. Once this command is executed, Turbo Basic checks between the execution of every subsequent statement for characters arriving at the specified serial port. If a character is detected, control passes to the associated subroutine. The ins and outs of using **ON COM** were covered in "Turbo Basic Communications," (*TURBO TECHNIQ*, November/December, 1987).

The pressing of a joystick button is the fourth type of event that can be trapped by a program, through **ON STRIG**. The fifth type of event is a press on the light pen button, through **ON PEN**. Finally, the sixth type of event is the emptying of the background music buffer—as the buffer empties, the program can check to see if more notes need to be written to the buffer. This is handled through **ON PLAY**.

Event trapping in Turbo Basic begins with these various **ON <event type>** statements. We'll look more closely at the most commonly used event trapping statements: **ON ERROR**, **ON KEY**, **ON TIMER**, and **ON PLAY**.

## TRAPPING ERRORS

Good programming practice requires dealing courteously with errors. There is nothing so irritating to a user as having the program bomb and display some cryptic message such as :

```
Error 53 at prm-ctr: 42
```

They will growl and yell, "What the heck does that mean?"

Even we mighty programmers would have to do a little thumbing through the Turbo Basic manual to figure it out. Error 53 is simply "File not found." The



# ON

# ERROR

# GO TO

program counter number refers to the location in the source code of the failed statement. None of this is of any value to an end user.

The two-line program that generated this error was:

```
INPUT "File Name ",FileName$  
OPEN FileName$ FOR INPUT AS #1
```

When compiled and run, the program responds with the cryptic error message quoted earlier. I entered the filename "B:GROK," and since there is no file on drive B named GROK, the program went down in flames. This is a very common error, given the all-to-common propensity for pressing the wrong key, not having the right disk inserted, or forgetting to specify the correct path to the file.

The answer? Event trapping, of course. So, let's modify the program to recover from Error 53 and another common filename error, Error 64, "Bad File Name." The expanded and considerably more courteous program is given in Listing 1.

The first line uses **ON ERROR** to turn the error event trapping on. Whenever an error occurs, program control is switched to the **HandleErrors** subroutine. The question marks in that routine, by the way, are Turbo Basic shorthand for the **PRINT** statement. The variable **ERR** is automatically loaded with the number of the last error that occurred.

Should Error 53 or Error 64 occur—caused by a typo or non-existent filename being entered—

the program gives a polite and understandable error message, and recycles to let the user try again. This is the way *good* programs work.

A single error-handling routine can test **ERR** and take different courses of action depending on which error it detects. Listing 1 deals with only two errors, but for real programming you will want to develop error-handling subroutines that allow the program to recover from most common errors, or at least to gracefully exit in the event of an unforeseen one.

Once an error has been dealt with, control must be returned to some point in the program. The **RESUME** statement handles this task. There are several variations on the **RESUME** statement. The reserved word **RESUME** by itself (or **RESUME 0** if you still think in terms of line numbers) causes execution to return to the *same* statement that caused the error. **RESUME NEXT** transfers control to the statement *following* the one that caused the error. **RESUME <label>** transfers control to the statement at *<label>*.

As shown in Listing 1, once the error handler has notified the user that the filename entered was

not valid, the **RESUME GetA-File** statement transfers control back to the code that prompts the user for a filename. Keep in mind that once you've entered an error-trapping subroutine, *error trapping is disabled* until you execute a **RESUME** statement of some kind. Any error that occurs within an error trap brings execution to a halt with a cryptic error message, just as though error trapping did not exist. Make sure that your error handlers are watertight!

Also note that **RESUME** and **RESUME NEXT** add four bytes to your code file for *every* statement in a program, even if only a handful of those statements are capable of triggering errors. If at all possible, use **RESUME <label>** to keep the size of your final executable file to a minimum.

**ERRADR**, **ERL**, and **ERROR** are also associated with **ON ERROR**. These statements give your program access to information about its own errors, and can be used to build elaborate debugging machinery into your programs. We will cover these Turbo Basic statements in a later article.

## KEYING IN ON EVENT HANDLING

The **ON KEY** statement checks between every executed statement for the depression of a specified  
*continued on page 112*



## EVENT TRAPPING

continued from page 111

keystroke or combination of keystrokes. The general format is

```
ON KEY(n) GOSUB <label>
```

where *n* is an integer describing the key to be trapped and **label** is the first statement of your trap routine for that key.

The key (as it were) to **ON KEY** lies in specifying *n*. There are two layers to the process. The first layer is to make *n* a number from 1-31, with the numbers associated with specific keys as shown in Table 1. An example drawn from Listing 2 to trap function key F10 would be:

```
ON KEY(10) GOSUB KEY10  
KEY(10) ON
```

This may be enough if you want to trap the function keys or the cursor keys. However, to trap keys other than the cursor or function keys, you must go through the second layer.

This second layer involves defining up to ten keystrokes or keystroke combinations through the **KEY** statement. (This is a separate statement from both **ON KEY(n)** and **KEY(n)**.) The syntax is:

```
KEY n,CHR$(<shiftstatus>)_  
+CHR$(<scancode>)
```

Essentially, if you use **KEY** this way, it "builds" a keystroke from a shift status byte and a scan code byte and assigns that keystroke to *n*. This allows you to trap nearly any key on the PC keyboard, either alone or with any combina-

<i>n</i>	KEY
1-10	F1 - F10
11	Up arrow
12	Left arrow
13	Right arrow
14	Down arrow
15-25	KEY <i>n</i> definitions
30	F11
31	F12

Table 1. Key numbers for **ON KEY(n)**.

tion of the Ctrl, Alt, and Shift keys. **<scancode>** is a number from 1-83, specifying a scan code for one of the keys on the keyboard. These codes are given in Table 2. **<shiftstatus>** is a number from 0-255. It is a bit map specifying the state of the various shift-type keys for the desired keystroke. The meanings of the various bits are given in Table 3. For detecting multiple keys, simply add each key's bit value together. For example, to detect *either* the right or left shift key, you must add 1 and 2 and use a value of 3. To associate trap key 15 (*n* from **ON KEY(n)**) with Ctrl-Shift-S, you would execute this key statement:

```
KEY 15,CHR$(6)+CHR$(31)
```

The scan code of the S key is 31, and the 7 value for shift status is obtained by adding 1, 2 and 4, which are the bit values for the two shift keys and the control key, respectively.

Using **ON KEY(n)** only associates a specific key or keys with a specified trapping routine. Trapping itself does not begin until you explicitly enable it. Use **KEY(n) ON** to enable trapping key *n*, and **KEY(n) OFF** to disable trapping key *n*. You can turn key checking on or off within particular segments of your program by bracketing the section between **KEY(n) ON** and **KEY(n) OFF** statements.

An example of **ON KEY** event trapping is shown in Listing 2.

The first few lines of Listing 2 establish trapping routines for function keys F1, F2, F3, and F10. The numbers 1-10 always refer to the 10 function keys.

The operation of our example program is simple in the extreme. It enters a continuous loop, repeatedly printing the time to the screen. Between each statement in the loop, the program checks to see if one of the four defined keys have been pressed.

If the F1 key is pressed, execution of the loop is interrupted and control goes to the **KEY1** subroutine. A beep sounds and the pro-

gram returns to the loop. The F2 key routine works identically save that it sounds two beeps, and the F3 key sounds three beeps. The F10 key terminates the program by calling the subroutine **KEY10**.

This example is simplistic, but it shows the technique of trapping keystrokes. Of course your own subroutines will be much more complex. The F1 key might invoke a help screen (this is becoming fairly standard in the PC industry), the F2 key might be used to halt program operation, save data to disk, then continue, and so on.

## TIME FOR EVENT TRAPPING

The **ON TIMER** statement allows the execution of a subroutine after a specified delay. Its general format is

```
ON TIMER (n) GOSUB <label>
```

where *n* is the number of seconds to wait, from 1 to 86,400 (there are 86,400 seconds in 24 hours).

This command can be considered a real "sleeper." You could put it in a game program for your kids (or you) so that if play goes on too long, it would give a message such as "Go to bed!" or "You've played this game long enough!" then exit.

Let's look at a somewhat whimsical example of both time and key event trapping, with a little music and text animation thrown in for flavor. It's called "The Dancing Lady," given in Listing 3.

The figure on the screen is animated continuously by the loop after the label **Dance**. The **ON TIMER** trap counts seconds. Its trap subroutine, **Update**, breaks out of the animation loop once per second—just long enough to display the seconds count on the screen. The **ON KEY** trap provides an exit to the endless loop represented by **Dance**. Whenever an F10 keypress is detected, the program ends.

The dancing lady's music is generated by the **PLAY** statement,



KEY	SCAN CODE IN HEX	KEY	SCAN CODE IN HEX
Esc	01	Left shift	2A
! 1	02	\	2B
@ 2	03	Z	2C
# 3	04	X	2D
\$ 4	05	C	2E
% 5	06	V	2F
^ 6	07	B	30
& 7	08	N	31
* 8	09	M	32
( 9	0A	< ,	33
) 0	0B	> .	34
- _	0C	? /	35
= +	0D	Right shift	36
Backspace	0E	Prt Sc *	37
Tab	0F	Alt	38
Q	10	Spacebar	39
W	11	Caps Lock	3A
E	12	F1	3B
R	13	F2	3C
T	14	F3	3D
Y	15	F4	3E
U	16	F5	3F
I	17	F6	40
O	18	F7	41
P	19	F8	42
{ [	1A	F9	43
} ]	1B	F10	44
Enter	1C	Num Lock	45
Ctrl	1D	Scroll Lock	46
A	1E	7 Home	47
S	1F	8 Up Arrow	48
D	20	9 Pg Up	49
F	21	Grey -	4A
G	22	4 Left Arrow	4B
H	23	Pad 5	4C
J	24	6 Right Arrow	4D
K	25	Grey +	4E
L	26	1 End	4F
: ;	27	2 Down Arrow	50
" '	28	3 Pg Dn	51
` ~	29	0 Ins	52
F11	D9	. Del	53
F12	DA		

Table 2. Keyboard scan codes.

MODIFIER KEY	BINARY VALUE	HEX VALUE
Right Shift	0000 0001	01
Left Shift	0000 0010	02
Ctrl	0000 0100	04
Alt	0000 1000	08
Num Lock	0010 0000	20
Caps Lock	0100 0000	40

Table 3. The bit values for <shiftstatus>.

and repeats endlessly because of the **ON PLAY(n)** statement. The **MB** command at the start of the music encoding string tells Turbo Basic to play the tune in the background while the rest of the program executes in the foreground. This is done by "spooling" commands to the **PLAY** statement in a music buffer and passing notes to the PC's sound-generating hardware through an interrupt-driven scheme. By executing an **ON PLAY(5)** statement, an event occurs when only five notes remain in the music buffer. If the event-trapping subroutine submits the same music encoding string to the **PLAY** statement, the tune repeats as soon as it is finished.

**ON TIMER** and **ON PLAY**, as well as the other event trappers that check for their events between each statement, have a *trap stop* feature. In other words, once the trap is triggered and control is given to the trap's subroutine, checking between each statement is turned off. Without this feature, the subroutine could be called repetitively from within itself, eating stack space on each event until stack space is exhausted and the program crashes.

When the closing **RETURN** of the **Update** subroutine is encountered, the event trapping is automatically turned back on. Here, it would be equivalent to the **TIMER ON** statement but, depending on the event trap, it could also be **COM ON**, **PEN ON**, **PLAY ON**, **KEY ON**, or **STRIG ON**—those being the other event trappers that check between statements for events.

### SPEEDING UP EVENT-TRAPPED PROGRAMS

We have been enthusiastic about event trapping so far, and indeed the techniques described above are very effective programming tools. You should be aware, however, that there are no free

*continued on page 114*



## LISTING 1: ERRORS.BAS

```

ON ERROR GOTO HandleErrors

GetAFile:

INPUT "File Name " FileName$
OPEN FileName$ FOR INPUT AS #1
PRINT "File located. Thanks you!"
PRINT "Press RETURN:"
END

HandleErrors:

IF (ERR = 53) OR (ERR = 64) THEN ? :
? "Sorry, no such file. Please try again.": ?

RESUME GetAFile

```

## LISTING 2: KEYEVENT.BAS

```

CLS
ON KEY(1) GOSUB KEY1 : KEY (1) ON
ON KEY(2) GOSUB KEY2 : KEY (2) ON
ON KEY(3) GOSUB KEY3 : KEY (3) ON
ON KEY(10) GOSUB KEY10 : KEY (10) ON

COLOR 0,7 : LOCATE 25,15
? " F1 = 1 beep, F2 = 2 beeps, F3 = 3 beeps, F10 = Exit ";
COLOR 7,0

LoopAroundEndlessly:
LOCATE 10,35 : ? TIME$;
GOTO LoopAroundEndlessly

KEY1:
? CHR$(07); : DELAY .5 : RETURN

KEY2:
? CHR$(07); : DELAY .5 : ? CHR$(07); : DELAY .5 : RETURN

KEY3:
? CHR$(07); : DELAY .5 : ? CHR$(07); : DELAY .5 : ? CHR$(07);
DELAY .5 : RETURN

KEY10:
END

```

## EVENT TRAPPING

*continued from page 113*

lunches, and that event trapping involves a performance overhead.

Because a program with event trapping must check between each statement for the indicated event or events, program speed is reduced. In some cases, such as in a sort routine, this reduction in speed could be unacceptable. Event trapping also exacts an additional penalty on compiled program size. In an event-trapped program, the Turbo Basic compiler adds at least one byte more per statement to the program's .EXE file than it does to one without trapping enabled.

The obvious answer is to only enable event trapping during certain portions of the program. It's simply not needed everywhere. You can use such statements as **TIMER OFF**, **KEY OFF**, and **PLAY OFF** to disable trapping through portions of your program, and then use **TIMER ON** or **KEY ON** to re-enable it. Note that this does *not* help the problem of the extra event-trapping code.

Turbo Basic does provide an answer. The **\$EVENT** metastatement allows you to control the generation of event trapping code. The statement **\$EVENT OFF** in your program stops the compiler from generating the extra event-trapping code until the compiler encounters an **\$EVENT ON** command. (Remember that metastatements must always be on a line by themselves.)

Be aware of an important difference between the use of the **\$EVENT** metastatement and turning off trapping with a statement such as **KEY(1) OFF**. Both do, in effect, the same thing. However, the **KEY OFF**, **PLAY OFF**, **TIMER OFF**, **STRIG OFF**, and **PEN OFF** statements simply stop the pro-



gram from *reacting* to their traps, but the tests for the events are still made, which consumes some small but appreciable amount of time.

The **\$EVENT** metastatement, on the other hand, is a compiler directive that prevents this code from being generated in the first place. Hence, with less code to execute, your programs run more quickly.

You must coordinate the use of these two techniques, however. If you do a **KEY (1) ON** statement in an area of your program governed by **\$EVENT OFF**, the **KEY** statement will be inoperative because the event-checking code was never generated. The *default* condition in Turbo Basic is **\$EVENT ON**.

### DISCRETION IS THE BETTER PART OF TRAPPING

The use and control of event trapping in Turbo Basic programs is not merely desirable, but a necessity. Error recovery, program branching, timer control, recognizing and acting on keystrokes—these are things that separate “toy” programs from professional software products. Admittedly, every advantage is balanced by some disadvantage. You sacrifice some speed of operation and must accept larger executable files. Your skill as a programmer will dictate how effectively you balance the two sides of the equation with such tools as the **\$EVENT** metastatement. Use trapping only where you must, and you will use it best. ■

*Ralph Roberts, WA4NUO, is a freelance writer who has written books on many topics, including Turbo Basic, Turbo Prolog, Reflex, and autograph collection.*

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

#### LISTING 3: LADY.BAS

```
' THE DANCING LADY: a program showing ON TIMER, and ON KEY event
' trapping plus some nifty text animation
' and some unforgettable music

CLS

Song1$ = "MBT255N44N42N41N37N37N32N37N37N41N37N41N44N42N41"
Song2$ = "N42N39N39N32N39N39N42N39N42N46N44N42"
Song3$ = "N41N37N37N32N37N37N41N37N41N44N42N41"
Song4$ = "N42N41N42N39N44N42N41N37N37N37"

$$SOUND 75

Jig$ = Song1$+Song2$+Song3$+Song4$

ON TIMER (1) GOSUB Update : TIMER ON
ON KEY(10) GOSUB EndIt : KEY(10) ON
ON PLAY(5) GOSUB LoadJig : PLAY ON

PLAY Jig$ 'Play the song the first time

COLOR 0,7 : LOCATE 25,30 : ? " THE DANCING LADY... Press F10 to End ";
COLOR 7,0
Dance:

LOCATE 1,1
? "      /.\ \ "
? "      ( _
? "      * ) "
? "      ) ) "
? "      ( ) "

LegMovement:
? "      | | "
? "      | | "
? "-----" : DELAY .1
LOCATE 5,1
? "      / -- | "
? "      _ \ " : DELAY .1

LOCATE 5,1
? "      | -- \ "
? "      _ / _ " : DELAY .1 : GOTO Dance

LoadJig:
PLAY Jig$
RETURN

Update:

X% = CSRLIN      ' save cursor position
Y% = POS(0)

TIME = TIME + 1  ' increment seconds
LOCATE 2,20 : PRINT "The lady has been dancing ";TIME;" seconds.";
LOCATE X%,Y%    ' restore cursor
RETURN

EndIt:

CLS : END
```



# PAL CONTROL STRUCTURES

Structuring programs in PAL will be familiar territory if you know either Pascal or C.

Dan Shafer

For better or for worse, a language is known by its control structures. Pascal is the land of **REPEAT..UNTIL** and **CASE..OF**. C is famous for **switch**. BASIC, alas, is known for **GOTO**. Understanding how languages control their own flow of execution is critical to working effectively within them.

In this article, we'll examine the control structures in PAL. If you have some experience in Pascal or C, much of this will look like familiar territory. PAL borrows heavily from these two languages for its syntax and command names. Even if you don't "speak" Pascal or C, however, this article will help you to understand and deal with control structures.

## WHAT ARE WE CONTROLLING?

Before we examine the actual structures in PAL, we'll review why we use control structures in PAL or any other programming language.

Left to their own devices, programs written in procedural programming languages follow a blind, straight-ahead execution strategy. They carry out the first instruction they encounter, then the second, then the third, and so forth.

Real-life programs almost never work this way. To deal with the complexities of real life in procedural programs, we often do two things: branch and loop. Control structures alter a program's execution flow either by branching to other instructions rather than the next one in line, or by repeatedly executing a series of instructions in a loop until some condition causes the loop to end.

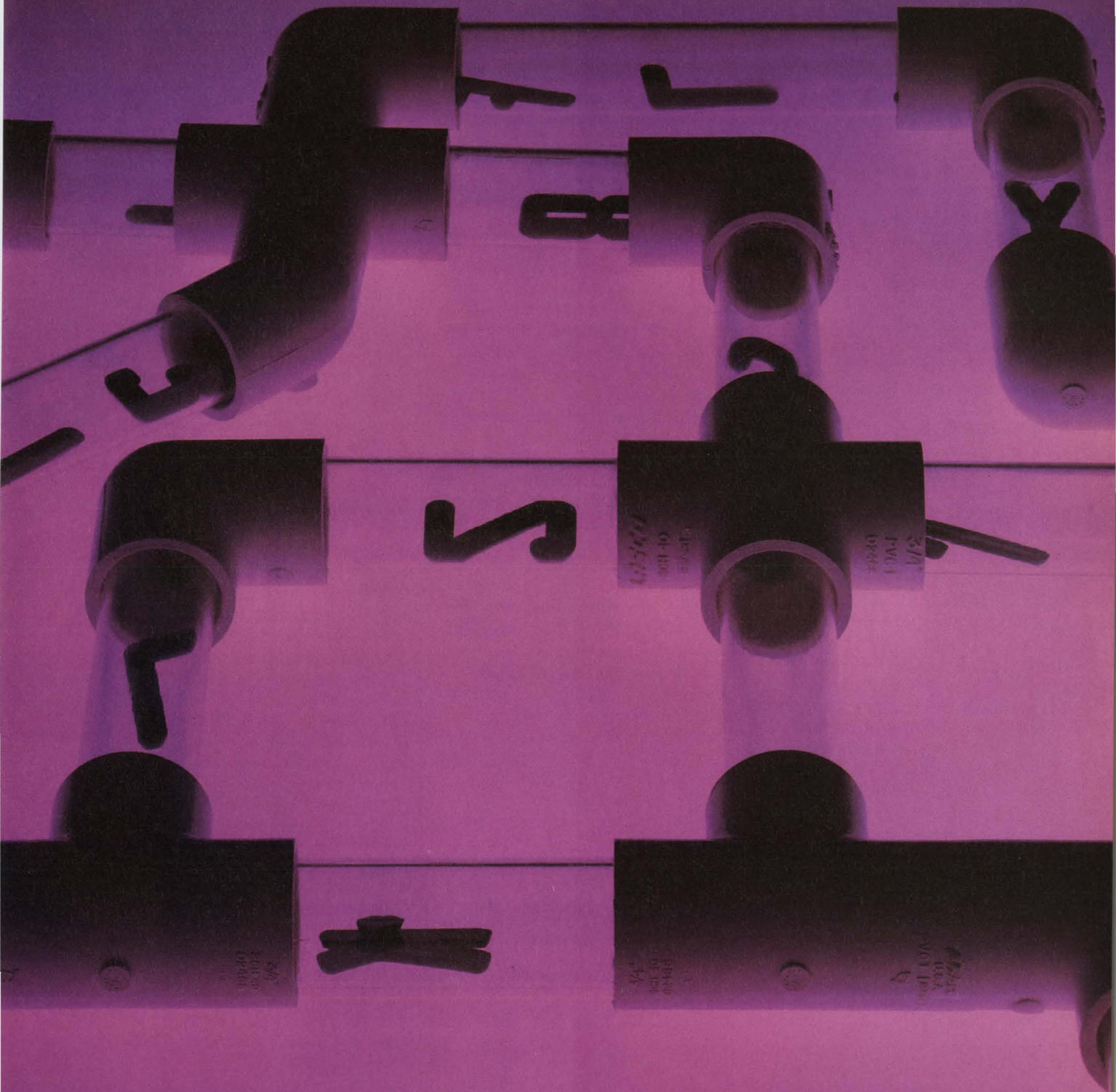
PAL offers two branching constructs: **IF..THEN..ELSE**, for binary branching; and **SWITCH**, for multiple-path branching.

