BUG HUNTING, BORLAND STYLE

Integrated and standalone symbolic debugging

Turbo Pascal 5.0 and Turbo C 2.0

Introducing Turbo Assembler

Definite clause grammars in Turbo Prolog
DESQview API Reference Manual

This is the primary source of information about the DESQview API. It contains all you need to know to write assembly language programs that take full advantage of DESQview's capabilities. The Reference manual comes with an include file containing symbols and macros to aid you in development. AVAILABLE NOW!

DESQview API C Library

The DESQview API C Library provides C Language interfaces for the entire set of API functions. It supports the Lattice C, Metaware C, Microsoft C, and Turbo C compilers for all memory models. Included with the C Library package is a copy of the API Reference Manual and source code for the library. AVAILABLE NOW!

DESQview API Debugger

The DESQview API Debugger is an interactive tool that enables the API programmer to trace and single step through API calls from several concurrently running DESQview-specific programs. Trace information is reported symbolically along with the program counter, registers, and stack at the time of the call. Trace conditions can be specified so that only those calls of interest are reported. AVAILABLE NOW!
Introducing DESQview 2.0 API Tools

Bringing new power to DOS

DESQview API Panel Designer

The DESQview API Panel Designer is an interactive tool to aid you in designing windows, menus, help screens, error messages, and forms. It includes an editor that lets you construct an image of your panel using simple commands to enter, edit, copy, and move text as well as draw lines and boxes. You can then define the characteristics of the window that will contain the panel, such as its position, size, and title. Finally, you can specify the locations and types of fields in the panel.

The Panel Designer automatically generates all the DESQview API data streams necessary to display and take input from your panel. These data streams may be grouped together into panel libraries and stored on disk or as part of your program. AVAILABLE NOW!

DESQview API Pulldown Menu Manager

The DESQview API Pulldown Menu Manager is an interactive tool to aid you in designing pulldown menus. This DESQview API tool assists you in giving your DOS program an OS/2-like look and feel. AVAILABLE OCTOBER 88

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

MS-DOS and IBM PC-DOS are both trademarks of Microsoft Corporation and IBM Corporation respectively.
Turbo Debugger arrives to provide multiple windows into your largest and most difficult programming projects. Advanced features such as 80386 virtual-86 partitions separate debugger and debuggee to allow you to have your full 640K of DOS memory—and debug in it, too.

Turbo Debugger: The View from Within
Michael Abrash

Turbo C 2.0: The Thrill of the Hunt
Kent Porter

Floating Point: The Second Wave
Roger Schlafly

A Directory Search Engine in Turbo C
Jake Richter

Seeing a bug happen is by far the greater part of fixing it. Turbo Pascal 5.0’s Integrated Debugger lets you run your program one step at a time and inspect your data between each step. Turn the lights on in your Pascal code and watch those critters run!

Turbo Debugger: The View from Within
Michael Abrash

Turbo C 2.0: The Thrill of the Hunt
Kent Porter

Floating Point: The Second Wave
Roger Schlafly

A Directory Search Engine in Turbo C
Jake Richter

Seeing a bug happen is by far the greater part of fixing it. Turbo Pascal 5.0’s Integrated Debugger lets you run your program one step at a time and inspect your data between each step. Turn the lights on in your Pascal code and watch those critters run!
The first steps toward natural language comprehension involve the analysis of a language's syntax—a task Turbo Prolog is uniquely suited to perform.
Michael Abrash has coined a new verb to describe what he does for a living: He "zens" 86-family machine code. Yes, he programs in assembler, but his methods differ so markedly from standard practice that he feels a new word is called for.

How one "zens" is hard to describe—though I'll try—but I'm convinced that it works, having seen an EGA-based graphics windowing interface that Michael wrote where a moving window doesn't "blank out" but retains its contents, and moves smoothly from top to bottom, without flicker or any annoying refresh "swoop."

And oh, right, I forgot to tell you: This was on a 4.77-mHz 8088 machine.

The aim of zen coding is to produce the fastest, most compact machine code possible. Following are the principles Michael uses to achieve this aim.

**Love the machine.** I was tempted to say, "Know the machine," but knowledge, while essential, isn't the germ of the principle. Too often we strive to know our hardware like we'd know an enemy, just in order to avoid getting trounced. The aim of zen coding is to wrap the program closely around the hardware so that every element of the hardware works for us, not against us. First of all, this means grabbing every available reference on PC hardware and digesting it down to the status bit level and even further whenever you can. But more than that, it means looking at the complexity of the hardware as an opportunity to fine-tune, and not as a tar pit to die in.

Once you know the hardware, use it. Write to the bare metal at every opportunity. Portability goes out the window because you're writing for this machine—next year you can begin the process again for some other machine.

Nothing says this kind of development comes cheap.

Above all, love it. If you can't quite shake the notion that this is somehow playing dirty, you're not cut out for zen coding.

**Assume nothing.** Optimizing by dead reckoning—that is, by writing a cycle count next to each instruction, adding them up, and then seeing what you can pull out—doesn't work. It doesn't work because instruction cycles aren't the whole story. Every machine has "cycle-stealers," including memory wait states, video wait states, and DMA refresh delays, that skew the total in ways that are nearly impossible to predict on paper. Furthermore, once you factor in the nondeterministic effects of the filling and purging of the prefetch queue, the paper chase is simply over. You cannot know how time-efficient a given solution will be unless you go in and measure the solution in action. Forget how fast a snippet of code must be—go in and see how fast it is.

**Look at all possible solutions.** Some folks build mini-interpreters. Others optimize by giving over subroutine calls and putting repeated instances of the same routine in one long sequence. Still others will do anything to keep values floating in registers. The art of zen coding requires the coder to keep an open mind and have a feel for which solution is right for the problem at hand.

Whether we realize it or not, we often recast a problem in familiar terms to conform to our familiar solutions. Keeping things in registers may help—but eliminating subroutine calls by keeping code inline may help more. If you don't try, you can't know.

None of this is easy. Nor is the above summary the final word: Michael emphasizes the importance of right-brain thinking to tie it all together, and that may be the toughest part of all. Still, I've persuaded him to take a shot at describing his methods in a book, and with some luck, *The Zen of Assembler* will appear next year.

One thing is clear: There comes a point when conventional methods in conventional languages fail us. At that point the only alternative is assembly language, where the programmer becomes the code generator and the rules get turned on their head. Zen coding throws away the precepts of breaking down a problem into independent modules, and demands that the programmer embrace the problem as an organic whole in the quest for a unified, optimal solution. Not everyone can do it—but our very competitive industry will be very good to those who can.
Blaise Passes the Screen Test.

POWER SCREEN

Best performance in a supporting role.

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

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

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

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

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

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

POWER SCREEN includes PAINT, the POWER SCREEN Runtime Library, as well as other utilities for creating help files and maintaining and documenting your screen database files. Language interfaces with source code are included for C, Turbo Pascal 4.0 and QuickBASIC.

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

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

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

Complete price: $129.

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

---

THE BLAISE MENU

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

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

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

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

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

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

- **EXEC** $95.00
  - Program chaining executive. Chain one program from another in different languages; specify common data area; 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

---

Microsoft, QuickC and QuickBASIC are registered trademarks of Microsoft Corporation. Turbo C and Turbo Pascal are registered trademarks of Borland International.
Slander not Aristotle; remember the RESTART; and how tightly is Bruce coupled to his parachute?

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

DIALOG
TURBO TECHNIX Magazine
1800 Green Hills Road
Scotts Valley, CA 95066-0001

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.

NOT PLATONIC
I want to correct an historical error on page 67 of your March/April, 1988 issue. Keith Weiskamp states that Plato is the “Father of Logic.” Nonsense. Plato had the first comprehensive organized philosophic system, but logic was not one of its attributes. It was Plato’s student, Aristotle, who was the true founder of logic (via his Prior and Posterior Analytics for the most part). This is well known and easily verifiable by reading Plato versus Aristotle, whose philosophies are very much opposed for the most part. Plato touted the philosophy of “two worlds”; Aristotle rejected this. Plato believed that all knowledge is innate, existing in people at birth, which is hardly conducive to logic or any logical theory of knowledge. Aristotle completely rejected this as well, stating that all babies are born “tabula rasa,” or like a blank slate, and acquire all knowledge after birth. Plato uses all sorts of illogical premises and arguments. Plato was basically a mystic and the founder of the idea of totalitarianism, via The Republic. So don’t ascribe logic, of all things, to Plato. Give credit where credit is due—to Aristotle.

—Philip Oliver
Indianapolis, IN

We sincerely hope that the old chap will forgive us.

—Jeff Duntemann

SILICON NOSTALGIA
First let me thank you for what is becoming a very excellent publication. There is little in the microcomputer field today (other than the continuing quality of Byte) that offers genuine technical content instead of business chatter.

However, at the risk of being accused of nit-picking, I must take issue with Jeff Duntemann’s statement (“Exploring the Interrupt Vector Table,” May/June, 1988) that “Until the development of the 8086 and 8088, all interrupts were hardware interrupts.”

Evidently Mr. Duntemann has never programmed a Z80 or 8080 chip. Both of these older processors have a software interrupt capability very similar to that of the 8086/88 family, although they only have eight vectors in contrast to the 255 available on later chips. I will not make any statements crediting the 8080 as the first microprocessor to offer this feature, since it may have existed even earlier.

MS-DOS is not the first operating system to take advantage of software interrupts, either. They were used at least as early as the CP/M-80 operating system, and the LDOS/LS-DOS operating systems used on the Z80 made extensive use of software interrupts long before MS-DOS was introduced.

Many writers today fall into the common trap of assuming that IBM, Intel, and Microsoft were innovators who virtually invented microcomputers and operating systems. This simply isn’t the case. They all adapted concepts and hardware that were already developed and in use at that time.

—Gary Lee Phillipps
Chicago, IL

Nay, nay; I was there. Only just last week Mr. Byte snuck into the garage and lifted his leg on my cobwebbed IMSAI 8080 S100 box, which I can’t sell or even give away. What passes for a software interrupt on the 8080 is the mysterious RESTART instruction, which I never used because none of my books ever bothered to explain what it was or how it worked. RESTART was roughly equivalent to an 8088 INT 1, except that RESTART transferred control to a JMP instruction in a calculated location in low memory, rather than to an address contained in a vector table. 8080 hardware interrupts worked in much the same way, so while it’s true continued on page 8
POWER SCREEN

Multiple Screens... give you the big picture.

BLAISE COMPUTING INC.

Presenting POWER SCREEN, a new high performance screen management system by Blaise Computing which provides everything you need to create lightning-fast window oriented applications.

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

POWER SCREEN has the appearance and performance of the popular integrated programming language environments. It helps you to design and modify screens, define fields and how they are formatted, specify range values and field output masks. All attributes and colors are supported including all VGA, EGA and monochrome text modes.

More than just a code generator, screens are stored in a Runtime Library that you can later access and modify without program code changes.

POWER SCREEN supports a variety of languages including Microsoft C 5.0 and QuickC, Turbo C, Turbo Pascal 4.0, and QuickBASIC.

POWER SCREEN includes The Norton Guides Online Instant Access Program ready to use with our database of online help information.

- Virtual screens!
- Context sensitive help!
- Total control over every keystroke during data entry.
- Write to any screen any time, even if it is not visible.
- Automatic physical screen update.
- Range checking is supported for all standard data types.
- Number of screens is limited only by the amount of available memory.
- Detects which display adapter and monitor are used.
- Fully configurable field editing keys.
- Well documented source code.
- No royalty payments.

This package is accompanied by a fully-indexed comprehensive User Reference manual describing POWER SCREEN procedures and utilities. Complete example programs are supplied as well as utilities for creating help files and maintaining and documenting your screen database files.

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

YOUR QUALITY CONNECTION...

Programmer's Connection wants to give you the big picture at a small price. Programmer's Connection carries a complete line of programmer's development tools including the Blaise Computing products for C and Pascal.

List  Our
POWER SCREEN 129 99
C ASYNCH MANAGER 175 135
C TOOLS PLUS/5.0 129 99
EXEC 95 79
KeyPlot 50 47
LIGHT TOOLS 100 99
Pascal ASYNCH MANAGER 175 135
Pascal TOOLS/TOLS 2 175 135
RUNOFF 50 47
Turbo ASYNCH PLUS 129 99
Turbo C TOOLS 129 99
Turbo POWER TOOLS PLUS 129 99

Please refer to our main advertisement in this journal for an expanded listing of our products. To receive our FREE comprehensive Buyer's Guide, which includes descriptions of over 750 products from over 250 software manufacturers, write or call one of our convenient toll free numbers.

Programmer's Connection
7249 Whipple Ave N.W.
North Canton, OH 44720
US 800-336-1166
Canada 800-225-1166
Oh & AK (Collect) 216-494-3781
International 216-494-3781
Fax 216-494-5260
TELEX 9102406879
DIALOG

continued from page 6

that RESTART acted like a software interrupt, nobody ever called it a software interrupt, and very few people ever made the connection.

Flipping through the yellowing pages of 1978-vintage books on the 8080 CPU and SI100 bus last night (I wired-wraped my first machine in 1976 and am by no means a neophyte to this business) made me appreciate how much more we know about our hardware and our operating systems than we did ten years ago. The 8080 and CP/M-80 were much more potent than we ever appreciated, because back then we were working almost blind. As I said in my January/February editorial, much of the power of the 8088 and DOS stems from the depth of our understanding of them. Had I known what RESTART was in 1979, I would have used it, and I would have explained to others how to use it, but the 8080 and CP/M vanished before the industry’s understanding of them achieved the critical mass that the 8088 and DOS enjoy today.

—Jeff Duntemann

MAKING TIME

I read Mr. Ron Sires’ feature “A Memory Resident Clock Utility,” May/June, 1988, with great interest, since I write numerous memory-resident programs. Mr. Sires described a manual procedure for determining the size of a program. He did a compile with the map option set in order to determine the size of the CLOCK.EXE program from the TLINK map. The value 1298H for _BSSEND was rounded up to 1300H and then divided by 16 giving the value 150H for the program size. This value was then used in his main() function in the keep(0, 0x0130) statement. This manual procedure could be replaced with an automatic procedure by changing the original KEEP statement to the following:

```
keep(0,
  ((unsigned int)sbrk(0)+15)/16);
```

Thus, if the size of the program changes, the second parameter to the keep function will automatically change to compensate. Note that the return value from the sbrk function is cast to an unsigned int so that values greater than 7FFFH will be processed correctly. The sbrk function is described in detail in the Turbo C Reference Guide, page 44.

—Alan Cohn

Irvine, CA

Neat hack, Alan. Thanks; I’d been looking for a way to do that. I’ve tested it in CLOCK.C and it works fine, and is a good general way to do the program-sizing job I described how to do manually. The only caution is that I’ve only tested it under the Tiny code model, and sbrk really doesn’t make sense under any but the Tiny and Small code models, since it depends on there being only a single data segment in the program.

—Ron Sires

MAC SCENE

Even though I do all of my programming on an Apple Mac+, I find all of the articles in TURBO TECHNIX help me to write better code. The best features of the Borland programming languages are that they are complete, up to date, similar in format and are thoroughly supported by good tutorials specific to the languages. Tutorials like the Borland/Osborne-McGraw Hill books are nonexistent for the Macintosh, and if there is anything that a beginner needs for the Macintosh, it’s a good tutorial specific to the language. I do have Borland’s Turbo Pascal Tutor for the Macintosh and it is complete but lacks the short programming examples that the Borland/Osborne-McGraw Hill books use to help a programmer get started. (I realize that in a book as big and complex as the Turbo Pascal Tutor this is not possible.)

I would like to see a Borland Turbo C and Turbo Basic, both supported by Borland/Osborne-McGraw Hill tutorials, for the Macintosh. Following that, a TURBO TECHNIX for the Mac would be great. Is there any possibility of that in the near future?

—Robert Orthman

Boulder, CO

Well, gee, given endless funds we can do almost anything—but software R & D and magazine publishing are two of the most expensive endeavors I can think of. Borland’s commitment to Macintosh developer tools is secure, and we can’t be much more specific than that. As for a TURBO TECHNIX for the Mac—that might be a long, long wait. In the meantime, you can’t do much better than Mac Tutor, The Macintosh Programming Journal (P.O. Box 400, Placentia, CA 92670). They publish monthly at $30/year, with 86 pages per issue. Their motto is “No fluff,” and they mean it, with the (minor) downside that they don’t publish what we would consider Square One material.

As for tutorial books, help is coming. The venerable Scott, Foresman & Company has concluded an agreement with Borland very similar to the one between Borland and Osborne-McGraw Hill, to copublish a series of books on Borland’s Macintosh products. All Mac products, including the business products, will be covered, and the books will begin to appear later this year. Watch for Complete Macintosh Turbo Pascal by Joseph Kelly as the first programming tutorial in the series. There will be more. If there were another two or three of me, I’d write one myself.

—Jeff Duntemann

AFTER YOU, BRUCE

Bruce Webster is an interesting man; I had the pleasure of jumping out of an airplane with him and a bunch of other distinguished programmers on a fine sunny day at an altitude of about 3000 feet. Bruce, of course, had impeccable taste. He wore an olive-drab parachute and used structured programming methodology to enter and leave the airplane: One way in, one way out.

I enjoyed his “How Loosely Are You Coupled?” article in the May/June, 1988 issue. It coincides with my recent learning about the topic, which has been around for about ten years. Coupling (and the associated topic, “cohesion”) will be, I predict, the next programming rage.

continued on page 10
in'de-pen'dent (in'di-pen'dent) adj. 1. not influenced by others in opinion, conduct, etc. 2. not affiliated; sovereign in authority. —n. (in'de-pen-den') someone or something independent.

FACT:

Many major dealers specializing in programming tools for personal computers are legal affiliates of companies who also publish development software.

FACT:

Programmer's Connection is not a publisher and is not affiliated to any company that has ever been in the business of publishing software.

When you come to Programmer's Connection, you'll find our knowledgeable, non-commissioned salespeople and technical consultants will give you an unbiased look at the products we carry.

**ORDERING INFORMATION**

**FREE SHIPPING.** Orders within the USA (lower 48 states only) are shipped FREE via UPS Ground. Call for APO FPO PAG, and 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. P.O.s 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 document 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 questions and send 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

**PROGRAMMER'S CONNECTION**

**database management**

**cliquer** by bitBakcDeck
695 519
**eBASE III Plus** by Ashton-Tate
695 439
**FoxBASE** by Fox Software
395 249
**FoxBASE II** with Fox Software
595 399
**FrontRunner** by Ashton-Tate
195 175
**Genie** by ByteWord
395 249
**HI-SCREEN** by Sofware
149 129
**Magic** by Her
199 179
**R/BASE for DOS** by Microprecise
725 539

**microsoft products**

**Microsoft C Compiler** with CodeView
450 299
**Microsoft C/386 Compiler** with CodeView New Version
400 565
**Microsoft FORTRAN** Compiler
450 299
**Microsoft Mac Assembler**
150 105
**Microsoft Mouse with 2 MB RAM**
250 199
**Microsoft Pascal Compiler**
100 99
**Microsoft QuickBASIC**
300 229
**Microsoft QuickPascal**
99 69
**Microsoft QuickBASIC 2.0**
99 69
**Microsoft Windows**
99 69
**Microsoft Windows 3.1**
195 129
**Microsoft Windows Development Kit**
500 229
**Microsoft Word**
450 299
**Microsoft Works**
195 129

**braise products**

**ASYNC MANAGER** supports Turbo C
175 125
**C TOOLS PLUS**/5.0
129 99
**Turbo ASYNCH PLUS**/4.0
129 99
**Turbo Developer**
129 99
**Turbo POWER SCREEN**
129 99
**Turbo POWER TOOLS PLUS**/4.0
129 99

**SoftCode by Software Bottling**

List $195 Ours $179

SoftCode is a screen editor and program generator which makes use of language templates. You simply design a screen with the editor and SoftCode generates the code using templates. You then use of language templates. You simply design a screen with the editor and SoftCode generates the code using templates. You then use.

**other products**

Briefly by Software Bottling
245 184
**Overlay Manager** by TurboPower Software
45 43
**TOBUG 4.0** by TurboPower Software
45 43
**Turbo Analyzer** by TurboPower Software
75 69
**Turbo Professional 4.0**
99 99
**Turbo Programming by ASCON**
389 309
**TurboWAD by IMS, Specify Turbo C or Pascal**
95 75

**Turbo Programmer by ASCON**

List $389 Ours $309
**Turbo Programmer/C**
List $499 Ours $399

Turbo Programmer is an application development system designed to quickly and efficiently produce database applications in Turbo Pascal or C. All you do is draw and paint your screens and tell Turbo Programmer how you want to retrieve your data. With Turbo Programmer you can create entire database application programs complete with b-tree indices, context-sensitive help, and automatic programmer documentation.

**Goldschmidt products**

**Instant Assistant**
100 89
**Instant Reply III**
100 89
**Turbo Plus Supports Turbo Pascal 4.0**
100 89

**peter Norton products**

**Advanced Norton Utilities**
150 89
**Norton Commander**
75 55
**Norton Editor**
75 59
**Norton Guide**
100 65
**For CSS**
150 109
**Norton Utilities**
100 59

**software bottling products**

**Flash-up**
89 79
**Flash-up Developer's Toolbox**
49 47

**Programmer's Connection**

Call for Products Not Listed Here
USA........ 800-336-1166
Canada......... 800-225-1166
Ohio & Alaska (Collect) .............. 216-494-3781
International .............. 216-494-3781
FAX .............. 216-494-5260
TELEX.............. 9102406879

Business Hours: 9:30 AM to 8:00 PM EST Monday through Friday

(216) 494-3781

©Copyright 1988 Programmer's Connection Incorporated

Established 1984
Coupling and cohesion were brought out of the closet by E. Yourdon and L. Constantine in their 1979 Prentice-Hall book, Structured Design. A lot of programmers are just now talking about it in the magazines. There is an excellent summary of it in P.J. Plauger's "Programming on Purpose" column in the January, 1988 issue of Computer Language. (See also an interesting related letter to the editor, entitled "The Zen of Plauger," in the April, 1988 issue.)

There were a few things I wanted to touch on in Webster's article. First, there were two disturbing points mentioned. Dealing with global variables by passing them as parameters to a module does not reduce coupling. Global variables are global variables. No matter how you access them, the trouble remains the same: You're never quite sure how other modules affect them, and you're never sure if what you're doing to them adversely affects some other module.

The other point is that the sort routine of Listing 3 is not quite "completely" decoupled. The complexity of its interface requires the programmer to worry about how the routine does its job: It needs to know the number of bytes in each array element, as well as the number of elements. In addition, coupling is raised with the implicit assumptions that only numbers will be sorted, and that numbers will be in the array. A completely decoupled sort routine procedure header would look something like this:

\[
\text{PROCEDURE Sort(VAR AnyStructure);}
\]\n
I don't think Pascal can handle such a declaration, but from what I've read, C can do it with function pointers.

Thanks, Bruce, for some thought-provoking reading. I'm looking forward to the next installment.

-Bill Parker
Culver City, CA

I take issue with Bill's assertion that passing global variables as parameters, instead of modifying them directly (based on scope), doesn't reduce coupling. One measure of coupling is the ability (or lack thereof) to pick up a routine from one program and drop it into another without modification; another is the ability to use the routine with various sets of parameters. Direct use of globals increases coupling in both of those respects.

As for the sort routine—there's a distinction between coupling and generality (though the two are related). A sort routine that I can drop into any program and use without having to add new global definitions (constants, types, variables, other routines) is loosely coupled. This doesn't mean that it has to handle all sorting situations; I can have a routine that sorts only arrays of integers, and it can still be loosely coupled if it meets the criteria above.

Bill's example of a general sort routine, though, is possible in Turbo Pascal, which does allow untyped \textit{VAR} parameters (all versions) and procedural parameters (version 5.0, though you can kludge them in earlier versions). You would need to pass the structure, the size of a given element in bytes, the total number of elements in the structure, and a pointer to the function that compares any two elements and returns \textit{True} if the first is less than the second, \textit{False} otherwise.

And yes, it's true, I did jump out of a plane with Bill and the other charter members of the PMS Commando Team (and we won't discuss having my right boot momentarily entangled in my suspension lines after the parachute opened). What "Colonel" Bill Parker failed to mention is that he's the one who proposed the jump in the first place. We all wore custom T-shirts stating that this was the "1st Annual Idiot Programmers' Jump," which would make Bill... now, it's too easy. Good to hear from you, Colonel.

-Bruce Webster

\section*{MANDELBROT MANIA}

Let me start off by saying that I enjoyed immensely Fred Robinson's article, "Plotting the Mandelbrot Set With the BGI," in your May/June, 1988 issue. I enjoyed it not only because I am a Mandelbrot Set fan but also because it illustrated very well the use of the BGI for making a program device independent.

You may not be aware of it, but there are a lot of us out there who work with the Mandelbrot Set—some seriously and others like me who do it for fun. As a matter of fact, we have our own newsletter called \textit{Amygdala}, which has a circulation of a few hundred copies and comes out about ten times a year.

\textit{Amygdala} is published by Rollo Silver, and costs $15 for ten issues. The address is:

\textit{Amygdala}
P.O. Box 219
San Cristobal, NM 87564

I look forward to seeing more interesting articles in your publication.

-Hector Santos
Los Angeles, CA

Fred's Mandelbrot Set article generated an astonishing volume of mail and CompuServe activity for something most of us here considered a sophisticated party game. Fred has rewritten and greatly improved his Mandelbrot Set generator, and now makes it available as a shareware product. Those interested may obtain it directly from Fred for $15:

Fred Robinson
29766 Everett
Southfield, MI 48076

Another TURBO TECHNIX author, Jon Shemitz, offers a very innovative Mandelbrot generator that plots a "sparse" image—setting only every fifth pixel and every fourth scan line—very quickly so that you can cancel the full plot if the image doesn't look interesting enough at first glance. The program, which also features mouse-based crosshair zooming, may be obtained directly from Jon for $25:

Jon Shemitz
Emerald City Software
1805A Felt Street
Santa Cruz, CA 95062

Many thanks to Hector for bringing \textit{Amygdala} to the readership's attention.

-Jeff Duntemann

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

For only $99.95, you can incorporate the power of a full-fledged Expert System into your TURBO PASCAL® programs. Seamlessly. Affordably.

Finally. Actual Expert Systems, developed for simple use by any Turbo Pascal 4.0 programmer.

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

You also have the ability to create large rule bases and still have plenty of room left for your program, thanks to conservative memory use. You can link multiple rule bases, you’ll be compatible with our Turbo Companion units, and you have available advanced features like date and time arithmetic, confidence factors, windowing, demons, agendas, blackboards, and more.

Imagine a single “EXE” file containing your user interface and data handling, and a full Expert System.

For a limited time, get a FREE copy of our Turbo SnapShot graphics package worth $79.95. We’ll give one away with every copy of Turbo Expert sold between now and September 30. This package will let you capture graphics images from other programs and use them in any Turbo Pascal program.

You can even convert images from any CGA or EGA format to any other.

On top of all that, Turbo Snapshot has routines for graphic gauges and dials as well as mouse support. You’ll have all you need for a sophisticated graphics front-end for your Expert Systems—free.

Call for more information or to order, (317) 876-3042. Software Artistry Inc., 3500 Depauw Blvd., Suite 2021, Indianapolis, IN 46268. Include $5.00 for shipping and handling.
My initial reaction to Turbo Pascal 4.0 was summarized well in a single word: Wow! If I had to characterize my initial reaction to release 5.0, it would be a different but no less enthusiastic response: I can see! The better part of doing what you must is seeing what you’re doing, and while you can work in the dark, you can work faster with the lights on. The whole thrust of 5.0 is to turn the lights on, via the Integrated Debugger.

LET THERE BE LIGHT

A Pascal program consists of code and data, and neither can be observed directly. Instead, you look to a program’s effects: what it puts on the screen, what it prints to the printer. There are always inferences to be drawn, and if you draw the wrong inferences, you lose.

The Integrated Debugger lets you look directly at both program execution and program data. The means is remarkably straightforward: With a program displayed in the editor window, a colored highlight bar (called the execution bar) covers the next statement to be executed. You press a function key. Bang! That statement executes, and the bar moves to the next statement. Press the function key again. Bang! The statement executes, and the bar moves yet another step forward.

All the while, in a separate window beneath the edit window, one or more variable names appear beside a display of their current values. After each statement is executed, the values of the displayed variables are rewritten to the screen. Thus, while the program runs, you can watch the ebb and flow of program data in the window, which is called the watch window.

The synergy between the execution bar and the watch window, and then step through the code until the variable changes. What took hours to solve by inference now takes seconds—simply because you can see.

INTEGRATED INSTRUMENTS

The Integrated Debugger’s tools fall into two categories:

- Tools that control program execution; and
- Tools that manage the display of data items.

Let’s look at execution control first.

The Integrated Debugger offers the choice of two methods to control program execution: single-stepping and breakpoints. A breakpoint is a stop sign that can be erected anywhere in the source code file. Once the breakpoint is set, simply run the program. When execution reaches the breakpoint, the program pauses, but nothing is lost: The state of the program is retained, and the program can be started again as though it had never stopped.

Single-stepping is just that: The program executes one line of code, and then pauses. As with breakpoints, the pause is not destructive. Single-stepping your way through a program is logically equivalent to simply running the program without interruption.

This is a good place to make an important distinction: The Integrated Debugger is a line-oriented system. The execution bar highlights an entire line of source code, not a single Pascal statement. If a line contains more than one statement, all of the statements on that line are executed in one single step.

Of course, much of the power of Pascal stems from its procedural nature, in which a number of statements are grouped together as a named procedure that’s invoked as a single statement. Do you execute the whole procedure as though it were a single statement? Or do you enter the procedure and then execute its component statements individually?

You do what you must. Turbo Pascal 5.0 lets you have it either way. Two separate commands control single-stepping: Step over (F8) and Trace into (F7).
Step over treats a subprogram as an indivisible statement, executing it completely before pausing again. Trace into enters the subprogram and allows you to single-step the subprogram's statements as well. The two commands are interchangeable (except for their effects). You can merrily step along the main program, treating subprograms as statements, until you reach a subprogram call that’s been acting suspiciously. Then you can duck into the call and take a close look around.

Breakpoints and single-stepping work very well together. In a larger program, you may have a strong hunch where a problem lies. Instead of tediously single-stepping to that point, set a breakpoint shortly before the point where you expect the trouble begins, and then execute the program without pausing until that breakpoint is reached. From the breakpoint, carefully single-step until trouble happens.

LOOKING FOR TROUBLE
Trouble, when you see it, may be a bad branch or some other failure of program control. More likely, trouble will mean that a variable is filled with the right stuff at the wrong time, or the wrong stuff at the right time, or the wrong stuff all the time. To spot that kind of trouble, variables as well as program code must be watched. The Integrated Debugger offers two mechanisms for this process: the watch window and the evaluation box. Both are ways of looking at the contents of program variables. The watch window allows you to watch a variable continuously while the program runs. The evaluation box lets you take a quick peek at something at irregular intervals, and also lets you change the values of program variables when program execution is paused.

The 5.0 watch window takes the place of the Turbo Pascal 4.0 output window on the screen whenever debugging is enabled. One or more variables can be placed into the watch window, and the display of their values is updated.

continued on page 14
in the window every time program execution pauses for a breakpoint or after a single-step.

Unlike the watch window, which always displays during debugging, the evaluation box appears only when summoned. When the name of a variable is typed into the box, the variable's current value appears below its name. Alternatively, you can "point and shoot" by placing the cursor on a variable name and pressing Ctrl-F4; the variable name appears in the evaluation box automatically. Whole expressions may be "grabbed" from the edit buffer by placing the cursor at the start of an expression and then using the right arrow key to copy as much text as desired into the box. A new value for the variable can also be entered; this value is then loaded into the variable, ready for use when program execution restarts.

Both the watch window and the evaluation box can display data in many different ways. Binary values may be displayed as sequences of bytes in decimal or hex. Records may be displayed with field labels or without. Pointers appear as pairs of segment and offset values. Dynamic variables are displayed as dereferenced pointers.

Sets are shown as set elements between set constructor brackets, with closed intervals identified and displayed as such. Files, when displayed, indicate their current mode (OPEN, CLOSED, READ, or WRITE) and the physical filename to which they have been assigned. Arrays are displayed in the same format that array constants are defined.

Furthermore, data may be displayed in terms of simple variables and expressions. The expression may include literals, constants, variables, all legal Turbo Pascal operators, typecasts, and a limited suite of standard functions that include SizeOF, Abs, Chr, Ord, Succ, Pred, Length, Addr, CSeg, DSeg, Seg, OfS, Ptr, SPtr, SSeg, IOResult, MemAvail, MaxAvail, Hi, Lo, and Swap.

Examples of various ways to display data in a watch window are shown in Figure 1.

### DISPLAY SWAPPING

The process of watching code in the edit window, and watching variables in the watch window, doesn't leave any room on the screen for the operation of the program being examined. Given that most modern programs use the entire screen, it seemed inappropriate to divide the screen yet another time for a run window. Instead, Turbo Pascal 5.0 uses a system called display swapping to share the screen between the two debugging windows and the application being debugged.