The language also offers three types of loops: **FOR**, **WHILE**, and **SCAN**.

PAL also includes three ways to stop a loop before its natural conclusion: **RETURN**, **QUIT**, and **EXIT**.

*continued on page 118*







## PAL

continued from page 116

### BRANCHING INSTRUCTIONS

Within PAL, the easiest and most common type of branching is *binary*, or two-way, branching. If you have procedural language programming experience, you are already familiar with the nearly universal **IF..THEN..ELSE** construct. Such branches take the general form:

```
IF <some condition is true>
  THEN
    <execute 1 or more commands>
  ELSE
    <execute other commands(s)>
ENDIF
```

The **ELSE** clause is optional, but **IF**, **THEN**, and **ENDIF** must be included in any **IF** statement. If you omit the **ELSE** clause, the **IF** statement terminates if the condition is found to be false.

In other words, the omission of the **ELSE** causes the **IF** statement to be exited when the condition is not true.

Notice that the **IF** clause is executed only once. It tests the condition, and if the condition is true, certain instructions are carried out. If the condition is not true, other instructions, possibly outside of the **IF** statement, are executed.

Figure 1 is an example of an **IF..THEN..ELSE** statement in a PAL program. The program fragment in Figure 1 looks at a variable called **Result**. If **Result** is negative, it sets up a display style attribute that displays the value in red type on a blue background. If **Result** is positive, it sets up the attribute to display white on black. The last line of the listing uses the PAL question mark operator, **?**, to print the value of **Result**. Because **Result** is a number, it must be converted to a string so that the **?** operator can display it. This conversion is handled by the **STRVAL** function.

You can *nest* **IF** statements inside of other **IF** statements. The additional **IF** statements can be part of the **THEN** clause of the original **IF** statement, or they can be part of the **ELSE** clause. PAL places no arbitrary limit on the depth of such nesting. Figure 2

```
IF Result<0
  THEN STYLE ATTRIBUTE 20
  ELSE STYLE ATTRIBUTE 15
ENDIF
? STRVAL(Result)
```

Figure 1. The **IF..THEN..ELSE** structure.

shows a nested **IF..THEN..ELSE** statement. Briefly, this set of commands fills in the last name field of a record with the last name field of the preceding record if the user authorizes it. (In fact, this procedure would probably never be used in PAL because there are nonprogramming ways to carry forward a name from one record to another, but the procedure is illustrative nonetheless.)

The outermost **IF** statement checks to see if the **LastName** field is blank. If not, execution falls through all of the **IF** constructs. If **LastName** is blank, then the second **IF** statement checks the *previous* record's **LastName** field. If a name is found there, the user is asked if he wants to use the same name in the new record. If the user responds "Y," the procedure assigns the previous record's **LastName** field to the current record's **LastName** field and continues processing. Otherwise, the user is asked to supply a last name string.

The other branching structure in PAL is the **SWITCH** statement. If you are experienced in C, you'll notice that the PAL **SWITCH** statement is virtually identical to its C counterpart. If your experience is in Pascal, you will still recognize the construct as being fairly similar to the **CASE..OF** statement in that language. A **SWITCH** statement has this basic form:

```
SWITCH
  CASE <condition1> :
    <1 or more commands>
  CASE <condition2> :
    <1 or more commands>
  CASE <condition3> :
    <1 or more commands>
  OTHERWISE :
    <1 or more commands>
ENDSWITCH
```

```
IF (ISBLANK(ThisRecord[LastName]))
  THEN IF(NOT
    (ISBLANK(PrevRecord[LastName])))
    THEN ? "Use " +
      (PrevRecord[LastName]) + "?"
    IF (GETCHAR(=ASC("Y")))
      THEN
        (ThisRecord[LastName])=
        (PrevRecord[LastName])
      ENDIF
    ELSE ?
      "Please supply a last name!"
    ENDIF
  ENDIF
```

Figure 2. A nested **IF..THEN..ELSE** group.

You can have as many **CASE** clauses as you like; we've stopped at three for convenience, not for any technical reason. Each **CASE** clause has a condition associated with it. This condition is like those used with the **IF** statement. The conditions must evaluate to a Boolean, true/false result. Generally, conditions use logical operators for their tests.

The **OTHERWISE** clause is optional. If present, it handles conditions not covered by one of the **CASE** clauses.

In PAL, one of the most frequent uses for the **SWITCH** statement is in creating and using menus that look like Paradox's single-line menus. Using another built-in PAL command, **SHOWMENU**, your application can produce such a menu and store the user's response in a variable. The **SWITCH** statement can then be used to carry out specific instructions relevant to the choice. Usually, these are either **PLAY** statements, that result in Paradox loading and running other scripts (what Paradox calls program files), or the name of a procedure you've defined in your program. But **SWITCH** can also parcel out control to other kinds of statements, such as those used in the discussion of the **IF..THEN..ELSE** statement.

Figure 3 is a sample menu-handling routine. It assumes that the **SHOWMENU** command has been used and that the user's response is stored in the variable **MenuChoice**.

### LOOPING INSTRUCTIONS

Loops alter the normal sequential flow of program execution by causing one or more instructions



inside the loop structure to be executed more than once or skipped completely, based on some condition.

Table 1 provides guidelines for determining when to use each of PAL's three built-in loop structures. The basic format for the **FOR** loop is:

```
FOR <counter> FROM <startvalue>
    TO <endvalue>
    STEP <stepvalue>
    <1 or more commands>
ENDFOR
```

The counter is a variable. The value following the optional **FROM** clause tells where to start counting, and the value following the optional **TO** clause tells where to stop counting and terminate the loop. The value following the **STEP** clause (also optional) determines the incremental value to add to the variable each time through the loop.

Each time through the loop, all of the instructions between the **FOR** and the **ENDFOR** are executed. Figure 4 demonstrates the use of a **FOR** loop. When the loop in Figure 4 finishes executing, the cursor will be five rows lower in the table on the screen than it was before the loop began. Multiple-position cursor movement is one of the most common uses of **FOR** loops in PAL.

A **WHILE** loop's general structure looks like this:

```
WHILE <condition>
    <execute 1 or more commands>
ENDWHILE
```

The **WHILE** loop differs from the **FOR** loop in two significant ways. First, it executes an indeterminate number of times rather than a set number of times.

**SWITCH**

```
CASE MenuChoice = "Schedule" :
    PLAY "skednew"
CASE MenuChoice = "Cancel" :
    PLAY "apptxcl"
CASE MenuChoice = "Reschedule" :
    PLAY "resked"
CASE MenuChoice = "Exit" :
    QUIT
OTHERWISE :
    BEEP
    MESSAGE "To quit, select Exit"
ENDSWITCH
```

Figure 3. Menu-handling implemented with **SWITCH**.

Second, because it executes as often as needed, the **WHILE** loop can also execute zero times (i.e., be skipped completely). Figure 5 shows a modified version of our previous example containing a series of nested **IF** statements. Here, we've substituted a **WHILE** loop for the first **IF** statement. If **ThisRecord's LastName** field is not blank, the loop never executes. On the other hand, unlike the earlier **FOR** loop, this one will not let the user escape until he gives the program *some* data for **LastName**. (We could have designed the earlier example to force this as well, but that would have complicated things unnecessarily.)

The last of PAL's loop struc-

tures, **SCAN**, is unique to Paradox. Because Paradox is a database management system, PAL programmers frequently need to read through all of the records in a file, then retrieve or update them based on some criteria. Sometimes you need to update all of the records in a file. While this could be accomplished using the other looping structures and some built-in functions, PAL provides a structure specifically for this purpose. The form is simplicity itself:

```
SCAN FOR <condition>
    <1 or more commands>
ENDSCAN
```

*continued on page 120*

Loop type	When to use it
<b>FOR</b>	Use <b>FOR</b> when you know the number of times you want the loop to be carried out, or when the number of executions needed can be determined and placed in a numeric variable.
<b>WHILE</b>	Use <b>WHILE</b> when the loop must be executed an indeterminate number of times (i.e., as long as some condition is valid).
<b>SCAN</b>	Use <b>SCAN</b> when performing some action on all or selected records in a Paradox data file (table).

Table 1. Using the right loop at the right time in PAL.

TURBO C QUICK C LET'S C DESMET C DATALIGHT C ECO-C  
LATTICE C MICROSOFT C AZTEC C COMPUTER INNOVATIONS C

---

NEW --- Limited time offer.

## Peacock System's CBTREE

Object library for only \$49!

Our FULL COMMERCIAL VERSION of CBTREE in object library format is being offered for the amazingly low price of \$49.


CBTREE provides you with easy to use functions that maintain key indexes on your data records. These indexes provide you with fast, keyed access, using the industry standard B+tree access method.

Everything you need to fully utilize CBTREE in your applications is included. The CBTREE source code can be purchased later at any time for the \$110 difference. Example source programs and utilities are included FREE.

CBTREE source library	\$159
Object library only	\$49

This limited time offer is simply too good to refuse. Peacock's standard ROYALTY FREE, UNCONDITIONAL MONEY-BACK GUARANTEE, AND FREE TECHNICAL SUPPORT applies to this offer.

To order or for additional information  
call 1-800-346-8038 or (703) 847-1743 or write:



PEACOCK SYSTEMS, INC.  
2108 GALLOWS ROAD, SUITE C  
VIENNA, VA 22180

Trademarks: Turbo C (Borland); Quick C (Microsoft); Let's C (Mark Williams); DeSmet C (DeSmet Software); Datalight (Datalight); Lattice C (Lattice); Microsoft C (Microsoft); Aztec C (Manx Software); Computer Innovations C (Computer Innovations); Eco-C (Ecosoft, Inc).



The **FOR** is optional. If it is present, then the condition following it becomes a selection criterion. Only records that meet this criterion are affected by the commands in the loop. If it is omitted, then *all* of the records in the file are affected by the commands in the loop.

Figure 6 shows the use of the **SCAN** structure. Here, the system will award a bonus of \$1,000 to all of the sales staff in the Sales file

```
FOR counter FROM 1 TO 5
  DOWN
ENDFOR
```

Figure 4. A **FOR** loop.

who have year-to-date sales of \$100,000 or more. If their sales are below \$100,000, nothing happens.

We might want to apply a variable bonus to sales representatives based on year-to-date totals. Figure 7 shows how to do this using a **SWITCH** statement inside of a **SCAN** loop.

### EXITING LOOPS

I doubt there is a programmer alive who has never accidentally created an infinite loop. Such a loop never terminates because it never meets the condition it needs to meet before it can end. For example, if the condition calls for the value of a variable **X** to be greater than 50, and inside the loop the program is resetting the value of **X** to 1 each time, **X** is never going to get to 50. The loop will keep executing forever.

There are three ways to get out of loops before they terminate normally. The first, **RETURN**, is used only in complex PAL scripts and is not discussed here.

The other two are **QUIT** and **EXIT**. Either of these commands can be placed anywhere inside a loop. When the **QUIT** command is encountered inside a loop, it returns the user to Paradox as it leaves the loop. When **EXIT** is

```
WHILE (ISBLANK(ThisRecord[LastName])
  IF (NOT(ISBLANK(PrevRecord[LastName])
  THEN ? "Use " + (PrevRecord[LastName]) + "?"
  IF (GETCHAR(=ASC("Y")))
  THEN (ThisRecord[LastName])=
    (PrevRecord[LastName])
  ENDIF
  ELSE ? "Please supply a last name!"
  ENDIF
ENDWHILE
```

Figure 5. A **WHILE** loop.

```
SCAN FOR (Sales[YTD] >= 100000)
  (Sales[Bonus]) = 1000
ENDSCAN
```

Figure 6. Paradox's **SCAN** statement.

```
SCAN FOR (Sales[YTD]) > 0
  SWITCH
  CASE (Sales[YTD]) >10000 AND (Sales[YTD]) < 50000:
    (Sales[Bonus]) = 100
  CASE (Sales[YTD]) >= 50000 AND (Sales[YTD] < 100000:
    (Sales[Bonus]) = 500
  CASE (Sales[YTD]) >= 100000 :
    (Sales[Bonus]) = 1000
  ENDSWITCH
ENDSCAN
```

Figure 7. **SWITCH** within **SCAN**.

## WATCH THE END ZONE

If you are an experienced C or Pascal programmer, be careful of the **END** statement in PAL programming. In traditional programming languages, a single **END** form is used to terminate loops, branch constructs, and even programs and procedures. PAL, however, has a specialized **END** construct for each type of activity. For example, in an **IF** clause group, an **ENDIF** is used. Similarly, **ENDWHILE** and **ENDFOR** terminate the **WHILE** and **FOR** structures.

If you're new to PAL, be watchful; this can take some getting used to. ■

—Dan Shafer

encountered, the user is taken out of Paradox and directly into DOS. **QUIT** and **EXIT** are normally used in conjunction with one of the branching constructs, when some specific condition arises during a loop that makes further execution of the loop unnecessary or even dangerous.

As we've seen, PAL's control structures are as powerful as those of any popular high-level language. They let you program in a readable, structured way without "stretching" for the solution. If you need further evidence, consider that PAL has been used to generate enormous vertical market packages without a trace of the oldest control structure of all: **GOTO!** ■

*Dan Shafer is an independent consultant and freelance writer living in Redwood City, California. He has been using Paradox for more than a year and has fielded a dozen applications in PAL.*



# BINARY ENGINEERING

## Preconditions and Postconditions

Bruce Webster



**D**id you know that it's possible (in theory, at least) to make your program bug-free? In other words, you can write a complete specification of your program, then design and write your program to match that specification exactly. An entire branch of computer science, dealing with *program correctness*, is dedicated to doing just that, using such tools as propositions, assertions, predicate calculus, abstract data design, and flow analysis. Entire books have been written on the subject, some of which are listed at the end of this article.

"So where does that leave me?" you may ask. "All I want to do is write a program to play tic-tac-toe, not get a degree in computer science." Fair enough. However, there are some simple techniques that you can use to make even your tic-tac-toe program work correctly. Let's look at two of them: data abstraction, and the use of pre- and postconditions.

### DATA ABSTRACTION

Programs have one function—to manipulate information. And they are built from two items: algorithms and data structures. *Data structures* are named storage locations in memory and on disk that hold the information. *Algorithms* are sequences of instructions that perform the manipulation. Since we spend most our time working on the algorithms, we tend to approach programming from that

direction, with the data structures as something of an afterthought. In other words, we use a code-driven approach to writing our programs. For example, in writing a tic-tac-toe program, we might quickly sketch out the main body of the program in the following pseudo-code:

```
start new game
REPEAT
  get move
  make move
  update display
UNTIL game over
show results (winner or draw)
```

We can turn this almost verbatim into a main program, write the indicated subroutines, and proceed until the program works. This represents a good, solid, top-down structured approach to writing the program.

However, what if we use an abstract data approach instead? Rather than diving into our code, we're going to define the data types and structures needed to represent a game of tic-tac-toe. Furthermore, we're going to do it without worrying (for now) about which programming language we're using or how we're going to implement it. This is what is meant by the "abstract" in "abstract data type." We're approaching the problem in terms of the game itself, instead of in terms of the game's implementation. So let's look at the game of tic-tac-toe to see what data types we might need.

Playing tic-tac-toe means making moves, alternating Xs and Os, on a 3-by-3 grid. The game is over as soon as one of three conditions is satisfied: there are three Xs in a

row (horizontal, vertical, or diagonal); there are three Os in a row; or all nine locations are occupied. Once the game is over, the winner is the player with three symbols (X or O) in a row. If neither player wins, then the game is a draw.

Our most general data type is going to be **game**. There are two basic operations we want to perform on a game: set up a new game, and make a move at a given location. There are also some pieces of information we want to be able to extract from a game, such as how many moves have been made, whose move is next, what move (if any) has been made at a given location, whether or not the game is over, and who the winner (if any) is. So we now have the definition shown in Figure 1, using a Pascal-like pseudo-code notation.

We've got two other data types to define here: **move** and **location**. The type **move** consists of three values: X, O, and blank (no move yet). The only operation we would probably be interested in is an "opposite" function: if passed X, it returns O, and if passed O, it returns X. So we might define **move** as shown in Figure 2.

The last data type, **location**, is used to specify locations on the tic-tac-toe grid. Since the grid is 3-by-3, we can use the numbers 1 through 9 as shown in Figure 3. We don't need to perform any operations on the **location** data type itself; it will only be used as a parameter to some of the operators for **game**. We can now pro-

*continued on page 122*



```

unit TicTac;
{
  TICTAC.PAS  unit for implement abstract data types MOVE,
              LOCATION, GAME
  author      bruce f. webster
  last update 12 dec 87
}
interface

{ definitions for abstract data type Move }

type
  Move      = (BLANK,X,O);

function Opposite(M : Move) : Move;

{ definitions for abstract data type Location }

const
  GLim      = 9;

type
  Location  = 1..GLim;

{ defintions for abstract data type Game }

type
  Board     = array[Location] of Move;
  Game      = record
    Grid    : Board;
    Next,Win : Move;
    Moves   : Integer
  end;

function GetLoc(var G : Game; L : Location) : Move;
function NextMove(G : Game) : Move;
function MovesMade(G : Game) : Integer;
function GameOver(var G : Game) : Boolean;
function Winner(G : Game) : Move;
procedure DoMove(var G : Game; L : Location);
procedure NewGame(var G : Game; First : Move);

implementation

function Opposite(M : Move) : Move;
{
  purpose    return opposite of value passed
  pre        m has a value of X, O, or BLANK
  post       if m = X, then returns O
              if m = O, then returns X
              else returns BLANK
}
begin
  case M of
    BLANK : Opposite := BLANK;
    X      : Opposite := O;
    O      : Opposite := X
  end
end; { of proc Opposite }

procedure SetLoc(var G : Game; L : Location; M : Move);
{
  purpose    sets a location in the game to a given value
  pre        l is in the range 1..9
              m has a value of X, O, or BLANK
  post       location l in the game has value m
}
begin
  G.Grid[L] := M
end; { of proc SetLoc }

```

vide a definition for **location** as given in Figure 4.

**DATA TYPE:** game  
**PURPOSE:** represent a game of tic-tac-toe  
**CONTENTS:** contains complete tic-tac-toe game information, including tic-tac-toe grid, moves made, next move, game status, and winner  
**OPERATIONS:**  
 newGame(G : game; First : move)  
 doMove(G : game; Loc : location)  
 movesMade(G : game) : integer  
 nextMove(G : game) : move  
 getLoc(G : game; Loc : location) : move  
 gameOver(G : game) : boolean  
 winner(G : game) : move

Figure 1. The definition of **game**.

**DATA TYPE:** move  
**PURPOSE:** represent moves in a tic-tac-toe game  
**VALUES:** blank, X, O  
**OPERATIONS:** opposite(M : move) : move

Figure 2. The definition of **move**.

1	2	3
4	5	6
7	8	9

Figure 3. Locations on the tic-tac-toe grid.

We've now defined the abstract data types we need to play tic-tac-toe. This is all very fine, and it's probably a novel approach for most of you, but what does it have to do with program correctness? That's what we're about to see.



**DATA TYPE:** location  
**PURPOSE:** enumerate location on tic-tac-toe board  
**VALUE:** the integers 1 through 9  
**OPERATORS:** none

Figure 4. The definition of **location**.

### PRECONDITIONS AND POSTCONDITIONS

In our definitions of the abstract data types, we listed a series of operators for the types **game** and **move**, and we gave brief or implied descriptions of what each did. Now let's get a little more formal and give *preconditions* and *postconditions* for each one.

A precondition for an operator is just that: a condition that must be true before the operator is called. It is a restriction placed on the use of the operator. The user then has the responsibility of ensuring that all preconditions for a given operator are true, prior to calling the operator. If the user fails to do this, then the operator is no longer guaranteed to give the correct results. For example, a precondition of the operator **doMove()** might be that the game isn't over yet.

A postcondition for an operator is simply a result of that operator. You, the implementor, are guaranteeing that if all of the preconditions for a given operator are met, then all of the postconditions for that operator will be true. For example, a postcondition of the operator **doMove()** might be that the indicated location in the grid has been set to X or O, depending upon whose move it is next.

Consider the following sets of pre- and postconditions for the operators mentioned earlier:

**newGame(G : game; First : move)**

*pre:* **First** has the value X or O

*post:* The grid in **G** has been cleared, the game status has been set to not over, the number of moves made has been set to 0, the winner has been set to blank (none), and the next move set to **First**

*continued on page 124*

```
function GetLoc(var G : Game; L : Location) : Move;
{
  purpose  returns value of a given location in the game
  pre      g has been initialized, l is in the range 1..9
  post     returns value of g at location l
}
begin
  GetLoc := G.Grid[L]
end; { of proc GetLoc }

function NextMove(G : Game) : Move;
{
  purpose  returns next move
  pre      g has been initialized
  post     if game is not over
           then returns X or O
           else returns BLANK
}
begin
  NextMove := G.Next
end; { of func NextMove }

function MovesMade(G : Game) : Integer;
{
  purpose  returns number of moves made in game so far
  pre      g has been initialized
  post     returns a value in the range 0..9
}
begin
  MovesMade := G.Moves
end; { of func MovesMade }

procedure InARow(var G : Game; I,J,K : Location);
{
  puporse  checks for three X's or O's in a row
  pre      g has been initialized, 0 or more moves made
  post     if locations i,j,k all have the same value
           and that value is not BLANK
           then the winner is set to that value
}
begin
  with G do begin
    if Win = BLANK then begin
      if (Grid[I] = Grid[J]) and (Grid[J] = Grid[K])
        and (Grid[I] <> BLANK) then Win := Grid[I]
      end
    end
  end
end; { of proc InARow }

procedure CheckForWin(var G : Game; L : Location);
{
  purpose  see if last move won the game
  pre      g has been initialized, 1 or more moves made,
           l is in the range 1..9, location l has X or O,
           last move was made at location l
  post     if l forms 3 X's or O's in a row
           then the winner is set to that value (X or O)
}
begin
  case L of
    1 : begin
      InARow(G,1,2,3);
      InARow(G,1,5,9);
      InARow(G,1,4,7)
      end;
    2 : begin
      InARow(G,1,2,3);
      InARow(G,2,5,8)
      end;
  end;
```



```

3 : begin
  InARow(G,1,2,3);
  InARow(G,3,5,7);
  InARow(G,3,6,9)
end;
4 : begin
  InARow(G,1,4,7);
  InARow(G,4,5,6)
end;
5 : begin
  InARow(G,1,5,9);
  InARow(G,2,5,8);
  InARow(G,3,5,7);
  InARow(G,4,5,6)
end;
6 : begin
  InARow(G,3,6,9);
  InARow(G,4,5,6)
end;
7 : begin
  InARow(G,1,4,7);
  InARow(G,3,5,7);
  InARow(G,7,8,9)
end;
8 : begin
  InARow(G,2,5,8);
  InARow(G,7,8,9)
end;
9 : begin
  InARow(G,1,5,9);
  InARow(G,3,6,9);
  InARow(G,7,8,9)
end
end
end; { of proc CheckForWin }

function GameOver(var G : Game) : Boolean;
{
  purpose    returns status of game (over or not)
  pre        g has been initialized, 0 or more moves have been made
  post       if game is over
             then returns TRUE
             else returns FALSE
}
begin
  GameOver := (G.Win <> BLANK) or (G.Moves = GLim)
end; { of func GameOver }

function Winner(G : Game) : Move;
{
  purpose    returns winner of game
  pre        g has been initialized, the game is over
  post       if there are 3 X's in a row, returns X
             if there are 3 O's in a row, returns O
             else returns BLANK (draw)
}

```

## doMove(G : game; Loc : location)

*pre:* **G** is not over, **Loc** is in the range 1..9, and location **Loc** in the grid is blank

*post:* Location **Loc** in the grid is set to either X or O (depending on who had the next move), the number of moves made has been incremented, the next move has been set to the opposite value, and the game status and winner (if any) have been updated accordingly

## movesMade(G : game) : integer

*pre:* **G** is initialized (via **newGame**), and 0 or more moves have been made (via calls to **doMove**)

*post:* Returns the number of moves made so far (0..9)

## nextMove(G : game) : move

*pre:* **G** is initialized, and 0 or more moves have been made

*post:* If the game is not over, then returns the next player to move (X,O); if the game is over, then returns blank

## getLoc(G : game; Loc : location) : move

*pre:* **G** is initialized, and 0 or more moves have been made, **Loc** is in the range 1..9

*post:* Returns the contents of location **Loc** in the grid (X, O, or blank)

## gameOver(G : game) : boolean

*pre:* **G** is initialized, and 0 or more moves have been made

*post:* Returns **TRUE** if at least one of the following conditions is met: nine moves have been made; there are three Xs in a row (horizontally, vertically, or diagonally); there are three Os in a row

continued on page 126



# Paradox: The perfect relational database manager for both novices and advanced users

*Paradox® is so technically advanced that it's easy to use.*

You don't have to be a genius to use the new Paradox 2.0. 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.

That's because Paradox incorporates advanced artificial intelligence technology.

*Paradox encourages you to do things your own way*

With Paradox you'll soon be creating professional customized applications.

Most important of all, as your information management skills grow, and you want more creative freedom, Paradox stays out of your way.

What all this means to you is that you can concentrate on running your business and stay well ahead of the competition while Paradox takes care of the details.

*With Paradox, networks really work*

Paradox 2.0 also handles multiple users on a network, making powerful multiuser applications a reality. This in itself is quite an achievement.

“ Paradox is, indeed a unique blend of a user-friendly relational database with a highly sophisticated and powerful programming environment.

*Rusel DeMaria, PC Week ”*

“ We find Paradox preferable to every database system we've encountered so far, including dBASE III PLUS, for both everyday use and application development.

*Jay Alpersen & Steve King  
Data Based Advisor ”*

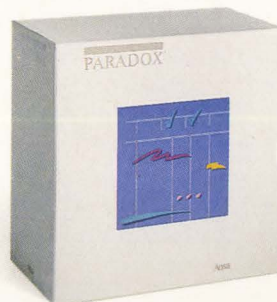
For a brochure or more information, please call (800) 543-7543

Paradox is remarkably easy to use.  
It's remarkably sophisticated.  
That's not a contradiction.  
That's innovation.

**PARADOX**

by Ansa

A Borland Company



PARADOX



```

}
begin
  Winner := G.Win
end; { of func Winner }

procedure DoMove(var G : Game; L : Location);
{
  purpose      make next move in game
  pre          g has been initialized, 0 or more moves made,
              the game is not over, l is in the range 1..9,
              getloc(g,l) is BLANK
  post        the next move is made at location l
              a possible win is checked
              if game is not over,
                then the next move is toggled
              else the next move is set to BLANK
}
begin
  with G do begin
    SetLoc(G,L,G.Next);
    Moves := Moves + 1;
    CheckForWin(G,L);
    if not GameOver(G)
      then Next := Opposite(Next)
    else Next := BLANK
  end
end; { of proc DoMove }

procedure NewGame(var G : Game; First : Move);
{
  purpose      initialize a new game
  pre          first has a value of X or O
  post        g has been initialized:
              locations 1..9 are set to BLANK
              the next move is set to first
              the winner is set to BLANK
              the number of moves is set to 0
}
var
  I          : Integer;
begin
  with G do begin
    for I := 1 to GLim do
      SetLoc(G,I,BLANK);
    Next := First;
    Win := BLANK;
    Moves := 0
  end
end; { of proc NewGame }

end. { of unit TicTac }

```

**winner(G : game) : move**

*pre:* The game is over  
(**gameOver(G)** is **TRUE**)

*post:* Returns X if there are three Xs in a row; else returns O if there are three Os in a row; else returns blank

**opposite(M : move) : move**

*pre:* M has the value X, O, or blank

*post:* Returns a value according to the following table:

M	returns
blank	blank
X	O
O	X

What have we accomplished by all of this? We've defined a set of operators that can be written bug-free. In other words, if the preconditions are met for each operator, we can guarantee the postconditions.

Notice that there is an important relationship between the preconditions and the set of operators: the set of operators is sufficient to let the user guarantee all preconditions. For example, the operator **doMove()** requires that the game not be over and that the location given be unoccupied. The operators **gameOver()** and **getLoc()** let the user test both of these conditions.

There are other subtle but important relationships between the operators. The pre- and postconditions of **doMove()**, for example, guarantee that there can never be both three Xs in a row and three Os in a row, avoiding potential bugs in **gameOver()** and **winner()**.

Finally, notice that we've done the bulk of our work without writing a single line of code! In fact, we haven't even locked ourselves into a given programming language. Let's see how we might implement this.



## IMPLEMENTING YOUR DESIGN

Most often, the best way to implement an abstract data design is as a separate module, such as a unit (in Turbo Pascal) or a separately compiled library (in Turbo C). This has three major benefits:

- *Program decomposition:* Breaks your program down into smaller, more manageable chunks, making it easier to code and debug.
- *Module independence:* Lets you create a module that you might be able to use in several programs.

**As you write your module, you may find you need additional operators, not for the end user, but for your own internal use.**

- *Information hiding:* Hides the actual implementation from the user, allowing you to make changes in the implementation itself without the user knowing or caring.

As you write your module, you may find that you need additional operators, not for the end user, but for your own internal use. This is perfectly normal and to be expected. You should write the same abstract specifications for these additional operators, complete with pre- and postconditions, to help ensure the internal correctness of your module. For example, here are three operators that you might end up creating for the **game** data type:

*continued on page 128*

### LISTING 2: GAMEIO.PAS

```
unit GameIO;
interface

uses CRT,TicTac;

type
  CharSet      = set of Char;
  MsgStr       = string[80];

procedure DisplayGame(theGame : Game);
procedure DrawGrid;
procedure ReadChar(var Ch : Char; Prompt : MsgStr; OKSet : CharSet);
procedure ReadInt(var Val : Integer;
                  Prompt : MsgStr;
                  Low,High : Integer);

implementation

const
  BoardX      = 10;      { positioning for tictactoe grid }
  BoardY      = 10;
  Bar         = #186;    { special characters used for grid }
  Line        = #205;
  Cross       = #206;

procedure DrawGrid;
{
  purpose     draws full-sized tictactoe grid,
              with smaller numbered one beside it
  pre         screen has been cleared
  post        two grids drawn on screen
}

procedure DrawHorz(X,Y : Integer);
{
  purpose     draws horizontal bar for tictactoe grid
  pre         x <= 63, y <= 23
  post        bar is written to screen
}
begin
  GoToXY(X,Y);
  Write(Line,Line,Line,Line,Line,Cross);
  Write(Line,Line,Line,Line,Line,Cross);
  Write(Line,Line,Line,Line,Line)
end; { of locproc DrawHorz }

procedure DrawVert(X,Y : Integer);
{
  purpose     draws vertical bars for tictactoe grid
  pre         x <= 78, y <= 16
  post        vertical bar appears on screen (with gaps for crosses)
}
var
  J,I        : Integer;
begin
  for J := 1 to 3 do begin
    for I := 0 to 2 do begin
      GoToXY(X,Y+I); Write(Bar)
    end;
    Y := Y + 4
  end
end; { of locproc DrawVert }
```



```

procedure DrawMoves(X,Y : Integer);
{
  purpose    draws 3x3 grid with numbered positions
  pre        x <= 77, y <= 21
  post       3x3 grid drawn on screen
}
begin
  GoToXY(X,Y); Write('1',Bar,'2',Bar,'3');
  GoToXY(X,Y+1); Write(Line,Cross,Line,Cross,Line);
  GoToXY(X,Y+2); Write('4',Bar,'5',Bar,'6');
  GoToXY(X,Y+3); Write(Line,Cross,Line,Cross,Line);
  GoToXY(X,Y+4); Write('7',Bar,'8',Bar,'9');
end; { of locproc DrawMoves }

begin
  DrawHorz(BoardX,BoardY);
  DrawHorz(BoardX,BoardY+4);
  DrawVert(BoardX+5,BoardY-3);
  DrawVert(BoardX+11,BoardY-3);
  DrawMoves(BoardX+20,BoardY)
end; { of proc DrawGrid }

procedure DisplayGame(theGame : Game);
{
  purpose    draws status of tictactoe game on screen
  pre        grid has already been drawn on screen
  post       contents of each grid location are displayed
}
var
  I,Col,Row : Integer;
  M          : Move;
begin
  for I := 1 to GLim do begin
    M := GetLoc(theGame,I);
    Col := BoardX + 2 + 6 * ((I-1) mod 3);
    Row := BoardY - 2 + 4 * ((I-1) div 3);
    GoToXY(Col,Row);
    case M of
      BLANK : Write(' ');
      X      : Write('X');
      O      : Write('O');
    end
  end
end; { of proc DisplayGame }

procedure ReadChar(var Ch : Char; Prompt : MsgStr; OKSet : CharSet);
{
  purpose    prompt for and get one character of a given set
  pre        okset is non-empty and contains valid uppercase
             characters (including digits, punctuation, etc.)
  post       readchar() returns some character contained in okset
}
begin
  GoToXY(1,1); ClrEol;
  Write(Prompt);
  repeat
    Ch := UpCase(ReadKey)
  until Ch in OKSet;
  Write(Ch)
end; { of proc ReadChar }

```

**setLoc(G : game; Loc : location; M : move)**

*pre:* G is initialized, and 0 or more moves have been made; Loc is in the range 1..9; M has the value X, O, or blank

*post:* Location Loc in the grid is set to the value M

**inARow(G : game; I,J,K: location)**

*pre:* G is initialized, and 0 or more moves have been made; I, J and K are all in the range 1..9

*post:* If the game is not over yet and the grid locations I, J, and K all have the same value, and that value is X or O, then the game is over, and the winner is set to that value; otherwise, there is no effect

**checkForWin(G : game; Loc : location)**

*pre:* G is initialized, and 1 or more moves have been made; Loc is the location of the last move made

*post:* All possible winning moves through Loc are checked

Should these operators be “visible” to the end user? That depends. If you’re writing a program that plays tic-tac-toe against the user, then these kinds of operators may prove useful for quick, low-level access. But they also increase the danger of “corrupting” the correctness of the game data structure, and thus introducing bugs into your program.

In defining the data types themselves, you can now use whatever tools the language provides. For type **move**, you can use an enumerated type (in either C or Pascal) or simply an integer value (C, Pascal, or BASIC). For type **game**, you can use a structure (C), a record (Pascal), or an array (C,



Pascal, or BASIC). You're free to use whatever you feel most comfortable with and whatever works best for your implementation.

In fact, the ideal situation is to have the user completely unaware of the internal structure of the **game** data structure, to remove any temptation for direct modification. There are a number of ways to accomplish this. One is to declare the public version of **game** to be an array of some number of bytes, with no hint as to the internal organization. Your operators can work on that array directly, or map it into a structure or record for internal use.

**Through type-casting and other techniques, your operators can reference the internal structure of the game, while the user sees it only as the referent of an untyped pointer.**

Another approach, which works particularly well in C, is to make **game** an untyped (**void**) pointer. You then need to have two new operators:

**createGame(G : game)**

*pre:* **G** does not currently point to an allocated block of memory

*post:* Memory is allocated for a game data structure, and **G** points to that memory

*continued on page 130*

```

procedure ReadInt(var Val : Integer;
                  Prompt : MsgStr;
                  Low,High : Integer);
{
  purpose      prompt for and get an integer value in a given range
  pre          low <= high
  post        readint() returns some value in the range low..high
}
begin
  {$I-}
  repeat
    GoToXY(1,1); ClrEol;
    Write(Prompt, ' (',Low,',',High,'): ');
    Readln(Val)
  until (IOResult = 0) and (Val >= Low) and (Val <= High)
  {$I+}
end; { of proc ReadInt }

end. { of unit GameIO }

```

#### LISTING 3: MOVES.PAS

```

unit Moves;
{
  MOVES.PAS  unit for making computer's moves for tictactoe
  author    bruce f. webster
  last update 12 dec 87
}
interface

uses TicTac;

var
  CFlag      : Integer; { 0 if computer moves first, 1 otherwise }
  CMove      : Move;    { contains computer's market (X,0) }

procedure GenerateMove(G : Game; var L : Location);

implementation

function WinFound(G : Game; var L : Location) : Boolean;
{
  purpose    checks for winning move in game
  pre       g has been initialized, 0 or more moves have been made,
            the game is not yet over
  post      if the next move can win the game
            then l is set to that move and winfound() returns TRUE
            else l is unchanged and winfound() returns FALSE
}

```



```

}
var
  Temp      : Game;
  I         : Integer;
begin
  I := 1;
  WinFound := FALSE;
  repeat
    if GetLoc(G,I) = BLANK then begin
      Temp := G;
      DoMove(Temp,I);
      if GameOver(Temp) and (Winner(Temp) <> BLANK) then begin
        L := I;
        WinFound := TRUE;
        Exit;
      end;
    end;
    I := I + 1;
  until I > GLim;
end; { of func WinFound }

function BlockFound(G: Game; var L : Location) : Boolean;
{
  purpose   checks for blocking move in game
  pre       g has been initialized, 0 or more moves have been made,
            the game is not yet over
  post      if the next move can prevent the following move from
            winning the game
            then l is set to that move and blockfound() returns TRUE
            else l is unchanged and blockfound() returns FALSE
}
var
  Temp      : Game;
  I         : Integer;
  J         : Location;
begin
  I := 1;
  BlockFound := FALSE;
  repeat
    if GetLoc(G,I) = BLANK then begin
      Temp := G;
      DoMove(Temp,I);
      if not WinFound(Temp,J) then begin
        L := I;

```

## destroyGame(G : game)

*pre:* G points to a game data structure

*post:* The memory G points to is deallocated, and G contains a null pointer value

Through typecasting and other techniques, your operators can reference the internal structure of the game, while the user sees it only as the referent of an untyped pointer.

I'm sure you're dying to see how this all turns out, so I've provided an implementation in Turbo Pascal. The code that we've been talking about all along is in the unit source file TICTAC.PAS, Listing 1. Additional code to display the game board and accept input from the user is provided in GAMEIO.PAS, Listing 2. Routines that allow the computer to play against the user are in MOVES.PAS, Listing 3.

## USING THE IMPLEMENTATION

Now that we have an implementation to work with, let's go back and look at the main body of our program that we created so long ago. We've cleaned it up a bit, so that now it looks like this:

```

theGame : game
Next    : location

```

```

startGame(theGame)
REPEAT
  getMove(theGame,Next)
  doMove(theGame,Next)
  displayGame(theGame)
UNTIL gameOver(theGame)
showResults(theGame)

```

You'll recognize two of these operators—**doMove()** and **gameOver()**—from the list of operators we implemented earlier. But what about the other four? These are operators for our entire game-playing program. Three of them—**startGame()**, **displayGame()**, and **showResults()**—



involve screen output. The fourth, **getMove()**, may involve input (if the user is playing one or both sides), and it may involve calculations (if the computer is playing one or both sides). Regardless of the function of the operators, though, we can still use the same pre/postcondition approach:

**startGame(G : game)**

*pre:* **G** exists (but is not necessarily initialized)

*post:* All game options have been selected, a new game (**G**) has been started, and the game has been displayed on the screen

**getMove(G : game; Next : location)**

*pre:* **G** is initialized, 0 or more moves have been made, and the game is not yet over; **Next** exists

*post:* **Next** contains a value in the range 1..9, and that location in the grid in **G** is blank

**displayGame(G : Game)**

*pre:* **G** is initialized and 0 or more moves have been made

*post:* The game is displayed upon the screen

**showResults(G : Game)**

*pre:* The game **G** is over

*post:* The results (X won, O won, a draw) are somehow indicated on the screen.

If we now implement these four functions according to their pre- and postconditions, we will have a program with no bugs. It will perform exactly as specified. The program file in Turbo Pascal (PLAY.PAS) is given in Listing 4.

Note how the specifications for each of these operators mesh with one another as the program runs. Let's look at our pseudo-code again, inserting the postconditions

*continued on page 132*

```

        BlockFound := TRUE;
        Exit
    end
end;
I := I + 1
until I > GLIM;
end; { of func BlockFound }

procedure GenerateMove(G : Game; var L : Location);
{
    purpose    generates next move for computer
    pre        g has been initialized, 0 or more moves have been made,
              the game is not yet over
    post       *l contains a value in the range 1..9,
              getloc(g,*l) returns BLANK
    strategy   goes first for move to the center (*l == 5)
              then focuses on corners, then moves randomly
              always looks for winning move
              after 3 or more moves, also looks for blocking moves
    analysis   not perfect, but simple and effective; won't always
              win when it could, but always plays to at least a draw
}
var
    NMoves      : Integer;
begin
    L := 5;
    NMoves := MovesMade(G);
    if NMoves <= 2 then begin
        if GetLoc(G,L) = BLANK
            then Exit
        end;
    if WinFound(G,L) then Exit;
    if (NMoves > 2) and BlockFound(G,L) then Exit;
    repeat
        if NMoves <= 4
            then L := 1 + 2 * Random(5)
            else L := 1 + Random(GLim);
        until GetLoc(G,L) = BLANK
    end; { of proc GenerateMoves }

end. { of unit Moves }

```



```

program PlayGame;
{
    purpose          implement tic-tac-toe on the computer
    author           bruce f. webster
    last update      12 Dec 87 -- 1100 mst
}
uses CRT,TicTac,Moves,GameIO;

procedure StartGame(var theGame : Game);
{
    purpose          set up a new game
    pre              none
    post             g,cflag,cmove have been initialized
                   no moves have been made
                   the tictactoe grid has been drawn on the screen
}
var
    Ans              : Char;
begin
    ClrScr;
    ReadChar(Ans,'Who moves first: H)uman or C)omputer? ',['H','C']);
    if Ans = 'C'
        then CFlag := 0
        else CFlag := 1;
    ReadChar(Ans,'Do you wish to be X or O? ',['X','O']);
    if Ans = 'X'
        then CMove := 0
        else CMove := X;
    if CFlag <> 0
        then NewGame(theGame,Opposite(CMove))
        else NewGame(theGame,CMove);
    DrawGrid;
    DisplayGame(theGame)
end; { of proc Initialize }

procedure GetMove(theGame : Game; var L : Location);
{
    purpose          select the next move for the game
    pre              g has been initialized
                   0 or more moves have been made
                   the game is not yet over
    post             l contains a value from 1..9
                   getloc(g,l) is BLANK
}

```

after each operator. The result is shown in Figure 5.

Note that each operator has its preconditions made by the postconditions of the previously called operator(s). It is this meshing that allows us to write programs that can be proven correct.

You may be wondering where the other operators (**getLoc()**, etc.) have gone. They are (most likely) being used within the new operators above. A sample list of possible calls is shown in Table 1.

```

theGame : game
Next : location
    theGame, Next exist
startGame(theGame)
    theGame has been initialized
    0 moves have been made
    the game is displayed on the screen
REPEAT
    the game is not over
    getMove(theGame,Next)
    Next has a value in the range 1..9
    Next points to a blank location in the grid
    doMove(theGame,Next)
    the next move has been made at Next
    the player of next move has been toggled
    (X->O, O->X)
    1 or more moves have now been made
    displayGame(theGame)
    the game is displayed upon the screen
UNTIL gameOver(theGame)
    the game is over
showResults(theGame)
    the results of the game are displayed on
    the screen

```

Figure 5. The meshing of pre- and postconditions.

```

startGame():    newGame(),
                 displayGame()
getMove():      getLoc(),
                 movesMade(),
                 nextMove(),
                 opposite()
displayGame():  getLoc(),
                 movesMade(),
                 nextMove()
showResults():  winner()

```

Table 1. Operators as used by other operators.



## CONCLUSION

Using pre- and postconditions for subroutines, combined with an abstract data approach, provides you with a simple but effective step toward program correctness. It avoids the pitfalls of the "dive-in-and-code" method, a temptation to which many of us succumb. It also encourages good software engineering techniques, including modularity, decomposition, and information hiding. Besides, it's kind of fun.

For those of you brave enough to delve into the more technical aspects of program correctness and abstract data design, I've listed a few below. Be warned: most are quite formal and are usually digestible only in small chunks. The exception is the book by Stubbs and Webre, which focuses more on data structures than on program correctness, and is a great book for any programmer's library. ■

## SUGGESTED READING

Dijkstra, Edsger W. *A Discipline of Programming*. Englewood Cliffs: Prentice-Hall, Inc., 1976.

Gries, David. *The Science of Programming*. New York: Springer-Verlag, 1981.

Jones, Cliff B. *Software Development: A Rigorous Approach*. Englewood Cliffs: Prentice-Hall International, 1980.

Reynolds, John C. *The Craft of Programming*. Englewood Cliffs: Prentice-Hall International, 1981.