During the debugging process, the Integrated Debugger ordinarily keeps control of the visible screen. A screen buffer for the application being debugged is maintained in memory. This buffer is brought into view only when the application needs to write to the screen, and then only long enough for the write operation to take place. Then the altered screen is saved back out to memory, and the Integrated Debugger takes control of the screen again. These steps happen very quickly, especially on fast 286 or 386 machines.

This feature, called smart display swapping, is the default mode. You can also specify that the application take over the display every time the application executes a statement, or that the application and the Debugger share the same screen. (This works acceptably well if the application does little or no screen I/O. If the Debugger screen is disrupted, the screen can be rewritten by a menu command.) Turbo Pascal 5.0 can also circumvent the display problem by allowing dual-screen operation, with the Debugger on the monochrome screen and the application on the color screen.

### GOIN' ON A BUG HUNT

Neil Rubenking was nice enough to share a bug he tangled with while developing his directory search engine (See "A Directory Search Engine in Turbo Pascal," p. 27 of this issue.) The bug would rear its ugly head during any use of the search engine, but let's track it down in the context of the Where program presented in Neil's article.

The bug came to light while testing WHERE.EXE in a directory that contained a number of files whose names included the string "ENGINE": ENGINE.PAS, ENGINE2.PAS, and ENGINE3.PAS, plus .BAK and .TPU versions of the aforementioned files. When WHERE was invoked as WHERE E**., all of the engine files were correctly found and displayed. However, when WHERE was invoked as WHERE ENGINE**., none of the files turned up.

The flawed copy of ENGINE.PAS is shown in Listing 1. (The source code for WHERE.PAS is the same as that given in Listing 3 of Neil's article.) You can download the buggy ENGINE.PAS from CompuServe if you want to follow along in real time—just don't mix up the buggy version with the working version from Neil's article!

Prepare the application for debugging by loading WHERE.PAS into the editor, and entering a command line string of "ENGINE**." through Options/Parameters. Be sure the source code for ENGINE.PAS is available to the Integrated Debugger.

Now, we can look at anything we want to. So what do we look at? A hacker's hunch tells us that the file spec must be getting stepped on under certain circumstances, so a good place to start is to watch the file spec as it wends its way through program logic. Since Where passes the file spec to SearchEngine in a variable called template, let's take the first step of setting a watch on template through either Break/watch/Add watch or its shortcut, Ctrl-F7. To avoid having to single-step through the procedure that validates the command-line parameters, let's set a breakpoint on Where's invocation of SearchEngine. To do so, move the cursor to the line that contains the call to SearchEngine, and toggle a breakpoint on by way of either Break/watch/Toggle breakpoint or its shortcut, Ctrl-F8. The line changes color. It's ready.

Run the program by bringing down the Run menu and choosing the Run option. (In Turbo Pascal 5.0, Run is a menu, and all continued on page 16
You'll love these utilities.

**Saywhat?!**

The lightning-fast screen generator

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

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

Whether you're a novice programmer longing for simplicity, or a seasoned professional searching for higher productivity, you owe it to yourself to check out Saywhat. For starters, it will let you build your own elegant, moving-bar menus into any screen. (They work like magic in any application, with just one line of code!)

You can also combine your screens into extremely powerful screen libraries. And Saywhat's remarkable VIDPOP utility gives all languages running under PC/MS-DOS, a whole new set of flexible screen handling commands. Languages like dBASE, Pascal, BASIC, C, Modula-2, FORTRAN, and COBOL. Saywhat works with all the dBASE compilers, too!

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

**Topaz.**

The breakthrough toolkit for Turbo Pascal 4.0

If you'd like to combine the raw power and speed of Turbo Pascal with the simplicity and elegance of dBASE, Topaz is just what you're looking for. You see, Topaz (our brand new collection of units for Turbo Pascal 4.0) was specially created to let you enjoy the best of both worlds. The result? You can create truly dazzling applications in a very short time. And no wonder. Topaz is a comprehensive toolkit of dBASE-like commands and functions, designed to help you create outstanding, polished programs, fast. Think of it. With Topaz you can write Pascal code using SAYs and GETs, PICTURE and RANGE clauses, then SELECT and USE databases (real dBASE databases!), SKIP through records, APPEND data, and lots more.

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

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

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

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

**Special Limited-Time Offer!**

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

To order, call toll-free

800-468-9273

In California: 800-231-7849

International: 415-571-5019

The Research Group

88 South Linden Ave.

South San Francisco, CA 94080

YES. I want to try.

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

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

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

NAME ___________________________________________

ADDRESS ___________________________________________

CITY ______ STATE ______ ZIP ______

☐ Check enclosed ☐ Ship C.O.D. ☐ Credit card

# __________________ Exp. date ______ Signature ______
of Run's options support debugging in various ways. The Run option is the normal way to run a program under the Integrated Environment, whether you're debugging or not.

Execution pauses at the breakpoint. The watch window shows the current value of template: ENGINE.* (see Figure 2). So far, so good—or, so far, no bug.

The light blue bar on the call to SearchEngineAll is the execution bar, which rests on the next statement to be executed, not the statement that was just executed. At this point, we can either execute SearchEngineAll as a single statement by pressing Step over (F8) or else descend into SearchEngineAll and single-step SearchEngineAll's statements by pressing Trace into (F7). Since the problem obviously isn't located in the main body of Where, press F7 to duck into SearchEngineAll and have a look around.

Nothing changes in the watch window. A quick look at the body of SearchEngineAll suggests that this routine is largely a frame for calling SearchEngine. In any event, nothing is done to the file spec within the body of Search-EngineAll, which suggests that the problem lies somewhere within SearchEngine. Before single-stepping, move up the source code and set a breakpoint at the first executable statement in the body of SearchEngine by moving the cursor to that statement and pressing Ctrl-F8. Once the new breakpoint is set, press Ctrl-F9 to start things running again.

The execution bar moves instantly to the first line of SearchEngine. template hasn't changed ... but whoa, hold on: As an actual parameter passed by value to SearchEngine, template isn't...
The Official Endorsed Books On

Quattro

Over 175,000 Copies In Print

"Borland® Osborne/ McGraw-Hill offers you the only full line of endorsed books on Quattro. These titles combine Borland's own technical expertise with Osborne/McGraw-Hill's publishing savvy. With these official Quattro titles, you'll have a comprehensive library that keeps pace with you as you develop greater skills with Quattro."

Philippe Kahn, President & CEO, Borland International, Inc.

**Quattro® Made Easy**
by Lisa Biow
Guides you through a step-by-step introduction
$19.95 600 pp.
ISBN: 0-07-881347-6

**Using Quattro®**
The Professional Spreadsheet
by Stephen Cobb
Gets you up and running fast with basic to more advanced techniques.
$31.95 684pp.
ISBN: 0-07-881330-1

**Quattro® Secrets, Solutions, Shortcuts**
by Craig Stinson
Unveils a clever selection of Quattro tricks.
$21.95 650pp.
ISBN: 0-07-881400-6
Available: 8/88

**Quattro®: Power User's Guide**
by Stephen Cobb
Unlocks Quattro's full power for serious business.
$24.95 600pp.
ISBN: 0-07-881367-0

**Quattro®: The Complete Reference**
by Yvonne McCoy
Details every Quattro feature, command, and function.
$24.95 720pp.
ISBN: 0-07-881337-9

**Quattro®: The Pocket Reference**
by Stephen Cobb
Puts essential commands and features at your fingertips.
$5.95 128pp.
ISBN: 0-07-881378-6

**ORDER TODAY!**
Available at Fine Book Stores and Computer Stores
Everywhere or
CALL TOLL FREE
800-227-0900
Visa, MasterCard & American Express Accepted
YOU CAN'T BEAT

APPLICAtION GENERATOR FOR TURBO PASCAL V4.0 AND TURBO C

The simple and revolutionary new 4GL used to develop sophisticated relational database applications.

YOU CAN'T BEAT THE POWER.

DATABASES DESIGNS. Databoss is revolutionary because it lets you design and paint data entry screens and datafile layouts, as well as menus and reports. DATABASES then automatically generates the solid, structured Pascal or C source code that makes up your finished system.

DATABASES GENERATES. Databoss is a program generator that takes program definitions and produces TURBO C or TURBO PASCAL 4.0 source code. The definitions are created by pulldown-menu driven input screens. Code is generated for menus, file and record editing, file re-index and recovery, reports, and file reconfiguration. No coding is necessary for most purposes.

DATABASES IS UNIQUE. This unique Skeleton File system allows programmers to change the way DATABASES generates code. This means that DATABASES can be used for any application, no matter how complex or unusual.

DATABASES can be modified to suit the individual programmer's style and requirements.

YOU CAN'T BEAT THE FRIENDLINES.

DATABASES IS IDEAL for beginner programmers as it lets them create professional-level Database systems with little or no programming required. It also allows beginners to learn Pascal and C more quickly and easily for professional applications.

YOU CAN'T BEAT THE SPEED.

DATABASES IS FAST. Professional programmers will find that DATABASES increases productivity by letting them concentrate on the more challenging aspects of their project. DATABASES will quickly generate thousands of lines of complex, bug-free and easily modified source code that would take even professional programmers months of work.

YOU CAN'T BEAT THE PRICE.

DATABASES is a true 4GL, providing more power to the user than dBASE or similar products. It is packaged specifically for ease of design and use. Which would you choose...

<table>
<thead>
<tr>
<th>dBASE (database management)</th>
<th>Genifer (code generator)</th>
<th>Clipper (compiler)</th>
<th>R&amp;R (relational report writer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1,500 PLUS</td>
<td>$1,500 PLUS</td>
<td>$1,500 PLUS</td>
<td></td>
</tr>
</tbody>
</table>

OR

DATABASES + Turbo Pascal V4.0 or Turbo C = LESS THAN $500.

DATABASES TOOLS

A function library to enhance the power of DATABASES. An ideal two-in-one Database Toolbox package for users of Turbo Pascal and Turbo C. An integrated, intelligent, high level interface to DOS for managing Files and Console input and output. Available as an independent package for $99. An invaluable adjunct to DATABASES Application Generator for the more sophisticated programmer.
DATABOSS COMPONENTS INCLUDE THE FOLLOWING:

MENU GENERATOR:
- Unlimited menu nesting
- Call internal DOS commands and external .EXE, .COM and .BAT files with parameters
- Include your own initialization and exit routines
- Display date and time, copyright notice and menu heading
- Nine security levels and modifiable password file with user

THE MOST POWERFUL
RELATIONAL REPORT
GENERATOR EVER
DEIVED
- Design any type of report
- Automatic structure definition for relational reports
- A report element can be a field, text, function
- Unlimited number of totals and subtotals
- Send report elements to CON, LST, RS232, DSK individually or simultaneously
- Paint and build report range selection screens to select specific data
- Print multiple records across a page

IMPEX QUERY BY
EXAMPLE MODULE
- Import external ASCII files into your DATABOSS database definitions
- Query datafiles using point and select cursor movements
- Select fields to be output and specify order
- Impose conditions for data selection

FIELD DEFINITION:
- Each field defined via a 4GL template
- Up to 16 related datafiles per application module
- 16 index keys per datafile unique or duplicate
- Up to 9 segments per index key
- Allows multiple use of fields in key segments
- Automatic datafile linking
- Dynamic traceback of linked datafiles
- Unlimited number of open files
- Character input control via pictures
- Any field default value allowed
- Full field validation via BOOLEAN check either expression or function
- User defined error messages
- Compute and key expressions
- Automatically generated re-indexing module
- Automatically generated datafile reconfiguration module

SCREEN PAINTER:
- Free form full screen editor
- Draw lines and boxes — full IBM extended character set displayed choice
- Copy, move, insert, center text etc.
- Color painting, foreground, intensity and background colors

PROGRAMMERS CAN
CUSTOMIZE AND MAKE
APPLICATIONS
MORE POWERFUL
- Write your own functions, initialization and exit routines and include them in the function table
- Customize a skeleton file and use this file at generation time

GENERATE AND
COMPILE USING
TURBO PASCAL
V4.0 OR TURBO C
- Generate 1000 lines of code in 10 seconds
- Compile to produce fast, executable object code
- No runtime licence fees
- We provide you with end user screen and printer installation modules to include in your menus
- Provide the IMPEX query and import program to your end users on your menu

Name: ________________________________
Company: ____________________________
Street Address: _______________________
City: __________________ State: __________ Zip: __________

Charge to my ________________ Account Name ________________
Account No. ________________ Expiry Date: ________________

☐ Yes! I've seen enough! You CAN'T BEAT THE BOSS!
   Please send me DATABOSS APPLICATION GENERATOR at $399 ☐ For Turbo Pascal V4.0, or ☐ For Turbo C.
   Name: ________________________________
   Company: ____________________________
   Street Address: _______________________
   City: __________________ State: __________ Zip: __________
   Account No. ________________ Expiry Date: ________________
   Charge to my ________________ Account Name ________________
   Signature: __________________________

Yes! I've seen enough! You CAN'T BEAT THE BOSS!
   Please send me DATABOSS TOOLS at $99.

☐ Yes! I've seen enough! You CAN'T BEAT THE BOSS!
   Please send me your CAN'T BEAT THE BOSS booklet.

☐ Yes! I want to know more about revolutionary, new
   DATABOSS! Please send me your CAN'T BEAT THE BOSS booklet.

Yes! I've seen enough! You CAN'T BEAT THE BOSS!
   Please send me DATABOSS TOOLS at $99.

☐ Check enclosed. Charge to my ________________ Account Name ________________
   Account No. ________________ Expiry Date: ________________
   Signature: __________________________
THE EVOLUTION OF A SYSTEMS LANGUAGE

Falling as they do in the shadow of Integrated Debugging, Turbo Pascal 5.0's other enhancements run the danger of being overlooked. This would be a mistake—5.0 would be a major upgrade even without its debugging power.

Perhaps most important—overlays are back. Bruce Webster covers the new unit-based overlay system on page 38; it's much smarter and faster than the scheme in Turbo Pascal 3.0, especially if EMS memory is present in your system.

EMS support has another wrinkle: The editor buffer is now placed in EMS memory if EMS memory is detected at runtime. This step frees up to 64K of DOS memory for the Integrated Environment and for your application.

Apart from overlays, units have been enhanced by permitting them to have private USES statements in their IMPLEMENTATION sections, thus allowing circular references among units to be resolved cleanly. The DOS unit contains new routines for parsing and reading the DOS environment, for reading and changing the state of the DOS verify flag, and for reading and changing the state of Ctrl-Break checking in DOS. ParamStr(0) returns the DOS Exec path.

Neil Rubenking explores another 5.0 enhancement, procedural types, in "A Directory Search Engine in Turbo Pascal" on page 27 of this issue.

Turbo Pascal 5.0 now aligns data items in the data segment and on the stack on machine word boundaries. (The heap is not affected.) This allows the CPU to fetch data from memory as much as 20 percent faster than before. A new compiler directive, {$A+}, has been provided to enable or disable this feature, which may affect assembly language routines that make assumptions about data offsets from BP in the subprogram stack frame.

FLOATING POINT EMULATION

4.0 supported several IEEE numeric types: Single, Double, Extended, and Comp. However, these types were supported only on machines that contain an 87-family numeric coprocessor. Turbo Pascal 5.0 now emulates the math coprocessor when it's run on machines that don't have a math coprocessor, by using the same system described by Roger Schlafly in "Floating Point in Turbo C," TURBO TECHNIX, January/February, 1988. In brief, when an .EXE file generated by Turbo Pascal 5.0 is run, the file tests for the presence of an 87, and then either uses the coprocessor directly (for the fastest possible floating point support), or else emulates the coprocessor at the cost of some performance.

CONSTANT EXPRESSIONS

In all previous versions of Turbo Pascal, a named constant could be defined only by equating it to some literal value. Defining a constant in terms of expressions that incorporate arithmetic operators and previously defined constants is standard procedure in many languages, including assembler and C. Turbo Pascal 5.0 now allows constant expressions that contain previously defined constants, most arithmetical, logical, bitwise and set operators, and a limited number of standard functions including Size-Of, Length, Abs, Chr, Ord, Succ, Pred, Length, Hi, Lo, and Swap.

The most important use of constant expressions is to create a "ripple down" effect that changes the values of many constants, based upon a single constant defined earlier in the program. A good example involves the many "magic numbers" sent out to UART control registers in telecommunications applications. These numbers differ depending upon which serial port is to be used. A set of constant expressions based upon a port number allows the source code to be altered for a new serial port simply by changing a single constant definition:

```
COMPORT = 1; (1=COM1; 2=COM2);
COMBASE = &H8F;
PORTBASE = COMBASE OR (COMPORT SHL 8);
THR = PORTBASE;
RBR = PORTBASE;
!ER = PORTBASE + 1;
!IR = PORTBASE + 2;
LCR = PORTBASE + 3;
```

Here, all you have to do to change to serial port COM2 is redefine the constant COMPORT to 2, and the change propagates through the rest of the constants automatically.

BGI ENHANCEMENTS

The Borland Graphics Interface has been considerably enhanced for Turbo Pascal 5.0 with the addition of several new drivers and many new procedures and predefined constants. The IBM 8514 is now supported in its 640 X 480 and 1024 X 768 modes, and the VGA driver suite includes support for the 320 X 200 X 256 color mode. The 8514 is fully supported by all BGI features (except that FloodFill does not work on 8514 graphics). Also, a new routine, SetRGBPalette,
performs palette management for all 256-color modes on the 8514 and the VGA; the earlier BGI palette routines don't work in 256-color mode. Another procedure, SetRGBColor, performs color management for 256-color modes.

A Sector procedure has been added to draw elliptical or circular segments that may be filled using the scan converter. A separate new routine, FillEllipse, draws full ellipses that are automatically filled with the current fill color and fill style.

New mechanisms enable the registration of BGI fonts and drivers provided by non-Borland sources. InstallUserDriver installs a third-party graphics driver into the BGI driver table. InstallUserFont performs the same function for third-party fonts. Other new BGI procedures and functions include GetMaxMode, which returns the maximum mode number for the loaded driver; GetModeName, which returns the name of a mode given its number; SetAspectRatio, which allows fine-tuning of X/Y ratios to correct for misaligned display screens; SetWriteMode, which specifies the binary operation (XOR or MOV) used in drawing straight lines; and SetUserCharSize, which allows the width and height of stroked fonts to be varied.

New predefined constants include CurrentDriver, for calls that require a driver ID number.

TURBO DEBUGGER SUPPORT
Turbo Pascal 5.0 fully supports Turbo Debugger for standalone symbolic debugging. In contrast to 5.0's Integrated Debugger, Turbo Debugger lets you follow the effects of your program through all levels of the underlying system including memory, stack, and machine registers. All of the features described by Michael Abrash in "Turbo Debugger: The View From Within" (p. 52 of this issue) may be used with Turbo Pascal 5.0 just as easily as with Turbo C 2.0.

Figure 1 shows a Turbo Debugger screen as it might appear while single-stepping a small program. The source code file is displayed in a module viewer window, while the generated machine code for each Turbo Pascal statement is shown with associated assembly language mnemonics in the CPU viewer window. The state of all machine registers and flags is updated after each statement is executed. A variable viewer window contains all variables visible in the current scope; any of these variables may be chosen for closer examination.

ALL SYSTEMS GO
With every new release since 1983, Turbo Pascal has moved more and more toward a true systems-implementation language. I now consider it to be the functional equivalent of C—no part of the PC system is beyond its grasp. Turbo Pascal still puts the much-maligned safety railing between you and the cliff edge, but if you really want to walk over that cliff, it'll gently help you past the railing—and then say ... g'day.

—Jeff Duntemann
TURBO PASCAL 5.0

continued from page 16

referenced from within SearchEngine. A watch was set—but on the wrong item. There's a lesson here: Keep things like scoping in mind while you debug, especially while you're learning the Integrated Debugger, and doubly especially if you're just learning to program.

At this level in the program, the file spec is held in a variable named Mask. A watch could be set on Mask, but the horse could already be out of the barn.

One way to check is to bring up the evaluation box and look at the current value of Mask. Place the cursor on Mask, press Ctrl-F4, and Enter. The evaluation box appears with Mask in the Evaluate field, and the current contents of Mask appear in the box's Result field (see Figure 3).

Aha! Look closely at the file spec: "C: \ENGINE*". The second asterisk is gone. As a result, DOS thinks that this file spec requires files that don't have any file extension at all. Nothing in the directory matches this file spec.

Don't get too excited just yet. This is the bug's spoor; the bug itself is still nowhere in sight. But where to look now? Sadly, execution can't be "backed up" a step at a time the way it can move forward. The wise thing to do here is to reset the program to its initial state by selecting the Run/Program reset item, and then start again. This time, set a watch on the right item and begin to single-step a little earlier.

Before we do so, however, let's use another feature of the Integrated Debugger, and take a quick look at the call stack. Select Debug/Call stack, or use its shortcut, Ctrl-F3. A box appears that contains a summary of the current state of subprogram nesting, in

continued on page 26
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™
Instant Assistant™
Screen Genie™
Word Genie™
NoBlink Accelerator™
Assembler Genie™
DOS Assistant™
Programming Libraries
Supports Turbo Pascal 4.0

HardRunner™
... and more

Nostradamus®

---

"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
UNIT Engine;

(***)

{SEARCH ENGINE}

(* Input Parameters: *)

(* Mask : The file specification to search for *)

(* May contain wildcards *)

(* Attr : File attribute to search for *)

(* Proc : Procedure to process each found file *)

(* Output Parameters: *)

(* ErrorCode : Contains the final error code. *)

*****************************************************************************

(**) INTERFACE (**)

*****************************************************************************

USES DOS;

TYPE ProcType = PROCEDURE (VAR s : SearchRec);

PROCEDURE SearchEngine(Mask : PathStr;
Attr Byte;
Proc : ProcType;
VAR ErrorCode : Byte);

FUNCTION GoodDirectory(S : SearchRec) : Boolean;

PROCEDURE ShrinkPath(VAR path PathStr);

PROCEDURE ErrorMessage(ErrCode Byte);

PROCEDURE SearchEngineAll(path PathStr;
Mask NameStr;
Attr Byte;
Proc ProcType;
VAR ErrorCode : Byte);

{procedure FSplit(Path: PathStr; var Dir: DirStr; var Name : NameStr; var Ext: ExtStr);}
If you think writing program code is a dirty business, we have something to help you clean up your act.

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

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

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

The full Layout package also comes with three additional programs:

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

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

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

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

**Video Tape Offer**

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

Do it today. Because once you see what Layout can do for you, we think you'll swear by it.
Turbo Pascal 5.0

Continued from page 22

including the actual parameters passed to each currently active subprogram. Examine Figure 4.

The call stack display shows that the program started at Where, called SearchEngineAll, and then called SearchEngine. But look closely at the actual parameters: The file spec was passed correctly to SearchEngineAll, but was already corrupted by the time it was passed to SearchEngine.

We assumed too much when we first descended into SearchEngineAll. Something in that very short and uncomplicated procedure corrupted the spec. Reset the program, reload WHERE.PAS using the pick list, and start the program running again. The breakpoints are still there, and execution pauses at the call to SearchEngineAll. Trace into SearchEngineAll with a single press of F7, and take a more careful look around.

What does the program do to Mask between the call to SearchEngineAll and the call to SearchEngine? Nothing! The same parameter, Mask, is passed through untouched. The call stack showed that the spec was passed intact down into SearchEngineAll. Look at Mask again by bringing up the evaluation box once more.

Elementary, My Dear Pascal

Surprise! Mask is corrupted already, to "ENGINE\*.*". If we had looked at Mask immediately upon entering SearchEngineAll, we could have avoided the trip onward into SearchEngine. However, we’d still be confronted by a mystery: The string "ENGINE\*.*" is passed to SearchEngineAll, and the string "ENGINE\*.*" comes out the other side. That’s a subtle point, but it should suggest something to you. Let’s look at the types of the formal and actual parameters here.

The actual parameter, template, is type STRING, which is 255 characters long. However, the formal parameter to which template is passed is type NameStr, which is a type defined within the DOS unit. If you look in the documentation for the DOS unit, you’ll find that NameStr is defined as STRING[8], and therefore is only eight characters long.

The string "ENGINE\*.*" is nine characters long. Mask literally isn’t long enough to hold this string. Since strict type-checking for strings is disabled through the {$V-} compiler directive at the start of Where, the final asterisk is truncated off into oblivion with no one the wiser.

The bug is that Mask is declared to be an inappropriate type. The solution is fairly simple: Declare a new type in the Engine unit that is large enough to hold any file spec that doesn’t also include a path:

```pascal
TYPE
  FullNameStr = STRING[12];
```

Next, redeclare all parameters or variables that must contain a file spec as type FullNameStr. Now recompile and test it out.

It works. The bug is dead. Long live Turbo Pascal 5.0!

Sight, Foresight, and Hindsight

Hindsight is always perfect, and also perfectly useless. Sure, this was an easy bug to spot—but only because we had the power to lift the hood and take a look. Logical deduction almost never works on bugs like this, because we rarely remember to think of the matching of formal and actual parameters as a real program action and not simply a formality. Sooner or later you’d spot it, but you’d probably waste half an hour in the process. I’ve wasted far more time on far wimpier bugs, simply because my mind gets locked into a set of assumptions that logic alone just won’t crack.

Debugging is a skill that takes some practice to develop. It requires that you study your chosen language and your machine. It requires that you keep an eye on your assumptions, especially the deadly one that insists that “nothing really happens between here and there.” Remember that Turbo Pascal 5.0 still requires that you learn how to look—but now, at least, it lets you see.

Listings may be downloaded from Library 1 of CompuServe forum BPROGA, as PASBUG.ARC.

---

Printed graphics in ultra-high resolution using Turbo Pascal 4.0? Introducing . . .

**GRAPHLINK™**

The powerful printer software that gives you the same control over your printer that Turbo’s BGI graphics gives you over your screen—in ultra-high resolution!

**GRAPHLINK** commands work exactly like Turbo Pascal’s BGI commands wherever possible, so you can quickly add printed graphics to your programs. No need to learn a whole new syntax—you already know the commands!

**GRAPHLINK** emulates every applicable BGI procedure and function, including viewport and image-transfer routines.

**GRAPHLINK** dynamically compresses images in conventional memory, so you can store an 8" x 10", 150 dpi image in as little as 150 kB. Optionally uses expanded memory!

**GRAPHLINK** supports printers to their highest resolution:
- HP LaserJet II to 300 dpi
- Epson LQ series to 180 dpi
- NEC and Toshiba 24 pin printers to 360 dpi!

Only $69 + $5 s/h (PA residents add 6% to total.)

Requires Turbo Pascal 4.0. Minimum 512 kB of memory recommended. Satisfaction guaranteed or your money back within 30 days.

**VISITECH SOFTWARE**

DS 3807 Ridgewood Court
Pittsburgh, PA 15239
412/733-4775

---

26 TURBO TECHNIX September/October 1988
A DIRECTORY SEARCH ENGINE IN TURBO PASCAL

Turbo Pascal 5.0 allows you to build completely generalized routines by supporting the passing of procedures—and hence program actions—as parameters.

Neil Rubenking

Whatever high-level language you use, eventually some of your programs will need the ability to search a disk directory. The machinery to do so is built into DOS, and Turbo Pascal 5.0's DOS unit provides FindFirst and FindNext directory search procedures that use DOS's directory search functions. To support FindFirst and FindNext, the DOS unit provides the SearchRec data type that models the DOS disk transfer area (DTA) as a Pascal record. Unit DOS also contains built-in constants for each of the DOS file attributes. These elements can be combined into a completely generalized file search "engine," placed into a unit, and then used for any purpose by any program that needs to search a directory or a directory tree.

Engine (Listing 1) is the unit that contains the generalized file search engine. Engine contains two major procedures, SearchEngine and SearchEngineAll. SearchEngine searches a single directory for a file that matches a file specification and a file attribute byte. SearchEngineAll traverses an entire directory tree or subtree during its search. Since the compiler and the DOS unit handle so much of the file search activity, the search engine unit can be quite compact. (For information about the theory behind DOS directory searching, see "A Directory Search Engine in Turbo C" on p. 75 of this issue. In this article, I cover the practical implementation of DOS directory searches in Turbo Pascal 5.0.)

PROCEDURES AS PARAMETERS?

Procedure SearchEngine takes four parameters. This procedure needs to know which file specification to seek, which attribute to match, and which procedure to call on every found file. SearchEngine then returns the final DOS error code returned by the DOS Find First and Find Next functions.

If you're alert, you're probably wondering how the SearchEngine procedure could have a procedure as a parameter. Procedural types (and function types) are a new feature of Turbo Pascal 5.0. Conceptually, procedural types allow you to think of program state-
SEARCH ENGINE
continued from page 27

Declaration of procedural type P:
TYPE
  P = Procedure(X : Integer; Ch : Char);
A valid procedure of type P:
PROCEDURE Manny(1 : Integer; MyChar : Char);
Invalid procedures for type P:
W is the wrong type:
PROCEDURE Moe(w : Word; Letter : Char);
X formal parm not VAR:
PROCEDURE Jack(VAR 1 : Integer; Ch : Char);
Wrong number of parms:
PROCEDURE Ray(NewCh : Char; K : Integer);
Wrong order of parms:
PROCEDURE Bob(MyGrade : Char);

Figure 1. Valid and invalid procedures for procedural type P.  Note that the names of the procedures and their parameters do not matter. The type and order of the declaration of parameters, and whether a parameter is passed by reference (VAR) or by value, are the only things that matter.

SearchEngine takes the parameter Proc, of type ProcType. Each of the example programs contains one or more procedures of this type. Within SearchEngine, a call to formal parameter Proc has exactly the same effect as a call to the procedure passed in Proc as the actual parameter.

THE ENGINE

ENGINE.PAS (Listing 1) contains the directory search unit. SearchEngine uses DOS unit procedures FindFirst and FindNext to find all matching files. Each time SearchEngine finds a matching file, it calls the user-specified procedure passed in procedural parameter Proc. Simple! SearchEngine also returns the final DOS error code. However, if SearchEngine finds at least one file during a search, it doesn’t consider not finding additional files to be an error.

Procedure SearchEngineAll searches the given path and all of its subdirectories for files that match the file specification. Passing a path that specifies a volume’s root directory, such as “C:\”, to SearchEngineAll tells the procedure to search the entire volume. SearchEngineAll uses the file specification, attribute, and user-specified procedure to call SearchEngine in order to find and process all matching files in a given directory. SearchEngineAll then calls SearchEngine a second time. This time, however, SearchEngineAll searches for subdirectories by specifying the directory attribute bit for the search. SearchEngineAll then uses procedure SearchOneDir, which is passed as a procedural parameter of type Proc, to process the subdirectories that have been found.

Like SearchEngineAll, SearchOneDir makes two calls to the regular SearchEngine—one call matches files, and the other call searches for more directories. Hence, SearchOneDir and SearchEngine form a recursive loop. At each level of nesting, SearchOneDir uses SearchEngine to look for any subdirectories. If SearchEngine finds any subdirectories, it calls SearchOneDir again. This process continues until all of the subdirectories located beneath the initial path passed to SearchEngineAll have been processed.

(For a discussion of recursion, see “Recursing without Cursing,” TURBO TECHNIX, July/August, 1988.)

Other handy routines for directory searches are included in ENGINE.PAS. Function GoodDirectory returns True only if its SearchRec parameter refers to a file that has the directory attribute and is neither the current directory nor the parent directory (i.e., neither “.” nor “..”). ShrinkPath removes the last subdirectory from a path, using Turbo Pascal 5.0’s new PSplit procedure. Procedure ErrorMessage then prints a message that’s appropriate to the DOS error code passed to this procedure. These other routines are used in the example programs.

INSTANT DISK UTILITIES

The various routines in Engine let you write useful DOS disk utilities with very little additional code. DirSum (Listing 2) shows just how tiny a program that uses the Engine unit can be. Small enough to fit on one 25-line screen, DirSum manages both to display the names of all of the files in the current directory and to tally their sizes into one total size value. How can DirSum be so small? Because all of the work happens elsewhere. DirSum passes procedure WriteIt to SearchEngine, which causes the engine to write the name of every file that it finds. When DirSum has displayed the names of all of the files in the current directory, it then displays the total number of bytes of disk space that these files occupy. That’s awfully easy, though. Let’s give the search engine more of a challenge.

WHERE.PAS (Listing 3) contains Where, a program to find files that match a file specification located anywhere on your disk. With SearchEngineAll, a task like this is almost ridiculously simple. Simply pass the path, file template, and file attribute to SearchEngineAll, along with the procedure for processing each found file. In this case, procedure ShowFile displays the full pathname of each found file and—as a bonus—updates a tally (as does DirSum) of how much disk space the found files occupy. ShowFile uses standard output for its screen displays (note that the Crt unit is not named in the USES statement). A handy disk file of found files can be created by redirecting Where’s output to a file. For example, the invocation WHERE *.txt > ALLLFILES.DIR creates a file named ALLLFILES.DIR that lists the name of every file located anywhere on the current volume.

DELBK.PAS (Listing 4) contains program DelBak. DelBak performs a useful housecleaning task—it deletes all .BAK files on the current volume. If you haven’t purged your .BAK file collection in a while, you may find that these files occupy tens or even hundreds of thousands of bytes of hard disk space.

DelBak is similar in structure to Where. Again, SearchEngineAll does all of the work. The file specification is fixed as “*.BAK.” The action procedure passed to SearchEngineAll is DelFile, which simply deletes the found file and notes how many bytes were saved.

story continues on page 36
listing begins on page 34
The revolution continues...
What started modestly enough in November of 1983 with the launch of our first program, Turbo Pascal®, became a revolution and it has been going like a rocket ever since.