Stubbs, Daniel F., and Webre, Neil W. *Data Structures with Abstract Data Types and Pascal*. Monterey: Brooks/Cole Publishing Company, 1985.

---

*Bruce Webster is a computer mercenary living in California. He can be reached via MCI Mail (as Bruce Webster) or on BIX (as bwebster.)*

---

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

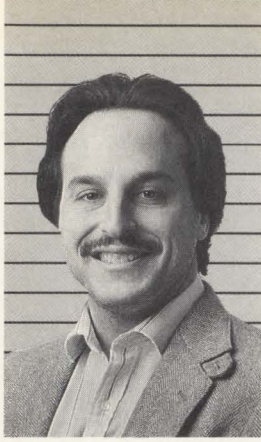
```
}
var
  I          : Integer;
begin
  if (MovesMade(theGame) mod 2) = CFlag
  then GenerateMove(theGame,L)
  else begin
    repeat
      ReadInt(I,'Enter move',1,9);
      if GetLoc(theGame,I) <> BLANK
      then Write('G')
      until GetLoc(theGame,I) = BLANK;
      L := I
    end
  end; { of proc GetMove }

procedure ShowResults(var theGame : Game);
{
  purpose      show results of tictacoe game
  pre          g has been initialized, 5 or more moves have been made
              the game is over
  post         the results of the game are displayed on the screen
}
var
  M          : Move;
begin
  M := Winner(theGame);
  GoToXY(1,1); ClrEol;
  case M of
    BLANK : Write('The game was a draw');
    X      : Write('The winner is X');
    O      : Write('The winner is O')
  end;
  Write(' -- press any key to continue (Q to quit) ');
end; { of proc CleanUp }

var
  theGame    : Game;
  Next       : Location;
  Ch         : Char;

begin { main body of program TicTacToe }
  repeat
    StartGame(theGame);
    repeat
      GetMove(theGame,Next);
      DoMove(theGame,Next);
      DisplayGame(theGame);
    until GameOver(theGame);
    ShowResults(theGame);
    Ch := ReadKey;
  until (Ch in ['Q','q']);
  ClrScr
end. { of program TicTacToe }
```





# LANGUAGE CONNECTIONS

Get the best of both worlds by linking Turbo Prolog to Turbo Pascal.

Peter Immarco

**T**he moment has arrived—the moment when you can call Turbo Pascal 4.0 routines from Turbo Prolog. At last, all those wonderful Turbo Pascal routines that you carefully crafted are available to your Turbo Prolog programs. The possibilities are endless, but at what cost? As you will see, it is easier than you might expect.

In this article, I'll show you how to directly link Turbo Pascal 4.0 routines with a Turbo Prolog program. In addition, we will take a firsthand look at the new utility from Borland that converts unit (.TPU) files to object (.OBJ) files.

## GETTING STARTED

Before we go too far, we should mention two restrictions. First, the Turbo Prolog-to-Turbo Pascal bridge is a one-way connection. You can call a Turbo Pascal procedure from Turbo Prolog, but you cannot call a Turbo Prolog predicate from Turbo Pascal.

Secondly, Turbo Pascal defines its real numbers to be six bytes in size, while Turbo Prolog stores its real numbers in eight bytes (conforming to the IEEE standard). Because of this incompatibility in real number storage, Turbo Pascal code that uses floating point operations must use a math coprocessor. If a coprocessor is not available, you cannot pass and manipulate real numbers between the two languages, and you will have to do all real arithmetic in either Turbo Prolog or Turbo Pascal (but not both). If you do have a math coprocessor, define your Turbo Pascal real numbers as **double** precision reals.

With that in mind, let's see how to link the two languages. The actual connection is a six-step process.

1. Compile all Turbo Pascal code to units.
2. Extract the **System** unit, along with any other standard units that may be used from the Turbo Pascal unit library (TURBO.TPL).
3. Convert all Turbo Pascal units (.TPU files) to object (.OBJ) modules.
4. Create an assembly language bridge to handle the naming conventions.
5. Compile any Turbo Prolog code to object modules.
6. Link all the object modules to create an executable (.EXE) file.

Let's examine each of these steps in detail.

In order to link Turbo Pascal code with Turbo Prolog code, all Turbo Pascal modules must be in object code format. Since Turbo Pascal does not compile object code, Borland provides a utility that converts Turbo Pascal units to object modules (we'll discuss this utility at greater length shortly). If you have Turbo Pascal code that is not in unit form, you must first convert that code to units before using the conversion utility. (We're assuming that you're familiar with units. If you feel a little shaky on units, refer to the *Turbo Pascal Owner's Handbook*.)

Before going on to step 2, we should do a little preparation

work. First, create a list of all the units that you intend to use. Start your list with the names of the units you have just converted (or created). Add the name of the Turbo Pascal **System** unit, which is used by all Turbo Pascal 4.0 units. In addition, if you plan to do any screen I/O, add **Crt** to your list of units. Other standard units you may consider are **Printer**, **Dos**, **Turbo3**, and **Graph3**.

Before you begin, make sure that a .TPU file exists for each of the units on your list. Notice that the **System** unit does not have a corresponding .TPU file (SYSTEM.TPU). This is because SYSTEM.TPU resides in the unit library file, TURBO.TPL. Therefore, you must extract the **System** unit from TURBO.TPL using the unit mover utility, TPUMOVER.EXE, on the Turbo Pascal 4.0 distribution disk.

The unit mover is painless to use. To extract the **System** unit from the unit library, simply enter TPUMOVER TURBO.TPL /\* SYSTEM at the DOS prompt. This command assumes that TPUMOVER.EXE and TURBO.TPL are in the current directory. The unit mover assumes that input files will have a .TPU file extension, so the .TPL extension is included in the command line. The /\* parameter tells TPUMOVER to extract the unit that is specified next in the command line (in our case, the **System** unit). The unit mover then extracts the **System** unit from TURBO.TPL, creating SYSTEM.TPU. (For more information on the unit mover, consult the *Turbo Pascal Owner's Handbook*.)

Once you've collected the .TPU



files, the next step is to convert them to object modules using the unit conversion utility, TPU2OBJ.EXE. Like the unit mover, using TPU2OBJ is straightforward. As an example, let's convert the **System** unit to an object module. With both SYSTEM.TPU and TPU2OBJ.EXE in the current directory, enter

```
TPU2OBJ SYSTEM
```

at the DOS prompt. TPU2OBJ assumes that the input file has a .TPU file extension and automatically names the converted file with a .OBJ file extension. In this case, the converted file is named SYSTEM.OBJ. If you have multiple files, you can convert them one at a time or you can convert them all at once. For example, to convert three units named **Unit1**, **Unit2**, and **Unit3**, you would type:

```
TPU2OBJ UNIT1,UNIT2,UNIT3
```

The converted files would be named UNIT1.OBJ, UNIT2.OBJ, and UNIT3.OBJ. TPU2OBJ reports the names of any units it cannot find.

### AN EXAMPLE

To make things concrete, let's link a Turbo Prolog program, shown in Listing 1, to a Turbo Pascal unit that contains a single procedure called **Square\_0**, shown in Listing 2. **Square\_0** has two integer parameters, **InNumber** and **OutNumber**. **InNumber** is passed by value, while **OutNumber** is passed by reference. **Square\_0** takes **InNumber** and multiplies it by itself, returning the result in **OutNumber**. We will call **Square\_0** from our Turbo Prolog program.

Before you can call a Turbo Pascal procedure, you must create a Turbo Prolog global predicate for that procedure. For instance, the global predicates section in Listing 1 contains a declaration for **square** that calls the Turbo Pascal procedure **Square\_0**. You'll also notice the global predicates **systeminit**, **systemexit**, **saveBP**, and **restoreBP**.

We've named the Turbo Pascal procedure **Square\_0**, instead of **Square**, for a reason. During compilation, Turbo Prolog generates a different predicate name for each

of the global predicate's possible flow patterns (see the *Turbo Prolog Owner's Handbook* for an explanation of flow patterns). Each subsequently generated predicate name consists of the original predicate's name, followed by an underscore character and a number. The number for the first predicate name generated during compilation would be 0. For each name created thereafter, the number would increment by one. If, for instance, the predicate **square** had three flow patterns, Turbo Prolog would generate three calls to procedures named **square\_0**, **square\_1**, and **square\_2**. Consequently, we must add the suffix to our Turbo Pascal procedures.

A global predicate begins with a predicate name, and is followed by its arguments in parentheses. The number of arguments the predicate has must *exactly* match the number of variables declared in the Turbo Pascal procedure. Also, the domain types of the parameters must exactly match the variable types declared in the Turbo Pascal procedure header. (Compare the **square** predicate declaration in Listing 1 to the **Square\_0** procedure declaration in Listing 2.)

After the predicate name comes the "-" separator, followed by the predicate's flow pattern. Since we pass the first argument and return the second argument, **square** has an (i,o) flow pattern. Turbo Prolog passes input parameters by value. An input parameter's value at the time of the call is pushed onto the stack before the call is made. Turbo Prolog expects that return arguments (corresponding to an output flow pattern) will be passed by reference. A double word pointer to the output parameter is pushed onto the stack before the call is made. The predicate **square** in Listing 1 has an (i,o) flow pattern, so the procedure **Square\_0** in Listing 2 passes **InNumber** by value, and **OutNumber** by reference as indicated by the **Var** identifier. There must always be a correlation between the input and output parameters of a Turbo Prolog global predicate and the value and reference variables of a Turbo Pascal procedure.

Following the flow pattern declaration (if one exists) comes the keyword **language**, which is fol-

lowed by the identifier **pascal**. The **pascal** identifier tells Turbo Prolog that the global predicate is a Pascal module.

### INITIALIZING UNITS

**systeminit**, the unit initialization procedure for the **System** unit, is also located in the global predicates section of Listing 1. When a Turbo Pascal program executes, it immediately calls each of the unit initialization procedures. If we plan to do any screen I/O from Turbo Pascal, we must include the unit initialization for the **Crt** unit as well. Turbo Prolog does not automatically make these calls for you, so you must call the unit initialization procedures from your Turbo Prolog program.

Similarly, as your Turbo Pascal program terminates, it calls all the unit exit procedures. Again, Turbo Prolog does not make these calls automatically. Table 1 shows a list of the initialization and exit calls for the **System** and **Crt** units.

Notice two additional global predicates, **saveBP** and **restoreBP**, in Listing 1. You must save the base pointer register before calling the Turbo Pascal unit initialization procedures, because some of the initialization procedures disturb the base pointer register. Turbo Prolog uses the base pointer register to reference variables created in Turbo Prolog predicates. Notice that **initialize** in Listing 1 calls **saveBP** *before* any of the unit initialization procedures are called. The **saveBP** predicate calls the **SaveBP\_0** procedure in the Turbo Pascal unit PROFACE.PAS, shown in Listing 3. The procedure **SaveBP\_0** saves the base pointer to the global variable **BPStorage**. After all the calls to the unit initialization procedures are completed, the complementary procedure **RestoreBP\_0** is called. **RestoreBP\_0** restores the base pointer from **BPStorage**, returning it to the value it had before the unit initialization procedures were called.

### THE BODY EXPLAINED

The main body (or **goal**) of Listing 1 consists of the predicate **run**.

*continued on page 136*



**LISTING 1: PRO2PAS.PRO**

```

/* PRO2PAS.PRO - a simple example of interfacing Turbo Prolog 1.1 to
Turbo Pascal 4.0 */

project "PRO2PAS.PRJ"

global predicates
systeminit - language asm          /* Calls the ASM module */
                                   /* SYSTEMINIT_0 which in turn */
                                   /* calls the Turbo Pascal SYSTEM */
                                   /* unit initialization code */

square(integer,integer) -         /* The Turbo Pascal proce- */
(i,o) language pascal            /* dure takes a number and */
                                   /* returns its square. There- */
                                   /* fore it has an (i,o) */
                                   /* flow pattern */

systemexit - language asm        /* This predicate calls the */
                                   /* exit code for the SYSTEM */
                                   /* unit */

saveBP - language pascal
restoreBP - language pascal

predicates
initialize
run
goodbye
goal
run.

clauses
run:-
    initialize, /* Initialize all units that have initialization */
               /* code and maintain BP */
    clearwindow,
    square(4,Result),
    write("The square of 4 is ",Result),
    nl,
    write("Press any key to continue."),
    readchar(_),
    goodbye. /* Call the exit code of any units that have them */

/* Call all the unit initializaton code */
initialize:-
    saveBP, /* Save the Base Pointer register from harm */
    systeminit, /* Make the calls to the unit initialization */
    RestoreBP. /* Restore the Base Pointer register */

/* Call any necessary unit exit code. */
goodbye:-
    systemexit. /* systemexit will terminate program */

/* END PRO2PAS */

```

**CONNECTION**

*continued from page 135*

**run** calls **initialize** immediately to make sure the units have been properly initialized. The calls to the unit initialization procedures within the predicate **initialize** should be made in the same order that the units would appear in a Turbo Pascal **uses** statement. The **System** unit initialization call is always first.

The program then clears the screen and makes the call to **square** with the value **4**. Turbo Pascal multiplies the value **4** by itself, and the result is bound to the variable **Result**. After writing the value of **Result** to the screen, the program pauses until you press a key. **run** then calls **goodbye**, making sure all the unit exit procedures are called, and terminates the program via **systemexit**.

**BRIDGING THE GAP**

We need to call the Turbo Pascal **System** unit initialization and exit procedures in Table 1. In order to do so, we need to create a global predicate for each of these procedures. However, as mentioned earlier, Turbo Prolog generates an **\_0** suffix at compile time, so we cannot directly call these procedures from Turbo Prolog. The solution is to write a small assembler program that passes on the calls to the (standard) unit initialization and exit procedures. Since these procedures do not require any parameters, the assembler program is straightforward.

Listing 4 shows the interface

Call	Function
SYSTEM_001	The initialization code to Turbo Pascal. Must be called before calling a Turbo Pascal routine from Turbo Prolog.
SYSTEM_003	The exit code for Turbo Pascal. Must be called at the end of a Turbo Prolog program.
CRT_000	The Initialization code from unit CRT.

*Table 1. Turbo Pascal unit initialization and exit calls for the **System** and **Crt** units.*



bridge (BRIDGE.ASM), which is written in assembly language. At the top of the listing is an external procedure statement (EXTRN), which lists all the Turbo Pascal 4.0 unit initialization and exit code procedure names used in our sample Turbo Prolog/Pascal program. Each *bridge* procedure consists of a procedure label with a *\_0* suffix, a call to the correct procedure, and a far return.

### THE CONNECTION

Now we are ready to link the object modules from our Turbo Prolog, Turbo Pascal, and assembly language routines. You should have already compiled PASUNIT.PAS (Listing 2) and PROFACE.PAS (Listing 3) to units. In addition, you should have extracted the **System** unit from the Turbo Pascal unit library (TURBO.TPL). The next step is to convert all the units to object modules, using TPU2OBJ:

```
TPU2OBJ PASUNIT,PROFACE,SYSTEM
```

Now that the units have been converted, the next step is to assemble BRIDGE.ASM to an object module. If you are using Microsoft's assembler MASM, enter

```
MASM BRIDGE
```

at the DOS prompt.

Our final step is to compile the Turbo Prolog module, linking all of the object modules together to create the executable file. You can create the executable file in two ways. You may use Turbo Prolog's auto-link facility, or you can create the executable file manually with the LINK.EXE utility provided with DOS. We'll look at the auto-link method first.

Going back to Listing 1, you'll see that the first statement in the program is a project statement:

```
project "pro2pas.prj"
```

PRO2PAS.PRJ (in Listing 5) contains a list of all the object modules that Turbo Prolog must link in order to create PRO2PAS.EXE. These modules include the Turbo Prolog program PRO2PAS, the assembly module BRIDGE, and the Turbo Pascal modules PROFACE, PASUNIT, and SYSTEM. To compile the project, select the **Project** option from

#### LISTING 2: PASUNIT.PAS

```
{ PASUNIT.PAS - which will result in the unit file, PASUNIT.TPU }
unit PASUNIT;

interface
  { Two integer parameters only }
  procedure Square_0(InNumber: Integer; Var OutNumber: Integer);

implementation

  { Square takes a number (InNumber) and returns its square in
    OutNumber }
  procedure Square_0;
  Begin
    OutNumber:=InNumber*InNumber;
  end;

end.
```

#### LISTING 3: PROFACE.PAS

```
{ Base Pointer protection procedures }
unit PROFACE;

interface

  procedure SaveBP_0;
  procedure RestoreBP_0;

implementation

  Var
    BPStorage: Word;

  Procedure SaveBP_0;
  Begin
    { Save the base pointer BP in a Turbo Pascal
      global variable - BPStorage. BP should
      be on the stack, pushed there by Turbo
      Pascal. Push it back onto the stack to
      insure a proper return for Turbo Pascal }
    InLine($58/      { pop Ax          }
           $50/      { push Ax         }
           $BF/BPStorage/ { mov Di,Ofs BPStorage }
           $89/$05   { mov [Di],Ax (DS) } );
  end;

  Procedure RestoreBP_0;
  Begin
    { Reset the Turbo Prolog base pointer (BP)
      what it was before initialization calls }

    { Remove the BP value pushed on the stack by
      by Turbo Pascal. Move the old (BPStorage) BP
      value that was pushed on the stack by
      Turbo Prolog into Ax, and push it onto the
      stack. when RestoreBP returns turbo will be
      fooled into resetting the BP to the value
      we pushed on the stack, not the value it
      placed there. }
    InLine($58/      { pop Ax          }
           $BF/BPStorage/ { mov Di,Ofs BPStorage }
           $8B/$05/      { mov Ax,[Di]    }
           $50          { push Ax         } );
  end;

end.
```

*continued on page 138*



**LISTING 4: BRIDGE.ASM**

```

; MASM assembler example of a bridge to the
; Turbo Pascal 4.0 unit initialization and
; exit procedures.

EXTRN SYSTEM_001:far,CRT_000:far,SYSTEM_003:far
PUBLIC SYSTEMINIT_0,CRTINIT_0,SYSTEMEXIT_0

BRIDGE segment
    assume CS:BRIDGE

; Pass on call to SystemInit_0 to SYSTEM_001.
SystemInit_0 proc Far
    call SYSTEM_001
    ret
SystemInit_0 endp

; Pass on call to CrtInit_0 to CRT_000.
CrtInit_0 proc Far
    call CRT_000
    ret
CrtInit_0 endp

; Pass on call to SystemExit_0 to SYSTEM_003.
SystemExit_0 proc Far
    call SYSTEM_003
    ret
SystemExit_0 endp

BRIDGE ends
end

```

**LISTING 5: PRO2PAS.PRJ**

```

pro2pas+system+pasunit+proface+bridge

```

**LISTING 6: BRIDGE2.PAS**

```

( Pass the Turbo Prolog 'underscore' procedure names
  onto the correct Turbo Pascal 4.0 procedure name )

unit BRIDGE2;

( Link in the external (EXTRN) declarations for
  the unit initialization and exit codes )

interface
    uses CRT;

    ( One for each Turbo Pascal procedure called)
    procedure ClrScr_0;

implementation

    ( Just pass the call on to the correct Turbo Pascal
      procedure )
    procedure ClrScr_0;
    begin
        CLRSCR;
    end;
end.

```

**CONNECTION**

*continued from page 137*

the Options pulldown menu, load PRO2PAS.PRJ, and select Compile. Turbo Prolog then compiles PRO2PAS.PRO to an object file and links it with the other object files specified in PRO2PAS.PRJ, creating PRO2PAS.EXE. Voila! Any errors that occur during the compilation and link process appear in the Message window.

**THE ROAD NOT TRAVELED**

The DOS LINK utility provides the second method of linking the object files to create the executable file. Keep in mind that Turbo Prolog requires LINK version 2.20 or greater in order to link correctly. First, compile PRO2PAS.PRO to an object file using the OBJ file compiler option. Next, Quit out of Turbo Prolog to perform the link. LINK links all of the object files, the Turbo Prolog initialization module, the symbol table created by compiling the Turbo Prolog module, and the Turbo Prolog Runtime Library, to create the executable image. My link command looked like this:

```

link INIT+PRO2PAS+SYSTEM+
    PASUNIT+PROFACE+BRIDGE+
    PRO2PAS.SYM,PRO2PAS,,PROLOG

```

**CALLING TURBO PASCAL FUNCTIONS**

There may be times when we wish to call Turbo Pascal library functions and procedures from Turbo Prolog. For instance, we can clear a window in Turbo Prolog, but clearing the entire screen may be a problem if several windows exist. One solution is to call the Turbo Pascal library procedure **ClrScr** to clear the screen.

Once again, we run into the problem of the **\_0** suffix generated by Turbo Prolog. In this case, we don't have to use assembly language to handle the naming conventions. We can create a **\_0** suffixed procedure in Turbo Pascal to pass the Turbo Prolog call on to the correctly named Turbo Pascal library procedure.

BRIDGE2.PAS in Listing 6 does exactly that for us. It defines a



procedure, **ClrScr\_0**, which makes a call to the library procedure **ClrScr**, and then returns to the calling program. To add **BRIDGE2.PAS** to our program, compile **BRIDGE2.PAS** to a unit, extract the **CRT** unit from **TURBO.TPL** and convert the units to object modules as described earlier. Then add **CRT** and **BRIDGE2** to the list of modules in **PRO2PAS.PRJ** (or the **LINK** command line). Finally, add a call to the **CRT** initialization procedure in the Turbo Prolog code. Listing 7 reflects these changes to our Turbo Prolog program. You can use this same method to call other Turbo Pascal library procedures or functions.

### WHAT TO WATCH OUT FOR

There are a couple of things to watch for. If you change a unit file, remember to convert the unit to an object file using **TPU2OBJ**. I forgot a couple of times and couldn't figure out why the changes weren't showing up in my program.

If you've created a unit containing initialization code, and you intend to interface that unit to Turbo Prolog, you have to convert that code into a procedure, putting the procedure header in the **interface** section of that unit. Otherwise, there is no way to call the initialization code, because no procedure name exists that can be referenced from a Turbo Prolog global predicate. If you want to maintain compatibility for that unit with other Turbo Pascal programs, change the initialization code in your newly created initialization procedure to a single cell. Then all should be well.

Also, as we mentioned earlier, Turbo Pascal code that uses floating point operations must have a math coprocessor. Otherwise, you will have to do your floating point arithmetic in Turbo Prolog. ■

---

*Peter Immarco is the national sales manager for Thought Dynamics, makers of the memory-resident utility Fetch. He can be reached at 1142 Manhattan Ave., Suite CP-310, Manhattan Beach, CA 90266.*

---

*Files may be downloaded from CompuServe as PROPAS.ARC.*

### LISTING 7: PRO2PAS2.PRO

```

/* PRO2PAS.PRO - a simple example of interfacing Turbo Prolog 1.1 to
Turbo Pascal 4.0 */

project "PRO2PAS.PRJ"

global predicates

  systeminit - language asm      /* Calls the ASM module that */
                                /* in turn calls the SYSTEM unit */
                                /* initialization code */

  crtinit - language asm        /* Initializes the CRT unit */

  clrscr - language pascal      /* Clear the screen, but using */
                                /* Turbo Pascal */

  square(integer,integer) -     /* The Turbo Pascal proce- */
    (i,o) language pascal      /* dure takes a number and */
                                /* returns its square. There- */
                                /* fore it has an (i,o) */
                                /* flow pattern */

  systemexit - language asm     /* This predicate calls the */
                                /* exit code for the SYSTEM */
                                /* unit */

  saveBP - language pascal
  restoreBP - language pascal

predicates
  initialize
  run
  goodbye
goal
  run.

clauses
  run:-
    initialize, /* Initialize all units that have initialization */
               /* code and maintain BP */

    clearwindow,
    square(4,Result),
    write("The square of 4 is ",Result),
    nl,
    write("Press any key to continue."),
    readchar(_),
    goodbye. /* Call the exit code of any units that have them */

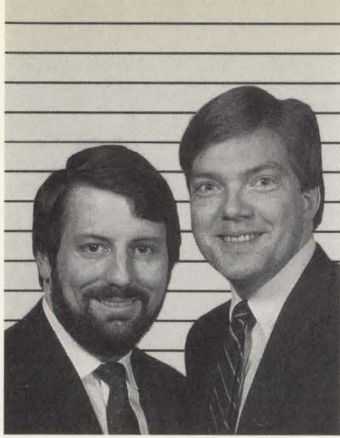
/* Call all the unit initialization code */
initialize:-
  saveBP, /* Save the Base Pointer register from harm */
  systeminit, /* Make the calls to the unit initialization */
  crtinit, /* procedures */
  RestoreBP. /* Restore the Base Pointer register */

/* Call any necessary unit exit code. */
goodbye:-
  clrscr,
  systemexit. /* Systemexit will terminate program */

/* END PRO2PAS */

```





# TALES FROM THE RUNTIME

## Memory models

Mark L. Van Name and Bill Catchings

In our last column (*TURBO TECHNIX*, January/February, 1988), we discussed the wildcard expansion routine. Like all of the Turbo C Runtime Library routines, this one works with any Turbo C programs. In particular, it works regardless of the memory model you use. Last issue we used a few coding tricks to make sure that our code would meet this goal. To save space we largely glossed over those techniques. This time around we take a deeper look at the various PC memory models and how they affect your work with Turbo C and its Runtime Library.

Many programmers treat memory models as nothing more than hindrances or annoyances that are best avoided. However, choosing the right memory model is crucial, due to the trade-offs in program speed, size, and capability. As we look further into the Runtime Library source code, you will need a solid understanding of memory models for two reasons. First, the Runtime routines support all of Turbo C's memory models. This support shows up in odd coding tricks, such as the few we used in our wildcard expansion routine, *expwild*. The second reason involves the extensive use of assembly language in the Runtime. While memory model support usually only affects a few declarations in Turbo C routines, it has far greater implications for assembly language programs.

### THE MEMORY MODEL BLUES

When programming on the PC, you quickly learn that Turbo C is not the source of the memory model blues. Instead, the cause of the trouble is the hardware architecture of the Intel 8086 family of processors. Although many PCs now use newer Intel CPUs, the newer systems still follow roughly the same rules that we will discuss here. The Intel 80286 and 80386 CPUs can operate in several modes, but when they are executing in real mode, they function in essentially the same fashion as the older 8086. While some of the details vary, an 80286 or 80386 running under OS/2 also has basically the same characteristics that we describe here.

The 8086 family employs a *segmented* architecture in which memory is divided into chunks, or *segments*. Each 8086 segment contains 64K. (The 80386 can accommodate segments of different sizes.) A program executing on the 8086 can have up to four different segments in use at any time: one for code, one for the stack, and two for data. Each segment is identified by one of the 8086's segment registers.

To locate a byte in memory you need an address with two components: the segment in which the byte resides, and its offset, or position, in that segment. Each component is typically a 16-bit number. If a byte is within any of the segments that you are currently using, all you need to define is

its address, its segment register, and the 16-bit offset of the byte. Any address that can be defined with only its offset and a current segment register value is called *near*. An address that is outside of the current segments, and therefore requires both an offset and a 16-bit segment identification, is called a *far* address.

Obviously, if some parts of a program think a byte's address is far, while others treat it as near, things will break down. A *memory model* is simply a set of rules that a program's modules follow in order to handle addresses consistently.

The memory model you use depends primarily on your program's size and data addressing requirements. If your program is small, you may be able to keep all of your code and data near. Larger programs often need more than 64K of code or data, and so must use far code or far data. Because they require only 16 bits, near pointers are smaller and can be processed more quickly than far pointers, which require 32 bits but give you a larger range of addresses (your *address space*).

### THE SIX MODELS

In order to let you pick the type of addresses most appropriate for your programs, Turbo C and its Runtime Library support six memory models. (These six are similar to those offered by most



other C compilers.) The Runtime object library for each memory model is in a different library file. To build a program that uses a particular memory model, follow the conventions of that model and then link the program with the appropriate Runtime Library.

We need to introduce a few terms before we explain the six memory models. *Global* memory is the memory used for a program's globally declared variables, while *static* memory is that used for its static variables. C keeps all of the local variables for its procedures on the *stack*, which is a first-in, first-out data structure maintained by dedicated pointers in the 8086 family hardware. Any storage that you dynamically allocate at runtime, such as that which you get by calling **malloc**, is part of the memory known as the *heap*.

The sizes we use in defining Turbo C's memory models are the maximums. The 8086 can address a maximum of 1MB of total memory, while the PC further restricts that to 640K.

Turbo C's smallest memory model is known as the *Tiny* model. Programs that follow the Tiny model use a single 64K segment for everything—code, global memory, static memory, stack, and heap are all in the same segment. You must use this memory model to build a DOS .COM file.

In the *Small* memory model you have two different 64K segments. All of the code goes into one, while the other holds global data, static data, the stack, and the heap.

The *Medium* model also forces you to keep all of your data in one segment. It allows your code, however, to run over many segments, up to a maximum of 1MB.

The *Compact* model is similar to the Medium, but with the space allocation reversed. It gives you only 64K for code, but you can have 64K for global and static memory, another 64K for the stack, and up to 1MB for the heap.

The *Large* model still limits you to 64K for global and static data, and 64K for the stack, but here you can have up to 1MB for the heap and up to 1MB for your code.

The *Huge* memory model is the biggest of the bunch. It is just like the Large model except that each source code file can have its own 64K segment for its global and static variables.

Again, remember that all of these sizes are limited by the 1MB address space of the 8086. There are also a few other restrictions. Regardless of the memory model, no single routine can be larger than 64K. Because even the Huge model has a maximum of 64K for the global and static data in each source code file, you cannot declare any single data item to be larger than 64K.

You can get around this last restriction by dynamically allocating heap for an object. Items from the heap can be larger than 64K as

long as you explicitly declare the pointer to them to be **Huge**. (Note that this is a declaration for a single pointer, which is not the same as having the entire program follow the Huge memory model.) When you use a Huge data object, prepare to pay a speed penalty. Turbo C uses *normalized* pointers to work with Huge data items. This is because arithmetic on a 32-bit segment and offset address does not correctly carry from the offset to the segment. Turbo C keeps pointers correct by working on them only in subroutines, which are obviously much slower than the machine instructions that process most pointers.

## MIX AND MATCH

Turbo C provides one other powerful construct: you can build programs that use *mixed* memory models, in which different routines follow different models. This lets you follow one of the smaller, faster memory models for most of your routines, while allowing individual data items (or even procedures) to be far or Huge. If most

*continued on page 142*

# THE PROGRAMMER'S SHOP

helps compare evaluate, find products. Straight answers for serious programmers.

## FREE Catalog & Advice

Over 40 products for Turbo X programmers PLUS over 700 more for programmers in other languages. Technical specialists help you choose the right product for you. Call today.

## Turbo BASIC Support

BASIC Development Tools PC \$ 89  
Turbo Finally! PC \$ 85

## Turbo C Support

Blackstar C Functions PC \$ 99  
C Utility Library PC \$ 119  
C Worthy Interface Library PC \$ 169  
with Forms PC \$ 249  
Curses - by Aspen PC \$ 109  
dB\_VISTA - single user MS CALL  
dBx - dBASE III to C MS \$ 299  
Essential Graphics PC \$ 185  
Greenleaf C Sampler PC \$ 69  
Greenleaf Comm Library PC \$ 129  
Greenleaf DataWindows PC \$ 155  
Greenleaf Function Library PC \$ 139  
Panel PLUS MS \$ 395  
PC-lint - v. 2.10 MS \$ 99  
Turbo C Tools - by Blaise PC \$ 95  
TurboHALO PC \$ 79  
TurboWINDOW/C PC \$ 75

## Feature

Turbo-to-C Tools - Translates Pascal to modular K&R. MS, Turbo library support, nested procedures, all data types (incl. structured constants), operators, control structures. 99% rate. PC \$459

## Recent Discovery

WKS Library - 30 C functions access spreadsheet data. Enter/change cell values, formulas, macros, range names, column widths, global defaults. Lotus 1-2-3 or compatible. MS, Turbo C, Lattice. Source included. No royalties. PC \$85

Vitamin C PC \$ 159  
Windows for Data PC CALL

## Turbo Pascal Support

Halo PC \$ 209  
Mach 2 - MicroHelp MS \$ 55  
Math Pak 87 PC \$ 79  
Report Builder PC \$ 159  
Screen Sculptor PC \$ 89  
System Builder PC \$ 129  
TP2C PC \$ 199  
Turbo Asynch Plus - Blaise PC \$ 99  
Turbo Extender PC \$ 65  
TurboHALO PC \$ 79  
Turbo Optimizer - object PC \$ 59  
with source PC \$ 109  
Turbo Power Tools Plus PC \$ 99  
Turbo Power Utilities PC \$ 79  
Turbo Professional PC \$ 79  
Turbo-Ref PC \$ 35  
TURBOsmith Debugger PC \$ 79  
TurboWINDOW PC \$ 79

Call for a catalog and solid value

**800-421-8006**

## THE PROGRAMMER'S SHOP

Your complete source for software, services and answers

5-x Pond Park Road, Hingham, MA 02043  
Mass. 800-442-8070 or 617-740-2510

Note: All prices subject to change without notice. Mention this ad. Some prices are specials. Ask about COD & P.Os. 200 formats plus 3" laptop. UPS surface shipping add \$3 normal item.

TX388



continued from page 141

of your program uses a larger model, you can also go the other way and declare individual routines and pointers to be near.

When assembling a routine, a symbol definition (done with the MASM /D parameter) specifying which Turbo C memory model you want can be one of the arguments to MASM. You pass these definitions in the form /D\_<memory model>\_, giving parameters like /D\_LARGE\_ or /D\_HUGE\_. For example, the command

```
MASM setargv /D_LARGE_ /MX
```

assembles the file SETARGV.ASM with the Large memory model. (The /MX parameter preserves case sensitivity for C's sake.) In our first column we provided two batch files that used the /D parameter as they built and updated the Runtime libraries. If you omit the /D parameter, the Turbo C assembly language include file, RULES.ASI, assumes the Compact memory model.

Regardless of the memory model you use when writing procedures to add to the Runtime, or changing those already in it, you want a single source code module that works with all six memory models. All of the source code supplied by Borland works with all six models.

That source code contains resources that you can use in writing routines that are independent of the memory model used by the caller. These resources are useful both when you are working with the Runtime and for other C and assembly language programming tasks.

RULES.ASI uses the symbol that you set with the MASM /D parameter to execute EQUATE statements when defining two important symbols: LPROG and LDATA. LPROG tells whether the code uses far calls and returns, while LDATA shows whether the

```
IFDEF  _HUGE_
    mov  ax, seg _stklen@ ; if we are using the huge model,
    mov  es, ax          ; we need the segment that holds
    mov  ax, es:_stklen@ ; the stack data
ELSE
    mov  ax, _stklen@    ; here we need only the stack's
                        ; length, as all data in one seg
ENDIF
```

Figure 1. Assembly code to handle global variables using the Huge memory model.

```
; The second arg is the destination string. We push first its
; segment if we are in a large data memory model. In any case,
; we then push its offset.
IF LDATA
    mov  ax, word ptr NewCmdLn+2 ; Segment for large data
    push ax
ENDIF
    mov  ax, word ptr NewCmdLn ; Destination offset
    push ax
```

Figure 2. Assembly code to handle near and far addresses.

data is far. Each is set to **True** or **False** as appropriate for the memory model you use.

Many routines then test these symbols in conditional statements so that they can take the action appropriate to the memory model. We used these symbols several times in the code in our last column; we have extracted four of those cases as examples here.

In Figure 1, we check to see if the program is being assembled as a Huge memory model routine. If **\_HUGE\_** is defined, then we know that, unlike in all the other memory models, global variables are not necessarily all in the same segment. Therefore, we must get the segment address as well as the offset of the global variable **\_stklen** before we can get its value. If **\_HUGE\_** is not defined, then there is only one global data segment. In this case, since the data segment, DS, points to the only global data segment, we now can load the contents of **\_stklen** directly. If we had not handled this special case, our routine might not work in the Huge memory library.

In Figure 2 we test the **LDATA** symbol. We use it rather than the memory model symbol so that one section of code will work with all

of the memory models that have far data. We test **LDATA** here to be sure that we are handling a subroutine argument correctly. Be sure to set up subroutine arguments correctly. If the routines you call use the wrong stack offsets to locate their arguments, you will get the wrong data. In Figure 2 we must make sure that, if a given data item is far, we *first* push the segment address for that item and then the offset address. Both addresses are necessary for far data, and the segment address must be first.

Subroutine arguments affect not only the routines you call, but also the work you do when cleaning up the stack after a routine returns to your code. When a routine returns to your code, you must adjust the stack pointer to remove the bytes that you pushed. If you forget to clean up the stack, your own code will attempt to return to what it thinks is the caller's address on top of the stack. This will not be an address at all, but leftover data, and your



program will die most ignobly. In Figure 3 we adjust the stack differently, depending upon the size of one of the pointer arguments. If the data for this program is far, we have to pop the extra two bytes of segment identification from the stack.

**LPROG** and **LDATA** are not the only memory model tools in **RULES.ASI**. That file also contains a few macros that make it easier for you to push pointer arguments. For some unknown reason, none of Borland's Runtime code uses these macros.

### MEMORABLE MACROS

One useful pair of macros is **pushDS\_** and **popDS\_**. If you are using any memory model with far data, these macros take the expected action: they push or pop the DS. If you are using a memory model with only near data, where the value of DS never changes, these macros do nothing.

Another macro, **PushPtr**, makes it simple for you to push pointers independently of memory model. **PushPtr**'s arguments are the memory locations containing a pointer's offset and its segment. In a routine with far data, **PushPtr** pushes both the segment and the offset, while in a routine with near data, it pushes only the offset.

When writing code that works with all memory models, you must consider the type of call, as well as the type of the arguments to that call. It's crucial if a procedure is called with a near call or a far call. The normal procedure in MASM is to declare every label as near or far, after which calls to that label are handled appropriately.

**RULES.ASI** provides some routine declaration macros that let you handle this problem independently of the memory model you are using. The **Proc@** macro opens a routine within a module, while **PubProc@** lets you open a public function. You can use **ExtProc@** to declare an external function.

All of these macros take as their arguments the name of the routine and an argument that says whether the routine follows C (**\_CDECL\_**) or Pascal (**\_PASCAL\_**) calling conventions. These macros then set up that routine so that it works correctly with the memory model being used. These macros also add an underscore to the front of the routine's name if it is a C routine. Once you use these macros to declare a routine, you can call that routine with the name `<symbol name>@`; you no longer have to worry whether it follows C or Pascal conventions, or whether it is near or far. Figure 4 shows the declaration of the Turbo C Runtime Library routine **sbrk** as an external routine, followed by a call to **sbrk**.

### CALL TO THE wild

We used one other trick in the last column that is worth revisiting now that we have discussed memory models. We needed to pass the DOS command line to **expwild** to expand any wildcards in it. Unfortunately, the command line is at a fixed place that is outside the single global data segment of the memory models with near data. For memory models with far data this is no problem, because they address all data with both a segment and an offset. But

we had to make sure that our C routine **expwild** could work with every memory model.

Turbo C's support of explicit pointer types came to our rescue. We declared the source string parameter to **expwild** to be a far pointer. In **expwild** we used that pointer to read the command line and copy all relevant portions of that line into a local variable. Then we passed that local variable to the routine **wild** for expansion. This let Turbo C handle the call to **wild** in the manner appropriate to the memory model being used, rather than also forcing **wild** (and any other routines that **wild** might in turn call) to declare its arguments as far.

No one likes to worry whether their code will work with all of the memory models, but because of the architecture of the 8086 family of processors, you have no choice. Fortunately, Turbo C's Runtime Library source code gives us some support primitives that help make the job a little easier.

Next issue we will delve into another part of the Runtime. As always, if there is a particular area that you would like us to explore, please write and let us know. ■

---

*Mark L. Van Name is a freelance writer. Bill Catchings is a freelance writer and a software engineer at Data General Corp.*

```

; Clean up the stack after the call. We adjust by 10 if in a large
; data memory model because of the extra segment id we pushed.
IF LDATA
    add    sp, 10        ; Clean up the stack
ELSE
    add    sp, 8        ; Clean up the stack
ENDIF

```

Figure 3. Assembly code to adjust the stack depending on pointer size.

```

ExtProc@    sbrk, _CDECL_
            call    sbrk@    ; Get some memory

```

Figure 4. Declaring and calling the Turbo C Runtime Library routine **sbrk** as an external procedure.



# ARCHIMEDES' NOTEBOOK

## Designing a two-band vertical antenna.

Augie Hansen, KBOYH

**E**ureka: The Solver is an ideal tool for radio engineers and experimenters. In particular, I have found it useful in designing tuned circuits and antennas. Although there are many fairly inexpensive antennas available over the counter, it is both fun and instructive to build your own antenna from scratch.

In this article, we will design a two-band vertical antenna, and in the process, examine radio frequency resonant circuits. A parallel-tuned circuit is at the heart of this easy-to-build antenna. By editing the Eureka equation file, you can extend the multibanding technique used here to vertical antennas that cover more than two bands, and to other types of antennas, including horizontal dipoles and the venerable inverted vee.

### HALF AN ANTENNA IS BETTER THAN NONE

If you are an experienced radio operator or short-wave listener, you know that the vertical antenna is a much-maligned antenna, often described as working "equally poorly in all directions." That's essentially true, but it's really a bad rap. If properly installed over a decent ground system, the vertical antenna has a low angle of radiation and is therefore an excellent long-distance (DX) antenna. It is one of the few antennas that is economically and mechanically feasible for low-frequency operation.

The vertical antenna is essentially half of a dipole stood on end. In its many variations, it can be ground-mounted, elevated above the earth's surface, or it can be cut to one of several common wavelengths for a given frequency to obtain the desired radiation angle. And it can be made from a wide variety of alternate materials, including suspended wire and aluminum tubing.

Typical ground-mounted single-band verticals are cut to some fraction of the desired wavelength. The length (in feet) of a 1/4-wave series-fed vertical antenna is equal to 234,000 divided by the frequency in kilohertz. The empirically derived number 234,000 is based on practical experience with antennas of various thicknesses, and works well for verticals constructed of aluminum tubing. The 1/4-wave vertical over a perfect ground has a feed-point impedance of about 36 ohms. Over typical lossy ground, even with a modest ground system, the vertical has a feed-point impedance that approximates the impedance of standard 50-ohm coaxial cable. This eliminates the need for complicated feed-point impedance matching. Other commonly used dimensions include 3/8-wave, 1/2-wave, and 5/8-wave verticals, each having unique vertical radiation patterns and feed-point characteristics.

Making a multiband vertical antenna involves constructing an element that looks like a resonant radiator in each of the target frequency bands (Figure 1). One way of making an antenna resonate in

more than one band is to use a trap that disconnects one section of the antenna from another at one frequency, but acts as a loading coil at another frequency. A *trap* is simply a tuned circuit that is also physically able to maintain antenna integrity under maximum wind loading and other environmental conditions.

### GOOD VIBRATIONS

A tuned circuit consists of capacitors and inductors in various combinations. We will restrict our attention to the simple parallel-connected type. At a given excitation frequency, an inductor exhibits some inductive reactance,  $X_L$ . Similarly, a capacitor exhibits a capacitive reactance,  $X_C$ . By convention, *inductive* reactance is considered "positive" and *capacitive* reactance "negative" because they are 180 degrees apart in phase.