We've changed the way you program. We invented integrated environments with Turbo Pascal and we brought that to all our languages—to make you instantly at ease with our languages. (No one else has even tried to do that for you.) Read these pages. You'll see that the revolution continues.

New! Turbo Assembler/Debugger

It's Assembler magic and a revolution in source-level Debugging.

New Turbo Debugger debugs all sizes

Nothing is too big or too small, too simple or too complicated. Nothing. With EMS support, remote debugging, and 386 virtual machine debugging, there's no limit to the size of program you can debug. In fact with 386 virtual machine mode, debugging takes zero, zip, nil, no bytes of conventional memory.

See what's happening

Multiple overlapping windows let you look at code and data and work at any level—down to CPU or up to source level. You can see it all with multiple views of the program you're debugging: source code, variables, CPU registers, call stack, watches, breakpoints, memory dump, and more. And a new "session-logging" feature tracks and records your every move.

We've brought "what if" to Debugging!

Our breakpoints give you more control than anyone else's. Ordinary debuggers only get you to a stop, then they stop. With ours you control When they happen and What happens next. When our breakpoints are triggered you can simply stop, or you can print expressions, run code, send messages to the session log, or even evaluate an expression with user-defined function calls. You can control when these breakpoints occur because all our breakpoints are conditional. In plain and simple terms we've brought "what if" to debugging.

Unique Data Debugging features

Plain Vanilla debuggers can only give you code debugging. Our new Turbo Debugger gives you data debugging too. Now it's easy to find the data you want. You can browse through your data from the simplest byte to the hairiest data structure, inspect arrays, and walk through linked lists. All by point and shoot. And once you've found the data you want, you can get all the information you want about it, and you can change it.

Feature highlights

Multiple overlapping views
- Source
- Watches
- Variables
- Breakpoints
- Call Stack
- CPU
- Registers
- Numeric Processor
Debugger, Assembler, &...  

Shown here is Turbo Debugger in action.

- Memory Dumps
- Session Log
- Files
- User Screen

Breakpoints
- Can perform these actions: Stop, Print Expression, Run Code, Log Expression
- Can break on arbitrary condition, memory changed, pass count; attach to specific line of code, or apply continuously
- Includes breakpoints, tracepoints, watchpoints, and conditional breakpoints

Debug Any Program
- Turbo Pascal, Turbo C, and Turbo Assembler
- EMS support
- 386 virtual machine debugging
- Remote machine debugging
- Supports CodeView® compatible executables

Data Debugger
- Follow pointers through linked lists
- Browse through arrays and data structures
- Variables view lets you see all defined data
- Displays type and value information
- Change data values

Minimum system requirements: For the IBM PS/2 and the IBM family of personal computers and all 100% compatibles. PC DOS (MS DOS) 2.0 or later. 384K minimum.

New Turbo Assembler® lets you write the tightest, fastest code

Turbo Assembler is faster than other assemblers: not just by a little, but by factors. You can use it on your existing code; it's fully MASM compatible, 4.0, 5.0, and 5.1.

You choose the level of compatibility—even MASM can't do that. Turbo Assembler takes you beyond MASM, with significant new Assembly language extensions, more complete error checking, and full 386 support. Turbo Assembler is designed for easy interfacing with high-level languages like Turbo Pascal and Turbo C. (We use Turbo Assembler on Quattro®, our best-selling spreadsheet program; now you can write your own best-seller with Turbo Assembler!)

Turbo Assembler and Turbo Debugger are two of our secret weapons, now they can be yours.

Feature highlights
- Factors faster than other assemblers
- Full MASM (4.0, 5.0, and 5.1) compatibility
- Significant new assembly language extensions
- Easy interfacing with high-level languages including Turbo C and Turbo Pascal

Turbo Assembler/Debugger: only $149.95

For the IBM PS/2 and the IBM family of personal computers and all 100% compatibles. PC DOS (MS DOS) 2.0 or later. 256K minimum.

*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.

† Run on an IBM PS/2 Model 60.

Prices and specifications subject to change without notice.

All Borland products are trademarks or registered trademarks of Borland International. Other brand and product names are trademarks or registered trademarks of their respective holders. Copyright (c) 1988 Borland International, Inc. Bl 1279
New! Turbo C® 2.0
With integrated source-level debugger

Borland’s revolutionary new Turbo C 2.0 is the one C compiler that does it all; nothing is half done or not done at all—instead, your every programming need is met. (We wrote our best-selling word processor Sprint with Turbo C 2.0; when you write with Turbo C 2.0, the word is “revolutionary.”)

At better than 16,000 lines a minute!, Turbo C 2.0 compiles your code 20-30% faster than its predecessor Turbo C 1.5 which was already faster than any other C compiler.

Make bugs bug off

Nice bugs are dead bugs, and Turbo C 2.0’s integrated source-level debugger lets you find them and flatten them in a flash. You can set multiple breakpoints, watch variables and evaluate expressions—all from inside your integrated C environment.

Turbo C 2.0 has the best of everything

- Includes the compiler, editor, and debugger, all rolled into one
- Integrated source-level debugger lets you step code, watch variables, and set breakpoints
- Develop and debug production-quality code in all six memory models
- Support for Turbo Assembler and Turbo Debugger
- Make facility with automatic dependency checking
- Graphics library with over 70 graphics functions including multi-font graphics text

- Faster than ever; compiles and links 20-30% faster than Turbo C 1.5
- EMS support
- Numerous levels of error checking with built-in Lint

Turbo C 2.0: only $149.95

Minimum system requirements: For the IBM PS/2® and IBM family of personal computers and all 100% compatibles. PC-DOS (MS-DOS) 2.0 or later. 448K minimum (320K for the command-line version).

Turbo C Professional

Turbo C 2.0 plus both Turbo Assembler & Turbo Debugger: all three programs rolled into one—the one C package that has everything. A complete set of tools that caters to every level of programming expertise.

Turbo C Professional: $250

New! Turbo Pascal® 5.0 with integrated source-level debugger

Turbo Pascal, the worldwide favorite with over a million copies out there, just got even smarter. The best got better. Meet Version 5.0. In a word, it’s revolutionary.

Not only do you go code-racing at more than 27,000 lines a minute!, you also now go into a sophisticated debugging environment—right at source-level. It’s completely integrated and bullet-fast.

Turbo Pascal’s new integrated debugger takes you inside your code for fast fixes. You step,
...new Turbo Pascal 5.0!

![Image of Turbo Pascal 5.0 features]

Shown here is the Evaluate/Modify window of Turbo Pascal: look at expressions, examine structured data types, change variables on the fly.

trace, set multiple breakpoints. You modify variables—as you debug—and watch full expressions at run time.

**Orbit with Units**

Break your code into Units. Compiled units make everything go faster! Your separately compiled Units can be shared by multiple programs and linked in a flash with Turbo Pascal's built-in Make utility and smart Linker. (We give you a powerful library of standard Units including the spectacular Borland Graphic Interface and our state-of-the-art overlay manager.)

**Debugging: The inside story**

Turbo Pascal’s new integrated source-level debugger takes you inside your code to fix errors fast. (Don’t worry about errors, everyone makes them; but with the right debugger, this one, it’s a fast fix.)

**Feature highlights**

- Includes the compiler, editor, and debugger, all rolled into one
- Integrated source-level debugger lets you step code, watch variables, and set breakpoints
- Support for Turbo Assembler & Turbo Debugger
- Overlays, including EMS support
- IEEE standard floating point emulation
- Smaller, tighter programs: Smart Linker strips both unused code and data
- Procedural types, variables, and parameters
- EMS support for editor

**Turbo Pascal 5.0**: only $149.95

Minimum system requirements: For the IBM PS/2* and IBM family of personal computers and all 100% compatibles. PC-DOS (MS-DOS) 2.0 or later. 448K minimum (256K for the command-line version).

**Turbo Pascal Professional**

Turbo Pascal 5.0 plus both Turbo Assembler & Turbo Debugger: all three programs rolled into one—the one Pascal package that has everything. A complete set of tools that caters to every level of programming expertise.

Turbo Pascal Professional: $250

As you can see from all these brand new programs, the revolution is alive and well. Borland continues to bring you the best

For the dealer nearest you, call (800) 543-7543.

BORLAND
UNIT Engine;
{SV-}

(* SEARCH ENGINE *)

(* Input Parameters: *)

(* Mask: The file specification to search for *)

(* Attr: File attribute to search for *)

(* Proc: Procedure to process each found file *)

(* Output Parameters: *)

(* ErrorCode: Contains the final error code. *)

(*----------------------------------------------------------------------*)

(** INTERFACE **)
Sophisticated User Interfaces in Minutes!

Put magic in your programs with turboMAGIC

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.
TWO LITTLE ENGINES

Keep the compiled ENGINE.TPU handy; you’ll find yourself thinking of new ways to use it all the time. When you need to search for files within a single directory, use SearchEngine. For searches that traverse a tree or a subtree, use SearchEngineAll.

Unit Engine illustrates an important principle of software development: The more general a tool, the more different problems it solves, and the more time that it saves you. Turbo Pascal 5.0’s procedural types make possible the creation of truly general tools whose tasks can be specified at runtime. Other “engines” suggest themselves, such as a graphics function plot engine that receives a function to plot via a function parameter, or a general-purpose sort unit that takes a function that specifies which of two data items is considered greater than the other on some sort sequence. Once you start thinking of program statements as just another kind of data, these kinds of solutions will seem the natural way to do business in a system-level language such as Turbo Pascal 5.0.

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

Listings may be downloaded from Library 1 of CompuServe forum BPROGA, as PASENG.ARC.

**The cost involved, in writing one of these geometric routines, is more than the price of the TurboGeometry Library.**

```
BEGIN
PROCEDURE Validate;
(Validate the command line parameter)
VAR P : Byte;
Ext : Extstr;
BEGIN
IF ParamCount <> 1 THEN
BEGIN
WriteLn('SYNTAX: "HERE [path]filespec"');
Exit;
END;
FSplit(ParamStr(1), path, template, Ext);     (*IF no path specified, search from root of current volume*)
IF path = '/' THEN
BEGIN
GetDir(0, path);
END;
END;
BEGIN
Total : = 0;
Validate;
WriteLn
('Searching for files below path"');
SearchEngineAll(path, template, Archive, ShowFile, ErrCode);
WriteLn
('These files occupy ',Total : 8,' bytes of disk space.');
END.
```

```
PROGRAM DelBak;
(* ********************************************************
(* Uses SearchEngine to find and delete all *.BAK files *)
(* in any subdirectory in the current volume. *)
(* ********************************************************
USES DOS,Engine;
VAR
path : PathStr;
ErrCode : Byte;
Number : Integer;
Size : LongInt;
BEGIN
PROCEDURE DelFile(VAR S : SearchRec; path : PathStr);
VAR F : FILE;
BEGIN
inc(Size, S.Size);
Assign(F, path + S.name);
Erase(F);
inc(Number);
END;
PROCEDURE Initialize;
BEGIN
Number : = 0;
Size : = 0;
GetDir(0, path);
IF Length(path) = 2 THEN path := path + '/';
ELSE path[0] := #5;
WriteLn('Going to delete ALL *.BAK files in the current volume.');
ReadLn;
END;
BEGIN
Initialize;
SearchEngineAll(path, '*.bak', AnyFile, Delfile, ErrCode);
WriteLn
('Erased ',Number,' *.BAK files for a saving of ',Size,' bytes.');
END.
```

TurboGeometry Library

Are you programming or planning to program CAD/CAM or graphics applications? Many hours, even days, can be spent in writing and debugging geometric routines. TurboGeometry Library can relieve you of those time consuming tasks that are part and parcel of every CAD/CAM or graphics program. There are over 150 routines in the library, supported by example programs and a 400 page manual. The source code is included. 30 day guarantee. Need IBMPC or Compatible, Turbo Pascal 4.0, Turbo C, or MS C. $149.95 plus $5.00 S&H in US. VISA, MasterCard, Check, PO, MO. No COD’s Send for additional information or call 214-423-7288.

Disk Software, Inc., 2116 E. Arapaho #487 Richardson, Texas USA 75081

“In CAD/CAM or graphics, it all comes down to using geometry”
The most important part of your program is the user interface... getting that face right used to be the hardest part of application development. Not any more.

Introducing Facelt™. Creating menus has never been easier with this new state-of-the-art user interface manager. Facelt's totally different approach to menu creation gives you perfect faces every time. You simply supply the data and Facelt does the rest. It creates the face you need—pop-ups, pull-downs, horizontal menus, help windows, dialog boxes or multiple column menus, automatically, based on the data you provide. And don't worry if your files contain lots of text, Facelt has built-in virtual windowing and scrolling capabilities.

How Facelt Works.
1. Define the contents of your menus using any editor or import the data directly from a .DBF file.
2. Then Facelt, using this data, designs the interface you want.
3. Use the interface as is. Or, go into the interactive mode to change it on the screen. Take total control over menu customization. Change window shapes, border styles and color every element of the menu right down to the individual menu item.
4. You're done. The Facelt faces are ready to be called directly from your program.

Laying Out Different Mening Systems Now Takes Minutes. Using a new technology called "dynamic menuing," you get a complete menuing system in minutes. Facelt automatically draws the boxes, puts the data into the menus, links them together, positions them, handles all cursor control, saves and restores the screen and provides mouse support. Create menus by specifying any or all records or line numbers. Plus, because all Facelt faces are live menus and not static, you can customize or totally redesign them right there on the screen. Turn a pull-down into a Lotus® style menu in seconds without any coding or compiling!

Facelt Speaks Your Language. Facelt includes the language specific modules (LSMS) for all Borland and Microsoft languages, dBASE™, FoxBASE+,™ Clipper™, QuickSilver™, DIXI™ and Emerald Bay's Eagle™. LSMS provide the two-way communication between your application and the Facelt engine. Facelt menus are called directly from your programming language so there's no need for wake-up codes or hot keys.

The Perfect Turbo Companion. Make your programs look like the Turbo environment you program in. Facelt has no royalties or runtime fees and is backed by our 30 day unconditional money back guarantee. So whether you program for yourself, your company or other people, Facelt will create the right face and give your application the look it deserves.

FaceIt

and put your best face forward.

Only $99
Call Today 212-787-6633

Black & White International, Inc.
P.O. Box 21108
New York, NY 10129
THE RETURN OF OVERLAYS

Turbo Pascal 5.0 uses disk storage and EMS to run your biggest program in as little memory as possible.

Bruce F. Webster

Overlays vanished in version 4.0 of Turbo Pascal because the internal changes to memory allocation made support of the old 3.0-style overlays impossible. With the advent of Turbo Pascal 5.0, however, a new and much better implementation of overlays has now appeared. In this article, we'll take a close look at how the 5.0-style overlays work, and the ways that you can use them.

THE MEMORY GAME

People who first met Turbo Pascal with version 4.0 may well be asking, "What are overlays anyway, and why would I want to use them?" Let's take the second part of the question first. When a Turbo Pascal program is run, the main program and all of the units that it uses are loaded into memory. The main program and each unit occupy separate code segments that can each be up to 64K in size. All declared variables are created in memory within a single data segment, which also has a 64K size limit. In addition, a program stack is allocated; its size is determined either by the Options/Compiler/Memory Size/Stack command or by the $M compiler directive in the main program. Finally, any remaining memory can be allocated to the heap through $M: this is the location in memory where any dynamic variables (which are created using the New or GetMem procedures) are allocated. Of course, a certain amount of memory is already occupied by DOS, any memory-resident programs you might have already loaded, and (if the programs are running under the Turbo Pascal Integrated Environment) by Turbo Pascal itself.

Although this may seem like a lot to have in memory all at once, most of the time there is memory room to spare. However, you can run out of available memory:

- If your program becomes very large;
- If you need to dynamically allocate large data structures; or
- If you have other programs loaded at the same time.

If your computer doesn't have a lot of memory, your program may not load; if it does load, it may halt prematurely with a memory allocation error.

A solution to this problem is to break your program up into relatively independent chunks, and then load those chunks into memory as they are needed. Once a given chunk is no longer needed, the memory that it formerly occupied can be reused for a different chunk.

This brings us back to the first part of the earlier question, "What are overlays?" Basically, overlays are those "chunks" I've been talking about. More specifically, overlays are separately compiled Turbo Pascal units that are loaded into memory as they are needed, and then removed from memory until they are required again. This process is handled for you in a painless and generally invisible way—you simply tell Turbo Pascal which units are to be used as overlays, and then perform a few other preparations. (For more information about units in general, see "Getting to Know Units," TURBO TECHNIX, November/December, 1987.)

HOW OVERLAYS WORK

When you compile a program that uses overlays, all of the executable code for the overlay units (the units that are designated as overlays) is written to an overlay file rather than to the usual .EXE file. The overlay file has the same filename as the .EXE file, with the extension .OVR instead of .EXE.

At the same time, a unit known as the "overlay manager" is linked into your program. The overlay manager determines which overlay unit or units should be in memory at any given moment, and loads them in from the overlay file as needed.

When a program that uses overlays is run, the main program, the overlay manager, and all nonoverlaid units are loaded into memory where they remain while the program executes. The data segment and the stack are also created and used in the same manner as with a nonoverlaid application.

continued on page 40
Lahey Computer Systems, Inc.
Sets a New FORTRAN Standard!
Introducing the latest addition to our line of PC FORTRAN Language Systems—
Lahey Personal FORTRAN 77 Version 2.0

What You Get When You Purchase Lahey Personal FORTRAN:

Lahey Experience.
We are experts in designing and implementing FORTRAN Language Systems. Lahey has been producing mainframe implementations since 1967 and PC FORTRANs (F77L) since 1984. In fact, F77L was named the "EDITOR'S CHOICE" among PC FORTRANs by PC Magazine. This 20-year span of specialization has been incorporated into the design of our revolutionary Lahey Personal FORTRAN 77.

LAHEY SLASHES COMPILATION TIME.
Compilation times (in seconds) for Whetstone Program (WHETS3H.FOR)

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lahey F77L</td>
<td>11.57</td>
</tr>
<tr>
<td>Microsoft F77L</td>
<td>54.08</td>
</tr>
<tr>
<td>Ryan McFarland F77L</td>
<td>88.95</td>
</tr>
</tbody>
</table>

Customer Support:
Our philosophy is that customer relationships begin, rather than end, at the point of sale. Services include free technical support, electronic bulletin board for fast service and information access, and newsletters to keep you up to date on our latest developments.

Purchasing the Lahey Personal FORTRAN 77 gives you software designed by FORTRAN experts, a feature-loaded product with industry-leading compilation speed, and quality technical support; all for $95.

International Representatives: Australia: Comp. Translations, Tel. (03)5372236 • Canada: Barry Mooney & Assoc., Tel. (902)9935941 • Denmark: Ravenholm Computing, Tel. (02)687249 • England: Gray Matter Ltd., Tel. (0364)93499 • Holland: Lemax Co. B.V. (0296)84210 • Japan: Microsoft Inc., Tel. (03)813622 • Norway: Polysoft A.S. (02)992240 • Switzerland: DST Comp. Services, Tel. (022)989188

MS-DOS & MS FORTRAN are trademarks of Microsoft Corporation.

Lahey is setting the PC FORTRAN Standard.
TO ORDER
1-800-548-4778
(specify disk size)
Lahey Computer Systems, Inc.
P.O. Box 6091
Incline Village, NV 89450
Telephone: (702) 831-2500
TELEX: 910240 1235
FAX: (702) 831-8123

We have a complete line of PC FORTRAN Language Systems. For developing or porting programs there is no substitute for a Lahey.

Lahey Personal.....So much for so little $95
F77L .................."Editor's Choice" PC Magazine $477
F77L-EM/16 ..........Ability to write programs as large as 15 MB $695
F77L-EM/32 ..........New 32-bit—Programs up to 4GB on 80386 $895

CALL FOR MORE INFORMATION
OVERLAYS WITH 5.0
continued from page 38

When a program that uses overlays is run, however, part of the heap is taken away and set aside as the overlay buffer. By default, this buffer is just big enough to hold the largest overlay unit; however, you can specify a larger buffer to improve performance during unit loading. The overlay manager then loads as many units as possible into the overlay buffer.

When a routine in an overlay unit is called, the overlay manager checks if that unit is already in memory. If the unit is not in memory, the overlay manager loads the requested unit from the overlay file into the overlay buffer, and removes other units from the buffer as needed. If the manager has a choice of units to swap out, it’s "smart" enough to remove the unit that was least recently called, based upon the assumption that the other units in the buffer are more likely to be called. This process is performed automatically, without any specific load or unload requests from your program.

The net benefit is that a large program can run in a limited memory space. While the costs are four-fold, they can be minimized by some attention to detail. First, your program may need to be restructured in order to make it feasible to use certain units as overlays. (This step may actually improve your overall program design.) Second, a disk access occurs each time a unit is loaded from disk into memory. These disk accesses can be minimized by either increasing the size of the overlay buffer, or (if your computer has expanded memory) by instructing the overlay manager to load the overlay file (not the overlay buffer) into expanded memory. Third, the overlay scheme requires that far calls be used throughout all procedure call chains that extend into an overlay. It also exacts an additional performance penalty when string literals and set constants are passed as parameters. Fourth, when floating point emulation is used, the interrupt vector "backpatching" scheme is reinitialized each time an overlay is loaded into memory. A small performance overhead occurs when the overlay’s floating point code is executed for the first time.

GET READY TO OVERLAY
Several steps are necessary in order to use overlays.

Units first. The program must first be structured to make overlays possible. Since only complete units can be treated as overlays, all sections that are to be overlaid must be broken out and put into units (if they’re not in units already). Overlaid units should be relatively independent—they should call one another’s routines as little as possible, but preferably, not at all. If one overlay calls routines in another overlay, disk "thrashing" may occur—where a distraught overlay manager loads one overlay and then another in rapid succession—bringing program performance to its knees.

The Overlay Unit. The main program must use the Overlay unit, which is part of the TURBO.TPL library. Overlay contains the overlay manager and provides several routines that allow the program to communicate with the overlay manager. Also, the unit name Overlay must appear in the USES clause before the names of any of the overlaid units. Preferably, Overlay should be the first unit named.

Compiler directives. Each unit that will be used as an overlay unit must be named in its own {$0 <unitname>}, where

Beat the Deadlines and thrill them with Performance!!

Now there is a better, more productive way to create programs that relieves your implementation worries and frees your mind, so conquering your next big project becomes child's play! Whether you program in Turbo Pascal or Turbo C, we've got you covered. Introducing The Developer's Library Series - not just a collection of handy routines like most libraries, but a complete programming environment. Both libraries are compatible, which makes switching from one language to another a snap.

Turbo C or Turbo Pascal

Developer's Libraries

Over 120 routines in each library for development of commercial software. Includes routines for: networks, multi-user file management, menus, utilities, sample applications, and more. Complete with 450 page text from Howard W. Sams Publishing and source code on diskette for IBM PC.

Only $69.95 each

New! The Floppy Librarian

A must for anyone with lots of floppy disks to manage!
Maintains physical locations of floppy disks, lists files stored on any and all disks, tracks file changes and backups and more!

Stop asking, "Where is it?!"

Order Now! Only $29.95 For IBM PC's & Compatibles

Only $29.95

Perpetual Data Systems, Inc.
63 Keystone Ave. Suite 206
Reno, Nevada 89503
(702) 348-8600

continued on page 42
Mainframe Power for your PC!

If you need or are accustomed to the throughput of a 32-bit mini, including any of DEC's VAX series, MicroWay has great news for you. The combination of our NDP compilers and our mW1167 numeric coprocessor gives your 386 PC, VAX speed! If you don't own a 386 PC, we provide a number of economical AT upgrade paths.

Many of our NDP Fortran-386 users are reporting turn around times that are two to six times faster than their VAX. The exact times are a function of the VAX processor being used, the speed of the 386, the number of users being served by the VAX, and the coprocessor being used with the 386. There are currently over 400 developers using our NDP tools to port 32-bit applications. To help the 386/1167 engineering standard emerge, MicroWay is co-marketing several mainframe applications that have been ported by our customers. In addition, this ad in.

Dr. Robert Atwell, a leading defense scientist, calculates that NDP Fortran-386 is currently saving him $12,000 per month in rentals of VAX hardware and software while doubling his productivity!

Fred Ziegler of AspenTech in Cambridge, Mass. reports "I ported 900,000 lines of Fortran source in two weeks without a single problem!" AspenTech's Chemical Modeling System is in use on mainframes worldwide and is probably the largest application to ever run on an Intel processor.

Dr. Jerry Ginsberg of Georgia Tech reports "My problems run a factor of six faster using NDP Fortran-386 on an mW1167 equipped 386/20 than they do on my MicroVAX II."

32-Bit Compilers and Tools

NDP Fortran-386™ and NDP C-386™ Compilers generate globally optimized mainframe quality code and run in 386 protected mode under PharLap extended MS-DOS, UNIX, or XENIX. The memory model employed uses 2 segments, each of which can be up to 4 gigabytes in length. They generate code for the 80287, 80387, or mW1167. Both compilers include high speed EGA graphics extensions written in C that perform BASIC-like screen operations $595 each.

- NDP Fortran-386™ Full implementation of FORTRAN-77 with Berkeley 4,2, VAX/VMS and Fortran-66 extensions.
- NDP C-386™ Full implementation of AT&T's PCC with Microsoft and ANSI extensions.

NDP Package Pricing:
387FastPAK-16: NDP Compiler, PharLap, and 80387-16 Coprocessor... $1299
1167FastPAK-16: NDP Compiler, PharLap, and mW1167-16 Coprocessor... $1695

NDP Windows™ — NDP Windows includes 80 functions that let you create, store, and recall menus and windows. It works with NDP C-396 and drives all the popular graphics adapters. Library... $125, C Source... $250

NDP Plot™ — Calcomp compatible plot package that is callable from NDP Fortran. It includes drivers for the most popular plotters and printers and works with CGA, Hercules, EGA and VGA... $325

NDP/FFT™ — Includes 40 fast running, hand coded algorithms for single and double dimensioned FFTs which take advantage of the 32-bit addressing of the 386 or your hard disk. Callable from NDP Fortran or NDP C with 1167 and 387 support... $250
387FFT for 16-bit compilers... $250

387BASIC™ — A 16-bit Microsoft compatible Basic Compiler that generates the smallest .EXE files and the fastest running numeric code on the market... $249

MicroWay®

80386 Support

Parallel Processing

Monoputer™
The world's most popular Transputer development product runs all MicroWay Transputer software using either a T414 or T800. The T800 processor has built-in numerics and provides performance comparable to an 80386 running at 20 MHz with an mW1167. The new 3L Parallel C and Fortran Compilers makes this an especially attractive porting environment. Can be upgraded to 2 megabytes. Monoputer with T414 (0 MB)... $995 Monoputer with T800 (0 MB)... $1495

Quadputer™
This board for the XT, AT, or 386 can be purchased with 2, 3 or 4 Transputers and 1, 4 or 8 megabytes of memory per Transputer. Two or more Quadputers can be linked together to build networks with mainframe power which use up to 36 Transputers. One customer's real-time financial application has gone from 8 hours on a mainframe to 16 minutes on a system containing five Quadputers. from $3495

Transputer Compilers and Applications
MicroWay and 3L offer Parallel languages for the Monoputer and Quadputer: MicroWay Parallel C... $595 MicroWay Occam2... $495 3L Parallel C... $995 3L Parallel Fortran... $895 jField — A specialty finite element analysis package targeted at Transputer networks, ideally suited to take advantage of the 6 Megalop speed of the Quadputer... $1600

Call (508) 746-7341 for our free catalog!

MicroWay Numeric Coprocessors

mW1167™ — Built at MicroWay using Weitek components and an 80387 socket. mW1167-16... $995 mW1167-20... $1595 mW1167/VGA-20 "SlotSaver"... $1995 8087... $99 8087-2... $154 80287-16... $259 80287-10... $295 80387-16... $475 80387-20... $725 287Turbo-12 (for AT compatibles)... $450 DRAM... CALL

All of our Intel coprocessors include 87Test.

PC and AT Accelerators

MicroWay builds a number of 8086 and 80286-based PC accelerators that are backed up by the best customer support in the industry.

Number Smasher™ (8087 & 512K)... $499 FastCACH-286/9 MHz... $299 FastCACH-286/12 MHz... $399 SuperCACH-286/12 MHz... $499 Intel Inboard™ (PC 1 MB)... $950

Intelligent Serial Controllers

MicroWay's AT4™, AT8™, and AT16™ are the fastest 8016-based intelligent serial controllers on the market. They come with drivers for UNIX, XENIX, and PC MOS.

AT4... $795 AT8... $995 AT16... $1295

32-Bit Applications

COSMOS-M/386 — SRC's finite element package for the 386/16 with an 80387 or mW1167 provides mainframe speed and capacity. Turn around times rival the VAX 8650 and are 6 to 15 times that of an AT. From $995

PSTAT-386 — This mainframe statistics package has been used by government and industry for over 20 years. The full version was ported. Requires 4 to 6 megabytes of memory: $1495

NDP/NAG™ — Features a library of 800 engineering and scientific numerical algorithms. Callable from NDP Fortran... $895

MicroWay is mW1167 boards in stock that run on the Compaq 386/20, IBM PS2/80, Tandy 4000, AT&T 6386, Acer 386/20, Everex Step 386/16(20), H.P. Vector RS16(20) and others. We now have a new board for the Compaq 386/20 which combines an 1167 with VGA support that is register compatible with IBM — the "SlotSaver". It features an extended 800x600 high res mode that is ideal for 386 workstations.

Finally, we still offer the 16-bit software and hardware which made us famous. If you own a PC or AT and are looking for the best 8087/80287 support on the market, call (508) 746-7341 and we'll send you our full catalog.
OVERLAYS
continued from page 40

uniname is the unit's name as it appears in the USES clause.

Also, an {$0+} compiler directive must be placed within each overlaid unit in order to show to the linker that the unit is to be treated as an overlay.

Far calls. The main program and all units should be compiled with the Options/Compile/Force far calls toggle set to On, or with the {$F+} directive present in each file. Far calls must be used with all of the routines that call the routines in overlaid units, with all of the routines that call those routines, and so on, back to the main body of the program. The safest way to enforce this requirement is simply to use far calls throughout the entire program.

The .OVR filename. The main program must tell the overlay manager the filename of the .OVR overlay file by calling Ovr-

{$F+}

program Ship;
uses
Overlay, Graph, MainLib, GameInit,
Navigation, Combat, Repair, Survey;

{$0 GameInit}
{$0 Navigation}
{$0 Combat}
{$0 Repair}
{$0 Survey}

var
GameState : States; { type defined in MainLib }

procedure SetupOverlays;
begin
Ovrlnit('SHIP.OVR');
if OvrResult <> 0 then begin
WriteLn('Overlay error: ',OvrResult);
Halt(1)
end
end; { of proc SetupOverlays }

begin
SetupOverlays;
Initialize; { in GameInit }
repeat
  case GameState of
    atHelm : DoNavigation; { in Navigation }
inCockpit : DoCombat; { in Combat }
inPanels : DoRepair; { in Repair }
atStation : DoSurvey { in Survey }
end
  until GameState = endGame;
SaveGame { in GameInit }
end. { of prog Ship }

Figure 1: The skeleton of a starship simulation game, which places each of the several distinct functions of starship operation into a separate overlay.

Figure 1: The skeleton of a starship simulation game, which places each of the several distinct functions of starship operation into a separate overlay.

Figure 1: The skeleton of a starship simulation game, which places each of the several distinct functions of starship operation into a separate overlay.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Multi-Edit</th>
<th>BRIEF 2.0</th>
<th>Norton Editor</th>
<th>Vedit Pro</th>
<th>PIZZA WITH EVERYTHING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit 20+ files larger than memory</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>C slices</td>
</tr>
<tr>
<td>Powerful high level macro language</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes Italian</td>
<td></td>
</tr>
<tr>
<td>Full UNDO</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Visual marking of blocks</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No Good</td>
<td></td>
</tr>
<tr>
<td>Line, stream and column blocks</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Use Knife</td>
<td></td>
</tr>
<tr>
<td>Automatic file save</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Online help</td>
<td>Extensive</td>
<td>Limited</td>
<td>Limited</td>
<td>Limited</td>
<td>Extensive</td>
</tr>
<tr>
<td>Choice of keystroke commands or menu system</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes Menu Available</td>
<td></td>
</tr>
<tr>
<td>Function Key assignments labeled on screen (may be disabled)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Word processing functions</td>
<td>Extensive</td>
<td>Limited</td>
<td>Limited Extra Cost</td>
<td>Difficult</td>
<td></td>
</tr>
<tr>
<td>Complete DOS shell</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No Deep Dish</td>
<td></td>
</tr>
<tr>
<td>Pop-up Programmer's Calculator and ASCII Table</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No ASCII</td>
<td></td>
</tr>
<tr>
<td>Unlimited 'Off the Cuff' keystroke macros</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes Sauce on Cuff after</td>
<td></td>
</tr>
<tr>
<td>Allocates all available memory to compiler when run from within editor</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes Lots of bytes</td>
<td></td>
</tr>
<tr>
<td>Intelligent indenting, template editing and brace/parenthesis/block matching and checking for C, PASCAL, BASIC and MODULA-2</td>
<td>Yes</td>
<td>C Only</td>
<td>No</td>
<td>Limited Intelligence</td>
<td></td>
</tr>
<tr>
<td>Flexible condensed mode display</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Definitely</td>
</tr>
<tr>
<td><strong>PRICE</strong></td>
<td><strong>$99</strong></td>
<td><strong>$195</strong></td>
<td><strong>$50</strong></td>
<td><strong>$185</strong></td>
<td><strong>About $12</strong></td>
</tr>
</tbody>
</table>

**MULTI-EDIT COMBINES POWER WITH EASE OF USE LIKE NO OTHER EDITOR ON THE MARKET TODAY.**

- Is your editor OUT TO LUNCH?
- Does it handle ALL OF YOUR NEEDS?
- Is it flexible, programmable and reconfigurable?
- MOST IMPORTANTLY, is it EASY TO USE?

**OR WOULD YOU RATHER BE EATING PIZZA?**

**Only MULTI-EDIT tastes this good!**

- Fully automatic Windowing and Virtual Memory
  - Edit multiple files regardless of physical memory size
  - Easy cut-and-past between files
  - View different parts of the same file
- Powerful, EASY-TO-READ high-level macro language
  - Standard language syntax
  - Full access to ALL Editor functions
- Language-specific macros for C, PASCAL, BASIC and MODULA-2
  - Smart Indenting
  - Smart brace/parenthesis/block checking
  - Template editing
  - More languages on the way
- Terrific word-processing features for all your documentation needs
  - Intelligent word-wrap
  - Automatic pagination
  - Full print formatting with justification, bold type, underlining and centering
  - Flexible line drawing
  - Even a table of contents generator
- Compile within the editor
  - Automatically positions cursor at errors
  - Allocates all available memory to compiler
- Complete DOS Shell.
  - Scrollable directory listing
  - Copy, Delete and Load multiple files with one command
  - Background file printing
- Regular expression search and translate
- Condensed Mode display, for easy viewing of your program structure
- Pop-up FULL-FUNCTION Programmer's Calculator and ASCII chart
  - and MOST IMPORTANT, the BEST USER-INTERFACE ON THE MARKET!

**Extensive context-sensitive help**

- Choice of full menu system or logical function key layout
- Function keys are always labeled on screen (no guessing required!)
- Keyboard may be easily reconfigured and re-labeled
- Easy, automatic recording and playback of keystrokes
- Anchovies easily removed

To Order, Call 24 hours a day:
1-800-221-9280 Ext. 951
In Arizona: 1-602-968-1945
Credit Card and COD orders accepted.
Tempe, AZ 85281

Requires IBM/PC/XT/AT/PS2 or full compatible, 256K RAM, PC/MS-DOS 2.0 or later.
Multi-Edit and American Cybernetics are trademarks of American Cybernetics, BRIEF is a trademark of Underware, Inc. Norton Editor is a trademark of Peter Norton Computing, Inc. Vedit is a registered trademark of ComputView Products Inc. Copyright 1987 by American Cybernetics.
Add power to your Turbo language programs with the Borland Turbo Toolboxes. They provide you with source code and routines to be added into your programs so you don't have to reinvent the wheel. And you don't pay royalties on your own compiled programs that include the Toolboxes' routines.

**TURBO C®**

**TURBO C 2.0 RUNTIME LIBRARY SOURCE CODE**
An indispensable tool for serious Turbo C programmers! The Runtime Library Source Code lets you get even more out of Turbo C's flexibility and control, with a library of more than 350 functions you can customize or use as is in your Turbo C programs. You get the source for the standard C library, math library and batch file to help with recompiling and rebuilding the libraries.*

**TURBO PASCAL®**

**TURBO PASCAL 5.0 RUNTIME LIBRARY SOURCE CODE**
Modify the runtime library source code or use it as is. You get the assembly language and Pascal source to the System, Dos, Crt, Printer, and Turbo3 units. Comes with a batch file to help with recompiling and rebuilding TURBO.TPL.*

**TURBO PASCAL DATABASE TOOLBOX**
With the Turbo Pascal Database Toolbox you can build your own powerful, professional-quality database programs. Included is a free sample database with source code and two powerful problem-solving modules.

**Turbo Access™** quickly locates, inserts, or deletes records in a database using B+ trees—the fastest method for finding and retrieving database information.

**Turbo Sort™** uses the QuickSort method to sort data on single items or on multiple keys. Features virtual memory management for sorting large data files.

**TURBO PASCAL NUMERICAL METHODS TOOLBOX**
Turbo Pascal Numerical Methods Toolbox implements the latest high-level mathematical methods to solve common scientific and engineering problems. Fast. Every time you need to calculate an integral, work with Fourier Transforms, or incorporate any of the classical numerical analysis tools into your programs, you don't have to reinvent the wheel. It's a complete collection of Turbo Pascal routines and programs that gives you applied state-of-the-art math tools. Includes two graphics demo programs to give you the picture along with the numbers. Comes with complete source code.

**TURBO PASCAL TUTOR**
Turbo Pascal Tutor is everything you need to start programming in Turbo Pascal. It consists of a manual that takes you from the basics up to the most advanced tricks, and a disk containing sample programs as well as learning exercises. It comes with thousands of lines of commented source code on disk, ready for you to compile and run. Files include all the sample programs from the manual as well as several advanced examples dealing with window management, binary trees, and real-time animation.

**TURBO PASCAL EDITOR TOOLBOX**
Turbo Pascal Editor Toolbox gives you three different text editors. You get the code, the manual, and the know-how. We provide all the editing routines. You plug in the features you want.

**MicroStar™**: A full-blown text editor with a complete pull-down menu user interface.

**FirstEd™**: A complete editor equipped with block commands, windows, and memory-mapped screen routines.

**Binary Editor**: Written in assembly language, a 13K "black box" that you can easily incorporate into your programs.

**TURBO PASCAL GRAPHIX TOOLBOX**
Turbo Pascal Graphix Toolbox is a collection of tools that will get you right into the fascinating world of high-resolution monochrome business graphics, including graphics window management. Draw both simple and complex graphics. Store and restore graphic images to and from disk.

**TURBO PASCAL GAMEWORKS**
Explore the world of state-of-the-art computer games with Turbo Pascal GameWorks. Using easy-to- understand example games, it teaches you theory and techniques to quickly create your own computer games. Comes with three ready-to-play games: Turbo Chess, Turbo Bridge, Turbo Go-Moku.*

---

*Does not include source for graphics or floating point emulator.
WITH TURBO TOOLBOXES!

TURBO PROLOG®
TURBO PROLOG TOOLBOX IS SIX TOOLBOXES IN ONE
More than 80 tools and 8,000 lines of source code help you build your own Turbo Prolog applications. Includes toolboxes for menus, screen and report layouts, business graphics, communications, file-transfer capabilities, parser generators, and more!

TURBO BASIC®
TURBO BASIC DATABASE TOOLBOX
With the Turbo Basic Database Toolbox you can build your own powerful, professional-quality database programs. Includes Trainer, a demonstration program that graphically displays how B+ trees work and a free sample database with source code. The Toolbox enhances your programming with 2 problem-solving modules:
- Turbo Access quickly locates, inserts, or deletes records in a database using B+ trees—the fastest method for finding and retrieving database information.
- Turbo Sort uses the Quicksort method to sort data on single items or on multiple keys.

TURBO BASIC® EDITOR TOOLBOX
Turbo Basic Editor Toolbox will help you build your own superfast editor to incorporate into your Turbo Basic programs. We provide all the editing routines. You plug in the features you want! We've included two sample editors with complete source code.

MicroStar: A full-blown text editor with a pull-down menu user interface and all the standard features you’d expect in any word processor.
FirstEd: A complete editor with windows, block commands, and memory-mapped screen routines, all ready to include in your programs.

To order, call (800) 543-7543

YES! I want the Borland Turbo Toolboxes® indicated below!
To order, simply complete this coupon, or call (800) 543-7543 and have your credit card number and the code ATIO ready to give to the operator. Mail coupon to: Borland International, 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95066-0001

Turbo C and Turbo Pascal Runtime Libraries each $150
Toolboxes $99.95 each
\( \times \text{Qty.} = \) \( \times \text{Total} = \)
CA and MA residents add appropriate sales tax
Shipping
\( \) \( \) \( \)
Diskette size: \( \) 5½" \( \) 3½"
Payment: \( \) Visa \( \) MC \( \) Check \( \) Money Order
Name _______________________
Shipping Address ______________________
City, State, Zip ______________________
Phone ______________________
Credit Card #: ______________________ /
Expiration Date _______ / _______

Outside U.S. mail payment by bank draft payable in U.S. dollars drawn on a U.S. bank. CODs and purchase orders will not be accepted.
OVERLAYS WITH 5.0

continued from page 42

DEBUGGING SUPPORT
The Turbo Pascal 5.0 Integrated Debugger fully supports overlays. You can single-step through calls to routines in overlay units, and those units will be loaded into and out of memory as needed. Again, this process is handled automatically and invisibly (except, of course, for the disk access that occurs as units are loaded into memory). You can set breakpoints within overlay units, use the Call Stack and Find Functions commands, and otherwise treat these units just like nonoverlaid units.

Overlays can be a great help when debugging very large programs. If kept fully intact, such programs may be too big to run in memory under Turbo Pascal. By breaking a very large program into overlays, the program may be made small enough to run under the Integrated Environment—which places the services of the Integrated Debugger at your disposal.

REMEMBER
A number of things should be kept in mind when using overlays. First, make sure that Ovrlnit is called before calls are made to any of the routines in overlay units. To be safe, call Ovrlnit either at the start of the main body of the program, or (as described below) in the initialization code of a nonoverlaid unit.

Avoid having initialization code in the overlay units. If such code is necessary, then ensure that Ovrlnit has been called before those units are initialized. The only way to do this is to put the call to Ovrlnit into the initialization section of a nonoverlaid unit that appears in the USES clause prior to any overlaid unit.

Be sure to call Ovrlnit before anything is allocated on the heap. Unless the heap is completely untouched, Ovrlnit won’t function correctly when called.

Make sure that Overlay appears in the USES clause before any of the overlaid units. The safest solution is to put Overlay first.

Also, make sure that all units, as well as the main program, are compiled with the {$F+} directive present, or (equivalently) with the Options/Compiler/Force far calls toggle set to On.

Finally, the DOS unit is the only one of the standard units shipped with Turbo Pascal that may be overlaid—and putting DOS out as an overlay is not a good idea. Any of your own units that contains interrupt handlers also may not be overlaid.

CONQUER SPACE
In order to write good programs, the needs of the program specification must be balanced against available DOS memory, expanded memory, and disk storage resources. The size of Turbo Pascal 4.0 programs is limited to available DOS memory space. Turbo Pascal 5.0's overlays feature raises that size limit well beyond the megabyte point. How far you can take the size of a single program depends upon how efficiently you use data space and symbol table space. With some care in design, your programs can (in most cases) be as large as they need to be.

Bruce Webster is a computer mercenary living in California. He can be reached via MCI MAIL (as Bruce Webster) or on BIX (as bwebster).

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

You'll write better Turbo Pascal 4.0 programs easier and faster using the powerful analytical tools of Turbo Analyst 4.0.

You get • Pascal Formatter • Cross Referencer • Program Indexer • Program Lister • Execution Profiler, and more. Includes complete source code.

Turbo Analyst 4.0 is the successor to the acclaimed TurboPower Utilities:
"If you own Turbo Pascal you should own the Turbo Power Programmers Utilities, that's all there is to it."
Bruce Webster, BYTE Magazine, Feb. 1986

Turbo Analyst 4.0 is only $75.

A Library of Essential Routines

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

Turbo Professional 4.0 is only $99.

Call toll-free for credit card orders.

1-800-538-8157 ext. 830 (1-800-672-3470 ext. 830 in CA)
Satisfaction guaranteed or your money back within 30 days.
NEW! MS-DOS File Compatibility Package (create, read, & write MS-DOS file systems on non-MS-DOS computers) $500
Blue streak Plus Communications (two ports, programmer's interface, terminal emulation) $400
PostScript, including GS+4 (COM, database, windows, file, user interface, DOS & CP) $35
CQL Query System (SQL retrievals plus windows) $325
GraphiC 4.1 (high-resolution, DISSPLA-style scientific plots in color & hardcopy) $325
NEW! Vmem/C (virtual memory manager; least-recently used pager, dynamic expansion of swap file) $250
NEW! TurboG (Graphics Library (all popular adapters, hidden line removal) $150
NEW! PC Curses (Aspen, Software, System V compatible, extensive documentation) $220
NEW! Greenleaf Data Windows (windows, menus, data entry, interactive form design) $200
NEW! Greenleaf Communications Library (interrupt mode, modem control, XON-XOFF) $175
TurboTeX (TRIP certified; HP, PS, dot drivers; CM fonts; LaTEX) $170
NEW! Essential resident C (TSRify C programs, DOS shared libraries) $165
Greenleaf Functions (296 useful C functions, all DOS services) $160
NEW! Essential Utility Library (400 useful C functions) $160
NEW! Essential Communications Library (C functions for RS-232-based communication systems) $160
NEW! Win Version 2.0 (C program interface to Lotus 1-2-3, AllBase, Quick, & Clipper) $155
OS/88 (U*X*x-like operating system, many tools, cross-development from MS-DOS) $155
NEW! ME Version 2.0 (programmer's editor with C-like macro language by Magina Software; Version 1.31 still $75) $140
NEW! TurboG Graphics Library (all popular adapters, hidden line removal) $135
PC Curses Package (full Berkeley 4.3, 4 menus, and data entry examples) $120
NEW! CIFTree (B+tree ISAM driver, multiple variable-length keys) $115
NEW! Minix Operating System (U*X*x-like operating system, includes manual) $105
NEW! PC/IP (CMU/MIT TCP/IP implementation for PCs) $100
NEW! B+Tree Library & ISAM Driver (file system utilities by Softfocus) $100
NEW! The Profiler (program execution profile tool) $100
NEW! Entelekon C Function Library (screen, graphics, keyboard, string, printer, etc.) $100
NEW! Entelekon Windows (menus, overlay, alarms, file handling, etc.) $100
NEW! TurboGeometry (geometry of routines for computational geometry) $90
NEW! QC88 C compiler (ASM output, small model, no longs, floats or bit fields, 80+ function library) $90
NEW! WPS (Word Processing System Construction Kit or PCNX, PC MS OS Shells) $80
NEW! C Windows Toolkit (pop-up, pull-down, spreadsheet, C/GA/EGA/Hercules) $80
NEW! IATE Async Terminal Emulator (includes file transfer and menu subsystem) $80
NEW! MultiDOS Plus (C-based multitasking, intertask messaging, semaphores) $80
NEW! Win Version 1.03 (C program interface to Lotus 1-2-3, program & files) $80
NEW! Professional C Windows (open & mean window and keyboard handling) $70
NEW! ip (flexible printer driver; most popular printers supported) $65
NEW! Query (interactive C interpreter) $60
NEW! EZ_ASM (assembly language macros bridging C and MASM) $60
NEW! PTree (pars tree management) $60
NEW! MicroFrm Toolkit (28 Uniques utilities for MS-DOS) $50
NEW! XTBIOS Kit (roll your own BIOS with this complete set of basic input/output functions for XT) $50
NEW! HELP! (pop-up help system builder) $50
NEW! Multi-User BBS (chat, mail, menus, syop displays; uses Galacticomm modem card) $50
NEW! Make (macros, all languages, built-in rules) $50
NEW! Vector-to-Raster Conversion (stroke letters & Tektronix 4010 codes to bitmaps) $50
NEW! Coder's Prolog (inference engine for use with C programs) $50
NEW! Virtual Memory System (least recently used swapping) $40
NEW! Virtual Memory System (least recently used swapping) $40
NEW! Biggs's System's Tools (multi-tasking window manager kit) $40
NEW! PC-XINU (Comer's XINU operating system for PC) $35
NEW! CLIPS (rule-based expert system generator, Version 4.1) $35
NEW! Tiny Curses (Berkeley curses package) $35
NEW! TELE Kernel or TELE Windows (Ken Berry's multi-tasking kernel & window package) $30
NEW! SP (spelling checker with dictionary and maintenance tools) $30
NEW! Compiler (C interpreter with extensive internals documentation) $30
NEW! Translate Rules to C (YACC-like function generator for rule-based systems) $30
NEW! 6-Pack of Editors (six public domain editors for use, study & hacking) $30
NEW! Crunch Pack (14 file compression & expansion programs) $30
NEW! UnPL (casing and list processing language, Version 7) $25
NEW! FLEX (fast lexical analyzer generator; new, improved LEX) $25
NEW! LEX (lexical analyzer generator; an oldie but a goodie) $25
NEW! Bison & FREP (YACC workalike parser generator & attribute grammar preprocessor) $25
NEW! AutoTrace (program tracer and memory trasher catcher) $25
NEW! Arrays for C (macro package to ease handling of arrays) $25
NEW! OOPS (collection of handy C++; classes by Keith Gorlen of NIH) $20
NEW! C Compiler Torture Test (checks a C compiler against K & R) $20
NEW! Benchmark Package (C compiler, PC hardware, and Unix system) $20
NEW! TN3270 (remote login to IBM VM/CMS as a 3270 terminal on a 3274 controller) $20
NEW! A68 (68000 cross-assembler) $20
NEW! List Proc (functions for lists, stacks, and queues) $20
NEW! XLT Macro Processor (general purpose text translator) $20
NEW! Cntreality (Eliza-based notetaker) $15
Data
WordCruncher (text retrieval & document analysis program) $275
DNA Sequences (GenBank $2.0 including fast similarity search program) $150
Protein Sequences (5,415 sequences, 1,302,966 residues, with similarity search program) $50
Webster's Second Dictionary (234,932 words) $60
U.S. Geodetic Datums & longitudes/latitude of 32,000 U.S. cities and 6,000 state boundary points $35
The World Digitalized (100,000 longitude/latitude of world country boundaries) $30
KST Fonts (15,200 characters in 139 mixed fonts: specify TeX or bitmap format) $30
USNO Floppy Almanac (high-precision moon, sun, planet & star positions) $20
NEW! NIST fonts (373 stroke characters in 14 fonts) $15
NEW! U. S. Map (15,701 points of state boundaries) $15

The Austin Code Works
WKS Version 2.0 $150
Voice: (512) 258-0785
11100 Leafield Lane acuinfo@uunet.uu.net
Austin, Texas 78750-5109 USA
Free surface shipping on prepaid orders
BBS: (512) 258-8831
FAX: (512) 258-1342
For delivery in Texas add 7% MasterCard/VISA
The Austin Code Works
WKS Version 2.0 $150
Voice: (512) 258-0785
11100 Leafield Lane acuinfo@uunet.uu.net
Austin, Texas 78750-5109 USA
Free surface shipping on prepaid orders
BBS: (512) 258-8831
FAX: (512) 258-1342
For delivery in Texas add 7% MasterCard/VISA
BUG HUNTING, BORLAND STYLE

Jeff Duntemann

ack in 1976, I wire-wrapped a computer that was based on a circuit diagram in *Popular Electronics*. My new computer used the CDP1802 CPU, which had a single-line serial output that could be set to one or to zero; I connected the line to an LED. A single instruction brought the line high, and another instruction brought it low again. Input to the machine was a row of eight toggle switches; output was a two-digit hexadecimal display. To test the machine, I toggled in the single-byte opcode that should have turned the LED on by bringing the serial line high. Nothing happened. I triple-checked the opcode (but seriously, how many ways are there to toggle in 7BH?) The hex display read 7B. The toggle switches were set to 0111 1011. The LED stayed off.

I assumed the CPU was bad, until I swapped it into a friend’s similar machine and found out that my CPU worked fine. I swapped all the ICs. I checked all the wiring. The machine appeared to be in perfect condition—yet it wouldn’t run a one-byte program. Time and lots of Mountain Dew uncovered the following problems:

1. By mistake, I had wired the toggle switches upside down; in other words, a switch whose bat handle was high (indicating a binary 1) was actually putting out a binary 0.
2. By mistake, I had socketed an octal inverter, rather than an octal driver, between the toggle switches and the hex display.

What this means is that the toggle switches were putting out an inverted byte, but that the inverting drivers that fed the hex display reinverted the inverted byte from the toggle switches, making the byte look normal again. The switches said 7BH. The displays said 7BH. But the machine was actually receiving an 84H byte, which did something harmless but incorrect. It was an accidental but diabolical partnership between two otherwise obvious screwups that hid one another perfectly for several weeks.

We don’t wire-wrap our machines any more, thank God, so hardware bugs like this have pretty much become extinct. But bugs will always be with us, and my 1976 experience says something absolutely basic about debugging: *Inspection is not enough.* You can fix some bugs by staring at your code after a good night’s sleep. You can fix a few more by pulling procedures out of a program piecemeal and plugging them into proven programs to get a second opinion. But even when all of the parts check out separately, the little devils often refuse to cooperate in peculiar ways when reassembled, no matter how carefully.

LIFTING THE LID

There’s no way around it: You have to lift the lid, go in there, and see what’s happening. Assume *nothing*. Watch every statement execute. Look at every variable at every step of the way. If you fail to do this, you’ll miss something, and the something you’ll miss will be the one thing you’ve been looking for for weeks.
The process of opening up the closed universe of a computer program for examination requires special tools. We call these tools debuggers, and commercial software development would be impossible without them. How debuggers work is the blackest of black arts, but what they do falls into two broad categories:

1. Debuggers stop and start program execution on command without losing the current state of the program. A program can be paused at a preset point in the code (called a breakpoint), or it can be made to pause after each program step. (This process is called single-stepping or tracing.) Tracing a program allows you to see "what it's doing in there." Breakpoints offer a chance to examine the effects of the program statements on program variables in medias res.

2. Debuggers let us examine and change the values of program data items. At the lowest level, this includes CPU registers, memory, and I/O ports. Some advanced debuggers (called symbolic debuggers) have the ability to relate memory, and occasionally machine registers, to program identifiers such as variable names.

POINTS OF VIEW

Even with respect to the way that they execute those two missions, debuggers are a pretty diverse lot. Every debugger falls into one of three categories that turn on the way the debugger (and, hence, you the programmer) view the program under examination. This matter of point of view is critical. There are two points of view from which to examine a computer program: the machine's point of view, and the programmer's point of view.

The machine sees the program as a series of executable binary instructions in memory, which are located alongside other memory locations that are set aside to store binary data. The machine's view also includes a set of values in machine registers that continually change as the program executes. In addition, there may be I/O ports that transfer data to and from the outside world.

Since the invention of high-level languages, such as C and Pascal, the programmer has had quite a different view of a program. A high-level language groups incomprehensible machine instructions together into higher-level program statements that are more easily read, remembered, and understood. The language also partitions data storage memory into named chunks that reflect familiar concepts in the human culture: yes/no answers, numbers, characters, values that are grouped into indexed arrays or named records, and so forth. The state of machine registers is usually hidden from the programmer's view, except in rare circumstances.

You can think of a program as a structure printed on a piece of paper that is suspended in space between the programmer (above) and the machine (below). To the programmer, who looks down on the program from above, the structure appears to be made up of program statements and named variables. The machine, which looks up at the program from below, sees a conglomeration of memory locations that contain either machine instructions or bi-

continued on page 50
BUG HUNTING

continued from page 49

nary data, plus a scattering of ever-changing registers. Two different views of exactly the same program.

Debuggers are classified based upon whose view they take. High-level debuggers look over the programmer's shoulder, and understand and display program statements and variables. They cannot display memory locations, machine instructions, or machine registers. Low-level debuggers can step through machine instructions and display blocks of memory. However, these debuggers are ignorant of high-level languages, and have no knowledge of program statements or variables. Full symbolic debuggers sit on the fence between the two worlds, embracing both of them. On the one hand, these debuggers understand high-level languages—they can step through a C or Pascal program line by line, displaying the contents of program variables as they go. On the other hand, full symbolic debuggers can also show the machine's view of memory, instruction opcodes, and machine registers. Best of all, these debuggers can show the synergy between the two views of a program—variables that are loaded into machine registers; program statements that display beside their equivalent machine instructions; and data that moves among variables, registers, and I/O ports.

The classic low-level debugger is DOS DEBUG, which is included with every copy of DOS. Inter-

INTEGRATED DEBUGGING

Both Turbo Pascal 5.0 and Turbo C 2.0 contain high-level debuggers that are intimately intertwined with both languages' Interactive Development Environments. We call these new debuggers the Borland Integrated Debuggers, because they're always beside the compiler, ready to go, while you're putting your programs together. I offer a close look at Turbo Pascal 5.0's Integrated Debugger on page 12 of this issue; Kent Porter leads a tour through Turbo C 2.0 and its Integrated Debugger on page 62.

The Borland Integrated Debuggers handle most program development, especially with respect to small programs and programs that don't perform a lot of black magic. On the other hand, the larger and the more ambitious your programs become, the greater the chances that you'll concoct a bug that is beyond the grasp of the Integrated Debuggers. The pursuit of system-level code crickets requires the synergy of a full symbolic debugger—and now you can turn to Turbo Debugger. (If that won't find 'em, you'd better go have a look at your toggle switches.)

Finally, Borland has released Turbo Assembler as a companion product that ships with Turbo Debugger. While retaining full compatibility with MASM 5.x, Turbo Assembler also offers Ideal mode, which is a new and more comprehensible syntax for assembly language, plus 286/386 support. Tom Swan introduces Turbo Assembler's features, including the new Ideal mode syntax, on page 120.

The more power you have, the more ways there are to go wrong. In future issues of TURBO TECHNIX, we'll pursue our ongoing mission of putting useful programming techniques in your hands. At the same time, we'll provide more information about fixing things that don't work the first time out. Remember: Assume nothing. Examine everything. And always use the best tools that you can bring to bear on the problem.
THE LIGHTNING WAY TO "C"

PRO-C. The first complete 'C' application tool that produces 'C' code. Programming can now be easier and routine coding problems are now eliminated. The only source code generator that runs on MSDOS, QNX, XENIX and UNIX, PRO-C has the ability to reproduce the normal functions of complex application software.

• DATA DEFINITION
Utilizing the integrated PRO-C generators to create menu, screen and report programs or batch processes.

• INTEGRATED GENERATORS
Screens Menus Reports Batch processes. Combined they can generate any business or database application.

• CONTEXT - SENSITIVE HELP
Complete context sensitive help is available at the touch of the help key.

• UTILITIES
Professional utilities allow the selection of your development environment, on line help for generated programs and full program documentation.

Call today for more information. Toll free
1-800-265-8887 North America Only.

Chancelogic Inc.
Allen Square, 180 King St. South, Waterloo, Ontario, Canada N2J 1P8.
Tel: 519-745-2700.
Fax: 519-746-1613.
LONG AGO, I made a comfortable living writing video games for the PC. Whenever I ran into a bug, I had no choice but to fire up DEBUG, the debugger IBM then provided free with DOS. DEBUG wasn’t much of a debugger, since it had just one kind of breakpoint, couldn’t display data structures, and could only debug at the assembly language level. In fact, the only thing DEBUG had going for it was that it was better than the alternative, which was nothing.

As you’ll read elsewhere in this issue, Borland has closed the debugging gap in a big way by adding integrated debugging to both Turbo Pascal (p. 12) and Turbo C (p. 48). Still, because each of the integrated debuggers has to squeeze into memory along with an editor, a compiler, a linker, and a user program, the integrated debuggers are inescapably less powerful than standalone debuggers. Certain debugging problems, such as runaway pointers, complex error conditions, debugging of assembler code, and the like, absolutely require a state-of-the-art symbolic debugger. Unfortunately, advanced debuggers tend to be difficult to use, and are generally more suited to debugging assembly language than Pascal or C programs. The ideal debugger would not only be state-of-the-art in terms of sheer power, but also would be as easy to use for high-level languages as for assembler.

Borland’s new Turbo Debugger fits that description to a T. Equally at home with Turbo Pascal, Turbo C, or Turbo Assembler programs, Turbo Debugger offers an intuitive interface and a suite of debugging features that take software-only debugging to the limits of possibility. On 386-equipped systems, Turbo Debugger can put the advanced capabilities of the 80386 CPU to work to provide limited hardware assistance in terms of hardware breakpoints. Another 386-based debugging breakthrough allows the debugger to run in 386 protected mode and the application being debugged to occupy a separate virtual-86 partition. This means that your application can be as large as necessary without crowding the Debugger out of DOS memory. Furthermore, the application can reside at the same addresses that it will occupy on its target system.

Let’s take a closer look at Turbo Debugger, and explore the situations when you might want to step up to Turbo Debugger from your Turbo language’s integrated debugging.

ADVANCED DEBUGGING FEATURES

At heart, there’s only one question to ask about a debugger: “How well does it let me catch error conditions in my programs?” The key to catching error conditions is breakpoint capability—and Turbo Debugger is extremely powerful in this area.

Breakpoint capability normally refers to the ability to instruct a program to stop for examination when a certain line of code is reached. Turbo Debugger has all of the standard breakpoint features. A breakpoint can be set simply by pressing the F2 function key on the line where you want the break to occur, or a break address can be specified by way of the Breakpoints menu. A program can also be executed either one source code line or one assembly language instruction at a time, and can either step over or trace into subroutines. Alternatively, you can just sit back and watch your PC screen change as Turbo Debugger runs a program line-by-line at a reduced speed. You can have Turbo Debugger run to a certain line by pressing F4 on that line; to run Turbo Debugger to the end of a function, simply press Alt-F8.