At the resonant frequency, the magnitudes of the inductive and capacitive reactances are equal ( $X_L = X_C$ ). The offsetting phase relationship produces a net reactance of 0 in a series circuit and a reactance that is infinitely large in a parallel circuit. The values of  $X_L$  and  $X_C$  are defined as:

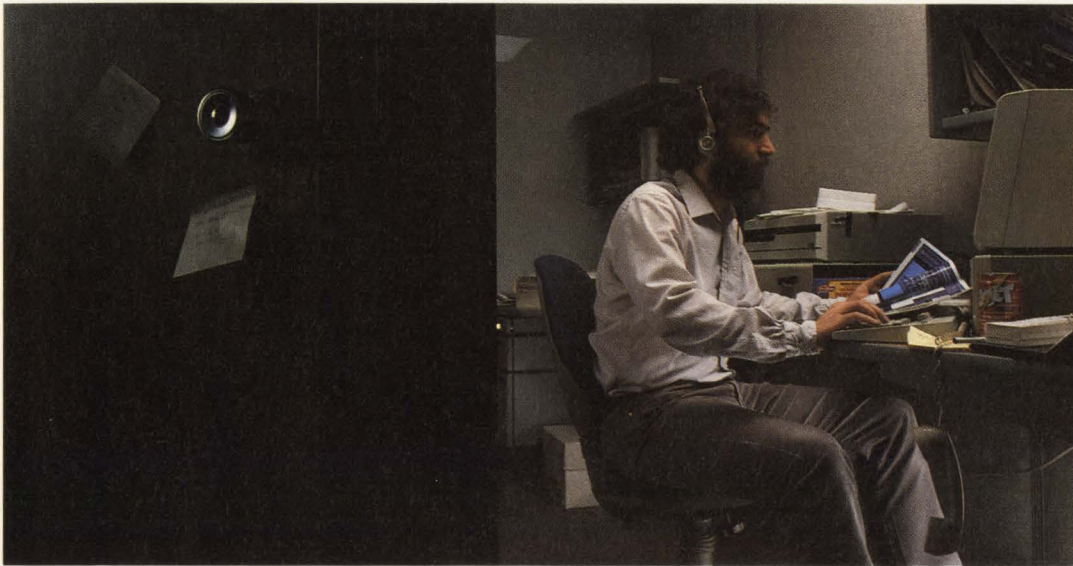
$$X_L = 2\pi fL$$

$$X_C = \frac{1}{2\pi fC}$$

*continued on page 146*



# *There's only one way to reach a programmer . . .*



## *Use the programmer's magazine:*

# **TURBO TECHNIX** THE BORLAND LANGUAGE JOURNAL

Our readers know that *TURBO TECHNIX* is the place to be when the focus is on development. They watch us for the tips and techniques that help them utilize the speed and power of Borland's programming languages. And they spend a lot of time in these pages.

Your ad should be here.

**JULY/AUGUST 1988**

ISSUE CLOSING DATE: APRIL 24

Create custom text device drivers in Turbo Pascal . . . build a meta interpreter in Turbo Prolog . . . add mouse support to your graphics applications . . . check printer status from within PAL programs . . . add a pattern locator to the MicroStar editor from the Turbo Pascal Editor Toolbox . . . save and load EGA screens in Turbo Basic . . . all the *TECHNIX* you've come to expect, and a whole lot more!

**SEPTEMBER/OCTOBER 1988**

ISSUE CLOSING DATE: JUNE 23

Multitask Turbo Pascal applications under DOS . . . learn how to use linked lists in Turbo C . . . write a code-generating script in PAL . . . use Turbo Basic to convert binary files to ASCII files for transmission as text through a modem . . . telecommunications with Turbo Prolog . . . understand how data is stored in and retrieved from 286 extended memory . . . our columnists, our critiques, and lots more!

### **CALL NOW**

### **RESERVE YOUR TURBO TECHNIX SPACE TODAY!**

Office of the Publisher  
(408) 438-9321

*Publisher*  
Marcia Blake

*Advertising Sales Manager*  
John Hemsath

Western Office  
(714) 858-0408  
Janet Zamucen

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

Southern Office  
(813) 394-4963  
Megan Patti



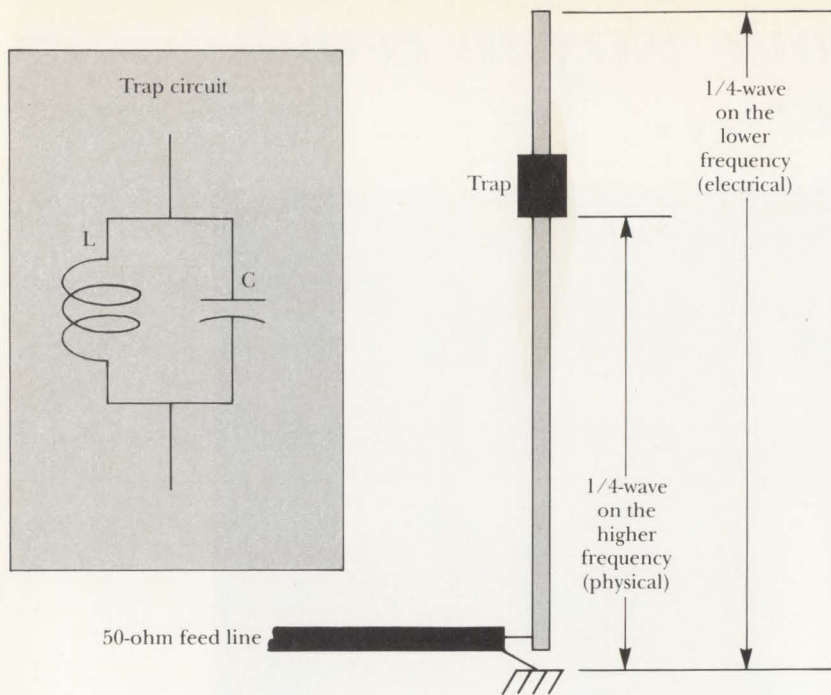


Figure 1. A two-band trap vertical antenna. At the high frequency, the trap effectively disconnects the upper portion of the antenna. At the lower frequency, the trap acts as a loading coil, and brings the physically short driven element into resonance.

## NOTEBOOK

continued from page 144

By equating the values and solving for the lumped constant  $LC$ , we obtain the following equation:

$$LC = \frac{10^{12}}{4\pi^2 f^2}$$

Once we have the value  $LC$  for a given frequency, we can rummage through our old parts bin for a suitable inductor or capacitor as one component of the resonant circuit and then calculate the needed value of the other. The solution recommends values for  $L$  and  $C$  based on a rule of thumb that the reactances should be about 200 ohms. However, you are free to assign a value to  $C$  that produces a capacitive reactance between 100 and 300 ohms. The value of  $L$  is calculated to balance the chosen  $C$ .

If we are inordinately lucky or well stocked, we might find both needed components on hand. It is more likely, however, that we will need to build or buy at least one of them. Fortunately it is easy to build inductors and capacitors out

of readily available materials.

For this design, we assume a capacitor value and determine the needed inductance, and then calculate the number of turns needed for a given form factor (coil diameter and length). The number of turns,  $n$ , is calculated by the formula

$$n = \frac{(L(18d + 40l))^{1/2}}{d}$$

where  $d$  is the diameter of the coil form and  $l$  is its length (both in inches).  $L$  is the inductance in microhenrys. This is a reasonable approximation for values of coil length equal to or greater than 0.4 times the coil diameter.

## EXPRESS YOURSELF

To express the formulas in terms acceptable to Eureka, we need to make a few changes. As shown in TRAP.EKA (Listing 1), the  $\pi$  symbol is replaced by the built-in function  $\text{pi}()$ , which returns the value of pi. I use parentheses to group expressions for clarity.

The \*, /, and ^ symbols indicate multiplication, division, and exponentiation respectively.

To make the equations readable, I have used descriptive names, such as *len* and *dia* instead of the cryptic  $l$  and  $d$  shown in the standard equations above. In addition, I have extensively commented the equation file so that it can stand alone. The comments contain detailed information about each step.

In the NUMBER OF TURNS calculation, a constraint is applied to check coil form values. Using too small a coil length will give inaccurate results.

Eureka's solution is given in Figure 3.

## HOT TIPS

I constructed the two-band vertical from 1 1/4-inch O.D. aluminum tubing. The base is insulated from ground by standoffs attached to a section of 2-by-4 driven mercilessly into the ground. A dozen radials of varying lengths are attached to a 1/4-inch copper ground rod and joined to the feedline shield. The feedline center conductor attaches to the vertical element.

Figure 2 shows one of many possible trap constructions. It uses two short aluminum tubing sections insulated from each other by a Plexiglas or treated wood dowel. I used 1 1/8-inch O.D. tubing which telescopes nicely with the 1 1/4-inch element sections. A 1-inch O.D. insulator provides enough rigidity to keep the assembled antenna upright under operating conditions.

The coil is a traditional air-coil type cut to the required dimension. The capacitor is formed by cutting a piece of coaxial cable to a length that produces the required capacitance. You will need to look up the cable's capacitance per foot in the manufacturer's literature, or measure the capacitance of a short piece with a capacitance meter. You could also build the calculation into the equation file and let Eureka do



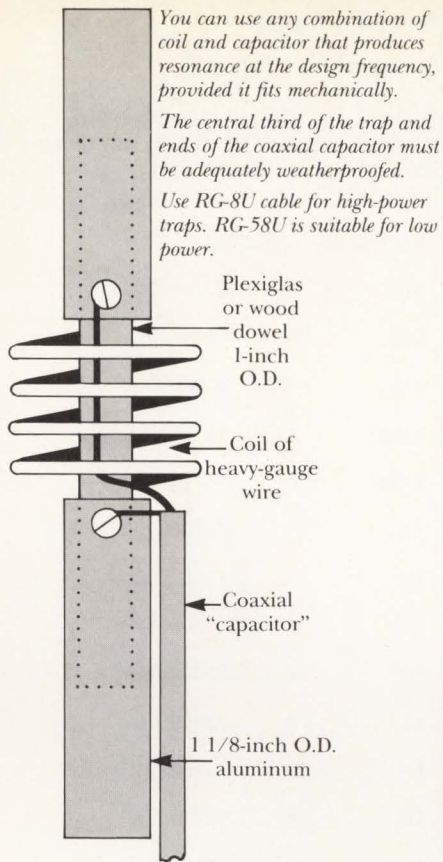


Figure 2. Mechanical details of a typical trap. The coaxial capacitor has the advantage that it can be "trimmed" by cutting between resonance tests—as long as it is not cut shorter than the resonant frequency requires.

the math for you. I chose to use a commercial 50 pF transmitting capacitor instead, but the coaxial capacitor is an inexpensive and adequate alternative. Be sure to weatherproof the cable ends with a sealant or at least good-quality tape.

If you use the coaxial capacitor, you will need to extend the coaxial cable stub along one element away from the trap. It must go in the direction *away* from the higher frequency portion of the element so that it won't detune the antenna during high-frequency operation. You should fine tune the trap so that it resonates at the higher frequency design center, providing a high

*continued on page 148*

#### LISTING 1: TRAP.EKA

```

;-----
; TRAP: Trap vertical calculations
; Augie Hansen, K80YH
;
; Formulas are based on the ARRL 1986
; Handbook for the Radio Amateur.
;-----

```

```

; Determine values of inductance (L) and capacitance (C)
; required to resonate at a specified frequency. The
; calculations are based on the formula
;

```

$$f = 10^6 / (2 * \pi * \sqrt{L * C})$$

```

; where f is in kilohertz, L in microhenries, and C is
; in picofarads, which are units suitable for tuned circuit
; construction with readily available components. Also
; calculate the lengths of the lower and upper portions of
; the radiating element.

```

#### ----- DESIGN DATA -----

```

;
; Adjust these values to match your design goals
; and parts on hand.
;
f1 = 14100 ; high-band (20M) design frequency (kHz)
f2 = 7100 ; low-band (40M) design frequency (kHz)
dia = 2.0 ; coil diameter in inches
len = 0.82 ; coil length in inches
C = 50 ; pF

```

#### ----- CALCULATE LC PRODUCT, L, AND C VALUES -----

```

;
; LC represents a lumped value of inductance times
; capacitance. Using the given C, the required value
; of L is calculated. The trap must exhibit a high
; impedance in the high-band operating range.
;
K1 = 1.0e+12 ; conversion constants
K2 = 1.0e+9
Xr = 200 ; recommended reactance value
LC = K1 / (4 * pi()^2 * f1^2)
Cr = K2 / (2 * pi() * f1 * Xr) ; recommended C value
L = LC / C ; L based on your C value
Lr = LC / Cr ; recommended L value
L(C) := LC / C

```

*listing continued on page 148*



```

;----- NUMBER OF TURNS -----
;
; Determine the number of turns required to produce
; an inductance of L (microhenrys) by using an air coil
; of dia (inches) diameter and len (inches) length.
; The calculation is a reasonable approximation for
; a coil length greater than or equal to 0.4 times
; the diameter. The value is based on the calculated
; L, not the recommended Lr value.
;
turns = (sqrt(L * ((18 * dia) + (40 * len)))) / dia
len >= 0.4 * dia ; constraint

```

```

;----- TURNS PER INCH -----
;
; Calculate the resulting turns per inch. You may need
; to spread turns or use a different form size to get
; the desired result.
;
tpi = turns / len

```

```

;----- ELEMENT LENGTHS -----
;
; The lower member is a full physical 1/4-wave element
; for the higher frequency. The upper member is cut
; to provide an overall electrical 1/4-wave at the
; lower design frequency. Compensation must be made
; for the loading effect of the trap coil by shortening
; the upper element. Use telescoping 2-piece elements
; to allow adjustment of center frequencies on both
; bands and center-loading compensation on the low
; frequency band.
;
K3 = 234000 ; 1/4-wave conversion constant
elem1 = K3 / f1 ; length of lower member
elem2 = (K3 / f2) - elem1 ; max. length of upper member

```

## NOTEBOOK

continued from page 147

Eureka: The Solver, Version 1.0  
 Monday November 30, 1987, 10:12 am.  
 Name of input file: B:\TRAP.EKA

Variables	Values
C	= 50.000000
Cr	= 56.437923
dia	= 2.0000000
elem1	= 16.595745
elem2	= 16.362002
f1	= 14100.000
f2	= 7100.0000
K1	= 1.0000000e+12
K2	= 1.0000000e+09
K3	= 234000.00
L	= 2.5481913
LC	= 127.40957
len	= .82000000
Lr	= 2.2575169
tpi	= 8.0735844
turns	= 6.6203392
Xr	= 200.00000

Figure 3. Eureka's solution.

impedance at that frequency. Adjust the trap in isolation for coarse resonance before installing it in the antenna. Use a dip meter or RX noise bridge to check the trap's resonant frequency.

The need to "cut and try" causes much of the aggravation associated with practical electronic experimentation; numbers worked out on paper are either approximate or simply wrong. Eureka can help by bringing the numbers into line. The rest is up to you. Happy DXing! 73, KB0YH. ■

*Augie Hansen is a computer consultant and trainer who owns and operates Denver-based Omniware. He is the author of several computer books, including Proficient C, Microsoft Press, 1987.*

Listings may be downloaded from CompuServe as ANTENA.ARC.



# CRITIQUE

## Turbo C Tools

Blaise Computing Inc.  
2560 Ninth Street, Suite 316  
Berkeley, CA 94710  
(415) 540-5441  
\$129.00

One of the advantages to programming in C is the availability of a variety of function libraries. By providing tested and debugged code for many programming tasks, a good library can significantly reduce the time and technical expertise required for applications development. Blaise Computing has been supplying high-quality C function libraries for years. They now bring us Turbo C Tools, a library designed specifically for Borland's Turbo C compiler.

It is impossible to completely describe the contents of this library without taking up most of the magazine. Turbo C Tools contains 183 different functions, ranging from the simple, such as sending a new-line to the standard output device, to the complex, like installing a C function as an interrupt service routine. For the sake of economy, I'll describe the more sophisticated functions in detail while providing a summary overview of the package's other capabilities.

Turbo C Tools is not merely a rewrite of the function libraries that Blaise publishes for other C compilers. It is tailored specifically for Turbo C, taking advantage of Turbo C's strengths while not duplicating functions found in Turbo C's own function library. No license or royalty agreement is needed to

distribute programs developed with Turbo C Tools.

### WINDOWS AND SCREEN FUNCTIONS

Turbo C Tools includes 33 screen-handling functions that provide a high-level interface to the video display. Most of the functions work via the BIOS, while a few use direct access to video hardware for maximum speed. Turbo C Tools supports five display types: monochrome display adapter, Color Graphics Adapter, Enhanced Graphics Adapter, Professional Graphics Adapter, and the PCjr. The EGA 43-line text mode is supported for either monochrome or color monitors.

The Turbo C Tools' screen functions enable the user to: determine the type of installed video equipment and its current mode; select display device, mode, and page; retrieve and set cursor location and style; clear and scroll whole or rectangular regions of screens; write to and read from the screen, and control color palettes.

Turbo C Tools provides powerful support for text-based windows. There are 27 functions for creating, displaying, modifying, and using windows. Examples of windows created with the Turbo C Tools windows functions are shown in Figure 1 (see page 152). Many of these functions provide control of advanced window features. The advanced features can be used if needed, but can also be safely ignored since windows are automatically created with a set of default characteristics that are suitable for most applications. This choice between default and user-specified window characteristics provides flexibility.

Creating a window requires only a single function call, passing the window's size and the attributes of its data area as parameters. Once created, a window exists as a data structure in memory. Displaying the window on the screen requires another call, to specify the window's screen location and border type. A window is required for almost all window functions, such as those that write to and read from a window, whether or not it is displayed. Window output functions include writing strings and single characters, wordwrap, and string formatting (in the manner of `printf()`). Input functions include echoing keyboard input in the window, and reading data from a window to a buffer.

Turbo C Tools allows precise control over a window's appearance. For example, screen attributes of an entire window or of a portion of a window can be changed. Similarly, a window or a rectangular block within a window can be scrolled horizontally or vertically. Windows can overlap, and the number of concurrent windows is limited only by available memory. All interactions between windows, such as saving and restoring previous screen contents, are handled automatically.

Turbo C Tools uses its window functions to implement an impressive set of menu functions. Three basic menu styles are available: vertical, horizontal, and grid. Selection is made with a moving highlight bar or by user-defined keystrokes. A fourth type of menu, Lotus-style, is a horizontal menu with automatic

*continued on page 150*



display of a menu description whenever a menu item is highlighted. The four types of menus that can be created with Turbo C Tools are illustrated in Figure 2 (see page 152). A nice feature is the ability to protect individual menu items when a menu choice is not available at a particular point in the program. A protected menu item is displayed in a different color, and cannot be selected by the highlight bar.

The menu functions allow complete control over the interactions between your menus, program, and keyboard input. By default, the Home, End, Tab, Shift-Tab, Enter, Esc, and arrow keys allow navigation within and between menus in the usual manner. The response to keypresses can be defined any way you like. For example, keys can be defined to enable selection by pressing the first letter in the menu item. The menu functions are very flexible, and can be used for everything from simple 2-choice menus to complicated multilevel menu systems like those used in the Turbo C environment.

### **INTERRUPT AND INTERVENTION ROUTINES**

Some of the most useful features of Turbo C Tools are those provided under the headings of "Interrupt Service Support" and "Intervention Code." The Interrupt Service Support routines provide all the support needed to utilize interrupt service routines (ISR) written in Turbo C, both hardware and software driven. This includes installing and removing ISRs, manipulating the interrupt vector table, detecting installed ISRs, and installing/removing TSR (terminate and stay resident) programs.

Blaise uses a clever approach to the special problems associated with ISRs. The address placed in the interrupt vector table does not

point directly to the ISR code. Rather, it points to a data structure called the ISR control block. The first few bytes of this structure are a **far** call to the interrupt dispatch routine. The dispatch routine does two things. First, it uses information stored in the ISR control block to prepare the processor to execute the ISR; for example, it sets the stack segment and stack pointer registers. Second, it transfers control to the ISR itself. When the ISR terminates, control passes back to the dispatcher, which restores the initial machine state and returns to the original caller via an IRET instruction. Your own code does not need to attend to these house-keeping chores because the Turbo C Tools routines take care of them.

## ***The Interrupt Service Support routines provide all the support needed to write hardware or software interrupt service routines (ISRs) in Turbo C.***

Turbo C Tools' intervention code routines are the icing on the interrupt service support cake. The intervention routines can schedule your ISRs to be invoked at regular time intervals, at a specified time of day, or when a "hot key" is pressed. This enables you to write programs that will, for example, automatically read data from laboratory instruments every 10 minutes, call an online service for stock quotes at noon every day, or pop up, SideKick-like, at the press of a key. While the Turbo C Tools routines greatly simplify writing of code for ISRs and TSRs, careful programming is still required. The Turbo C Tools manual does a good job of explaining the pitfalls that lie in wait for inexperienced program-

mers. Topics covered include hardware interrupt priorities, reentrancy, interrupt filtering, and background floating point operations.

### **OTHER HELPFUL FUNCTIONS**

Turbo C Tools includes a raft of other functions which, while not as impressive as the screen and interrupt routines, are nonetheless very useful and an important addition to any C programmer's arsenal. String functions perform filling, searching, character conversions, and tab expansion and compression. Graphics functions draw colored dots and lines. Keyboard functions perform various types of keyboard input and manipulation. File functions maintain volume labels, and lock or unlock portions of files. Printer functions provide an interface to the DOS PRINT utility. Utility functions—some of which are implemented as macros—provide 57 miscellaneous services. These include manipulation of pointers and addresses, memory transfers, obtaining program environment information, clock and speaker control, port I/O, and data conversion. Particularly useful are two functions for detecting pointer errors, which can help in tracking down the sources of those pesky null pointer assignments.

While Turbo C Tools certainly covers a wide range of programming needs, it does not do everything. The major deficiencies are in graphics, for which there is only very rudimentary support, and asynchronous communications, for which there is no support. This is not necessarily a problem, because those programmers who do not need graphics or communications functions will not want to pay for them. If you need a communications library, Blaise's "C Asynch Manager" now supports Turbo C as well as Microsoft C. Specialty graphics routine libraries for Turbo C are, or will soon be, available from several firms.

### **EVALUATING A FUNCTION LIBRARY**

Of course, a C function library must be evaluated on more than



the number of functions it provides. The key concern is: Do the functions do what they are supposed to? Of almost equal importance is how well integrated and robust the functions are, how errors are handled, availability of customer support, and the quality of the documentation.

Turbo C Tools scores very high marks in these areas. While time limitations prevented me from performing extensive tests on every function, I did give the more complicated ones a thorough workout. All performed smoothly and without problems. Given the complexity of some of the Turbo C Tools functions, this suggests that the code is well written and thoroughly debugged. Runtime errors are handled well. For example, if you try to lock part of a file that has not been opened, display a window that is already displayed, or select a non-existent menu item, the function does not crash but instead returns an error code, allowing your program to trap the error. The Turbo C Tools header files are written to take advantage of Turbo C's

strong data-typing and function-prototyping capabilities, which is very helpful in detecting and preventing bugs. Complete sample programs are included that illustrate the use of many Turbo C Tools functions, particularly the more complicated ones such as interrupts and intervention code.

Turbo C Tools comes with four compiled library versions, one each for the Small, Medium, Compact, and Large memory models. The library for the Small model can also be used for the Turbo C Tiny model, with the limitation that several speaker-control functions are not available. When using the Turbo C Huge memory model, the Turbo C Tools Large model library provides reliable support for programs that don't contain data objects (such as arrays) exceeding 65536 bytes in size. Since programs lacking data objects larger than 64K can be compiled with the Large memory model, the point of this support is unclear. For programs that really require the Huge memory model, a Huge model

library can be constructed by re-compiling the Turbo C Tools source code modules using the **-mh** compiler option.