These are fairly standard debugger breakpoint capabilities—and Turbo Debugger goes even further. Turbo Debugger lets you stop a program either when a memory location is changed, or when an expression becomes true. (By the way, expressions can be evaluated in the notation of the language of your choice—Pascal, C, or assembler—at any time, and these expressions can even contain functions.) What’s more, you can select the number of times that a breakpoint condition must occur before it causes a break, so that you don’t have to wait through 100

continued on page 54
struct TextBlock

```c
#include "demo.c"

int main()
{
    struct TextBlock *NextTextBlock;
    /* pointer to the next text block
    that
    */
    struct TextBlock
    
    Stack
    
    _main()

    CPU 80286
    _main: main()
    cs:01EE push bp
    cs:01EF mov bp,sp
    51A7:01EC 04 53 55 8B EC 83 EC 0
    7
    c=0
    z=0
    buffer *
    
    Watches
    F2-Close F3-Bkpt F4-Here F5-Zoom F6-Next F7-Trace F8-
iterations of a loop if the case you're interested in
occurs during the 101st pass. Alternatively, Turbo De-
bugger can record occurrences of a given breakpoint
in its ongoing log; later, you can refer back to the log
to see how the current state was reached. You can
also record comments and data dumps into the log,
and can record the log to disk. Perhaps most remark-
ably, you can instruct Turbo Debugger to execute the
expression of your choice at a given breakpoint;
since such expressions can modify variables, this
gives you a way to temporarily patch a line of code
into a program without leaving—or even restarting—
a debugging session. As I'll show later, these sophis-
ticated data breakpoints let you catch bugs that might
otherwise take hours to find.

There's a price to be paid when the more sophis-
ticated breakpoints are used. Programs run more
slowly when a changed memory location breakpoint
is active, for example, since Turbo Debugger must
stop after each line to see whether the breakpoint
condition has been met. To speed things up, Turbo De-
bugger offers an option that combines code and
data breakpoints. You can specify that a given data
breakpoint should only be checked when a given
line is executed; if you know where but not when a
bug occurs, you can quickly reach that point with a
combined code and data breakpoint.

To facilitate the use of hardware breakpoints,
Turbo Debugger contains a device-driver interface
that allows it to work with third-party hardware de-
bugger products from vendors such as Atron and Periscope. The first such Turbo Debugger-compatible
product has already appeared, in the form of Purart's
Trapper board (see accompanying sidebar). A device
driver is shipped with Turbo Debugger that allows
the use of the 386 CPU's built-in hardware debug-
ing features without additional hardware.

THE USER INTERFACE

While breakpoints are a prominent feature of any
debugger, the user interface makes the power of a
debugger readily available. Turbo Debugger expands
upon the familiar Borland windowing interface in a
number of ways.

Look at the context. For starters, Turbo Debugger is
highly context-sensitive. Help is context-sensitive.
Local variables are popped up for inspection from a
default scope that is determined by the cursor's loca-
tion in the source code. The default language con-
vention by which expressions are evaluated depends
upon the type of source module being debugged. If
Turbo Debugger thinks you're looking at a string, it
displays that data as text; otherwise, the data is dis-
played as hex bytes. Default responses to prompts
are based on the text located below the cursor. In
many cases, text can be highlighted on the screen
and then can serve as the response to a prompt; this
saves considerable typing.

Open a window. Turbo Debugger's basic screen con-
ists of any number of windows, with pull-down
menus on the menu bar at the top of the screen. At
your option, windows may overlap to any degree or
not at all. Figure 1 shows a Turbo Debugger screen
that contains three windows and a pull-down menu.
You can readily rearrange, resize, and move between
the windows with either hotkeys or menu com-
mands. Window configurations can be saved to disk
and reloaded later. In addition, Turbo Debugger al-
 lows you to undo the last window close, so you can
quickly recover if you close a window and then de-
cide you need that window after all.

Local menus. Each type of window has a specific
purpose. There are windows for viewing source
code, viewing the CPU state, inspecting data struc-
tures, watching variable values, dumping memory,
and more. Consequently, different actions are ap-
propriate to different types of windows. Rather than
try to cram the commands for all of the windows
onto the single menu bar at the top of the screen,
Borland instead chose to implement local menus. A
local menu is a popup menu specific to the window
that is currently active. The local menu for the cur-
rent window can be popped up at any time by press-
ing Alt-F10. Figure 2 shows the local menu for the
module viewer window.

Hotkeys and other tricks. As usual, Borland has pro-
vided hotkeys as a quick way to select many menu
items. To help you remember the many hotkeys, the
bottom line of the screen (known as the help line)
shows the available hotkeys at any given time. If you
hold the Alt key down, the help line shows the Alt
hotkeys. Hold down the Ctrl key, and the Ctrl hot-
k eys are displayed. The Ctrl hotkeys are also hotkeys
into the current local menu, so holding down Ctrl is
a good way to see the local menu commands that are
available at any time.

The Turbo Debugger interface provides other
handy features. For instance, it maintains history lists
of your responses to prompts. When a given prompt
is issued, your recent choices are displayed as well;
you can save considerable typing by reusing or mod-
ifying one of your earlier choices.

As another convenience, whenever Turbo De-
bugger presents an alphabetized list (such as the list
of global variables in the variables viewer window),
you can start typing the name of any item in that list.
As you press each key, Turbo Debugger instantly dis-
plays the next item in the list that matches the key-
strokes you've entered so far. This is quite handy if
you use hundreds of variables and can't remember
all of the letters in a given variable.

THE VIEW FROM WITHIN

As you can see, Turbo Debugger's user interface is
designed to let you work as efficiently as possible—
but what does it actually let you do? Briefly put, the
interface offers very flexible ways to view and modify
code and data within an executing program.
Pick a level. You can view code at either the source code or assembly language level. If you view code at the source code level (in a module viewer window), you don’t need to see individual instructions, registers, or flags unless you want to. At this level, code can be single-stepped a source-level statement at a time. If you view code at the assembly language level (in a CPU viewer window), you can see every detail of the program as it executes. Here, code can be single-stepped an instruction at a time. As an alternative to viewing code with either method individually, both module and CPU viewer windows can be displayed simultaneously so that you can watch code execute at both levels.

Any or all of the source modules in a program can be viewed at any time. You can search the source code for a text string, just as you would search for a text string in a text editor.

Follow the trail. The stack viewer window shows the function calling trail that led to your current location in the program. You can move to any function in the stack viewer window and see that function’s local variables and actual parameters.

Code can be assembled directly into the program for patching purposes, although those changes are only made to the program in memory. Such changes are lost as soon as the debugging session is ended, or the program is reloaded.

continued on page 56
Show me your data. Now we come to viewing data. The dump viewer window lets you display any area of memory in hex and ASCII. You can specify the area of memory to dump with any expression that resolves to a memory address. You can modify memory while you view it in either hex or ASCII. Hex values can be displayed in a variety of formats, including bytes, words, longs, and IEEE floats, and can be followed as pointers via local menu commands.

Expressions can be evaluated at any time, and the format in which the result displays can be controlled. Expressions can modify variables directly by assigning values to them. In many cases, an expression can generate a value that is stored into memory.

WATCHES AND INSPECTORS
Turbo Debugger also understands variables at the source code level—and that's where the real power of Turbo Debugger's data access features becomes apparent. Turbo Debugger not only knows about local variables (automatic and static) and global variables, but also knows about data types, pointers, arrays, structures, and unions. Named variables are automatically displayed according to their original source code data types in either the watches viewer window or the data inspector window.

The watches viewer window, which normally occupies the bottom of the screen, is the standard way to keep an eye on the values of selected variables during program execution. This window lets you select one or more variables for display. (Actually, any expression that resolves to a value may be displayed.) Structures and arrays can also be displayed. In addition, any memory location displayed in the watches viewer window can be modified.

Data inspector windows are something else altogether. These windows not only show the source form of variables, but can also readily follow pointers, scroll through arrays, display nested structures, and the like. Where watches viewer windows are useful for posting the values of several variables, data inspector windows are ideal for delving into the details of a specific variable or data structure. If the cursor is located on the name of an array, pressing Ctrl-I pops up a data inspector window for that array on the spot. If the cursor is located on the name of a pointer, an inspector can be popped up to show that pointer's referent, and another data inspector window can even be popped up from the first inspector to show additional information about the referent. Data inspector windows can be chained to follow a linked list of pointers, or to examine an array of structures or a structure that contains arrays.

The data addressed by any expression that resolves to an address can be inspected, and type-casting can be performed on any such expression. Variables that appear in the module and watches viewer windows can be inspected simply by pressing Ctrl-I. Variables in a data inspector window can be modified. Functions, local variables, and passed parameters can even be inspected by selecting them directly from the source code in a module viewer window.

More than any other feature, the data structures in modern programming languages set these languages apart from their predecessors. With data inspector windows, Turbo Debugger puts data structures at your fingertips.

THE DEATH OF HEISENBERG
Turbo Debugger offers a number of advanced features that let you take on debugging problems that go beyond the merely difficult to the brutal. One such problem is the debugging of very large programs. The difficulty here is that a large program, a debugger, and the information that the debugger needs to maintain about the program often can't all fit into the 640K DOS address space at the same time. Turbo Debugger provides three different solutions to this problem.

EMS storage. First of all, Turbo Debugger can store the table of information about a program's symbols in EMS memory (if EMS memory is present), thereby freeing the DOS memory that the table normally occupies and making that DOS memory available to the program being debugged. Furthermore, EMS memory can be shared between Turbo Debugger and the application being tested.

Separate but linked. Second, if two computers are available during development, Turbo Debugger can be moved away from the target application to run on another PC altogether, with debugging control taking place over a serial link between the two machines. In this configuration, Turbo Debugger needs only about 10K RAM on the target computer. This leaves plenty of memory for the application.

Virtual-86 partitions. Turbo Debugger's third solution to the problem of debugging large applications is particularly exciting. Turbo Debugger can take advantage of the virtual-86 feature of the 80386 and split memory into a virtual-86 partition for your application being debugged and a 386 protected mode partition for Turbo Debugger. This arrangement carries two benefits: First, any program that runs on a PC system can be debugged, no matter how large the program is. Second, the program being debugged loads at exactly the same memory location in the virtual PC as the program would in a standard PC if that program weren't being debugged, and the normal amount of memory is available in the PC for the program to use. As a result, in 80386 mode Turbo Debugger eliminates the interference with the target program that other debuggers inevitably introduce. This interference is sometimes called the "Heisenberg effect," after the famous physicist who demonstrated that it's impossible to observe subatomic interactions without altering them. With the combi-
nation of Turbo Debugger and a 386, it's possible to observe a program's inner workings without the observer getting in the way.

SCREENS AND KEYS
Another problem that arises during the debugging process is that both the debugger and the target application want to use the entire screen display. Turbo Debugger offers a number of screen-handling solutions. The debugger can switch between the user screen and the debugger screen, use a second display, use the extra text pages of color adapters, or turn off user display updating altogether. If none of these options is ideal for a particular program, the two-machine remote debugging approach described earlier, which solves all display-related problems, can be used.

Turbo Debugger allows the text editor of your choice to be invoked directly from the debugging environment. You can then return to the debugger and make changes to programs (or to data files) the instant you recognize a bug. Similarly, files can be viewed and modified directly from a file viewer window.

Keystroke sequences can be assigned to keys, and those keys can then be used instead of lengthy hand-typed command sequences. These keyboard macros are useful for quickly returning to a specific place in a program; once the key sequence that gets you to a given point is recorded, you can return to that point at any time with a single keystroke.

Turbo Debugger can disassemble all 8086, 80286, 80386, and 80387 instructions, both real- and protected-mode. It can also assemble all 8086, 80286, 8087, and 80387 instructions, plus most 80386 instructions. Turbo Debugger provides full support and a special window for the 87-family numeric coprocessor.

Turbo Debugger is, as you'd expect, designed to complement the latest generation of Turbo languages: Turbo C 2.0, Turbo Pascal 5.0, and Turbo Assembler 1.0. The current releases of Turbo Basic and Turbo Prolog are not supported, but future releases will be supported. If you use a compiler or assembler from another vendor, you may still be able to use Turbo Debugger, since it also supports programs compiled for use with Microsoft's CodeView debugger through a conversion utility. In addition, you can always debug any program at the assembler level with Turbo Debugger, regardless of the language with which the program was created.

WHEN DO YOU NEED TURBO DEBUGGER?
Now that you have an idea of what Turbo Debugger can do, the next question is when you might need to move up from integrated debugging with your favorite Turbo language to Turbo Debugger.

More and better. Turbo Debugger can help when you feel that you need more sophisticated breakpoints, or better display of data structures, than integrated debugging offers. For example, if a given flag is set to an incorrect value every 50 times that a function is called, you'd be much better off having Turbo Debugger break on the incorrect value, rather than break on the function 50 times in the integrated debugger so that you have to manually check the value of the flag each time.

Similarly, if you're having problems with nested structures, structures of arrays, or complex pointers, Turbo Debugger is the way to go. The data inspector windows of Turbo Debugger are simply the best tool around for examining complex data structures.

Low-level action. Turbo Debugger becomes absolutely necessary when you need to observe low-level machine functions in action. Integrated debugging is confined to entities that are defined by and understood by the high-level language that this debugging serves: constants, variables, and high-level language statements. If your program directly accesses DOS functions, BIOS functions, BIOS variables, interrupt vectors, display memory, or I/O ports; if you need to access memory directly from the debugger or need to know the actual addresses of variables; or if you're...
/* Turbo C 2.0 program for use in a sample Turbo Debugger 
* debug session. The bug: The Text array in the 
* TextBlock structure does not include space for the 
* terminating zero byte. The solution: Dimension the 
* Text array to (BUFFER_LENGTH +1) characters in length. 
* By Michael Abrash 6/18/88 
*/ 
#include <stdio.h> 
#include <stdlib.h> 

#define BUFFER_LENGTH 20

main() /* Structure we'll use to store text in. These structures 
* Number of characters buffered per text block. */

/* The text array to (BUFFER_LENGTH 
* text buffer */

struct TextBlock { 
    char Text[BUFFER_LENGTH]; /* text buffer */
    struct TextBlock *NextTextBlock; /* pointer to next 
* text block */
};

int c; /* temporary storage for a character */
int Done = 0; /* set to 1 when all text is buffered */
int TextCount; /* location in the current text buffer */
struct TextBlock *FirstTextBlock; /* Points to the text block that 
* starts the linked chain. */
struct TextBlock *CurrentTextBlock; /* Points to the current text block */
struct TextBlock *NewTextBlock; /* When the next text block */
/* Get the initial text block */
if ( !(FirstTextBlock = CurrentTextBlock = 
    malloc(sizeof(struct TextBlock))) ) ( /* We couldn't get any memory */
    printf("Out of memory!\n");
    exit(1);
)
/* Buffer the text the user types, allocating memory as 
* it's needed */

TextCount = 0; 
while ( !Done ) ( 
    c = getchar();
    if ( c == EOF ) ( /* It's the end of the file, so we're done */
        /* Put a zero at the end of the current buffer, 
         * to make it a string */
        CurrentTextBlock->Text[TextCount] = 0;
        /* Mark that this is the last text block in the 
         * linked list */
        CurrentTextBlock->NextTextBlock = 0;
        /* We've gotten all the text */
        Done = 1;
    )
    else ( /* Buffer the character */
        CurrentTextBlock->Text[TextCount++] = toupper(c);
        /* This buffer's full, so allocate another 
         * text block */
        if ( !(NewTextBlock = 
            CurrentTextBlock->NextTextBlock = malloc(sizeof(struct TextBlock))) ) ( /* We couldn't get any more memory */
            printf("Out of memory!\n");
            exit(1);
        )
        /* Put a zero at the end of the current buffer, 
         * making it a string */
        CurrentTextBlock->Text[TextCount] = 0;
        /* Start buffering at the beginning of this 
         * text block's text buffer */
        TextCount = 0;
        /* Make the newly allocated text block the 
         * current text block */
        CurrentTextBlock = NewTextBlock;
    )
)
/* Print out the uppercase result, starting with the 
* text stored in the first text block and continuing 
* until the last text block (the text block with a 
* null link) has been displayed */
CurrentTextBlock = FirstTextBlock;
do ( /* Print the text */
    printf("%s", CurrentTextBlock->Text);
    CurrentTextBlock = CurrentTextBlock->NextTextBlock;
) while ( CurrentTextBlock != 0)

THE VIEW FROM WITHIN
continued from page 57

interested in the actual assembly language code generated by Turbo Pascal or Turbo C, you need Turbo Debugger.

A SAMPLE SESSION
In this section, I'll show how Turbo Debugger lets you catch a subtle bug that could be infuriating to find when using a less-capable debugger. Listing 1 shows a Turbo C program that stores any amount of typed text (converted to uppercase) in a linked list of structures, which are allocated on the fly as they're needed. When all of the text is entered, the program prints the uppercase text. The task is simple enough, but a bug turns up when the program is run and the following lines are typed in:

First line
Second line
Third line
Fourth line
Fifth line
Sixth line

When these lines are entered, the text shown in Figure 3 results.

Clearly, something is wrong—but where? To get a handle on the problem, load the program into Turbo Debugger and move the cursor to the following line, located just before the final do loop:

CurrentTextBlock = FirstTextBlock

Pressing F4 at this point instructs Turbo Debugger to execute the program to this line and then to stop. After the six lines of text are entered, Turbo Debugger breaks at the selected line and brings up the debugging interface. At this point, all of the entered text is supposed to have been stored in a linked list of TextBlock structures.

Here, data inspector windows can be used to great advantage. To create a data inspector window, press Ctrl-I with the cursor positioned over any occurrence of FirstTextBlock in the module viewer window. The window that appears shows the first TextBlock structure, which contains the first 20 text characters and a pointer. This structure looks fine, so move the cursor to the NextTextBlock field of the structure and press Ctrl-I again to pop up another inspector that follows the link to the next block. The result is shown in Figure 4.

Figure 4 makes it plain that something is wrong with the location to which the NextTextBlock field of FirstTextBlock points. The data inspector windows show clearly that the block of data to which the first link points does not contain the correct text. Two explanations are possible: at some point in the program, either the second TextBlock structure is filled with garbage, or else the NextTextBlock field of FirstTextBlock is set to point somewhere other than to the second TextBlock structure.

Turbo Debugger lets you check both cases simultaneously. First, set the program back to the start by selecting Program Reset from the Run menu. Then
move the cursor to the second occurrence of `CurrentTextBlock->Text[TextCount] = 0` and press F4; this step runs the program up to the point at which `FirstTextBlock->NextTextBlock` is set. After the following text is typed in, the breakpoint is reached, and the debugger interface comes up:

```
First line
Second line
```

Now, put a watch on `FirstTextBlock->NextTextBlock` by selecting Watch... from the Data menu and entering the following:
```
FirstTextBlock->NextTextBlock
```

The watch shows that the next text block is at offset 8C6H. Use the Changed memory global... selection in the Breakpoints menu to instruct Turbo Debugger to stop whenever the value of the `FirstText-

Figure 4. Using inspectors to trace pointer referents.

**Figure 3.** The bug’s telltale. Bugs have an affinity for “garbage” in programs, much as they do in real life.
THE VIEW FROM WITHIN

continued from page 59

acters are being stored to their proper location. Once you've verified that the block is filled properly (if it is), you can set a breakpoint on any modification of the first character of the Text field of the second TextBlock structure in order to catch any statement that might be trashing that block.

We're ready to catch the bug. Run the program by pressing F9, then sit back and watch the results come in.

In this case, you won't have to wait long. The program breaks on the very next line:

TextCount = 0

This means that the following line changed FirstTextBlock->NextTextBlock:

CurrentTextBlock->Text[TextCount] = 0

The watches viewer window agrees, reporting that FirstTextBlock->NextTextBlock has changed to 800H.

How could this possibly have happened? Veteran programmers will spot the problem right away: TextCount points past the end of the Text array, so that the final zero is stored right over the variable that resides immediately after Text; this variable just happens to be NextTextBlock. Since the lower byte of NextTextBlock is forced to zero, NextTextBlock now points not to the next text block, but rather to some random area of memory. Thus, the link between the text blocks is broken. The fix is a simple matter of dimensioning the Text array to BUFFER_LENGTH+1 characters in size.

Let's examine how to narrow the cause of the bug further (as we would have needed to do in this example if we hadn't immediately recognized the nature of the problem). Bring up a watch on TextCount (which reveals that TextCount is 20 at this point in the program), and then bring up an inspector on FirstTextBlock->Text and scroll to the end of the Text array (at this point, the data inspector window appears as shown in Figure 5). The inspector shows that Text is only 20 characters long, and won't let you scroll past element 19; at the same time, the watches viewer window shows that TextCount is 20. To go further still, we could create two dump viewer windows to dump the memory at both FirstTextBlock->Text[TextCount] and FirstTextBlock->NextTextBlock; these windows would show that both variables refer to the same address. That should narrow it down enough for anyone!

WINDOWS WITH A VIEW

The ability to see what happens within a program is by far the largest part of finding any bug. Turbo Debugger offers the power to watch every part of a program in action, from the high-level statements of the host language through the binary representations of large data structures, down to the bare machine registers and memory locations. In a program, many things happen at once—Turbo Debugger's windowed architecture lets you keep an eye on them all. It makes good use of any machine's resources, but it's especially powerful when paired with the 80386 CPU.

Turbo Debugger makes large-scale development with the Turbo languages easier and faster than ever before. The view is the power—look into it.

Michael Abrash is a senior software engineer at Orion Instruments, in Redwood City, California.

Listings may be downloaded from Library 1 of CompuServe forum BPROGB, as DBDEMO.ARC.
A Deal You Can’t Refuse ...700 Functions, 20 Disks, Free Software

**Entelekon’s**

**C Business Library**

or **C STARTER PACKAGE**

<table>
<thead>
<tr>
<th>FREE*</th>
<th>FREE*</th>
<th>FREE*</th>
</tr>
</thead>
<tbody>
<tr>
<td>TURBO C (R) *</td>
<td>QUICKC™</td>
<td>C MATH TOOL BOX</td>
</tr>
<tr>
<td>Borland</td>
<td>Microsoft</td>
<td>89 advanced</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td>math/stat functions</td>
</tr>
</tbody>
</table>

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

What You Get With Entelekon Libraries

- **A C COMPILER** without a good add-on library is like a PC without a keyboard... it won’t do what you want it to do!
- **GAIN C POWER** Add capabilities your compiler library does NOT have. e.g.:
  - New! Qwick Menuing—full 1-2-3 like menus & more
  - Flexible powerful windowing + new Qwick windows
  - Powerful cursor, video and attribute control
  - Time and date arithmetic
  - Sample code and working examples
- **SAVE TIME, TIME, TIME:** man-years on development, calendar months on schedule!
- **SAVE MONEY:** Lowest Cost, Highest Quality Library/Windows Available!
- **SMALLER PROGRAM SIZE:** your application program can be up to 50% smaller!
- **EASY** for beginners! **POWERFUL** for professionals!
- **INSTANT INSTALLATION UTILITY** included!
- **SUPERB DOCUMENTATION:** time saving, helpful, clear, complete, instructive.
- **BUSINESS USERS:** FREE 3 machine site license (C Library & Power Windows)
- **FULL SOURCE CODE** included! **NO ROYALTIES** on products you develop.
- **FREE UTILITY:** To convert Turbo Pascal code to C code.

SAVE MONEY! SAVE TIME! DON'T WAIT!

**Satisfaction Guaranteed**

(Direct from Entelekon only)

**Order Now!**

Call (713) 468-4412

### POWER WINDOWS*

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

Full source code $159.95

**SPECIAL OFFER**

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

### C FUNCTION LIBRARY

**BEST YOU CAN GET**

OVER 500 FUNCTIONS

FULLY TESTED

BETTER FUNCTIONS

Full source code $159.95

### C BUSINESS LIBRARY

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

ALL for $299.95

(A $500.00 VALUE)

### B-TREE LIBRARY & ISAM DRIVER

**POWERFUL DATA MANAGER**

FAST! EASY TO USE!

16.7 MILLION RECORDS/FILE

16.7 MILLION KEYS/FILE

Fixed/Variable length records.

Full source. No royalties $129.95

Multi-User option available.

### C STARTER PACKAGE

INCLUDES C FUNCTION LIBRARY, POWER WINDOWS, SUPERFONTS FOR C

ALL for $199.95

(A $370.00 VALUE)

SINCE 1982

12118 Kimberley, Houston, TX 77024 713-468-4412 VISA/MASTERCARD/CHECK-COD
Turbo C 2.0 goes one better with integrated debugging!

Kent Porter

Turbo C has always offered a great “bang for the buck.” The initial release of Turbo C provided over 350 library functions, an integrated development environment, and a variety of utilities to aid in program development. Turbo C 1.5 (introduced last winter) took a giant step forward with the addition of the BGI graphics library. Now, Turbo C 2.0 is here—and with the improvements to the toolset, including integrated debugging, the language takes another quantum leap.

Since Turbo C’s debugging features have garnered so much interest, this article deals primarily with the Integrated Debugger. First, however, let’s take a quick tour of all of the enhancements in Turbo C 2.0.

MEET THE NEW TURBO C

For convenience, I’ve grouped Turbo C’s new and expanded features into three categories.

Language Enhancements.

• Floating point emulation is faster.
• Long doubles are now supported for greater numeric precision.
• The obsolete ssignal and gsignal functions (which are leftovers from Unix System III) have been dropped in favor of signal and raise. This change improves compatibility with Unix System V.

Expanded Utilities.

• Turbo C 2.0 contains a new .OBJ file cross-reference utility.
• TLINK now generates .COM files from programs that are compiled in the Tiny model.
• MAKE supports autodependencies.

New Tools of the Trade.

• Compiles and links are 10-20 percent faster.
• The Turbo C editor can use EMS for the edit buffer. This can save up to 64K of memory for compiling and running the program.

```
/* FACTOR!.C: Computes factorial of a keyed number */
/* Repeats until user enters 0 */
#include <stdio.h>
main()
    
int value, atoi();
long fact();
char input [6];
do (  
printf ( 11 
Value? 11 );
gets (input);
value = atoi (input);
if (value > 0)  
 printf ("Factorial = \%ld\n", fact (value));  
else  
 puts ("Cannot take factorial of negative number\n");  
) while (value);
long fact (int val)
    
    long result = 0;
if (val)  
result = val * fact (val-1);
return (result);
```
Wildcards can be expanded on the application program's command line.

The integrated environment takes advantage of dual monitors.

The editor supports unindent, block indent/unindent, and optimal fill.

And, of course, Turbo C 2.0 offers interactive debugging.

The most apparent changes are in the integrated environment. The menu bar across the top of the screen is a little more crowded by the addition of a Break/watch selection. Every selection (except Edit) now has an associated pull-down menu for greater control over various aspects of the environment and the programming/debugging session (more on this presently).

Overall, the environment—while more comprehensive and flexible—retains the same general look and feel of Turbo C's previous generations. Unlike the transition from Turbo Pascal 3.0 to 4.0, there's no "culture shock" in moving to Turbo C 2.0. But there is plenty to learn, so let's take a look.

INTEGRATED DEBUGGING

It's tempting to say that Turbo C 2.0 has added a debugger, but the fairer statement is that debugging has been integrated into the Turbo C environment. Unlike most debugging packages, Turbo C's Debugger is not a standalone utility. Rather, it's an integral part of the environment, seamlessly folded into the process of writing, making, and testing programs.

During a debugging session, for example, you can edit and remake the source code to fix errors, then resume the debugging process. The recompile doesn't lose track of breakpoints and watches that were set earlier; they remain in effect even if source code is added or removed. This allows you to work out the bugs systematically, without disrupting the natural workflow.

The power of the C language has its price: no matter how skilled the programmer, it's almost impossible to write a C program that runs right the first time. The language's flexibility and sometimes obscure syntax encourage new techniques, and this experimentation inevitably introduces bugs. Therefore, Turbo C and integrated debugging go hand-in-glove.

THE HUNT

To see how a debugging session proceeds, let's develop and debug a simple program to compute the factorial of a number. To refresh your memory (in case your algebra has gotten rusty), a factorial is the series product of a value. For example, the factorial of 5 (written 5! in mathematical notation) is computed as $1 \times 2 \times 3 \times 4 \times 5$, which equals 120.

FACTORYLC (Listing 1) has a loop in main that repeatedly asks for a value and prints the value's factorial until the user types 0. FACTORYLC uses the recursive function fact to solve for the factorial. The program as listed contains a bug; we'll hunt the bug down in order to examine a few of the Debugger's features.

The process of editing and making a program in the development environment has not changed from Turbo C 1.5. The process of running the program, however, is a little different. The Alt-R command now produces a menu that includes some debugger controls. You can circumvent the menu and run the program by using the new hotkey, Ctrl-F9.

When the program is run, it returns 0 as the factorial of any number. Thus, the fact function appears to contain a bug.

To prepare a program for debugging, an environmental condition must be set: toggle Source debugging (located on Turbo C 2.0's revised Debug menu) to On (see Figure 1). Also, since you're dealing with a recursive function, you might want to check the call stack to make sure that the recursion is working properly. To do so, set Standard stack frame On from the Options/Compiler/Code generation menu. Now, remake the program and you're ready to go.

continued on page 64
THE HUNT
continued from page 63

THE PROBLEM
The program runs normally until the fact function is called from within printf. At that point, stop the program and observe what's going on. Set an automatic stop (called a breakpoint) by moving the cursor to the statement if(value) and pressing Ctrl-F8. (As an alternative approach, select Toggle breakpoint from the Break/watch menu.) Notice that the source line is highlighted to indicate that it's a breakpoint. Now, run the program.

The new “smart screen” option, which is on by default, automatically swaps between the edit screen and the program display. Whenever screen I/O occurs, the program display appears. Consequently, the program runs normally and asks for and receives values until it hits the breakpoint. The edit screen then reappears. A bar, called the execution bar, highlights the source line where the program stopped.

One way to proceed is to single-step, executing one line at a time, and watch what happens. The Turbo C 2.0 Debugger has two single-step hot keys. F7 activates Trace, which single-steps through all function calls. F8 activates Step, which “steps over” functions, executing them but not tracing their execution. In this case, since the bug is probably in the function, select F7.

However, there's an easier way to locate the source of the problem. Whatever the value you keyed, fact calls itself that number of times before it encounters the return statement. You can save a lot of single-stepping by setting a temporary breakpoint at the return. To do so, position the cursor on the return statement, then select Go to cursor from the Run menu. The program halts when it reaches the line where the cursor is positioned.

Now you can examine the state

Figure 1. Examining the call stack.

Figure 2. The evaluation window.
of affairs. First, check the call stack to see if recursion is working correctly (select Call stack on the Debug menu). Assuming that the keyed value is 3, this produces the display shown in Figure 1. Note that each invocation of fact is passed an argument that is one less than its predecessor. This is as it should be. So where’s the bug?

The program consistently reports 0 as the factorial. Examine the argument of return to see what the fact function is returning. To do so, place the cursor on result, then press Ctrl-F4 (or select Evaluate from the Debug menu).

Ctrl-F4 pops up the expression evaluation window shown in Figure 2. A number of things can be done in this window, such as typing expressions using C syntax and viewing the results, or changing the value of a variable. In this case, you simply want to see result’s value. The Debugger copies the variable name from the cursor position into the evaluate field. Press Enter and the value appears in the middle box. The value that displays is zero, which explains why the program returns incorrect results.

THinking IT THROUGH

A debugger is a tool for interactively controlling and watching the execution of a program, and for examining the program’s internal conditions. The debugger can tell you what’s happening, but it can’t think for you. The question is, “Why is the value returned in result equal to 0?” To find the answer, you have to inspect the algorithm that yields this value.

In the fact function, result is initialized to 0. Then, if the argument has a nonzero value, result is assigned the value of the expression, which triggers a recursive call. When the passed argument reaches 0, the if statement fails and the return statement sends back the original value of result (0). This value becomes one of the multipliers in the factorial series:

$$0 \times 1 \times 2 \times 3 \times \ldots \times N$$

Zero times anything else is zero. Consequently, the bug is the result of flawed logic; result should be initialized to 1 so that the function cannot return 0.

To fix the program, change the initializer, remake FACTORIAL, and test the program again. This time the program returns the correct answer, and the bug is fixed.

STICKY BREAKPOINTS

An interesting thing happens when you remake a program that contains set breakpoints; the breakpoints are retained, even if source lines are added or removed. The Turbo C 2.0 Debugger tracks the physical source lines that have breakpoints. When a program is complex and buggy, this automatic tracking process saves you the hassle of reestablishing breakpoints every time you fix and retry. These “sticky breakpoints” are one of the great advantages of having the Debugger integrated into the editing environment, rather than designed as a separate utility.

WATCHING VARIABLES

Another feature of the Turbo C 2.0 Debugger is the ability to watch one or more variables in a window while the program executes. This is particularly valuable when a loop counter goes berserk, or when some variable appears to have been corrupted for reasons unknown. The easiest way to set watches on variables is to position the cursor on some occurrence of the variable to be watched. For each variable, press Ctrl-F7 (or use the Add watch selection in the Break/watch menu). The watch window appears at the bottom of the display, similar to the error warning message window that appears during compiles. Also, F5 can be used to toggle between full (zoomed) and split-screen mode, and F6 switches between full-screen edit and watch windows. In split-screen mode, the watch window grows upward dynamically to accommodate the number of watched variables.

Local variables are visible only while control resides within the routine that owns them. Therefore, as execution proceeds from one routine to another, the auto variables located outside of the current routine become undefined. The watch window only shows values for the variables that it sees. (This explains the “Unde-

continued on page 66

THE WINDOW BOX

A windowing toolbox for C programmers.

Enhance the beauty of your C applications with THE WINDOW BOX.

ADD SOME PIZAZZ!

THE WINDOW BOX lets you ELECTRIFY your programs with pop-up windows, pull-down menus with highlight bar selection, and context sensitive help. Watch your screen go blank when your program is idle. Assign functions to the function keys. Much more!

ADD SOME POWER!

Read many fields with one operation. Data entry windows offer many formats, complete cursor navigation, and let you tie verification functions to any field. Use scrolling and text-editing windows, too. Print a window, not necessarily the whole screen. (Super for mailing labels!)

FAST AND COMPATIBLE!

Stores directly in video RAM. If your environment prohibits this, we can store in the alternate display pages, or use DOS calls exclusively.

SOURCE CODE PROVIDED!

Contains no assembler code! Only standard C code. See how things work. Change how things work. Compatible with all major C compilers. Requires MS-DOS/PC-DOS.

REASONABLE PRICE!

And no royalties. Only $49.50 including shipping and tax. Including source code! Overseas add $5 and we will Air Mail. US add $10 and we will overnight.

SATISFACTION GUARANTEED, or return it in 30 days for a full refund.

Mastercard/Visa: Call 412-487-4282.
Or, send checks (U.S. funds) to:

Vertical Horizons Software
113 Lingay Drive
Glemshaw, PA 15116
fined symbol" message associated with value in Figure 3. The execution bar shows that control is currently in fact; value, however, is local to main.) Global variables are, of course, visible from anywhere in the program.

The Break/watch menu provides options for changing and deleting watches, as well as for removing all watches at the same time. Similar capabilities are available for controlling breakpoints.

If you want to watch certain elements within an array, you can use a watch editor feature called repeat counts. For example, to watch elements 4 through 8 of an array called num, specify the elements as the following:

```c
num[4],5
```

This statement tells the Debugger to watch the five elements of num, starting at subscript 4. The watch window then contains individual entries for each element, building upward from num[4].

If you're working with a large application that involves many source modules, you can qualify variable names from other modules in order to watch their values. This is true even if the variable comes from a module outside of the module you're currently debugging, and the variable is local to a specific function. The general form of the syntax for the watch editor is:

```
.module.function.variable
```

The module must be made with the debugging options on. However, the module doesn't have to be in the editing environment in order to watch its indicated variable while the program runs.

SMART SCREEN SWITCHING

With many debuggers, the process of debugging graphics applications is often tricky (and sometimes impossible)—but not with the Debugger in Turbo C 2.0. The Turbo C 2.0 Debugger operates in text mode, while the program display is in graphics mode. The smart screen management built into the Debugger lets you switch back and forth readily. The program display can be viewed at any time, regardless of its mode, by pressing Alt-F5. Pressing any key returns you to the edit/debug screen.

BAILING OUT

You may be wondering what happens if your program hangs the system. If your program is running under the debugging environment, you can usually (but not always) escape by pressing Ctrl-Break, which returns you to the edit/debug mode. The Program reset option from the Run menu (or Ctrl-F2) reinitializes the program as if it had never been run before. With this option, you can start over and utilize breakpoints, watches, and other Debugger tools to pinpoint the problem.

Once you've perfected your program by using the Integrated Debugger, simply turn off the environment's debugging option and recompile. The debugging options insert some additional information into the end of the .EXE file; the final application will run with this debug information still in place, but the .EXE file size will be smaller without it. This is in contrast to some debuggers that place int calls in your code, forcing you to remove the debug information before running the final application.

THE VELVET GLOVE

The essence of Turbo C 2.0 lies in its enhancements to the toolset—primarily in its addition of a powerful Integrated Debugger that lets you test, fix, remake, and retest until your program works like it's supposed to. Almost by definition, C is a language that encourages both extraordinary power and its accompanying bugs. Turbo C 2.0 fits integrated debugging over the hand of C like a velvet glove—and hurdles the last obstacle to true programmer productivity.

Kent Porter is a frequent contributor to TURBO TECHNIX. His next book, Stretching Turbo C, is due to be released this fall.

Listings may be downloaded from Library 1 of CompuServe forum BPROGB, as FACTRL.ARC.
FLOATING POINT: 
THE SECOND WAVE

When a number isn’t exactly a number, Turbo C 2.0 can handle it.

Roger Schlafly

Turbo C is very good at handling numbers that represent quantities in the real world, as I explained in "Floating Point in Turbo C," TURBO TECHNIX, January/February, 1988. Turbo C 2.0 now enhances floating point support with better precision, better exception handling, and a means of dealing with numbers that aren’t exactly numbers.

Turbo C 2.0 fully supports the IEEE standard for computer arithmetic (IEEE standards 754 and 854) when used with an 8087 math coprocessor. (If an 8087 is not present and emulation must be used instead, certain exceptions exist that are primarily related to denormals, as explained in my earlier article on floating point.)

GREATER PRECISION

There is a very long road in California called El Camino Real. Near Silicon Valley, FORTRAN programmers call it El Camino DOUBLE PRECISION. Lisp programmers, who claim it originally extended all the way to Mexico City, call it El Camino Bignum. Turbo Pascal programmers call it El Camino Extended. Turbo C programmers can now call it El Camino long double, in honor of Turbo C’s new data type, which is called the “long double.” This type was created by the ANSI C committee (X3J11) to accommodate the IEEE extended precision. Quite simply, long doubles are very long reals.

Turbo C 1.0 and 1.5 actually allowed the long double syntax, but long doubles were identical to doubles. In Turbo C 2.0, the long double is a 10-byte data type, whereas floats and doubles are 4 and 8 bytes in size, respectively (as in earlier releases of Turbo C). Turbo C automatically performs conversions among these types.

Long doubles are just as fast as floats and doubles. The only penalty is the additional data space required by long doubles. The following examples show the usefulness of long doubles.

Increased precision. Suppose you want to take the sum of some number of real values in a vector. Such calculations are prone to roundoff error, which is why numerical analysts use tricks to carefully reorder the numbers in order to minimize the loss in precision. A simpler alternative is to use long double precision to compute the sum, as shown in the following example:

```c
double vect_sum(int n, double x[n]) {
    int i;
    long double sum = 0;
    for (i = 0; i < n; ++i)
        sum += x[i];
    return sum;
}
```

Avoiding overflow and underflow. Turbo C has a hypot() library function, which returns the hypotenuse of the right triangle given the two remaining sides. This function is quite useful in calculating the modulus of complex numbers, in polar coordinate conversions, and in many other situations. If you were to create such a function, you would probably write it as follows:

```c
#include <math.h>

double hypot(double x, double y) {
    return sqrt(x*x + y*y);
}
```

The trouble with defining hypot() in this way is that it's susceptible to underflow or overflow of intermediate results. For example, if x = 3e200 and y = 4e200, then hypot() overflows even though the correct answer is 5e200, which is nowhere near the overflow threshold of 1.8e308. Worse yet, hypot(3e-200, 4e-200) underflows and returns 0, when it should return 5e-200. (This is worse because underflows are ignored by Turbo C Runtime code, and you'll have no idea that the function underwent a complete loss of precision unless you explicitly check for underflows.)

Now, examine the following hypot() function:

continued on page 68
FLOATING POINT
continued from page 67

double hypot (double x, double y)
{
    long double z = x*x + y*y;
asmandbyte ptr z
    asm FSQRT
}

This hypot() function is similar to the hypot() function included with Turbo C, except that the Turbo C library function calls matherr() if an overflow or underflow occurs in the result (i.e., if the resulting long double is outside the limits allowed for doubles). Thus, the hypot() function shown above avoids the difficulties that occur with overflows and underflows.

Some inline assembler was used here because the Turbo C sqrt() function requires a double argument, rather than a long double argument. The 8087 operation FSQRT, however, returns a result of any desired floating point type. The process of returning a long double is the same as that of returning a double or a float, and the 8087 supports all three types. The results are returned on top of the 8087 stack, and the 8087 chip automatically performs the required conversion when a number is unloaded from the 8087 stack.

READING AND PRINTING LONG DOUBLES
Long doubles can be read with scanf() or printed with printf() by using the L modifier for the usual floating point conversion specifications. For example, the following code reads \( \pi \) from a string and prints it to 18 decimals:

```c
#include <stdio.h>
#include <math.h>
define STRINGIZE(p) #p
long double pi;
sscanf(STRINGIZE(M_PI),"%Lf",&pi);
printf("pi = \%21.18Lf\n",pi);
```

PASSING PARAMETERS
The choice of using three floating point data types complicates parameter passing conventions, and makes it all the more likely that a function will be called with the wrong type parameter. I recommend using ANSI C prototypes. Be extra careful with functions that cannot be adequately prototyped, such as printf().

When passing a floating point expression without a prototype, Turbo C passes either a double or a long double, depending upon the longest type in the expression. ANSI C stipulates that the L suffix can be added in order to tell the compiler to consider a constant to be a long double. This step is demonstrated in the following example:

```c
printf("double constant = \%g\n",3.2);
printf("long double constant = \%g\n",3.2L);
```

Some ANSI C compilers may also require the L suffix in order to achieve full accuracy in situations such as the following:

```c
long double x, y;
y = 3.2L * x;
```

With Turbo C, however, an L is superfluous in this situation. Turbo C automatically stores such constants to long double precision.

PRECISION LOSS
Let's consider a simple example of an appropriate use of high precision. Suppose you want to compute \( 1/3 \) to the power of \( n \) for various positive integers \( n \), and the answer must be accurate to about three decimal digits. Several avenues of approach are possible:

• Method 1. Call the Turbo C library function pow(3,-n).
• Method 2. Recursively calculate \( x[n] = 3^{-n} \):
  \[ x[0] = 1 \]
  \[ x[n] = x[n-1] / 3 \quad \text{for} \ n > 0 \]
• Method 3. Recursively calculate \( x[n] = 3^{-n} \):
  \[ x[0] = 1 \]
  \[ x[n] = (31 * x[n-1] - 10 * x[n-2]) / 3 \quad \text{for} \ n > 1 \]

Method 1 is both the most accurate and the preferred method. It should yield a result that is accurate to full double precision.

Method 2 is the most straightforward approach if no library function is available. While method 2 is quite accurate for small values of \( n \), each operation causes a roundoff error. Still, roundoff errors tend to average out, and two or three significant digits are lost only when \( n \) gets to be about 600 (which is near the range limits for double precision anyway).

Computing the powers with long double precision is a reasonable approach, and provides the reassurance that the answer is as accurate as double precision allows.

Method 3 is a rather silly way to compute powers, but it's mathematically correct and similar to the methods that are frequently encountered in practice. Unfortunately, this method is almost completely useless. In single precision, it delivers an answer that is accurate to three digits only if \( n \leq 4 \). Larger values cause the method to yield garbage. Method 3 returns answers for some higher values of \( n \) by using high precision, but the gain is minimal. Double precision only works for \( n \leq 10 \), and long double precision only delivers three-digit accuracy when \( n \leq 12 \).

The lesson here is that most good numerical algorithms are stable with respect to roundoff error, and that they deliver much more precision than could ever be used anyway. Poor numerical algorithms can lose so much precision that they're often useless, even when plenty of precision is available in the variables.

DEALING WITH THE INFINITE
The enemies of numerical analysts are roundoff error, overflow, underflow, and division by zero. All of these situations involve numeric values that cannot be fully expressed in a finite number of bits. These anomalous values can infiltrate your program and create havoc. The usual countermeasure used by Turbo C and other C compilers is a form of Mutual Assured Destruction (MAD). If you give Turbo C a floating point expression that blows up, it retaliates by nuking your program, which abruptly terminates with a message such as:

Floating point error: Overflow.

The alternative is to negotiate your own INF treaty. The idea is to come to terms with the infinite, and learn to live with it.
The numbers won't get out of control as long as the 8087 control word is set properly. The control word can be set to mask numeric exceptions via a call to _control87() as follows:

```
#include <float.h>
_control87(MCW_EM-EM_DENORMAL, MCW_EM);
```

The second argument is the mask that tells _control87() which bits are being changed in the 8087 control word. The first argument specifies the new bit values that correspond to the exceptions that are to be masked. The invocation shown above masks all of the exceptions except the denormal exception. Denormal exceptions are largely harmless, because the Turbo C 2.0 Runtime Library contains a denormal exception handler.

The creation of denormals can be regarded as mildly criminal behavior on the part of the 8087 chip. In dealing with denormals, the 8087 tries to get something for nothing. A denormal is a number so small that it should be zero, but the 8087 gives the number a probationary nonzero status. This petty offense probably won’t bother your program. However, an annoying feature of the 8087 chip is that it doesn’t have much of a rehabilitation program for denormals. If a denormal value increases beyond a certain point, the denormal can reenter the range of the normals—in doing so, however, the value does not become normal. Instead, it becomes unnormals. Unnormals are like convicted felons who have not been rehabilitated. They are not normal, they corrupt whatever values they touch, and they cannot even be stored in float or double format. You don’t want these animals in your neighborhood.

Fortunately, Turbo C 2.0 goes the 8087 one better with a denormal exception handler that normalizes denormals before they mutate into unnormals. Turbo C normalizes denormals automatically when the denormal exception is left unmasked. Note that if an 80386 machine has an 80387, it doesn’t matter whether the denormal exception is masked or not. The 8087 has a built-in normalizer, and doesn’t generate unnormals at all.

**THREE NEW "NUMBERS"**

All of the other exceptions may be safely masked (and, in fact, that approach may be preferred for bulletproof programs). With denormals properly normalized, the IEEE standard allows every arithmetic operation to have a defined result. The standard accomplishes this end by adding the following three new numbers:

```
+INF plus infinity
-INF minus infinity
NAN not-a-number
```

**Two infinities.** Having two infinities is new to Turbo C 2.0. Early drafts of the IEEE standard called for two infinity modes—"projective" and "affine." While the 8087 supports both, it defaults to projective infinity; Turbo C 1.0 and 1.5 only supported projective infinity.

---

The numbers won’t get out of control as long as the 8087 control word is set properly. The control word can be set to mask numeric exceptions via a call to _control87() as follows:

```
#include <float.h>
_control87(MCW_EM-EM_DENORMAL, MCW_EM);
```

The second argument is the mask that tells _control87() which bits are being changed in the 8087 control word. The first argument specifies the new bit values that correspond to the exceptions that are to be masked. The invocation shown above masks all of the exceptions except the denormal exception. Denormal exceptions are largely harmless, because the Turbo C 2.0 Runtime Library contains a denormal exception handler.

The creation of denormals can be regarded as mildly criminal behavior on the part of the 8087 chip. In dealing with denormals, the 8087 tries to get something for nothing. A denormal is a number so small that it should be zero, but the 8087 gives the number a probationary nonzero status. This petty offense probably won’t bother your program. However, an annoying feature of the 8087 chip is that it doesn’t have much of a rehabilitation program for denormals. If a denormal value increases beyond a certain point, the denormal can reenter the range of the normals—in doing so, however, the value does not become normal. Instead, it becomes unnormals. Unnormals are like convicted felons who have not been rehabilitated. They are not normal, they corrupt whatever values they touch, and they cannot even be stored in float or double format. You don’t want these animals in your neighborhood.

Fortunately, Turbo C 2.0 goes the 8087 one better with a denormal exception handler that normalizes denormals before they mutate into unnormals. Turbo C normalizes denormals automatically when the denormal exception is left unmasked. Note that if an 80386 machine has an 80387, it doesn’t matter whether the denormal exception is masked or not. The 8087 has a built-in normalizer, and doesn’t generate unnormals at all.

**THREE NEW "NUMBERS"**

All of the other exceptions may be safely masked (and, in fact, that approach may be preferred for bulletproof programs). With denormals properly normalized, the IEEE standard allows every arithmetic operation to have a defined result. The standard accomplishes this end by adding the following three new numbers:

```
+INF plus infinity
-INF minus infinity
NAN not-a-number
```

**Two infinities.** Having two infinities is new to Turbo C 2.0. Early drafts of the IEEE standard called for two infinity modes—"projective" and "affine." While the 8087 supports both, it defaults to projective infinity; Turbo C 1.0 and 1.5 only supported projective infinity.

---

The creation of denormals can be regarded as mildly criminal behavior on the part of the 8087 chip. In dealing with denormals, the 8087 tries to get something for nothing. A denormal is a number so small that it should be zero, but the 8087 gives the number a probationary nonzero status. This petty offense probably won’t bother your program. However, an annoying feature of the 8087 chip is that it doesn’t have much of a rehabilitation program for denormals. If a denormal value increases beyond a certain point, the denormal can reenter the range of the normals—in doing so, however, the value does not become normal. Instead, it becomes unnormals. Unnormals are like convicted felons who have not been rehabilitated. They are not normal, they corrupt whatever values they touch, and they cannot even be stored in float or double format. You don’t want these animals in your neighborhood.

Fortunately, Turbo C 2.0 goes the 8087 one better with a denormal exception handler that normalizes denormals before they mutate into unnormals. Turbo C normalizes denormals automatically when the denormal exception is left unmasked. Note that if an 80386 machine has an 80387, it doesn’t matter whether the denormal exception is masked or not. The 8087 has a built-in normalizer, and doesn’t generate unnormals at all.

**THREE NEW "NUMBERS"**

All of the other exceptions may be safely masked (and, in fact, that approach may be preferred for bulletproof programs). With denormals properly normalized, the IEEE standard allows every arithmetic operation to have a defined result. The standard accomplishes this end by adding the following three new numbers:

```
+INF plus infinity
-INF minus infinity
NAN not-a-number
```

**Two infinities.** Having two infinities is new to Turbo C 2.0. Early drafts of the IEEE standard called for two infinity modes—"projective" and "affine." While the 8087 supports both, it defaults to projective infinity; Turbo C 1.0 and 1.5 only supported projective infinity.

---

**C-Index™ for Turbo C®**

**Database Toolkit**

Your Turbo C programs will really fly with fast C-Index access. C-Index is ready-to-use with Turbo C, right out of the box. No need to compile or change the source code. Nine simple function calls give you the power of fast B+Tree indexing and automatic variable-length records. C-Index makes programming easy with a friendly Application Program Interface. The excellent documentation includes extensive examples and an interactive tutorial program. Order now and we will send you a free database application, along with five additional utilities and fully commented source code for everything.

C-Index is the database toolkit that has been successfully used in commercial applications for the mass market, such as Prime Time. C-Index is the library of choice for major financial institutions that have used it for sensitive monetary transfers. Wayne Ratliff, author of dBase II and dBase III, thought C-Index was so "terrific" that he uses it as the foundation for his new Emerald Bay database system. Just link C-Index into your programs and you can be a database superstar too.

Every day thousands of people depend on C-Index to manage their data. Once you discover the speed and power of C-Index, you won’t write a program without it.

- Fast B+ Tree Indexing
- Single and Multi-user Access
- Works with any standard MSDOS LAN
- 170 page manual
- Proven Reliability
- Variable Length Records
- Complete Random and Sequential Access
- Data and Indexes in same file
- Multiple Record Formats per file
- Proven Reliability
- No Application Royalties

"I heartily recommend this package."

*Dr. Dobbs Examinining Room, June '88*

**C-Index™ for Turbo C®**

**$99**

Single/Multi-User version with complete source code.

Order now at this introductory price and receive the free database application.

**Trio Systems**

213/394-0796
2210 Wilshire Bl. Suite 289 Santa Monica, CA 90403
While the difference between the previous cases is that \texttt{NAN} will suffice for this discussion, 8087 generates only one, and that \texttt{8087()}, as shown in the following example:

```c
#include <float.h>
#include <math.h>

_double x = 1e+200 * 1e-200;
returns x = 0
```

Essentially, you have to divide by 0 in order to see which zero is present. The rule is \(1/0 = +\text{INF}\), and \(1/0 = -\text{INF}\).

The 8087 and 80287 (but not the 80387) support pseudozeros, which can occur when unnormals multiply and the product gets too close to (true) zero. Pseudozeros are also equal to zero, but they retain the taint of the unnormals that produced them. With the Turbo C 2.0 denormal handler preventing unnormals from occurring, pseudozeros shouldn't appear either.

Arithmetic can also be performed on these special numbers, as the following example demonstrates:

```
+INF + 5 = +INF
1/+INF = 0
+INF/+INF = \text{NAN}
5 * \text{NAN} = \text{NAN}
```

The constants +\text{INF}, -\text{INF}, and \text{NAN} as used in this example are not predefined in Turbo C. However, you can easily create constants that have these special numbers as their values by remembering that Turbo C (like most C compilers) evaluates constant expressions at compile time. Thus, \text{INF} and \text{NAN} can be created as constants with the following definitions:

```c
#define INF (1./0.)
#define NAN (0./0.)
```

What are all these crazy numbers good for? When performing computations on a computer, it's very important to have a closed arithmetic system. A closed arithmetic system means that every arithmetic operation yields a quantity that is somehow representable within the system. If a long sequence of operations is performed and the result is a \text{NAN}, then a mathematically invalid operation was performed somewhere along the way. Since the result of every expression—including an invalid result—is represented, the Runtime Library never has to throw up its hands in despair and crash.

Another use for \text{NAN} is in creating "uninitialized" data. In C, all uninitialized data are initialized with 0 at startup. (That's +0, not -0.) Occasionally, a variable must truly be recognizable as uninitialized through some unique nonzero value. A constant that is defined as a \text{NAN} (as shown earlier) can be used to initialize the variable with the value \text{NAN}. Since any operation on a \text{NAN} yields a \text{NAN}, a faulty answer won't occur when calculations are accidentally performed with uninitialized data.

**RECOGNIZING INF AND NAN**

Since many situations require special treatment of \text{INF} and \text{NAN}, it's necessary to be able to recognize these values when they occur in your program. For example, if a function returns a \text{NAN}, you may need to know immediately that the function failed.

Turbo C 2.0 handles +\text{INF} and -\text{INF} correctly in comparisons. The following method can determine if \(x\) equals \text{-INF}:

```c
#define INF (1./0.)
if (x == -INF) ...
```

Unfortunately, Turbo C does not support comparisons between floating point values and \text{NAN}s. This support is not present in Turbo C for two reasons. First, ANSI C does not require it; and second, due to the way that the Intel CPU and coprocessor chips work, this support could not be added without slowing down every floating point compare operation. Therefore, unless the invalid operation exception is masked, a comparison that involves \text{NAN}s generates the exception and ter-
If whether or not a number is a NaN, inconsistent results, as shown in Table 1. Therefore, I recommend using a procedural test, such as the ieee_type() function given in Listing 1, in order to determine whether or not a number is a NaN.

The function ieee_type() in IEEE_TYPE.C (Listing 1) identifies numbers as belonging to one of four categories: normal, +INF, -INF, and NaN. Zeros, normals, unnormals, and denormals are all classified as normals for simplicity. As long as a prototype can be used before ieee_type() is called, this function can be used for classifying float, double, or long double arguments. Because ieee_type() requires that long doubles be in the 10-byte format, this function will not work with Turbo C versions that are earlier than 2.0.

INFINITE PHILOSOPHIES

Different people have different attitudes towards floating point overflows. The traditional (and common) view is that debugged programs don't overflow. On many mainframes, this may truly be the case, because the hardware may prevent the program from continuing after an overflow occurs. Therefore, your program had better be debugged. In deference to this view, the default Turbo C behavior is to terminate the program in the event of an overflow.

If you share this traditional view, Turbo C 2.0 has some new features to help you. You can trap the overflow, and even though you may consider the overflow to be fatal, your program can print some useful diagnostics before it dies.

The more progressive view is to not discriminate against infinities and NaNs, and to not trap any floating point exceptions. This view seems more appropriate for C programs. After all, C is the language that assumes that the programmer knows what to do and then lets the programmer do it. Currently, Turbo C 2.0 library functions such as exp() will not return a value larger than 1.8e+308. Tradition requires Turbo C to return representable numbers, and 1.8e+308 is the largest such number. If the answer should be larger, then matherr() is called to notify the programmer of the error. However, the new IEEE standard has caused people to become more broadminded about the definition of a number—now a number can be INF, or even NaN. The latest ANSI C draft allows these special numbers to be considered representable.

In keeping with this trend, some future Turbo C release will probably assume that C programmers are ready to play as fast and loose with floating point numbers as they currently do with pointers and other data types. INFs and NaNs will be declared representable numbers, just as the ANSI C draft allows. When exp(1e10) is called, it will just return +INF, and possibly not even call matherr(). A call to sqrt(-1) might just return NaN.

In the meantime, the same thing can be accomplished under Turbo C 2.0 by replacing the library's matherr() with a matherr() of your own devising, and then modifying the variable _huge_dble. _huge_dble occurs in <math.h> in the following context:

```
#define HUGE_VAL  _huge_dble
```

The purpose of _huge_dble is to contain the largest representable value for programs that need this variable. The library functions that need this value must simply reference _huge_dble. The default is 1.8e+308. This value can also be defined as +INF. (Turbo C 1.0 and 1.5 used a function called _huge_val() for HUGE_VAL.)

If you include MATHERR.C (Listing 2) in your program, and call startfp() when the program first runs, then all exceptions other than the denormal exception are masked, all library errors are ignored, and the library functions return INF under appropriate circumstances.

CONTINUED FRACTIONS

Here is a typical example where arithmetic with infinities is useful, even when a finite result is being calculated. Consider the following formula:

\[ \tan x = \frac{x}{1 - \frac{x^3}{3 - \frac{x^5}{5 - \frac{x^7}{7 - \ddots}}}} \]

The formula converges to \( \tan(x) \) for any value of \( x \). This type of formula is called a continued fraction, and can be thought of as being analogous to a power series. In this case, the continued fraction can be more useful for approximating the tangent of \( x \) since the formula converges everywhere, and converges more rapidly than the power series. (The power series is only good for \( |x| < \pi/2 \), as the tangent function has a singularity at \( \pi/2 \).)

The code in TAN.C (Listing 3) uses long doubles for intermediate results. The calculation is likely to lose only a couple of bits of long double precision due to roundoff error, which won't matter once the calculation is rounded again to double precision. Thus, an answer will be accurate to the limits of double precision.

The nice thing about this example is that infinities can occur in the calculation, yet it always gives the correct finite answer if

\[ x = \text{NAN} \quad \text{always TRUE} \]
\[ x = \text{NAN} \quad \text{always FALSE} \]
\[ x < \text{NAN} \quad \text{unreliable} \]
\[ x <= \text{NAN} \quad \text{unreliable} \]

Assume this definition for the above comparisons: #define NAN(0.10 .)

Table 1. Results of comparisons involving NaN.

<table>
<thead>
<tr>
<th>Comparison</th>
<th>True</th>
<th>False</th>
<th>Reliable</th>
</tr>
</thead>
<tbody>
<tr>
<td>x != NAN</td>
<td>always TRUE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x != NAN</td>
<td>always FALSE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x &lt; NAN</td>
<td>unreliable</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x &lt;= NAN</td>
<td>unreliable</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CONTINUED ON PAGE 72
FLOATING POINT
continued from page 71

enough terms are used. In fact, because of the way the calculation is coded, it divides by 0 the first time through the loop!

Not all calculations are so fortunate. If a calculation produces an infinity, there’s the risk that a 0*INF, INF-INF, or INF/INF might produce a NAN. (0/0 also produces a NAN) Any calculation that depends upon a NAN yields a NAN. If the introduction of a NAN into a calculation is a possibility, then the calculation must check the result to see if the result is a NaN, or if the invalid bit was set in _status87(). For example, consider the following expression:

```
if(_status87() & (SW_INVALID | SW_ZERODIVIDE | SW_OVERFLOW))
```

If this expression evaluates to a nonzero value, then an invalid operation, a divide by zero, or an overflow must have occurred after either the start of the program or the last call to _clear87() or _preset(). Arithmetic operations on NANS are considered invalid operations.

Note that approx_tan() treats \( x = 0 \) as a special case. If this were not so, then approx_tan() would encounter 0/0 and return a NAN. A better fix for this problem is to initialize y with some nonzero value.

USING signal() TO TRAP EXCEPTIONS

ANSI C specifies a portable way to trap floating point exceptions. This method involves using the signal() function to install a floating point exception handler. Turbo C 2.0 fully supports this scheme, as shown in SIGTEST.C (Listing 4).

Call signal(SIGFPE, fphandler) to install the handler, and call setjmp(jump1) before doing any floating point calculations. Every time the handler is triggered, it must reinstall itself, because each signal causes the main program to revert to its default signal handler. (This is an old UNIX quirk.)

Following are the reasonable alternatives for a floating point exception handler. Items 3 and 4 require a physical coprocessor.

1. Print a suitable error message and exit (this is the process performed by the default handler). A program that wants to do the same thing may still wish to replace the handler in order to do some additional housecleaning or to print a more informative error message.

2. Perform a long jump to a safe place in the program. If this is done, the program must pay attention to all of the usual hazards of long jumps. In addition, the program should call _fpreset() to reset the coprocessor or emulator. (The library function _fpreset() resets the coprocessor. If for some reason a special value is maintained for the 8087 control word, then the control word must be reset to that special value because _fpreset() installs the default Turbo C control word.) Since interrupts occur asynchronously, there is more than the usual danger here that an interrupt will happen while the code is in an inconsistent state.

3. Set a flag and continue. As with case 3, most programs may prefer the simpler strategy of masking the exceptions. The occurrence of the exception can still be detected by examining the status word with _status87(). The status word can then be cleared with _clear87().

4. Attempt to analyze the damage and repair it. This is nearly impossible, because the 8087 is a very complex chip with many instructions, data types, registers, and special cases. However, the Turbo C Runtime Library Source does include a C interface to handle floating point exceptions, in which some additional information is provided.

Anyone who traps exceptions should be aware that some versions of DOS 3.2 contain a rather nasty bug, where DOS only allows eight exceptions before it halts the machine. Microsoft has a patch that fixes the problem. If you are using DOS 3.2 and a coprocessor, I strongly recommend that you either obtain the patch or else switch to DOS 3.1 or 3.3.

R.I.P. UNARY PLUS

As described in my earlier article on Turbo C floating point, the ANSI C draft had proposed a unary plus sign to force expressions to be evaluated in a particular order. This was needed by numerical analysts because C compilers traditionally reserve the right to ignore parentheses in an expression such as the following:

\[ x = (y - 2.1) + z; \]

Turbo C 1.0 and 1.5 supported a unary plus to force a particular order of expression evaluation. At the ANSI C meeting in December 1987, however, the decision was made that compilers should always evaluate parenthesized expressions first, unless it’s provable that the expression evaluation order doesn’t make any difference. Turbo C 2.0 supports this change. Thus, the unary plus is obsolete in Turbo C, yet still supported.

MAKING POINTS

The C language is becoming increasingly popular for numerical work. Its old defects (such as re-arranging parenthesized expressions and not type-checking function arguments) are no longer present. Turbo C now has features that FORTRAN programmers can only dream about: extended precision, trappable exceptions, INF, and NAN. These, along with all of the usual advantages of C (portability, preprocessor, dynamic memory, convenient data types, and control structures) and the advantages of Turbo C (speed, integrated environment, third-party support) make Turbo C the language of choice for nearly all numerical tasks.

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 Library 1 of CompuServe forum BPROGB, as CFLT20.ARC.
LISTING 1: IEEETYPE.C

```c
enum ieee {
  ieee_normal,
  ieee_plNF,
  ieee_mlNF,
  ieee_NAN,
};
enum ieee ieee_type(long double x)
{
  unsigned int *a = (unsigned int *) &x;
  if ((a[4] & 0x7FFF) == 0x7FFF) return ieee_normal;
  return a[4] & 0x8000 ? ieee_mlNF : ieee_plNF;
}
```

LISTING 2: MATHERR.C

```c
#include <math.h>
#include <float.h>
#define INF 1.0/0.
#define NAN (1./0.)

void startfp(void)
{
  /* mask all exceptions but denormal */
  _control87(MCW_EM_DENORMAL, MCW_EM);
  HUGE_VAL = +INF;
}
/* this gets called by library functions
if a domain or range error occurs */
int cdecl metherr(struct exception *e)
{
  /* return nonzero to show error has been handled */
  /* lib functions will return something sensible, */
  /* if you let them */
  /* we don't need no steenkin' errors! */
  return 1;
}
```

LISTING 3: TAN.C

```c
#define NUM TERMS 15
double approx_tan(double x)
{
  int i;
  long double x2 = x*x, y = 0;
  if (x == 0) return 0;
  for (i = 2*NUM TERMS-1; i >= 0; i -= 2)
    y = i - x2 / y;
  return x / y;
}
```

LISTING 4: SIGTEST.C

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
extern jmp_buf jump1;

void cdecl fhandler(int sig)
{
  fprintf(stderr, "Floating point error.\n");
  _fpreset();
  /* reinstall the exception handler */
  signal(SIGFPE, fhandler);
  /* jump to a safe place */
  longjmp(jump1);
}
```
A DIRECTORY SEARCH ENGINE IN TURBO C

Here's a truly generalized directory search routine that calls a procedural parameter each time it finds a matching file.

Jake Richter

DOS would not have made it very far without the ability to use wildcard characters for certain file operations, such as COPY, DIR, and ERASE. Imagine copying all 142 of your C source code files from your hard disk to a floppy disk by typing their names on the DOS command line, one at a time. The means to avoid this sort of mindless drudgery are the low-level DOS functions Find First and Find Next.

Find First searches for the first occurrence of a given file specification in a specified directory. The file specification may contain the wildcard characters "*" and "?", and thus can match more than one file. Find Next simply attempts to locate the next occurrence of the same file spec, and can be called repeatedly until no more matching files are found. Find First and Find Next comprise DOS's built-in file search toolkit. In this article, we'll examine the workings of Find First and Find Next, and will build them into a generalized file search "engine" for use with Turbo C. (On page 26 of this issue, Neil Rubenking implements a file search engine under Turbo Pascal 5.0, also using Find First and Find Next.)

ENTER THE DTA

Under DOS 2.x and later, Find First and Find Next are implemented as DOS functions 4EH and 4FH, respectively. Both functions require that a filename template (with optional path) and a file attribute value be specified. Using Find First and Find Next also requires the use of the DOS Disk Transfer Area (DTA).

The Disk Transfer Area is used by DOS for exactly what its name implies: Disk data is transferred to and from this area of memory. When Find First and Find Next are called, the information returned by DOS is placed into the DTA. When a DOS application first starts up, the DTA is set to a 128-byte region at offset 80H into its Program Segment Prefix (PSP). The Program Segment Prefix is a 256-byte block that is allocated by DOS in memory, in front of a loaded program. The DTA can also be moved to a more convenient place, such as your program's data space. This move is accomplished by using the DOS function Set DTA Address (1AH), which is called through DOS interrupt 21H using the following register values:

- **AH = 1AH** Specifies the Set DTA Address function
- **DS:DX = Segment:Offset of new DTA**

Function 1AH returns no errors.

When moving the DTA to your own program space, make sure that enough space is allocated for whatever DOS operation you plan to use. For the Find First and Find Next functions, the minimum DTA size is 43 bytes.

DIRECTORY ENTRIES AND ATTRIBUTES

When Find First and Find Next find a file, they return information in the DTA that comes from the found file's disk directory entry. There are three basic types of directory entries: volume labels, subdirectories, and normal files. Each entry in the directory structure uses the same amount of directory space. The entry types are differentiated from one another by the values in the file attribute field.

Six file attributes are currently supported by DOS, and each file attribute has its own bit flag in the attribute field.

**Bit 0 (01H): Read-Only.** This attribute applies only to regular files. When set, it indicates that the file cannot be deleted or written to. A subdirectory entry's Read-Only flag can be set, but the flag doesn't affect the use of that subdirectory. The Read-Only flag can be modified by using the ATTRIB program under PC-DOS and some versions of MS-DOS.

**Bit 1 (02H): Hidden File.** This flag applies to files and subdirectories. When it's set, the file or subdirectory can't be seen in a DIR listing, and a hidden file can't be deleted from the command line. However, the file can still be accessed by a program, or by other DOS utilities such as TYPE or COPY. Hidden subdirectories can be accessed by RMDIR and CHDIR.
Bit 2 (04H): System File. The System File attribute’s effects are similar to those of the Hidden File attribute. The reason for the existence of the System File attribute lies in the DOS boot process. When IBM versions of DOS boot up, they search for two hidden system files, IBMBIO.SYS and IBMDOS.SYS, which are required in order to complete the boot procedure.

Bit 3 (08H): Volume Label. This attribute identifies its directory entry as the current volume’s volume label. Each DOS volume can only have one valid volume label. If multiple directory entries have the Volume Label bit set in their attribute fields, then only the entry that is listed first in the directory is recognized. No other attributes can be set in conjunction with this attribute. Once it’s flagged as a volume label, the directory entry can only be modified by using an extended FCB (as explained in “Taking Charge of DOS Volume Labels,” TURBO TECHNIX, November/December, 1987).

Bit 4 (10H): Subdirectory. All subdirectories have this flag set in their attribute field.