Did I say "source code"? Yes indeed—source code is included for every Turbo C Tools function. Almost all of the modules are written in C. The few exceptions, mostly the routines for direct video access, are in assembly language. The source code is fully and clearly commented, and can be modified and recompiled if you have the need, or the code can simply be used as an educational tool.

The manual is a bit over 300 pages and is supplied in an IBM-size binder. While I wish the type was slightly larger, the manual is otherwise quite good. It begins with a general overview of the Turbo C Tools functions, instructions for installation, and information on programming style, compiler warnings, and printing and modifying the source code. Next, there is a chapter on each category of function—string, menu, window, etc.

*continued on page 152*



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

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

- Lowest price guaranteed
- Latest versions
- Huge inventory, immediate shipment
- Knowledgeable sales staff
- Special orders
- 30-day money-back guarantee

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

**We'll Match Any Nationally Advertised Price.**

	LIST	OURS		LIST	OURS
<b>ARTIFICIAL INTELLIGENCE</b>					
ARITY STANDARD PROLOG	95	79	FINALLY! XGRAF	99	89
MULISP-87 INTERPRETER	300	199	FLASH-UP	89	79
PC SCHEME	95	85	FLASH-UP TOOLBOX	49	45
SMALLTALK/V	SPECIAL 99	79	GRAPHPAK	69	59
EGA/VGA COLOR OPTION	50	45	INSIDE TRACK	65	55
GOODIES DISKETTE #1	50	45	MICROHELP UTILITIES	59	49
SMALLTALK/COMM	50	45	PEKS & POKES	45	39
TURBO PROLOG	100	69	QBASE	89	79
TURBO PROLOG TOOLBOX	100	69	QBASE REPORT	69	59
VP-EXPERT	100	89	QUICKBASIC	SPECIAL 99	65
			QUICK-TOOLS	130	109
<b>ASSEMBLY LANGUAGE</b>			QUICKPAK	69	59
EZ_ASM	70	65	QUICKPAK II	49	45
MS MACRO ASSEMBLER	150	99	QUICKWINDOWS	99	89
OPTASM	SPECIAL, NEW 195	165	TRUE BASIC	100	79
THE VISIBLE COMPUTER-8088	80	65	W/RUNTIME	150	105
THE VISIBLE COMPUTER-80286	100	89	TURBO BASIC	100	69
TURBO EDITASM	99	85	DATABASE TOOLBOX	100	69
			EDITOR TOOLBOX	100	69
			TELECOM TOOLBOX	100	69
<b>BASIC</b>					
DB/LIB	139	119	C LANGUAGE		
EXIM SERVICES TOOLKIT	50	45	C TOOLS PLUS/5.0	129	99
FINALLY!	99	89	ESSENTIAL C UTILITIES LIB.	185	125

**Terms and Policies**  
 • We honor MC, VISA, AMERICAN EXPRESS  
 No surcharge on credit card or C.O.D. Prepayment by check. New York State residents add applicable sales tax. Shipping and handling \$3.95 per item, sent UPS ground. Rush service available, prevailing rates.  
 • Programmer's Paradise will match any current nationally advertised price for the products listed in this ad.  
 • Prices and Policies subject to change without notice.  
 • Hours 9AM EST—7PM EST  
 • We'll Match any Nationally Advertised Price  
 • Mail Orders include your phone number  
 • Ask for details. Some manufacturers will not allow returns once disk seals are broken.  
 Dealers and Corporate Buyers—Call for special discounts and benefits!

**1-800-445-7899**  
**In NY: 914-332-4548**  
 Customer Service:  
**914-332-0869**  
 International Orders:  
**914-332-4548**  
 Telex: 510-601-7602

	LIST	OURS		LIST	OURS
ESSENTIAL COMM LIBRARY	185	125	TURBO WINDOW/PASCAL	95	79
GREENLEAF C SAMPLER	95	69	TURBO-TO-C TOOLS	495	449
GREENLEAF COMM LIBRARY	185	125	UNIVERSAL GRAPHICS LIBRARY	150	119
GREENLEAF FUNCTIONS	185	125			
MICROSOFT QUICK C	SPECIAL 99	65	<b>OTHER LANGUAGES</b>		
PANEL/QC OR /TC	129	99	LAHEY PERSONAL FORTRAN 77	95	89
PERISCOPE II-X	145	105	LOGITECH MODULA-II COMP KIT	99	79
PFORCE	295	215	MICROFOCUS PERSONAL COBOL	149	119
RESIDENT C	99	85	PC/FORTH	150	109
TURBO C	100	65			
TURBO C TOOLS	129	99	<b>UTILITIES</b>		
TURBO WINDOW/C	95	79	DAN BRICKLIN'S DEMO PROGRAM	75	59

**TURBO PROFESSIONAL 4.0**

New library of over 400 routines for the latest version of Borland's Turbo Pascal (4.0). Includes pop-up resident routines, BCD arithmetic, virtual windows and menus, EMS and extended memory access, long strings, large arrays, macros, and runtime error recovery. Complete source code is included.

List: \$99      **Special Price: \$79**

	LIST	OURS		LIST	OURS
<b>PASCAL LANGUAGE</b>			<b>BORLAND PRODUCTS</b>		
ALICE	95	69	EUREKA	167	115
AZATAR DOS TOOLKIT	NEW 75	69	REFLEX: THE ANALYST	150	99
FLASH-UP & MOUSE TOOLS	89	75	SIDEKICK	85	59
FLASH-UP	89	79	SUPERKEY	100	69
FLASH-UP DEVELOPER'S TOOLBOX	49	45	TURBO BASIC COMPILER	100	69
MACH 2	75	59	TURBO BASIC DATABASE	100	69
METRABYTE D/A TOOLS	100	89	TURBO BASIC EDITOR TOOLBOX	100	69
SCIENCE & ENGINEERING TOOLS	75	69	TURBO BASIC TELECOM TOOLBOX	100	69
SCREEN SCULPTOR	125	95	TURBO C COMPILER	100	65
SPEED SCREEN	35	32	TURBO LIGHTNING	150	95
SYSTEM BUILDER	150	129	W/WIZARD	100	69
IMPEX	100	89	TURBO PASCAL DBASE TOOLBOX	100	69
REPORT BUILDER	130	115	TURBO PASCAL DEV. TOOLKIT	395	289
TURBO ADVANTAGE	50	45	TURBO PASCAL EDITOR TOOLBOX	100	69
TURBO ADVANTAGE COMPLEX	90	79	TURBO PASCAL GAMESWORKS TB	100	69
TURBO ADVANTAGE DISPLAY	70	65	TURBO PASCAL GRAPHIX TOOLBOX100	100	69
TURBOASM	99	69	TURBO PASCAL NUM. METHODS	100	69
TURBO ASYN PLUS	129	99	TURBO PASCAL TUTOR	70	45
TURBO GEOMETRY LIBRARY	NEW 100	89	TURBO PROLOG COMPILER	100	69
TURBO HALO	99	85	TURBO PROLOG TOOLBOX	100	69
TURBO MAGIC	199	179			
TURBO PASCAL	100	69			
TURBO PASCAL DEV. TOOLKIT	395	289			
TURBO PLUS	100	89			
TURBO POWER UTILITIES	95	79			
TURBO PROFESSIONAL 4.0 SPECIAL	99	79			

**Programmer's Paradise™**  
 A Division of Hudson Technologies, Inc.  
 42 River Street, Tarrytown, NY 10591





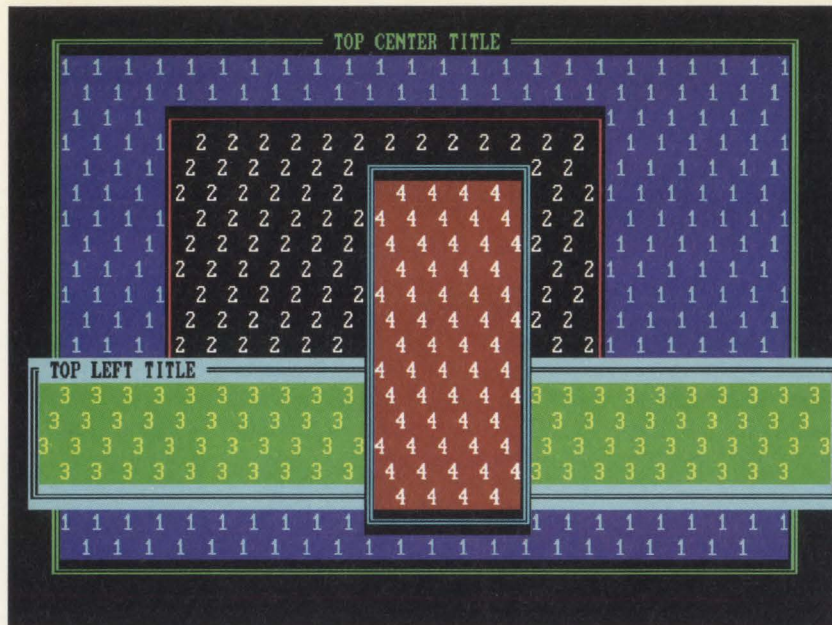


Figure 1. Turbo C Tools provides a complete set of functions for management of screen windows. The programmer has total control over size, position, border style, and color, plus a flexible set of window input/output functions. Interactions between windows, such as saving and restoring previous screen contents, are all handled automatically.

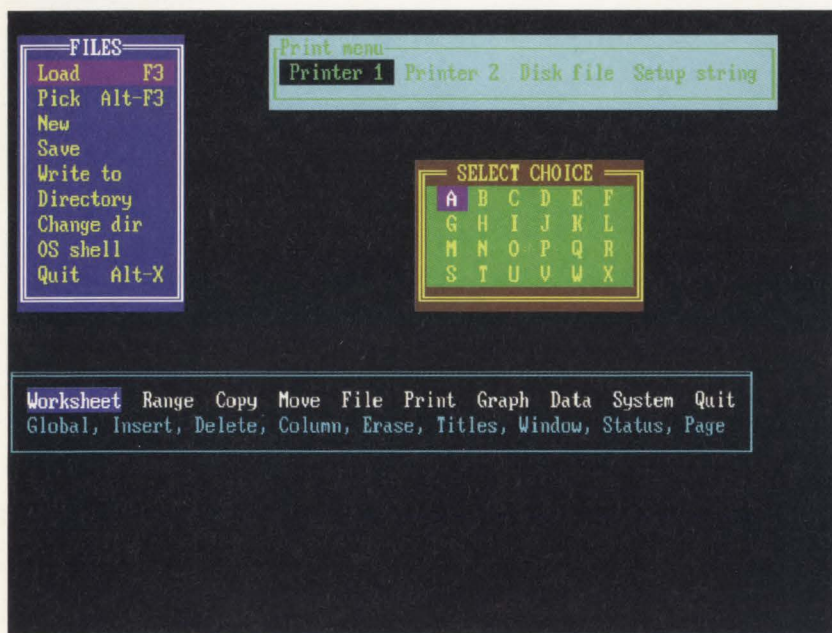


Figure 2. Turbo C Tools makes it easy to include sophisticated menus in your programs. The four available menu types are illustrated here: vertical, horizontal, grid, and Lotus-style. Menu colors can be tasteful and discrete or loud and garish depending on the programmer's aesthetic preferences.

These chapters not only summarize the capabilities of the functions, but provide a quick overview of those aspects of DOS and the computer hardware that are relevant to the functions under discussion. For example, the section on screen functions discusses attribute bytes, color palettes, and the video modes available on the five supported display adapters. The sections on interrupt service support and intervention code are particularly useful.

**Customer support seems excellent. I called Blaise a few times and on all occasions I received prompt and knowledgeable advice.**

The remainder of the manual is made up of an alphabetical function reference. Each entry summarizes the function's action, lists any necessary include file, and defines the function's return value and parameters. An example program fragment is provided for every function. There are useful appendices on troubleshooting, header files, error messages, etc., plus a detailed index.

Customer support seems excellent. I called Blaise a few times for clarification on some points that weren't clear in the manual. On all occasions I received prompt and knowledgeable advice.

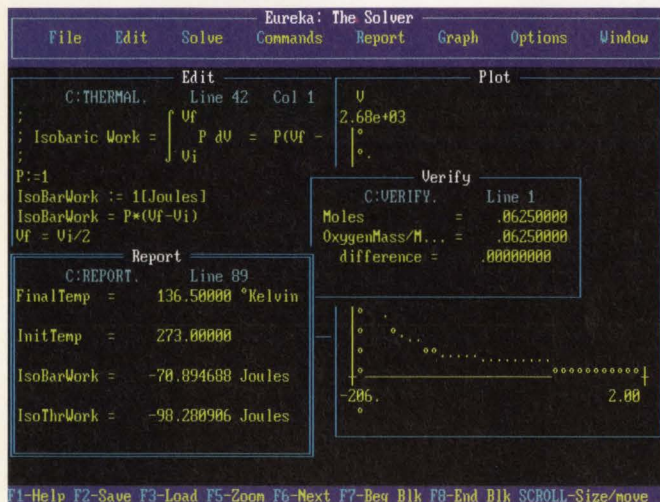
As you may have surmised, I am very impressed by Turbo C Tools. It is a powerful, well-implemented function library that should be a welcome addition to any Turbo C programmer's shelf. ■

—Peter Aitken



# How Eureka: The Solver instantly solves equations that used to keep you up all night

The state-of-the-art answer to any of your scientific, engineering, financial, algebraic, trigonometric, or calculus equations = Eureka: The Solver™



Eureka instantly solved this Physics equation by immediately calculating how much work is required to compress isobarically 2 grams of Oxygen initially at STP to 1/2 its original volume. In Science, Engineering, Finance and any application involving equations, Eureka gives you the right answer, right now!

**E**ureka 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 non-linear 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.  
Copyright 1987 Borland International.

BI-11458

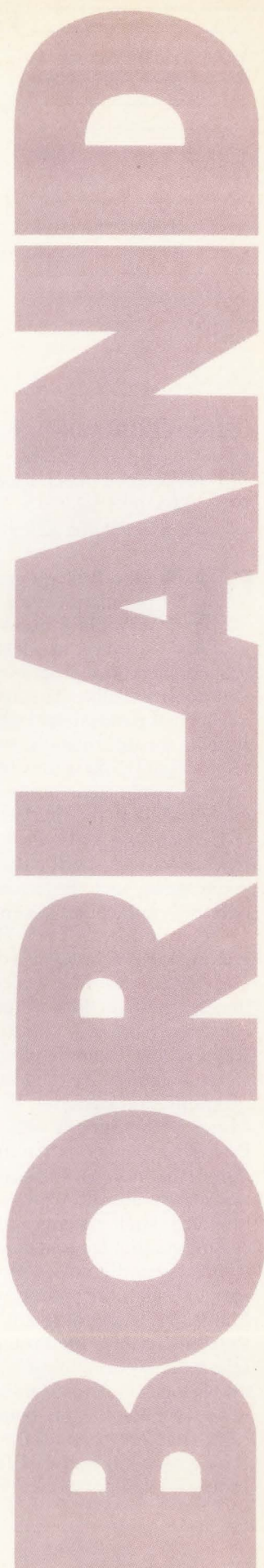


For the dealer nearest you or to order by phone

**Call (800) 255-8008**

In CA: (800) 742-1133;

In Canada: (800) 237-1136









simple program representing a medical diagnosis expert system gives an idea of how to develop such an expert system. One chapter is also dedicated to processing natural language. Here, programs demonstrate the use of keyword analysis in processing user queries, lexical sentence analysis, and so forth.

Within each chapter of this section, the author begins with the basic concepts. For example, when discussing expert systems, the author briefly discusses expert system fundamentals, such as the structure of an expert system, knowledge representation, and methods of inference. Structure charts and data flow diagrams illustrate program design and implementation.

Throughout the book, Dr. Yin illustrates the use of structure methodology in building programs by first creating smaller modules (similar to Pascal's procedures) that can be combined to create larger programs. Later chapters carry this a bit further by borrowing routines created in earlier chapters to build larger applications.

For the most part, example programs are short and simple, and the coverage on topics is limited to basics. This is both the book's strength and weakness: strength from the beginner's perspective, and weakness from an advanced user's perspective. The book is a great value for novice programmers, providing a clear, concise, and comprehensive guide to Turbo Prolog.

In contrast, experienced programmers looking for in-depth coverage on specific topics may be disappointed. Basic coverage is applied to most topics, but a few topics of interest to experienced users are missing. For example, there is no discussion on interfacing other languages with Turbo Prolog. In spite of this, the book is worth the price and should be considered by any Turbo Prolog programmer. ■

— Sanjiva Nath

## ADVANCED TECHNIQUES IN TURBO PROLOG

Carl Townsend, SYBEX, Inc.,  
Alameda, CA: 1987, ISBN 0-89588-428-3, 398 pages, softcover, \$18.95, disk \$29.95.

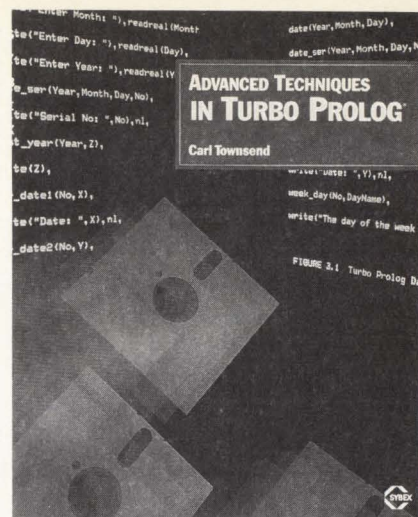
Despite the somewhat intimidating reference to "advanced techniques" in the title, you don't have to be a Prolog wizard to benefit from this volume. In the book's preface, Townsend describes his audience as "experimenters interested in developing small expert systems containing up to 750-1000 rules." The concept behind this book is that intermediate-level Turbo Prolog programmers can best develop their skills through a combination of study and experimentation. To this end, Townsend takes the reader through an impressive collection of source code, which is available for further study and modification.

The book is divided into two parts—the Basic Library and the Applications Library. Townsend builds the Basic Library as a set of Turbo Prolog tools, which he uses in the Applications Library.

### BASIC LIBRARY

The introduction to the first section describes the similarities and differences between Turbo Prolog and other procedural and declarative languages. I was pleased to see that this discussion, especially the comparison with standard "Edinburgh" Prologs, is presented in a well-balanced manner. Although the reader presumably owns Turbo Prolog, Townsend's observations are also valuable for improving one's overall *Prolog literacy*.

Chapter 1 examines Prolog in the context of procedural languages. After noting the difference between declarative and procedural



programming, Townsend presents the Turbo Prolog equivalents for the familiar **IF..THEN..ELSE**, **DO..WHILE**, and other control structures. For those just starting the process of mastering Prolog, these ten pages are a nugget.

The next three chapters build a library of *tool* predicates that can be incorporated into different programs. Each predicate is documented with its name, the data type and flow pattern for each argument, and a short description of the predicate's application. General-purpose predicates, benchmarking code, and eight pages of date-processing routines are included. An important discussion at the end of this chapter covers two fundamental Prolog predicates: **true** and **repeat**.

The pace picks up in the next chapter as the author presents predicates for string-handling operations. These predicates are particularly useful, since they mesh well with the existing string-handling predicates in Turbo Prolog. This chapter's only fault is that it's too short.

The text proceeds with a discussion of list-processing predicates, which Townsend groups into four  
*continued on page 156*



categories: predicates for displaying the contents of lists, predicates for sorting lists, predicates that apply statistical operations (like standard deviation and variance) to lists, and general list predicates. The sorting predicates predictably cover the bubble sort, the insertion sort, and the Quicksort, and are all briefly annotated in the text. I suspect that someone new to Prolog or to recursion may need an outside reference to completely understand how this code works. But then again, this is not a beginner's book.

Townsend also presents a chapter on arithmetic operations that include predicates for finding the maximum, minimum, or greatest common denominator of two numbers; as well as code for performing the *Sieve of Eratosthenes* benchmark. The chapter concludes with an example mini-program demonstrating the use of the **random** predicate, and with suggestions for using the **random** predicate in applications.

Databases are of prime importance to Turbo Prolog programmers. Although the author devotes an entire chapter to the dynamic database, including disk-based databases, Townsend's coverage falls short of the mark. The predicates for the disk-based database only hint at what can be done using the routines from the Turbo Prolog User's Manual. Similarly, I felt that the code for querying and sorting a database was lean.

The last two chapters in the first part of the book undertake some truly advanced topics: BIOS- and DOS-level support, and interfacing Turbo Prolog with other languages. These chapters are idea-oriented, so they contain no presentation of separate predicates. Rather, these chapters present concepts and

mini-programs on topics such as the use of a mouse in a Turbo Prolog application. The chapter on BIOS- and DOS-level programming is a good introduction to the use of the Turbo Prolog **bios** predicate.

The chapter on compiling and interfacing Turbo Prolog code with other languages is alone worth the price of the book. This discussion covers the theory behind interfacing and troubleshooting programs that use Turbo Prolog with C (Lattice, Microsoft, and Turbo) and assembly language.

### APPLICATIONS LIBRARY

In the second part of the book, the author sets off on a grand tour of applications ranging from the design of forward- and backward-chaining expert systems to natural language parsers. These four chapters examine the strategies, models, and predicates used to build expert systems in Turbo Prolog. Here, Townsend incorporates the Turbo Prolog tools built in the first section of the book into the design of actual applications. The ideals of readability, reliability, and efficiency are stressed throughout.

The first chapter in this section introduces fundamental concepts of expert systems, and presents a simple medical diagnostic system. In the next chapter, Townsend does a first-class job developing concepts such as knowledge representation and search strategies. His explanations mesh well with the systems he presents.

Townsend uses a second medical diagnostic system, which includes certainty factors in the reasoning, in a discussion of backward-chaining expert systems. This discussion leads the reader through all the steps of building the expert system, including accumulating, charting and clustering facts about various vitamin deficiencies. The next two chapters implement a planning expert system using a forward-chaining mechanism, and a frame-based weather-forecasting expert system.

The natural language processing chapter presents a quick rundown of noise-disposal, state-machine, and DCG parsers, with example code. I found the DCG parser code pertaining to airline schedules to be particularly interesting.