Bit 5 (20H): Archive. This flag is set each time a file is altered. DOS programs such as BACKUP and XCOPY use this bit to perform incremental backups (i.e., to back up only those files that have changed since the previous backup). When the file is copied by these utilities, its Archive flag is cleared; the flag remains clear until it’s set again by a subsequent modification. The Archive flag has no effect on subdirectories. Like the Read-Only flag, the Archive flag can be modified by the ATTRIB program.

DOS FUNCTION 4EH

The DOS Find First function is called via INT 21H by using the following register protocol:

1. AH = 4EH
2. CX = File attribute
3. DS:DX = Segment:Offset of ASCIIZ pathname string

Here are some things to keep in mind when setting up and using Find First.

1. The DTA must have been previously set to a buffer that contains at least 43 bytes of free memory.
2. The file attribute parameter specifies which file attributes must be present in order for a match to be legal. Four attributes are valid when using Find First: Hidden File, System File, Volume Label, and Subdirectory. If no attribute bits are specified, regular files (those with no attribute set) are searched for, as well as those files whose Archive or Read-Only attributes are set. If only the Volume Label attribute is set, then only a volume label is searched for.
3. The ASCIIZ string can contain both a path and the file specification. The file specification can consist of a combination of valid characters and the two wildcard characters, “*” and “?”.

If the Carry flag is set upon return from Find First, then one of the following errors occurred in the code returned in AX:

- File not found—02H
- Path not found—03H
- No more files/No match found—12H

If the Carry flag is not set, then no error occurred and the DTA contains the information returned by this call. In essence, if DOS reports an error on Find First or Find Next, then there are no more files to be found (assuming that you’ve previously validated the file path).

The information returned by DOS is placed in the DTA, and can be represented by the C structure shown in Figure 1.

```
struct ffblk
{
    char ff_reserved[21]; /* Reserved by DOS */
    char ff_attrib;    /* Attribute found */
    int ff_ftime;      /* File time */
    int ff_fdate;      /* File date */
    long ff_fsize;     /* File size */
    char ff_fname[13]; /* Found file name */
};
```

Figure 1. A C structure to divide the DTA into named fields.

The ffblk structure is defined by Turbo C in the DIR.H include file. The ff_reserved field is used by DOS to store information pertinent to the search, such as current index into search, search mask, and so forth. The ff_fname field is an ASCIIZ string that contains the name of the file that was just found by Find First (and Find Next), with all spaces removed and a “.” added to separate the filename and extension. ff_attrib, ff_ftime, ff_fdate, and ff_fsize are the attribute, the time and date of last update, and the size of the found file, respectively.

Certain constant definitions in Turbo C’s DOS.H file can help break down the ff_attrib field into its individual bit flags. The definitions and their meanings are summarized in Table 1.

Turbo C implements a function that calls Find First as find.first(); the function definition is shown in Figure 2. find.first() returns a nonzero value if no files that match the filename are found. The Turbo C version of the call requires a pointer to an ffblk structure because Turbo C sets the DTA to the specified ffblk structure, prior to calling the DOS-level Find First. This approach is very useful when several ffblk structures are active at the same time, as I’ll describe shortly.

```
/*File date. */
/* File time. */
/* Attribute found. */
Reserved by DOS. */
Found file name. */
```

Table 1. Predefined constants in DOS.H that specify individual bit flags in the file attribute byte.

DOS FUNCTION 4FH

The DOS Find Next function is also called via INT 21H, with register AH set to 4FH. No other registers need to be set. As its name implies, Find Next re-

continued on page 76
#include <dir.h>
#include <dos.h>
typedef FFBLK struct ffblk; /* For cleaner decl. */

int findfirst(char *filename, FFBLK *ffblkPtr, int attrib)
        char *filename; /* File mask w/optional path */
        FFBLK *ffblkPtr; /* Pointer to a ffblk struct */
        int attrib; /* Valid attributes for search */

Figure 2. Function findfirst() and its associated definitions.

quires a DTA that has been initialized by a Find First call. (Without a properly initialized DTA, DOS won't
know what file spec or attribute to search for, nor even where to start looking.) When it locates an ad­
ditional match, Find Next updates the information in the DTA.

If the Carry flag is set upon return from the Find
Next call, then either error code 02H (file not found)
or 12H (no more files found) is returned in AX. If
the Carry flag is cleared, then no error has occurred.

In Turbo C, Find Next is accessed through library function findnext(), which is defined as shown in
Figure 3. As with findfirst(), the value returned by
the findnext() function is nonzero if no files that
match the file specification (which is already in the
DTA) are found. findfirst() and findnext() both use
the ffblk structure to divide the DTA into fields.

THE SEARCH ENGINE

As you may have gathered from the discussion so far,
the findfirst() and findnext() routines work best in
combination. They suggest a general-purpose file
search "engine" that searches a specified directory
for files that match a given file spec and file attribute
value. When a file is found, the engine takes some
action by calling a function that is passed to the en­
gine through a procedure pointer. I've implemented
such a search engine function as a separate code
module that can be linked with other Turbo C pro­
grams. The SearchEngine() function definition is
given in Figure 4. The actual code for SearchEn­
gine() can be found in ENGINE.C (Listing 1).

SearchEngine() takes a file spec (which may in­
clude a path), an attribute value that specifies which
file attributes are valid to search for (see the earlier
explanation of Find First), and a pointer to a proce­
dure. This procedure is called each time a file that
matches the file spec and attribute is found. The pro­
cedure's definition (assuming that you name the function MyFunc()) is as follows:

#include <dir.h>
#include <dos.h>
typedef FFBLK struct ffblk;

void MyFunc(FFBLK) FFBLK *ptrFFBLK;

If the step of passing a procedure to the search
engine doesn't appear useful at first glance, let me
provide an example. Let's assume that it's necessary
to view the names of all of the current directory's C
source code files that have been modified since your

Figure 3. Function findnext() and its associated definitions.

last backup. The code for this task is provided in
MODIF-C.C (Listing 2).

MODIF-C contains two routines, main() and DisplayModC(). main() serves as the program entry
point, and initiates the call to SearchEngine(). Note
that SearchEngine() is passed a file spec that con­
tains a wildcard character "*" as the filename, with
the extension fixed as "C." The attribute that is
passed is "0," which indicates that only plain files are
valid (Read-Only and Archive attributes are consid­
ered plain for our purposes). Also, a pointer is
passed to DisplayModC() so that SearchEngine() can
call DisplayModC() on each "hit" during the search.

Note that the function called by SearchEngine() has complete access to the found file's directory in­ormation, via the pointer to the found file's DTA.
This means that the file's name, date, time, size, and
attribute are available to the procedure called
through the procedure pointer. If necessary, the di­
crctory for the file can be determined by making a
call to the Turbo C library function, getcwd() (Get
Current Working Directory), as I'll demonstrate later.

To re-create MODIF-C.EXE with the command­
line Turbo C compiler, execute the following DOS
command-line commands:

    tcc -c modif-c.c
tcc -c engine.c
tcc modif-c.obj engine.obj

The -c option indicates that only an object file
should be produced for the given source code file.
The last line specifies that the two object code files
are to be linked into an executable file. The .PRJ file
that creates MODIF-C.EXE using the Turbo C In­
tegrated Development Environment contains just two
lines:

    modif-c
    engine

SEARCHING A DIRECTORY TREE

Let's go one step further than the previous example,
and say that we want to display the names of all of
the modified C source code files that are located any­
where on the current drive, even though these files
might be in different subdirectories. This is not an
easy problem to solve with typical iterative program­
ing methods. Fortunately, this kind of problem is
easy to solve by using recursion.
sizeof() to determine the variable's size. Each parameter requires at least two bytes (again, the number of bytes depends upon the parameter's type). Each level of recursion (and, hence, each directory to be searched) has its own FFBLK structure. This is necessary in order to determine whether terminating condition #2 (as given above) has occurred. The DTA for a specific search contains a place marker that DOS uses to determine the starting position for Find Next. Therefore, DOS knows when its search on any given directory is complete. This allows the transparent use of findfirst() and findnext() in a recursive directory tree search, as long as a pointer is passed to the correct FFBLK structure for any given level.

The association of each level of recursion with its own FFBLK is performed by declaring an array of FFBLK structures named fileBlock. The number of elements in fileBlock is given by the constant MAXDIRDEPTH (which, at 15, allows more nesting levels than anyone is ever likely to encounter). A variable named curDepth acts as the index into the array of FFBLK structures. Each successive call to GetNextDir() increments curDepth, and each return from GetNextDir() decrements it.

If the FFBLK structures were declared as local to GetNextDir(), DirTree() would be significantly simplified, since as the array and its index would no longer be required. Each recursive level's FFBLK would be created on the stack when each recursive call is instantiated, and the different FFBLK structures would never get mixed up. This method, however, uses a great deal more stack space, and the aim here is to use as little stack space as possible.

Turbo C's findfirst() and findnext() also make it convenient to integrate the recursive directory search routine GetNextDir() with SearchEngine(). Each time a normal terminating condition is encountered, a call is made to SearchEngine(). The normal terminating conditions are designed such that each directory in a tree causes only one terminating condition. As an example, consider Figure 5, which schematically shows a directory tree whose root is a directory named "A."

When called to process the subdirectories in Figure 5, SearchEngine() processes them in the following order: E, I, F, J, K, G, B, C, H, D, and A. Subdirectories E, I, J, K, and H cause termination condition #1 (notice that they have no child directories). The rest of the subdirectories cause condition

continued on page 78
Listing 1: ENGINE.C

#include "dir.h" /* Required by findfirst, findnext, getcwd. */

#define TRUE !FALSE
#define FALSE 0

static FFBLK procBlock; /* Declare a file info block for the specific proc */

Listing 2: MODIF-C.C

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

typedef struct ffblk FFBLK;

extern void SearchEngineO;

void DisplayModCCsearchRec)

#2 after all of their child directories have been searched.

TWO ENGINES

What we now have are two different routines, both of which are general-purpose search engines for DOS directories. SearchEngine() searches a single directory, and DirTree() searches the entire directory tree of the current drive. Use whichever routine is appropriate; their parameter lists are identical.

For example, to incorporate full recursive tree search into the simple MODIF-C demo program, just substitute the following line for the original call to SearchEngine():

DirTree("*.C", 0, DisplayModC);

Then recompile DIRTREE.C, recompile MODIF-C.C, and link the final .EXE file to the Turbo C command-line compiler using the following commands:
tcc -c dirtree.c
tcc -c modif-c.c
tcc modif-c.obj engine.obj dirtree.obj

If you're using the Integrated Development Environment, the .PRJ file would look like this:

	modif-c
	engine

dirtree

The resulting program, MODIF-C.EXE, finds and displays the names of all of the modified C source files that are located in the current directory and in all of the directories below it.

To make the interface to the two search engine routines clear, MODIF-C has been kept bare-bones simple. Your first enhancement should almost certainly be to retrieve command-line parameters so that the program can be set to search for more than just C source code files. [Editor's note: In future issues of TURBO TECHNIX, we'll publish short articles that present file utilities built around the search engines—watch for them.]

DON'T SOLVE PROBLEMS—BUILD TOOLS!

Because the search engines' action is specified by the calling logic at runtime through procedure pointers, the search engines can be applied to a variety of tasks, such as building linked lists of directory entries, deleting files, printing file headers, moving files out to a backup drive, and so forth. The possibilities are virtually unlimited, and need not be specified at compile time. That's the advantage of an "engine" concept, as opposed to simply hard-coding fixed solutions to individual problems. When you solve a problem, work a little longer to turn the solution into a tool—and you'll work less the next time the problem comes up. ■

Jake Richter is the President of Panacea, Inc., a PC consulting company in Woburn, Massachusetts. He can be reached on MCI MAIL and on BIX as jrichter.

Listings may be downloaded from Library 1 of CompuServe forum BPROGB, as CENGN.ARC.
DIRTREE.C - by Jake Richter

LISTING 3: DIRTREE.C

Provides core routines for traversing a directory tree, using a "bottom-most first" algorithm.
As presented, code will search the directory tree and for each directory found, will call a routine called
SearchEngine(), which in turn will process certain files in
that directory in some fashion.

Program Definitions

# define FALSE 0
# define TRUE

typedef struct ffblk FFBLK;

static FFBLK fileBlock[HAXDIRDEPTH];
static int done;
static int curDepth;
static char *filename;
static char *attribute;
static void (*funcPtr)(

void GetNextDir()
{
    /* This is the recursive routine that traverses the
directory tree.
    */
    if (curDepth >= HAXDIRDEPTH)
        return;
    /* Since this section is encountered only when
going down to a new level, (re)initialize the
current level's file block by calling
findfirst. FindFirst and findnext return a
TRUE (non-zero) value when all files in
the current directory have been "found". A
separate block is needed for each level
because previously determined information
(set by findfirst() and subsequent findnext() calls)
must be maintained until an entire
directory level has been searched.
*/
    done = findfirst(".*", &fileBlock[curDepth], FA_DIREC);
    /* It is important to remember that ".*" and
".*", are valid directory names, but that
they also should be ignored while
traversing the tree. The following
conditional in pseudo-code:
DEFINITE CLAUSE GRAMMARS IN TURBO PROLOG

A parser is only as good as its grammar.

Barbara Clinger, Ph.D.

Since microcomputers have become faster and contain more memory, producers of software are under pressure to create friendlier software. If “user friendly” is a euphemism for the ability to exchange information with computers in English, then programs need a process to extract key information from the average user’s input. One such process uses a definite clause grammar (DCG).

The investigation of definite clause grammars is the primary purpose of this article. We’ll also examine parsers as a means to scan and interpret English sentences. In addition, two methods of partitioning—simple partitioning, and parsing by difference lists—will be explored and compared. Finally, we’ll examine a simple mathematical expression parser.

GRAMMARS

Languages are built with words; the lexicographic level is the dictionary which gives the definition, as well as the function, of a word (noun, verb, and so on). A language syntax imposes structure upon words. In English, phrases and sentences are part of the syntactic structure. In a programming language, such as Pascal, syntax is often provided through syntax diagrams. Figure 1 depicts a syntax diagram for a Pascal identifier (name). This diagram shows that the identifier must begin with a letter and may be followed by a combination of letters and digits.

Grammar provide another method for describing a language. A grammar allows a language to be precisely described by the use of a specific syntax. One popular grammar, called Bacus-Naur Form (BNF), is used to define the Turbo Prolog language (see Figure 2). To see how BNF syntax is read, consider the following statement:

\[
\text{name} ::= (\text{letter} | _) \{\text{letter} \text{digit} | _\}^* \\
\text{name-list} ::= \text{name} | \text{name-list}, \text{name-list} \\
\text{variable} ::= (\text{capital-letter} | _) \{\text{name} \text{digit} | _\}^* \\
\text{functor} ::= (\text{small-letter} | _) \{\text{name} \text{digit} | _\}^* \\
\text{letter} ::= \text{small-letter} \{\text{name} \text{digit} | _\}^* \\
\text{digit} ::= 0|1|...|9
\]

Figure 1. Syntax diagram for a Pascal identifier (name).

Letters and digits are also defined in BNF notation. As any programmer knows, failure to follow the syntax of a language results in the ubiquitous “SYNTAX ERROR” message, and rejection of the program by the compiler. A general discussion of BNF syntax can be found in Chapter 7 of the Turbo Prolog Toolbox Owner’s Handbook.

Using these simple concepts, I can define a very simple context-free grammar. My dictionary consists of three nouns (dog, cat, and water) and one verb (drinks). The syntax of this language has one rule: A
sentence takes the form of a noun, followed by a verb, followed by a noun. Using BNF notation, this grammar is defined as:

<sentence>::=<noun> <verb> <noun>
<noun>::= dog | cat | water
<verb>::= drinks

In this language, the following are all correct sentences:
dog drinks water
cat drinks water
water drinks cat

The last sentence is correct since it adheres to the syntax of a sentence; this sentence emphasizes why this grammar is called "context-free." The next higher level of a grammar imposes *semantics* (the meaning of words) on the language, and is beyond the scope of this article.

**DCG NOTATION**

A *definite clause grammar* (DCG) is simply a grammar that is expressed as logic statements; parsing is the execution of the statements. Although I'll use DCGs in context-free grammars in this article, keep in mind that they can be used for more powerful grammars.

The notation used with a DCG differs slightly from the BNF notation used in Figure 2. However, the translation between the BNF notation and the DCG notation (given in Figure 3) is quite simple. For instance, the DCG notation for the simple grammar in the previous example is the following:

```
sentence --> noun, verb, noun
noun --> dog
noun --> cat
noun --> water
verb --> drinks
```

"sentence," "noun," and "verb" are called *nonterminals*. The *tokens* (also called *terminals*) are "dog," "cat," "drinks," and "water."

The beauty of using DCGs to define a grammar is that the implementation in definite clause grammars follows naturally from the grammar's English description. For instance, the next example defines a grammar to parse sentences of the following form:

*John likes Mary.*
*The man sees a dog.*
*Mary likes the dog.*
*John eats.*

The grammar for these sentences can be described in English as listed below:

- A *sentence* takes the form of a noun phrase, followed by a verb phrase.
- A *noun phrase* takes either the form of a *determiner* (definite article) that is followed by a noun; or else the form of a noun.

*continued on page 82*
The dictionary words are called tokens or terminals.

These are words used in the grammar which are not terminals; they are given in terms of other language elements.

This symbol is the equivalent of the ::= in BNF form and is read "takes the form of".

The comma is read "followed by".

Figure 3. The translation between BNF and the DCG notation.

DEFINITE CLAUSE
continued from page 81

A verb phrase takes either the form of a verb that is followed by a noun phrase; or else the form of a verb.

Since the DCG form of the grammar will be converted into executable Prolog predicates, acceptable Turbo Prolog names are used in the following definitions:

sentence --> noun_phrase, verb_phrase
nour_phrase --> determiner, noun
noun_phrase --> noun
verb_phrase --> verb, noun_phrase
verb_phrase --> verb

Before we can implement a DCG, we must decide upon the form of the data that goes into and comes out of the parser. The decision to use a list of tokens as input is almost universal. Therefore, the reader’s output should consist of a list of tokens to be parsed. XPARS.SCA in the Turbo Prolog Toolbox is an example of a reader that’s designed for input into predicates produced by the parser generator. The predicate reader in Listing 1 produces a list of strings that are used as tokens in that program.

Before the parser can be implemented, the form of the data that’s required by the rest of the program must be known. The output can be as trivial as a true or a false to indicate that the parsing was successful or unsuccessful, respectively (as shown in Listing 1). Alternatively, the output can be a list of keywords, a numerical value, or a more complicated structure that represents a parse tree.

A parse tree is a structure that shows the overall construction of the original source input. In Pascal, the implementation of a tree structure is accomplished through pointers and records, where a record contains some information along with pointers to other nodes in the tree. In Turbo Prolog, a tree structure is represented through the use of compound objects. For instance, the sentence "the man sees a dog" can be represented by the following Turbo Prolog structure:

sentence(noun_phrase( 
determiner(the), 
noun(man)), 
verb_phrase(verb(sees), 
noun_phrase(determiner(a), 
noun(dog))))

Figure 4 shows the parse tree for this sentence.

With this sentence structure as output, the appropriate predicates must be written to extract the information from the tree and then evaluate that information. Recall the DCG for the sentence structure:

sentence --> noun_phrase, verb_phrase

The translation of this DCG requires the input list to be partitioned into two sublists A and B, in such a way that the list A is a noun phrase and the list B is a verb phrase. To see this clearly, consider the following input list:

[the, man, sees, a, dog]

This list can be partitioned into the sublists:

[the, man]
[sees, a, dog]

The next step is to test whether the sublists [the, man] and [sees, a, dog] satisfy the criteria of being a noun phrase and a verb phrase, respectively.

A very simplistic method for partitioning the input list is to use the predicate append. This two-way predicate not only appends two lists to produce a third, but it can also return all of the partitions of a list as two sublists. In simple grammars (such as the grammar implemented in Listing 1), append is adequate for the job, and uses less stack than does the difference list method (which is described shortly). In a more complicated grammar, append requires a lot of backtracking and is less efficient.

DIFFERENCE LISTS

A more efficient method of partitioning is based upon an incomplete data structure called the difference list. This alternative to list processing can greatly simplify list-processing programs.

In order to use this partitioning method, a "subtraction" between two lists must first be defined. Let’s examine the list A = [a, b, c]. A can be considered, in many ways, to be the difference of two
In fact, the following statement is true for any set $T$, where the arbitrary $T$ makes the data structure incomplete:

$$[a, b, c] = [a, b, c] \setminus T,$$

In general, if $A = [a, b, \ldots, d]$, then $A$ is the difference between the list $X = [a, b, \ldots, d]T$ and $T$, where $T$ can be any list; this difference is denoted by $A \setminus X$. Note that the empty list $[]$ is expressed as $X - X$.

To apply difference lists to DCGs, let $A = \text{[the, man, sees, a, dog]}$ and look at the following grammar in terms of difference lists:

$$\text{sentence} \rightarrow \text{noun phrase}, \quad \text{verb phrase}$$

According to this particular grammar, the difference list $A - T$ is a sentence if $A - Y$ is a noun phrase and $Y - T$ is a verb phrase, for some list $Y$.

To represent this as a Turbo Prolog clause, one would like to write:

$$\text{sentence}(A - T):-
\text{noun phrase}(A - Y),
\text{verb phrase}(Y - T).$$

The minus sign, however, normally implies subtraction between real numbers. We could define a predicate, such as $\text{difference}(X,Y,Z)$ where the difference of $X - Y$ is returned in $Z$, and use $\text{difference}$ in the clause for $\text{sentence}$. The same idea can also be coded with two arguments as in the following clause:

$$\text{sentence}(A, T):-
\text{noun phrase}(A, Y),
\text{verb phrase}(Y, T)$$

Keep in mind that the two arguments in this clause refer to the difference lists.

Listing 2 is similar to Listing 1, except that difference lists perform the partitioning process in Listing 2. The difference lists are handled in the following clauses:

$$\text{sentence(List in, Rest):-}
\text{noun phrase(List in, Y)},
\text{verb phrase(Y, Rest)}.\quad \text{noun phrase}(X, Rest):-
\text{determiner}(X, Y),
\text{noun}(Y, Rest).\quad \text{verb phrase}(X, Rest):-$$

Figure 4. Parse tree for the sentence "the man sees a dog."

**PARSING MATHEMATICAL EXPRESSIONS**

My final example of the use of DCGs scans a mathematical expression. This example illustrates several points, including how to handle a DCG that requires specific symbols (such as the arithmetic operators "+" or "/"); how to handle functions (such as the trigonometric functions); and how to return information from a scanner.

For this example, Listing 3 returns the numeric value of an expression, and Listing 4 returns the parse tree of an expression.

Before defining the DCGs to parse an expression, let's think for a moment about the precedence of operations. Consider the following expression:

$$2 \times 3 - 4 + 5 \times (\sin(1.5) + 8^2) + 6/7$$

The precedence of operations dictates that expressions within parentheses are evaluated first, then the individual terms that use multiplication and division are evaluated, and finally, the terms are summed. In evaluating such an expression, parentheses have the highest priority, followed by exponentiation, then by multiplication and division, and finally by addition and subtraction. Also, $\sin(1.5)$ represents a number that must be "looked up" before the expression inside the parentheses can be evaluated. I've defined a DCG in which the terminals are numbers (including pi and function continuation on page 84
DEFINITE CLAUSE

continued from page 83

tions that return numbers), with
the syntax imposed by the oper-
ators. The order in which the
grammar is stated determines the
priority of the operators.

The previous expression con-
tains four terms: 2*3, 4, sin(1.5)+ 8'2), and 6/7. These terms are
summed together to give the value
of the expression. The word
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
"sum" is used here because the
operations that return numbers), with
...
PolyAWK™ - The Toolbox Language.
For C, Pascal, Assembly & BASIC Programmers.

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

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

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

Saves You Time & Effort
The most compelling reason to use PolyAWK is that you can literally accomplish in a few lines of code what may take pages in C, Pascal or Assembler. Programmers spend a lot of time writing code to perform simple, mechanical data manipulation—changing the format of data, checking its validity, finding items with some property, adding up numbers and printing reports. It is time consuming to have to write a special-purpose program in a standard language like C or Pascal each time such a task comes up. With PolyAWK, you can handle such tasks with very short programs, often only one or two lines long.

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

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

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

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

Math Support
PolyAWK also includes extensive support for math functions such as strings, integers, floating point numbers and transcendental functions (sin, log, etc.) for scientific applications. Conversion between these types is automatic and always optimized for speed without compromising accuracy.
LISTING 1: GRAMMAR.PRO

/* Simple DCG parser
Barbara Clinger, 1988

This program illustrates the expansion of a simple DCG.
Its vocabulary consists of:
Nouns: John, Mary, man, dog;
Verbs: likes, sees

Sample input: The man sees a dog.
Output: True or False, for success or failure of parsing.
*/
domains
toklist = string

predicates
reader(string,toklist) /* the reader */
remove_period(toklist,toklist)
append(toklist,toklist,toklist)
do
/* The grammar */
sentence(toklist,toklist,toklist) /* the parser */
noun_phrase(toklist)
verb_phrase(toklist)
determiner(string)
noun(string)
verb(string)
goal
do.

clauses
/* The clause do parses a sentence and returns true or false. Its
writing is informational only. */
do :-
  nl,write("Enter a sentence --> "),
  readin(S),nl,nl,
  reader(S,List), /* use the reader */
  write("Output of the reader: ",List),nl,
  remove_period(List, List_in),
  sentence(List_in,Noun_phrase,Verb_phrase),
  write(" Noun phrase: ",Noun_phrase),nl,
  write("Verb phrase: ",Verb_phrase),nl.

/* Using append to split the list into possible noun phrases and
verb phrases is not efficient, but for simple grammars it is
adequate. */

/* expansion of:
  sentence --> noun_phrase, verb_phrase
*/
sentence(List_in,Noun_list_out,Verb_list_out) :-
  append(Noun_list_out,Verb_list_out,List_in),
  noun_phrase(Noun_list_out),
  verb_phrase(Verb_list_out).

/* expansion of:
   noun_phrase --> determiner, noun
*/
noun_phrase(A,B) :- determiner(A),noun(B).
  noun_phrase(A) :- noun(A).

/* expansion of:
   verb_phrase --> verb, noun_phrase
*/
verb_phrase(A,B) :- verb(A),noun_phrase(B).
  verb_phrase(A) :- verb(A).

/* the dictionary */
determiner["the"],
  determiner["a"],
  noun["man"],
  noun["John"],
  noun["Mary"],
  noun["dog"],
  verb["likes"],
  verb["sees"]
/* end of dictionary */

/* reader */
(1) the empty string returns the empty list,
(2) if the string is not empty, it recursively takes the front
   token, converts it to lower case, then reads the rest of the
   list, until the string is empty.
reader("",[]) := []
reader(Str, [Token|Rest]) :=
  frontendon(StripTok,Token),
  upper_lower(Tok,Token),
  reader(Str,Token),
/* end of dictionary */

LISTING 2: DIFFERENCE.PRO

/* Parsing by difference lists
Barbara Clinger, 1988

For input and output, this program is identical to
Program Listing 1. However, the expansion of the grammar
is done with difference lists rather than using append to split
the list of tokens into noun phrases and verb phrases. */
domains
toklist = string

predicates
do
  reader(string,toklist) /* the reader */
  remove_period(toklist,toklist)
  append(toklist,toklist,toklist)
do
/* The grammar */
sentence(toklist,toklist,toklist)
  noun_phrase(toklist)
  verb_phrase(toklist)
  determiner(toklist)
  noun(toklist)
  verb(toklist)
  goal
do.

clauses
/* sentence:
List_in is a sentence if List_out - Y is a noun phrase and
Y - Rest is a verb phrase. If the predicate sentence succeeds
in parsing the entire list then Rest is the empty list. */
sentence(List_in,Rest) :-
  noun_phrase(List_in,Y), verb_phrase(Y,Rest).

/* noun_phrase */
X is a noun phrase
if X - Y is a determiner and Y - Rest is a noun
  or if X = Y is a noun.

/*
  noun_phrase(X,Rest) :- determiner(X,Y), noun(Y,Rest).
  noun_phrase(X,Y) :- noun(X,Y).

/* verb_phrase */
X is a verb phrase
if X - Y is a verb and Y - Rest is a noun phrase
  or if X - Y is a verb and Y - Rest is a noun
  or if X - Y is a verb.

/* verb_phrase(X,Rest) :- verb(X,Y), noun_phrase(Y,Rest).
  verb_phrase(X,Y) :- verb(X,Y), noun(Y,Rest).
  verb_phrase(X,Rest) :- verb(X,Rest).

/* the dictionary */
determiner["the"],
determiner["a"],
noun["man"],
noun["John"],
noun["Mary"],
noun["dog"],
verb["likes"],
verb["sees"]
/* end of dictionary */
/* Mathematical Expression parser

This program parses a mathematical expression and returns the value of the expression. It allows the use of: for exponentiation, grouping using parentheses, evaluation of functions (sin, cosine, ...). Decimals in the range from -1 to +1 must be entered with a leading zero (i.e., 0.25). A warning is issued if negative numbers are raised to fractional powers; the indeterminant zero raised to the zero power stops execution of the program.

sample input: 2 * 3 + (sin(2*pi/3) + 1)^2 - ln(0.123)

Not all expressions of the form X - Y are possible. The clause if_can_do allows the obvious cases to be evaluated.

/ * goals do. */
classes do:
  write("error in secant argument "), nl, nl, fail.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is:"), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
  write("The value of the expression is ", N),nl,nl.
  if_ineven_int(real)
    write("Cannot evaluate the expression."), nl,
    write("Unevaluated remainder list is: "), N),nl,nl.
The clause if can do tests some cases for the evaluation of expressions of the form $V_1 \cdot V_2$.

if can do($V_1$, $V_2$) :-
  $V_1 = 0$, /* positive base, all ok */
if can do($V_1$, $V_2$) :-
  $V_1$ is positive base, all ok
X = exp($V_2$ \* ln($V_1$)).
if can do($X$, $V_1$, $V_2$) /* 0 raised to 0 is indeterminant */
  $V_1$ = 0, $V_2$ = 0, !,
write("*************** ERROR **************
expression contains indeterminant form 0 \* 0
***************

X = ln($V_1$). /* automatic stop of program*/
if can do($X$, $V_1$, )
  $V_1$ = 0, /* 0 raised to nonzero power is 0 */
X = 0.
if can do($X$, $V_1$, $V_2$) :- /* negative root to an odd*/
  X is odd int($V_2$), /* integer is ok */
X = exp($V_2$ \* ln(abs($V_1$)))).
if can do($X$, $V_1$, $V_2$) :- /*negative root to an even*/
  is even int($V_2$), /* integer is ok */
X = exp($V_2$ \* ln($V_1$))).

Negative number to a fractional power can swing right or wrong. For example:

(-32)^(1/5) (the fifth root of -32) is -2
(-1024)^(1/10) (the 10th root of -1024) does not exist.*

if can do($X$, $V_1$, $V_2$) :-
  X = exp($V_2$ \* ln($V_1$)).
write("*************** ERROR **************
expression contains (11, $V_1$, 11) \* $V_2$)
write("had to use (abs(11, $V_1$, 11) - $V_2$)
write("***************

is odd int($X$) :- $X$ = round($X$), Cr<><. and($X$) lllOd 2) = 1.
is even_int($X$) :- $X$ = round($X$).