The last two chapters in the book describe techniques for solving logic problems and spanning tree problems. Although the material on logic puzzles is interesting, the accompanying discussion is skimpy. The same can be said for the chapter on spanning trees. Somehow, I got the feeling that although they contain nice pieces of code, both of these chapters were included as an afterthought; they're far too short and seem out of place both physically and logically.

Overall, I like both the content and presentation of this book. Physically, it is a sturdily bound paperback. The publisher made good use of typography to set off headings, text, and code. If you like to write in your books, you'll appreciate the wide margins. The index is adequate, and the appendices are genuinely useful (I particularly like the glossary). For those who dislike typing source code, an optional diskette is available for \$29.95.

Personally, I might have coded some of the Turbo Prolog predicates presented in this book differently, but that's as it should be. After all, the goal of the book is to get the reader to think about, and in, Turbo Prolog. Despite the imposing title, novices as well as intermediate programmers can profit from this book, because its topics range from fundamental to very advanced. Despite some weaknesses, this book is a valuable addition to the library of both the novice and seasoned Turbo Prolog programmer. ■

—Alex Lane

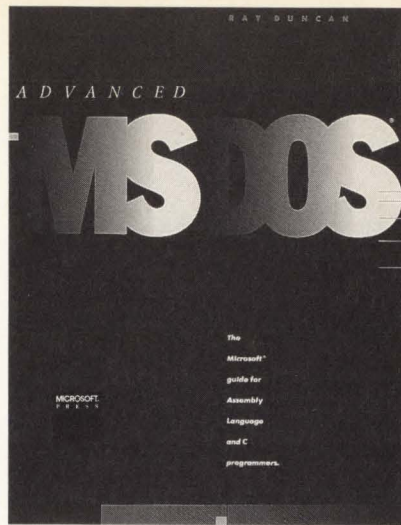


## ADVANCED MS-DOS

Ray Duncan, Microsoft Press, Redmond, WA: 1986, ISBN 0-914845-77-2, 468 pages, paperback, \$22.95.

There are millions of IBM-PC and PC-compatible computers in use today, and almost every one of them uses the MS-DOS operating system (the version sold by IBM, called PC-DOS, is essentially identical). Many computer users, intent only on working with applications programs, can successfully master their spreadsheets, databases, and word processors without having the slightest idea about the internal workings of DOS. For those of us who write programs, however, knowledge of the operating system is essential. Dozens of books have been published on DOS, some intended for beginners and some for more advanced users. As its title implies, *Advanced MS-DOS* falls into the latter category. It is an excellent book, providing the clearest and most complete exposition of DOS that I've seen (and I've read most of the competition).

The emphasis of this book is not only on understanding how DOS works, but (quoting the Introduction) on providing "the detailed information necessary to write robust, high-performance applications under MS-DOS." It does this by explaining in detail how programs run under DOS, and how programs can access the many DOS services that are available. This book is not for beginners, but is intended for experienced programmers who are familiar with the architecture of the 8088/80286 microprocessor family. The author does not attempt to provide an assembly language tutorial, an introduction to programming logic, or an explanation of PC hardware.



The first 270 pages of the book comprise a section called "Programming with MS-DOS." It begins with a history of DOS, starting with its origins in the CP/M-80 operating system and detailing the changes made as DOS evolved from version 1.0. The second chapter discusses the bootstrap procedure, how DOS is loaded in memory, and the general functions of the three major components of the operating system: the Basic Input/Output System (BIOS), the DOS kernel, and the command processor. Among the topics covered in other chapters are file and record manipulation, programming of character devices, and details of disk access. The chapters on memory allocation, the EXEC function, interrupt handlers, and installable device drivers are themselves worth the price of the book because of their clear treatment of these complex subjects.

The remainder of the book is devoted to programmers' reference material. First, there is a complete reference to DOS interrupts 20H through 2FH. Most of this section is devoted to the many operating system services that can be accessed via software interrupt 21H. Next is a section detailing the ROM BIOS interrupts 10H (video), 13H (floppy disk), 14H (serial port), 16H (keyboard), and 17H (printer). The last section refers to interrupt 67H,

which accesses the Lotus/Intel/Microsoft expanded memory manager. Sample code, in assembler, is included for each interrupt.

Duncan feels, as I do, that functioning, documented programs are an indispensable learning tool. The book contains many code fragments and complete programs, most of which can be used directly, or with minor modification, in your own programs. These include an interrupt handler for the divide-by-zero critical error; a boilerplate installable device driver; a simple interrupt-driven terminal program, and perhaps most interesting of all, a simple command shell for DOS that replaces COMMAND.COM as the user's interface to DOS. (COMMAND.COM is called via EXEC to process commands that the shell itself does not handle.) The shell may be easily extended by the seasoned assembly language or Turbo C programmer.

The programming examples are in assembler or C, which makes the book most appealing to users of those languages. Even so, the clarity of explanation and the completeness of program comments are such that the book will be of value to users of other languages, such as Pascal and BASIC. A companion disk, containing the programming examples as both source code and executable files, is available by mail for \$15.95.

Although I did not try every program example, the book appears remarkably free of errors. This speaks well for the author's care in writing and knowledge of the subject. Coupled with his clear and straightforward writing style, this makes for an eminently useful and readable book. If you have the necessary background, this book provides you with the information you'll need to write the next 1-2-3 or SideKick. ■

—Peter Aitken



# TURBO RESOURCES

---

## YOUR SUBSCRIPTION

You're looking for Borland language information. Where to go? Well, for starters, right here. A **free** 12-month subscription to *TURBO TECHNIX* is yours for the asking when you register any of the Borland languages (including Quattro, Paradox, Eureka, and Sprint) or language toolboxes. A subscription request card is packaged with each of those products—fill it out and return it to be sure you get every issue. If your copy of a Borland language product was shipped without the subscription request card, just write, "I would like to subscribe to *TURBO TECHNIX*" in the bottom margin of the registration card.

---

## COMPUSERVE

The best online information about the Borland languages can be found on CompuServe. Subscribing to CompuServe can be done through the coupon enclosed with every Borland product (which also includes \$15 worth of online time for your first month) or by calling CompuServe at (800) 848-8199. You'll need a modem and some sort of communications software that supports the XMODEM file transfer protocol.

Learning your way around CompuServe takes some time and practice, but good books have been written about it, including Charles Bowen's and David Peyton's *How To Get The Most Out Of CompuServe and Advanced CompuServe for IBM Power Users* (New York: Bantam Computer Books, 1986). Howard Benner's TAPCIS shareware utility can automate sessions and help you minimize connect time. It is available for downloading on CompuServe from DL 12 of the Word Perfect Support Group forum (**GO WPSG**). The TAPCIS file is 239,297 bytes long—plan to spend some hours downloading it.

### How to access the Borland Forums on CompuServe:

*TURBO TECHNIX* listings for Turbo Pascal and Turbo Basic are available

in DL1 (Data Library 1) of the BPROGA Borland programming forum (**GO BPROGA**). Turbo C and Turbo Prolog listings are stored in DL1 of the BPROGB forum (**GO BPROGB**). Listings for Business Language articles are also available in DL 1 of the Borland Applications Forum (**GO BORAPP**). From the initial CompuServe prompt, type **GO <forum name>** or follow the menus.

### How to download TURBO TECHNIX code listings from CompuServe:

At the Functions prompt, type: **DL 1**  
This will take you to the *TURBO TECHNIX* data library, where all listing files are stored. Listing files are archived using the ARC52 archiving scheme. You will need the ARC-E.COM program or one compatible with it to extract listing files from downloaded archives.

Archive files are organized two ways: by article and by issue. In other words, there will be one .ARC file for every article that includes listings, and a single, larger .ARC file for each issue containing all the individual .ARC files for that issue. You can therefore download listings for individual articles, or download the entire issue's listings in one operation.

The all-issue files follow a naming convention such that NVDC87.ARC contains all listing archives from the November/December 1987 issue, JAFB88.ARC for the January/February 1988 issue, and so on. The name of an article's individual listings archive file is given at the end of each article.

To download an archive file, type **DOW <filename>/PROTO: XMO** at the DL 1 prompt. After pressing Enter, start your own communications program's XMODEM receive function. After you have completely received the file, you must press Enter once to inform CompuServe that the download has been completed. Once you have downloaded an archive file,

you can "extract" its component files by invoking ARC-E.COM at the DOS prompt this way:

```
C>ARC-E <filename>
```

---

## NATIONAL USER GROUPS

### TUG

The national user group for Turbo languages is TUG, the Turbo User Group. TUG publishes a bimonthly newsletter called *Tug Lines* that contains bug reports, programming how-tos, and product reviews. Extensive public-domain utility and source code libraries are available to members. Dues are \$22.00 US/year (\$23.72 in Washington State); \$26.00 Canada and Mexico; \$38.00 overseas.

### TUG

PO Box 1510  
Poulsbo, WA 98370

### TPro Users

TPro Users was founded specifically to support Turbo Prolog programming. Their bimonthly newsletter contains technical articles, application stories, tips and techniques, and more. TPro also maintains an electronic bulletin board for source code download and message posting. Dues are \$25.00 US/year; \$35.00 overseas.

### TPRO USERS

3109 Scotts Valley Drive, Suite 138  
Scotts Valley CA 95066  
BBS: (408) 438-6506

---

## LOCAL USER GROUPS

One of the best places to look for advice and face-to-face assistance with your programming problems is at a local user group meeting. Most user groups in the larger cities have special interest groups (SIGs) devoted to the most popular programming languages, usually with strong Turbo presences. We will be listing some of the largest and most active user groups in major urban areas across the country; obviously, there are thousands of user groups that we cannot list due to space limitations. If no listed group is convenient to you, ask about local user groups at a local com-



puter store or check with a faculty member at a high school or college with a computer curriculum.

#### BOSTON COMPUTER SOCIETY

Information: (617) 367-8080

BBS: (617) 353-9312

One Center Plaza

Boston, MA 02108

#### CAPITAL PC USER GROUP (DC)

4520 East-West Highway, Suite 550

Bethesda, MD 20814

C SIG: Fran Horvath

AI/Prolog SIG: Dick Strudeman

BASIC SIG: Don Withrow

#### CHICAGO COMPUTER SOCIETY

Information: (312) 942-0705

BBS: (312) 942-0706

Pascal SIG: Bill Todd (312) 439-3774

C SIG: Ed Keating (312) 438-0027

AI/Prolog SIG:

Jim Reed (312) 935-1479

Basic SIG:

Hank Doden (312) 774-5769

#### HAL/PC (HOUSTON)

Information: (713) 524-8383

BBS: (713) 847-3200 or (713) 442-6704

Pascal SIG:

Charles Thornton (713) 467-1651

C SIG: Odis Wooten (713) 974-3674

Compiled BASIC SIG:

Larry Krutsinger (713) 784-9216

AI SIG (Prolog):

George Yates (713) 448-7621

#### NEW YORK PC USER GROUP, INC.

Information: (212) 533-6972

BBS: (212) 697-1809

40 Wall Street Suite 2124

New York, NY 10005

#### PACS (PHILADELPHIA)

Information: (215) 951-1255

BBS: (215) 951-1863

PACS, c/o Lasalle University

Philadelphia, PA 19141

#### SAN FRANCISCO PC USERS GROUP

Information: (415) 221-9166

444 Geary Blvd, Suite 33

San Francisco, CA 94118

#### ST. LOUIS USERS' GROUP

Information: (314) 968-0992

BBS: (314) 361-8662

Pascal SIG:

Jeffrey Watson (314) 481-4239

C/Assembler SIG:

David Rogers (314) 968-8012

BASIC SIG:

Dennis Dohner (314) 351-5371

#### TWIN CITIES PC USER GROUP

Information: (612) 888-0557

BBS: (612) 888-0468

PO Box 3163

Minneapolis, MN 55403

#### Independent CBBS systems with programming orientation

Questor Project Washington, DC

(703) 525-4066 24Hr \$

Illinois BBS Chicago, IL

(312) 885-2303 24Hr \$

PC-TECH BBS Santa Clara, CA

(408) 435-5006 24Hr

\$ = membership fee required

## Coming Up

### The Borland Graphics Interface ...

*An explosion in the number of different graphics devices for the PC has made life difficult for software developers in recent years. The CGA, EGA, VGA and others all need to be supported, but how to do it? Borland now provides the answer with the Borland Graphics Interface (BGI), a device-independent graphics system that detects an installed graphics device at program startup and loads an appropriate graphics driver. The rest is up to you ... but with a toolkit of 60 function calls, your graphics programming should be easier and more powerful than ever. Read Tom Swan's introduction to the BGI and get ready to see your graphics screen in a whole new light.*

### Solve your mouse mysteries ...

*You probably haven't ever gotten the whole story on attaching your cursor to that little cake of soap on its electronic rope. Some registers, a software interrupt, and just a little magic are what it takes to make your code mouse-knowledgeable. In Part 1 of this two-part series, Kent Porter provides a thorough discussion of all text-based mouse calls, and also explains interrupt-driven mouse event handling through the mysterious mouse function call 12H. Working mouse-interface libraries will be provided for both Turbo Pascal and Turbo C. Graphics-oriented mouse programming will wrap up the series in our July/August issue.*

### And the TECHNIX keep on coming ...

*Paradox's PAL language has numerous built-in functions for financial analysis, and Todd Freter takes us on a tour of present and future value, PAL-style. Learn about local variables in Turbo Basic, and list manipulation in Turbo Prolog. Get a glimpse of the future, as Safaa Hashim shows us how the hyper-text data linking paradigm can be implemented in Turbo Prolog. Our book reviews, critiques, and honored columnists will all return to keep you informed in the true Turbo fashion.*

## C:>CLASS.ADS

### TURBOGEOMETRY LIBRARY

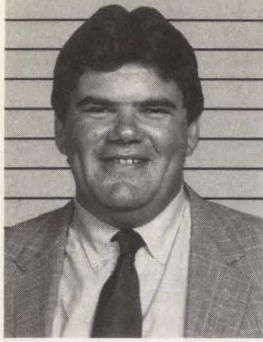
Turbo Pascal, C, Mac and Microsoft C Over 150 geometric routines that include: Intersections of Lines, Arcs, Planes, Circles 2D and 3D Transformations Equations of Lines, Circles, Planes. Hidden Line, Perspective, Curves Surface Areas & Volume Routines Clipping, Composite Matrices, Vectors. Distance Computations. Decomposition of Concave Polygons Req. IBM PC(Comp)/MAC. VISA,MC,MO Source Code,Manual for \$99.95 +\$5 S&H Disk Software, Inc. 2116 E.Arapaho #487. Richardson, TX 75081 (214)423-7288

C:>CLASS.ADS is *TURBO TECHNIX* magazine's display classified advertising section. We welcome to these pages all those who would like to take advantage of the special sizes and rates available for C:>CLASS.ADS—\$300 per column inch, with a 2-inch minimum. (A minimum ad, for example, measures exactly 2 1/16" wide by 2" long.) All C:>CLASS.ADS must be pre-paid and submitted in camera-ready form (black-and-white PMT or Velox) to:

C:>CLASS.ADS  
TURBO TECHNIX  
4585 Scotts Valley Drive  
Scotts Valley, CA 95066

For additional information, please call Production Assistant Annette Fullerton at (408) 438-9321.





# PHILIPPE'S TURBO TALK

Let's take humor seriously.

*Philippe Kahn*

**W**oody Allen was recently asked what memories and legacy he wanted to leave behind to his movie fans and readers. The question was, "Would you like to live on in the minds of your readers?" He responded, "No, I'd rather live on in my apartment."

So a pompous question got a humorous answer—but people with a less-developed or non-existent sense of humor (and you must agree that the high-tech industry seems to have more than its fair share of both types) would have answered differently.

The typical high-tech industry answer would have gone on about "The Cosmic Importance of our State-of-the-Art Gizmo" or about "The Utmost Significance of our Achievements in Bringing Soul to Machines" or some other miracle.

Give me a break!

If you dare to take yourself that seriously, people will laugh at you anyway! All this is not to say that the high-tech industry is shallow or that everything's a joke; far from it. But we all need perspective, and humor has a funny way of restoring perspective—not to mention creating enthusiasm for what you're doing.

It's possible, and healthful, to laugh *about* what you're doing without laughing *at* what you're doing. We should be able to laugh about the high-tech industry.

In some of our ads, we've used a quotation which runs, "The rea-

■ *It's possible, and healthful, to laugh about what you're doing without laughing at what you're doing. We should be able to laugh about the high-tech industry.*

son angels can fly is because they take themselves lightly," and that's our attitude at Borland.

## HIGH TECH CAN ALSO BE HIGH PRESSURE

In our business, we are constantly under pressure. We are expected to keep on building new and exciting software. And that's fun. But I have noticed that the most creative times are often the times when laughs are the rule rather than the exception.

Humor frees up the subconscious ... lets the ideas roll. Hackers know this. Berkeley UNIX has a utility called "Fortune" that displays a little joke when you log in. Maybe PC-DOS should do something like that.

We've all heard about the yoga master who says that the best form of meditation is a big laugh in the morning. I say that's a good start. There is something fundamentally

healthy about a good laugh, and it sure takes the pressure off our shoulders.

## SOFTWARE BUZZWORDS REVISITED

Entire books—very popular books, at that—have been written about this industry and its jargon. We've really left ourselves wide open to the jabs of "outsiders," so why not just join the fun?

For instance, do you know what Imelda Marcos and a "multitasking operating system" have in common? That's a tricky one, but let's just say that Imelda has three thousand pairs of shoes and only one pair of feet!

Or how about looking at what a TV evangelist and "distributed database access methods" have in common: There are going to be solutions real soon now, provided you make a contribution today.

What about "desktop connectivity" versus having the America's Cup in San Diego: Lots of water, but not much wind!

... And we can go on and on.

## WE'RE SERIOUS ABOUT GOOD SOFTWARE

It might be true that humor is the only test of gravity and gravity the only test for humor. So, like the swordsmiths of ancient times, and unlike Woody Allen (who probably has much more humor than we do—but who doesn't program) we at Borland would like to be remembered as forging the best blades of all. With humor.

Seriously. ■



# Instant Replay

## Version III

SOFTWARE  
AHEAD  
OF ITS TIME

*Instant Replay™* is a unique authoring system for creating:

Demonstrations • Tutorials • Music Presentations • Prototypes • Menus

When active, *Instant Replay™* is a DOS Shell that runs, memorizes, and replays programs. It has the unusual ability to insert prompts, pop-up windows, prototypes, user involvement, music, and branching menus into replays.

Building a Demo or Tutorial is easy. Just run any program with *Instant Replay™* and insert explanation pop-up windows, prompts, user involvement, music and so on. *Instant Replay™* remembers everything; it builds a demo that will re-run the actual program or a screens only prototype version.

*Instant Replay™* includes a *Screen Maker* for building pop-up windows, prototype windows, and menu windows. Other useful tools include a *Prototyper*, *Keystroke Editor*, *Music Maker*, *Menu Maker*, *Presentation Text Editor*, *Control Guide*, and *Insertion Guide*.

The screen editor *Screen Genie™* is designed to be memorized by *Instant Replay™*. Creating animations is easy and fun; just run *Screen Genie™* and memorize your activity.

*Instant Replay™* for IBM and True Compatibles, requires DOS 2.0 or greater. *Instant Replay™* is not copy protected. **There are no royalties required for distribution of Demos.**

*Instant Replay™* at \$149.95 is an exciting new product. Because of the quality of this product, *Nostradamus®* provides a 60-day satisfaction money back guarantee. Call or write, we accept VISA, AmEx, C.O.D., Check or P.O. with orders. Demo diskettes and free product brochure available.

**Nostradamus, Inc.** 3191 S. Valley Street,  
(ste. 252) Salt Lake City, Utah 84109  
voice (801) 487-9662

Data/BBS 801-487-9715 1200/2400,n,8,1

### Instant Replay™

#### Features:

- Illustrated manual with index
- Music Maker
- Text Editor
- Online Help
- Screen Painter
- Magic Demo Animator
- Dynamic Menu Maker
- Memorize and Replay actual programs
- Memorize and Replay screens only
- Insert: prompts, pop-ups, prototypes, music, and user involvement into replays
- Make insertions while creating or reviewing
- Generate Vapor Ware from actual programs
- Resident Screen Painter for creating and grabbing windows on the fly
- Prototyper that includes slide shows, menus, and nesting
- Keystroke/time editor, inserter, and merger
- Replay chaining and linking
- Modular demo making facilities
- Fast forward and single step modes
- Self Made Tutorial included
- Timed Keyboard Macros
- Numerous and powerful operator input options for Tutorials
- Text File Presentation facility
- Transparent Windows
- Change Defaults
- Foreground or Background Music
- Canned special sound effects
- Unlimited replay branching
- Compressed screens
- Object oriented programming
- Tracking editor
- Plus much more . . .

"Instant Replay™ is one of those products with the potential to go from unknown to indispensable in your software library." *PC Magazine*

"Incredible . . . We built our entire Comdex Presentation with Instant Replay.™" *Panasonic*

"Instant Replay™ brings new flexibility to prototypes, tutorials, & their eventual implementation." *Electronic Design*

"I highly recommend Instant Replay.™" *Computer Language*

"You need Instant Replay™!" *Washington Post*

**Instant Replay™**

**Instant Assistant™**

**Screen Genie™**

**Word Genie™**

**NoBlink Accelerator™**

**Assembler Genie™**

**DOS Assistant™**

**Programming Libraries**

*Supports Turbo Pascal 4.0*

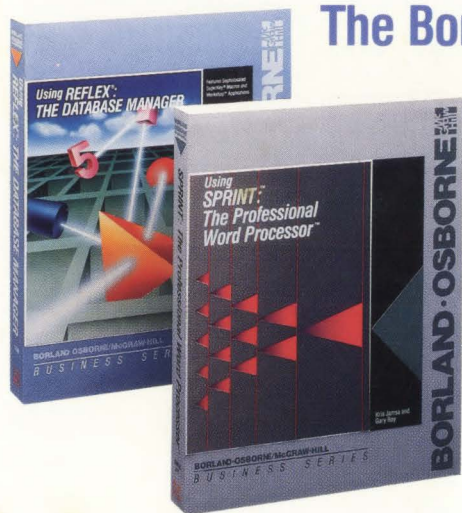
**HardRunner™**

. . . and more

**Nostradamus®**



# Announcing Two Dynamic New Imprints



## 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



## 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

BORLAND-OSBORNE/McGraw-Hill