LISTING 4: PARSTREE.PRO

/* Parse Tree example Barbara Clinger, 1988
This program illustrates a parser for simple algebraic expressions,
(no exponentation, parentheses, or functions). It returns the parse
tree of the expression. The tree is built using the structure node,
which is essentially an operator or functor with left and right
branches.
Sample input: 2 \* 3 - 4
The output is a tree which represents the root functor form)
(\+\ DOCUMENT
predicates
toklist = string*
read(token, toklist, toklist)
give_result(node, toklist, toklist)
append(toklist, toklist, toklist)

goal
do.

clauses
do :-
  nl,write("Enter an expression \to \n),
readin(String), nl,
read(token, Str, List_in),
read(token, List_in, Rest),
give_result(Tree, List_in, Rest).
give_result(1, T, 1) :-
  T = []
write("The structure of the expression is: \n", T).
give_result(\nil, T, 1) :-
  write("Cannot evaluate the expression.\n", T).
read(token, Str, Rest) :-
  weak(token, Tok, Str1),
  reader(Str1, Rest), 1.

domains
toklist = string*
token = op(string) ; leaf(real)
node = branch(item,node,node) ; empty

predicates
token = string*
read(token, toklist, toklist)
give_result(node, toklist, toklist)
append(toklist, toklist, toklist)

goal
do.

clauses
do :-
  nl,write("Enter an expression \to \n),
readin(String), nl,
read(token, Str, List_in),
read(token, List_in, Rest),
give_result(Tree, List_in, Rest).
give_result(1, T, 1) :-
  T = []
write("The structure of the expression is: \n", T).
give_result(\nil, T, 1) :-
  write("Cannot evaluate the expression.\n", T).
read(token, Str, Rest) :-
  weak(token, Tok, Str1),
  reader(Str1, Rest), 1.

domains
toklist = string*
token = op(string) ; leaf(real)
node = branch(item,node,node) ; empty

predicates
token = string*
read(token, toklist, toklist)
give_result(node, toklist, toklist)
append(toklist, toklist, toklist)

goal
do.

clauses
do :-
  nl,write("Enter an expression \to \n),
readin(String), nl,
read(token, Str, List_in),
read(token, List_in, Rest),
give_result(Tree, List_in, Rest).
give_result(1, T, 1) :-
  T = []
write("The structure of the expression is: \n", T).
give_result(\nil, T, 1) :-
  write("Cannot evaluate the expression.\n", T).
read(token, Str, Rest) :-
  weak(token, Tok, Str1),
  reader(Str1, Rest), 1.

domains
toklist = string*
token = op(string) ; leaf(real)
node = branch(item,node,node) ; empty

predicates
token = string*
read(token, toklist, toklist)
give_result(node, toklist, toklist)
append(toklist, toklist, toklist)

goal
do.

clauses
do :-
  nl,write("Enter an expression \to \n),
readin(String), nl,
read(token, Str, List_in),
read(token, List_in, Rest),
give_result(Tree, List_in, Rest).
give_result(1, T, 1) :-
  T = []
write("The structure of the expression is: \n", T).
give_result(\nil, T, 1) :-
  write("Cannot evaluate the expression.\n", T).
read(token, Str, Rest) :-
  weak(token, Tok, Str1),
  reader(Str1, Rest), 1.

domains
toklist = string*
token = op(string) ; leaf(real)
node = branch(item,node,node) ; empty

predicates
token = string*
read(token, toklist, toklist)
give_result(node, toklist, toklist)
append(toklist, toklist, toklist)

goal
do.

clauses
do :-
  nl,write("Enter an expression \to \n),
readin(String), nl,
read(token, Str, List_in),
read(token, List_in, Rest),
give_result(Tree, List_in, Rest).
give_result(1, T, 1) :-
  T = []
write("The structure of the expression is: \n", T).
give_result(\nil, T, 1) :-
  write("Cannot evaluate the expression.\n", T).
read(token, Str, Rest) :-
  weak(token, Tok, Str1),
  reader(Str1, Rest), 1.

domains
toklist = string*
token = op(string) ; leaf(real)
node = branch(item,node,node) ; empty

predicates
token = string*
read(token, toklist, toklist)
give_result(node, toklist, toklist)
append(toklist, toklist, toklist)

goal
do.

clauses
do :-
  nl,write("Enter an expression \to \n),
readin(String), nl,
read(token, Str, List_in),
read(token, List_in, Rest),
give_result(Tree, List_in, Rest).
give_result(1, T, 1) :-
  T = []
write("The structure of the expression is: \n", T).
give_result(\nil, T, 1) :-
  write("Cannot evaluate the expression.\n", T).
read(token, Str, Rest) :-
  weak(token, Tok, Str1),
  reader(Str1, Rest), 1.

domains
toklist = string*
token = op(string) ; leaf(real)
node = branch(item,node,node) ; empty

predicates
token = string*
read(token, toklist, toklist)
give_result(node, toklist, toklist)
append(toklist, toklist, toklist)

goal
do.

clauses
do :-
  nl,write("Enter an expression \to \n),
readin(String), nl,
read(token, Str, List_in),
read(token, List_in, Rest),
give_result(Tree, List_in, Rest).
give_result(1, T, 1) :-
  T = []
write("The structure of the expression is: \n", T).
give_result(\nil, T, 1) :-
  write("Cannot evaluate the expression.\n", T).
read(token, Str, Rest) :-
  weak(token, Tok, Str1),
  reader(Str1, Rest), 1.
New Turbo Prolog® 2.0 lets you harness powerful AI techniques. And you don’t have to be an expert programmer or artificial intelligence genius!

You get an all-new Prolog compiler that’s been optimized to produce smaller and more efficient programs than ever before. An improved full-screen, completely customizable editor with easy pull-down menus. All-new documentation, including a tutorial rich with examples and instructions to take you all the way from basic programming to advanced techniques. Even online help!

Turbo Prolog Toolbox is 6 toolboxes in one!

More than 80 tools and 8,000 lines of source code help you build your own Turbo Prolog applications. Includes toolboxes for menus, screen and report layouts, business graphics, communications, file-transfer capabilities, parser generators, and more!

Toolbox requires Turbo Prolog 1.1 or later

Just $99.95

More new features!
- An external database system for developing large databases. Supports B+ trees and EMS
- Source code for a fully-featured Prolog interpreter written entirely in Turbo Prolog. Plus step-by-step instructions to adapt it or include it as is in your own applications!
- Support for the Borland Graphics Interface, the same professional-quality graphics in Turbo Pascal, Turbo C, and Quattro
- Improved windowing
- Powerful exception handling and error trapping features
- Full compatibility with Turbo C so the two languages can call each other freely
- Supports multiple internal databases
- High-resolution video support

Just $149.95!

If I had to pick one single recommendation for people who want to try to keep up with the computer revolution. I’d say, ‘Get and learn Turbo Prolog.’

—Jerry Pournelle, Byte

An affordable, fast, and easy-to-use language.

—Darryl Rubin, AI Expert

60-Day Money-back Guarantee*

For the dealer nearest you
Call (800) 543-7543
Most computer programs are reasonably well-behaved. In the absence of pernicious bugs, a program will dutifully follow its algorithm, feeding on data along the way, then produce its results and call it a wrap. Artificial intelligence programs deviate from this procedural pattern, however. These eccentrics show not the slightest reluctance in plunging headlong into an unexplored search space in pursuit of an answer. All too often their nonchalant entry into such a system results in their program counter being irresistibly attracted to a black hole from which it never returns—and the program is lost in space.

The above scenario unfolds when the programmer fails to provide the program with an appropriate navigation system. The many techniques for guiding a program through its problem space are called search strategies. This article examines three of the simplest search strategies: depth-first search, breadth-first search, and best-first search. The depth-first and breadth-first searches are known as blind (or uninformed) methods since they utilize no heuristic information (or rules of thumb) about the problem. A best-first search, on the other hand, uses problem-specific information to traverse the search space more efficiently. Despite their differences, it turns out that all three approaches can be described in a uniform framework. We will look first at the general principles that are involved, and then I'll discuss their implementation in Turbo Prolog.

STATE SPACE

One popular problem-solving technique, known as “state space,” is used in a wide variety of AI applications including puzzles and games, natural language, and pattern-directed inference systems. This technique uses a directed graph of nodes to represent a given problem. Each node in the graph, called a state, represents a particular problem situation. One node is connected to another node by an arc. An arc between nodes exists if it’s possible to get from the first node to the second node by a legal move (sometimes referred to as a transition).

Now suppose that we have a directed graph that models a search space. One of the nodes of the graph, called the start node, is designated as the beginning point for the search. Also, some of the nodes of the graph are designated as goal nodes, and represent the states that we want to reach. The object is to find, if possible, a path from the start node to some goal node. A small example of such a graph is given in Figure 1, where node 0 is the start node and nodes 13, 14, and 15 are goal nodes. I'll use this graph as an illustration, and will presume that the successors of a given node are generated in increasing numerical order.

Some preliminary bookkeeping prevents going around in circles, exploring a section of the graph over and over. I therefore assume the existence of a mechanism for marking the nodes of the graph. In addition, a couple of simple data structures are required. The first data structure is the list L of nodes that have been discovered but not fully explored. These nodes are the possible starting points for further probes into the graph. The other data structure is a collection P of pointers joining pairs of nodes that have been discovered. When a goal node is found, this collection is used to construct a path from the start node to the goal node. Initially, only the start node is marked, L contains just the start node, and P is empty.

The general approach can now be described, beginning with the start node. If the start node is also a goal node, the search has succeeded with no effort, and the trivial path can be returned as the answer. If the start node is not a goal node, then proceed with the search as follows:

1. Choose the next node; and
   a. If the list L is empty, report failure; or
   b. If the list L is not empty, remove a node N from L and expand the node (i.e., generate a list S of all of the node's unmarked successors);
2. a. If one of these successors (say G) is a goal node, use P to generate a path from the start node to N, add the move from N to G, and return this as the successful result of the quest; or

b. If no goal nodes have been generated, mark the elements of S and add them to L. Also add a pointer, which points from N to each element of S, to the collection P;

3. Go to step 1.

All three of the search techniques that we are concerned with follow the outline just given. The difference lies in the manner with which the next node to be expanded is chosen. As you'll see, varying the way that this choice is made leads to strategies with widely disparate philosophies.

**DEPTH-FIRST SEARCHES**

Beginning at the start node, a depth-first search traverses down the levels of the search tree, choosing the lefthand node whenever more than one node exists. In this way, a depth-first search travels down the left side of the search tree first. If no solution is found, the search backs up one level and tries the righthand node. This process continues until all nodes have been examined.

In Figure 1, a depth-first search generates the path (0,5,8,9,14) from the start node 0 to the goal node 14. The source of the name "depth-first" becomes clear as the progress of the search is traced. The search moves as far down into the graph as it can go before giving up and seeking alternate routes. When the path (0,5,8,3,1) is generated during the search, a dead end has been reached. The process then backs up, first to 3 and then to 8, before taking the step from 8 to 9, which eventually leads to success.

In terms of our general description, some care is needed when adding new nodes to the list L. In particular, always put S at the beginning of L. When the time comes to pick a new node for expansion, choose the first element of L. In this way, the list L behaves like a stack—the last element in is the first element out.

The reference to "backing up" the search tree is reminiscent of the backtracking mechanism of Turbo Prolog—and that's no accident. Turbo Prolog searches for solutions in a depth-first fashion. One reason for using a depth-first search is that relatively little information needs to be maintained in order to recover the entire path once a goal node is found. The collection P of pointers that our implementation maintains is more than is needed for depth-first searching. At any stage, in fact, it's only necessary to track the pointers along the current path.

Depth-first searching, however, is not without difficulties. A major problem stems from the fact that the graphs involved in practical problems are very large, and sometimes infinite. It's relatively easy for a depth-first search to be led astray and to begin investigating a hopeless path—like the (0,5,8,3) route in our example. If the graph along such an avenue is large or infinite, a black hole develops and absorbs our intrepid explorer.

**BREADTH-FIRST SEARCHES**

One way to avoid such a demise is to adopt a more cautious strategy for moving around in the graph. A breadth-first search is one such approach. The basic idea behind a breadth-first search is simple. Investigate the graph level by level, beginning with the root. Look next at the nodes that are one step removed from the root, then examine nodes that are two steps away, and so forth. In terms of our general paradigm, simply add the elements of S to the end of L instead of to the beginning of L, and continue to choose the first element of L as the next node to be expanded. In this case, the list L is used as a queue (a first-in-first-out list).

When applied to the graph in Figure 1, the breadth-first search yields the path (0,6,9,14). At a length of three, this is one step shorter than the path that is generated by the depth-first search. Indeed, it's easy to see that a breadth-first search always finds a path of minimal length between the start node and a goal node, since all paths of length n are investigated before any paths of length n+1.

You may now be wondering, "If a breadth-first search always yields the shortest possible path, why not use it all the time?" A primary reason is that the goal nodes may be quite far away from the start node. In such a case, a depth-first search may well get lucky and reach a

continued on page 92
goal quickly, having explored relatively few false leads along the way. A breadth-first search, on the other hand, fans slowly downward, looking at the entire width of the graph until it reaches a goal. If all of the goals are far removed from the start node, the breadth-first search may take an intolerably long time.

Another concern with using a breadth-first search is memory usage. With a depth-first search, only the links that lead from the start node to the node currently being investigated need to be remembered in order to recapture the final path. A breadth-first search needs to remember all of the links in the whole bushy tree that it's built in order to function, since the search jumps to far-removed sections of the tree as it progresses. This exorbitant memory requirement prevents practical implementations of logic programming languages, which are based on a breadth-first search.

It's normally better to use a depth-first search when the search graph, as viewed from the perspective of the start node, is long and deep. If the search graph is short and wide, breadth-first searching is more appropriate. Both methods are prone to considerable difficulties, and it's often necessary to provide additional information about the graph (over and above the successor relation) in order to obtain an effective technique. This is what a best-first search tries to do.

BEST-FIRST SEARCHES

In order to describe the best-first search, it's necessary to make one additional assumption about our graph. Suppose that there is a rule by which two nodes in the graph can be compared in order to select which node is more likely to lead to a goal node. Such a rule is typically based on heuristic knowledge about the particular problem being solved, and the rule may be quite inaccurate. As an example, take the number of each node in our sample graph as a measure of the "goodness" of the node. The higher the number, the more likely our heuristic thinks that the corresponding node will lead to success.

The description of the best-first search is clear. Always expand the node whose heuristic value is the largest of all the nodes in L—in other words, follow your best guess. Whether this represents an improvement over the earlier blind methods depends entirely upon how good the heuristic is. With a typical "good but not perfect" rule, a best-first search explores the graph in a depth-first fashion for a while. If success is not forthcoming, the rule causes a jump to another part of the graph in a manner similar to the breadth-first search. A carefully chosen heuristic can often get the best of both worlds. In this case, it's worth noting that the list L behaves like a priority queue.

A best-first search yields the path (0,7,10,9,14) when applied to Figure 1. The best-first search finds this path with less exploration of the graph than either the depth-first or breadth-first searches, because it only looks at one short deadend (involving the step from 10 to 4).

IN TURBO PROLOG

A complete implementation of all three search techniques is given in SEARCH.PRO (Listing 1). Since Turbo Prolog is perfectly suited to problems of this type, the majority of the code is straightforward. I'll touch only on the highlights here, paying particular attention to those items which need to be changed in order to handle different problems.

The vertices of the search graph are represented by entries of type node. In Listing 1, node is simply a new name for integer. In general, the definition of the node domain should be modified to fit the problem at hand. The remaining domains—pointer, pointers, and path—need no adjustment for other problems. The graph itself is described by the predicates start_node, goal_node, and arcc. Naturally, the clauses for these predicates must be modified to apply to other search spaces.

The predicates search and continue_search form the heart of the search mechanism. The first clause of search says that the search (of whatever type) is over if the first node in the list of unexpanded nodes is a goal node. If this is not the case, the second clause of search expands the first (unexpanded) node and passes its arguments, together with the list of new nodes that it found, to continue_search. In turn, continue_search checks to see if a goal node is to be found among the newly generated nodes. If a new goal node is found, the search is over; otherwise, the new nodes are marked so that they will not be found again and are then merged into the list of unexpanded nodes. Finally, the pointer list is updated and control is passed back to search.

The difference between the three search techniques manifests itself in the merge predicate. For the depth-first search, new nodes are appended to the front of the list of old nodes. For the breadth-first search, new nodes are appended to the end of the list. In the case of the best-first search, an insertion sort inserts the new nodes into the list of old nodes. The only predicate related to merge that requires modification for other problem setups is better, which determines the ordering of nodes in the best-first search.

NAVIGATIONAL CONTROLS

These search methods can provide adventurous programs with reliable controls. In general, blind searches should only be used in situations where no guiding information is available—they're the hallmark of programs that are intended to be of such general application that no particulars can be assumed. The time spent in manufacturing an accurate navigational heuristic pays sizable dividends in performance.

Dr. Robert Crawford is a professor of computer science at Western Kentucky University.

Listings may be downloaded from Library 1 of CompuServe forum BPROGB, as SEARCH.ARC. Listing begins on page 94
It's Easy To See Why Quattro Is The Spreadsheet Of Choice!

In fact, it's hard not to see. Because one look at Quattro shows you a lot more for your money. More speed, more power, and the most spectacular presentation-quality graphics anywhere—built in.

Dazzling and diverse

If you went out looking, you'd be hard pressed to find spreadsheet graphics as dazzling and diverse as Quattro's. If you did, they'd be in a separate standalone package with a separate standalone price. And they still wouldn't be integrated with your spreadsheet's menu commands the way Quattro's are.

Brilliance built in

Quattro lets you choose from 10 different types of presentation-quality graphs and a huge selection of fonts, fill patterns and colors. Quattro supports PostScript too. So you can use today's most popular laser printers and typesetters to make your work—and yourself—look positively brilliant.

Hard copy made easy

Quattro makes it easy to get hard copies of your graphics—with a printer or plotter, directly from the spreadsheet. In fact, you don't even have to leave the spreadsheet.

Seeing is believing!

Dazzling graphics are just one of Quattro's eye-opening features; your dealer can show you the others. Quattro is easy to use and fully compatible; it even accepts familiar 1-2-3® compatible commands and uses data files created with other spreadsheets and databases. But Quattro gives you a lot more—in fact, twice the speed and power of the old standard. For only half the price.

60-Day Money-back Guarantee*

For the dealer nearest you call (800) 543-7543

"Quattro contains the most comprehensive presentation graphics capability available in a spreadsheet... The graphs Quattro can produce surpass even those available through add-on products like Lotus Graphwriter or Freelance Plus. If Borland wanted to, it could certainly sell the graphics portion of the spreadsheet on its own merit as a standalone graphics application.

Robert Alonzo, Personal Computing

Quattro's presentation-quality graphics output capabilities rival those that 1-2-3 can obtain only in conjunction with separate presentation graphics software... For me, at least, Quattro has certainly become the character-oriented spreadsheet program of choice.

William Zachmann, Computerworld

In the few years since Lotus Development Corp. introduced 1-2-3, many companies have attempted to unseat the king of the spreadsheet hill. The latest contender, Borland International Inc.'s Quattro, succeeds where other spreadsheet packages have failed... Quattro is at least two steps ahead of 1-2-3.

Ricardo Birmele, PCResource"
/* Graph Search */
node = integer /* Modify this to fit the problem */
pointer = pointer
path = pointer

/* Predicates defining the search space */
/* Change these to fit the problem */
start_node(node)
goal_node(node)
reverse(node)

clauses
start_node(0).
goal_node(13, 14, 15).
arcc(0, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15).

/* General purpose predicates */
better(node, node) /* Modify this to suit the problem. */
mark(node)

clauses
better(X, Y) :-
X > Y.
mark(X) :-
not(mark(X)).
markall(T).
insert(X, T, X).
insert(X, T, New) :-
better(X, New),
insert(X, New, New).
merge( "depth", New, R, New) :-
append(New, R, New).
merge( "breadth", New, R, New) :-
append(R, New, New).
merge( "best", New, R, New) :-
append(New, R, New).
merge( "best", New, R, New) :-
append(R, New, New).
update_pointers(N, New, Path) :-
update_pointers(N, Path, New, New).

goal,
markall(T),
my_retractall(mark),
write("What type of search (depth, breadth, best)? "),
readin(Type),
clearwindow,
start_node(S),
search(Type, S, TheGoal, Path),
write("A path leading from a start node to this goal is:
"),
write(Path),
nl, nl.

/* The search mechanism */

predicates
better(node, node) /* Modify this to suit the problem. */
search(string, path, pointers, node, path)
continue_search(string, path, pointers, node, path, path)
merge(string, path, pointers, node, path, path)
findpath(pointers, path, path)
markall(path)
update_pointers(node, path, pointers, pointers)

clauses
better(X, Y) :-
X > Y.
unmarked_successor(N, M) :-
arcc(N, M),
not(mark(M)).
goal_node(TheGoal),
findpath(P, TheGoal, Path),
1.
search(Type, [N | R], P, TheGoal, Path) :-
findall(X, unmarked_successor(X, N, New),
continue_search(Type, [N | R], P, TheGoal, Path, New).
continue_search(_, [N | R], P, TheGoal, Path, New) :-
member(TheGoal, New),
goal_node(TheGoal),
findpath(P, [N, TheGoal], Path),
1.
continue_search(Type, [N | R], P, TheGoal, Path, New) :-
markall(New),
merge(Type, New, R, New),
update_pointers(K, New, P, New),
search(Type, New, P, TheGoal, Path).
findpath(P, [N | T], Path) :-
member(ptr(X, H), P),
findpath(P, [X, H, T], Path).
findpath(_, Path, Path).
markall(I) :-
assert(mark(I)).
markall(T).
markall(I).
insert(X, T, [Node, H | T]) :-
better(Node, H),
insert(Node, T, NewT) :-
insert(Node, H, New). insert(Node, T, New) :-
insert(Node, T, New).
merge("depth", New, R, New) :-
append(New, R, New).
merge("breadth", New, R, New) :-
append(R, New, New).
merge("best", New, R, New) :-
append(New, R, New).
merge("best", New, R, New) :-
append(R, New, New).
update_pointers(N, New, Path) :-
update_pointers(N, Path, New, New).

typedef Window("depth", 1, 7, "breadth", 5, 5, "best", 10, 65),
my_retractall(mark),
write("What type of search (depth, breadth, best)? ")
readin(Type),
clearwindow,
start_node(S),
search(Type, S, TheGoal, Path),
write("A path leading from a start node to this goal is:
"),
write(Path),
nl, nl.
TAKING TO THE SCREEN

Take control of the Turbo Prolog Toolbox for your next generation of screens.

Gaylen Wood

The Turbo Prolog Toolbox offers an array of screen layout tools that allow you to easily design input screens. One such tool, the *screen definition* tool, is a program that lets you interactively design a form on the screen. Once the screen has been designed, the screen definition program saves the Turbo Prolog description of this screen as database facts. This definition file can then be consulted by other programs. With the aid of other tools in the toolbox, called *screen handlers*, the program displays and uses the screens that are defined by the screen layout tool. This approach allows the programmer to design a screen visually, rather than by the trial-and-error methods that are usually required by programming the screen manually.

A problem that arises is that many of the keys that are used by the screen handlers for input, such as the Tab key or the F10 key, are predefined to perform in a specific manner. Other keys, including most of the function keys, are not defined at all. These tools must be modified during the development of a “user familiar” application so that they perform in a way that the end user expects. Fortunately, the source code for the screen handling tools is included in the Turbo Prolog Toolbox, and it’s relatively easy to modify them to suit your specific needs.

In this article, I’ll explain how to modify these tools to emulate a specific user interface. In particular, I’ll show how to enable all of the function keys, and how an additional key for user input can be defined. I’ll also show how the Tab function can be given a “wrap around” capability, and we’ll look at a method for correcting the cursor position when a field is full. Finally, I’ll define a function to “back tab” from the middle of an input field. The specific changes that are involved in these tool modifications may not be of interest to everyone. The modification techniques, however, should interest anyone who wishes to customize input screens.

**THE BASICS**

The process of creating a screen with the screen layout tool SCRDEF.PRO (which is on the distribution disk) is fairly straightforward, and is described in Chapter 3 of the *Turbo Prolog Toolbox User’s Guide*. The result of this screen creation process is a consult file that describes text and input/output fields. This file is ready for consulting by the application program, and contains database facts that correspond to the following:

- `field(FName,Type,Row,Col,Length)`
- `textfield(Row,Col,Length,FieldString)`
- `windowsize(Height,Width)`

Once the screen values have been consulted, the presentation of the screen and the acceptance of input are handled by the tools in SCRHND.PRO. All screen handling capabilities can be invoked by a single call to the tool `scrhnd`:

```
scrhnd(STATUSON,KEY):-
   settopline(STATUSON),
   mkheader,
   writescr,
   field(FNAME,__,R,C,__),!,
   cursor(R,C),
   chng_actfield(FNAME),
   showcursor,
   repeat,
   writescr,
   keypressed,
   readkey(KEY),
   scr(KEY),
   showcursor,
   endkey(KEY),!.
```

The predicates `settopline` and `mkheader` establish a top line status window. The fields and associated screen text are then presented by `writescr`. The cursor is placed into the currently active field, which is defined by `chng_actfield`. Finally, `showcursor` displays the cursor's row and column position in the top line status window.

Processing begins with the `repeat` loop, which presents the fields and screen text with `writescr`. The `keypressed` predicate keeps the program “idling” un-
DEFINING NEW KEYS

The first changes to be made to the screen handling tools define a new key for user input and enable the use of all ten function keys. My particular user environment requires the + key located next to the numeric key pad to be used as an input key—after filling in the fields on the screen, the user presses the + key to tell the computer that input is finished. An additional requirement of my application is that the user terminate the session by using any of the function keys. (The original tool only provides the F10 key or the Esc key for this purpose.) Naturally, the function keys can be defined to perform any action you wish.

To define a new key, we must first look at the object KEY in TDOMS.PRO (provided on the Turbo Prolog Toolbox distribution disk). TDOMS declares the domain names for all of the keys that are recognized by the tools. To define the new key, simply pick an appropriate symbolic name and add that name to the domain list. (I chose the symbolic name plus.) Note that function keys are already defined by the domain declaration:

```prolog
fkey(INTEGER)
```

There is no need to modify this declaration. The new version of TDOMS is shown in Listing 1.

The readkey predicate, which reads an input character and returns its symbolic name, must now be modified to recognize the new key. readkey and its associated predicates can be found in TPRES.PRO (also on the distribution disk). readkey reads a character from the keyboard, converts that character into its ASCII code equivalent, and passes that code to readkey1. Extended keys, such as the function keys or Ctrl-key sequences, actually generate two characters; the first value for an extended key is always 0. If readkey1 detects an extended key, the rest of the ASCII code is passed to readkey2, as shown in the following code:

```prolog
readkey1(KEY,_,0):-
    readchar(T),
    char_int(T, VAL),
    readkey2(KEY, VAL).
readkey1(cr,_,13):-!.
readkey1(esc,_,27):-!.
readkey1(break,_,3):-!.
readkey1(bdel,_,8):-!.
readkey1(ctrlbdel,_,127):-!.
readkey1(fkey(0),_,43):-!.
readkey1(char(T),T,_) .
```

The + key has an ASCII code of 43, and doesn't generate an extended key code. Therefore, the + key is included by simply adding another readkey1 clause:

```prolog
readkey1(fkey(plus),_,43):-!.
```

Again, fkey is already defined in readkey2, and there's no need to modify its definition.

The modified version of TPRES.PRO is shown in Listing 2.

```prolog
SCRHND.PRO defines the actions that will be initiated by each of the function keys and by the + key. At this point, all ten function keys can be enabled. First, add a clause to scr for each additional key. Inspection of the clauses for scr reveals that a clause for fkey(10) is already present. Therefore, clauses need to be added only for function keys 1 through 9, and for the + key.
```

```prolog
scr(fkey(1)) :- not(typeerror),!.
scr(fkey(2)) :- not(typeerror),!.
scr(fkey(3)) :- not(typeerror),!.
scr(fkey(4)) :- not(typeerror),!.
scr(fkey(5)) :- not(typeerror),!.
scr(fkey(6)) :- not(typeerror),!.
scr(fkey(7)) :- not(typeerror),!.
scr(fkey(8)) :- not(typeerror),!.
scr(fkey(9)) :- not(typeerror),!.
scr(plus) :- not(typeerror).
```

The functioning of the Tab key is defined in the clause scr(tab) in SCRNHND.PRO, as shown:

```prolog
scr(tab):-
    cursor(R,C),
    nextfield(R,C).
```

nextfield(R,C) determines the current cursor position. nextfield establishes the next field in the sense of left to right, top to bottom:

```prolog
nextfield(_):=typeerror,!,fail.
nextfield(R,C):-
    field(FNAME,_,ROW, COL,_,)
    ,gtfield(ROW,R,COL,C),
    chang_actfield(FNAME),!,
    cursor(R,COL).
```

The first clause simply verifies that the definitions of the fields are consistent, and then it fails. Invalid fields are skipped. The second clause succeeds until the cursor is in the last field. At that point, a field whose cursor values qualify it as the "next field" cannot be found, and the second clause fails. Turbo Prolog backtracks to the next clause, which always succeeds. As the clause is currently written, however, no action is taken—nothing happens on the screen. To make the Tab key wrap around, simply change the third clause to:

```prolog
nextfield(_):=scr(home).
```

Now, when the second clause of nextfield fails, the third clause always succeeds, and the cursor is placed in the first defined field of the screen.

continued on page 98
SideKick Plus Gives Your PC the Power of Communication!

Get full communications capabilities without leaving Quattro—or any other application you’re using.

Online Help is always available.

It’s a full-fledged communications program for data and voice... plus a lot more!

Communication is power. And with SideKick* Plus, it’s at your fingertips. Because SideKick Plus is the only communication software you need. To send your message around the world. Or to pick up messages from MCI or Dow Jones or any other electronic service. Automatically—even if you’re down the hall in a meeting. Or doing something else you do on your PC. (Try asking CrossTalk* software to do that!)

SideKick Plus saves you time and keystrokes with sample scripts for popular programs like MCI Mail, CompuServe.* and BIX.* You can create scripts by simply recording your keystrokes, or edit scripts to access the full power of the script language.

Turbo charge your Phonebook

SideKick Plus lets you create the most high tech address books you’ve ever seen—entering names and addresses in the form you choose. Searching electronically for the information you need. And attaching notes and comments about each person listed.

SideKick Plus is communications and more: seven powerful software packages in one!

- A complete outliner that lets you open nine files at once
- A sophisticated DOS file manager
- A calendar you can use as a common scheduler if you’re on a local area network
- Multiple notepads
- A phonebook with full communications
- Your choice of four different calculators
- An ASCII table

Plus:

- Support for both expanded and extended memory. If you have an Intel Above* Board, you can take full advantage of your 640K of RAM and yet use all your SideKick Plus desk accessories at any time.
- All completely integrated and instantly accessible over any other application you’re working in
- All taking up as little as 72K of your computer’s RAM

Minimum System Requirements: For IBM PS/2, IBM family of personal computers, and all 100% compatibles. Operating system: PC-DOS (MS-DOS) 2.0 or later. Minimum system memory: 384K bytes. Minimum resident memory size: 72K. Hard disk required. Supports both EMS and extended memory.

*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.

Ready-to-use scripts make it easy to log onto MCI, CompuServe, or BIX

Define your password and encrypt it for security

Add in your local access number with a simple entry

“...Unlike most communications programs (including some that cost twice as much as SideKick), the new SideKick lets your computer communicate with another machine while you are running another program.

—Lawrence Magid, Washington Post

Get the power!

To buy this kind of communication power and all the other SideKick Plus features separately, you’d spend hundreds of dollars and drain your computer’s memory dry. Instead, just see your Borland dealer and get the power of SideKick Plus!

Hard disk required.

60-Day Money-back Guarantee*

For the dealer nearest you

Call (800) 543-7543

All Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks of their respective holders.

Copyright ©1987 Borland International, Inc.
TRACKING MECHANISM TO DO THE WORK. THE FOLLOWING CODE DEMONSTRATES THIS PROCESS:

```
prevfield(R,_,fail).
prevfield(R,C):-
field(FNAME,_,ROW,_,COL),
chk_found(FNAME,R,C,ROW,_,COL),!,
actfield(F1),
field(F1,_,RR,_,CC),!,
cursor(RR,CC).
```

The second clause of `chk_found` now checks if the current cursor position, which is provided by `prevfield`, is located in a defined field. If the current cursor position is in a defined field, then `chk_found` establishes that field as the currently active field, and allows `chk_found` to succeed.

Now, when the cursor is located in the middle of a field and the back tab function is used, the cursor returns to the first character of that field. If used further, the back tab function will act as originally defined.

If you're using Turbo Prolog 2.0, you must make one other change. SCRHD defines a predicate called `trunc` to truncate strings. In Turbo Prolog 2.0, `trunc` is a built-in predicate that truncates a real number and returns its integer value. Therefore, you need to change the name of the toolbox predicate from `trunc` to something else, such as `trunc`.

Listing 3 incorporates all of the changes that were made to SCRHD.PRO. The file TESTPROG.PRO (Listing 4) contains a short program that tests the changes. (HNDBASES.PRO from the distribution disk was used as a template for creating this test program.) Run these programs and observe the changes. I'm sure you'll find that your own personal requirements can also be easily incorporated into the already powerful Turbo Prolog Toolbox.
LISTING 1: XTDOMS.PRO

/* Listing 1 : XTDOMS.PRO */

/****************************
DOMAINS

ROW, COL, LEN, ATTR = INTEGER
STRINGLIST = STRING
INTEGERLIST = INTEGER*

KEY = cr; esc; break; tab; btab; del; bdel; ctrlbdle; ins; end; home; fkey(INTEGER); up; down; left; right; ctrlleft; ctrlright; ctrldel; ctrldelhome; pgup; pgdn; ctrptup; ctrptdown; char(CHAR); plus; otherspec

LISTING 2: XTPREDS.PRO

/* Listing 2 : XTPREDS.PRO */

/****************************
PREDICATES

nondeterm repeat

CLAUSES

/****************************
*/

CLAUSES

nondeterm repeat

/****************************
*/

CLAUSES

maxlen(STRINGLIST, COL, COL)

/****************************
*/

CLAUSES

maxlen(T[1], MAX, MAX1) :-

clist(t1, 1, LEN, LEN),
maxlen(t1, LEN, MAX1),
maxlen(t1, T[1], MAX),
maxlen(t1, T[1], MAX1).

clist(t[1], T[1], MAX),
maxlen(T, T[1], MAX),
maxlen(T, T[1], MAX1).

clist(0, T[1], MAX),
maxlen(T, T[1], MAX),
maxlen(T, T[1], MAX1).

clist(0, 0, N),
maxlen(T, T[1], MAX).

writestring(LIST, L, LIST, LIST).

writestring(list1, L, LIST, LIST).

writestring(list2, L, LIST, LIST).

field_str(t1, 0, ANTKOL, H),
L1=L+1,
write(list1, L1, ANTKOL, T).

min(X, Y, Z) :- X<Y, Z.
min(X, Y) :- X<Y, Y.

max(X, Y, Z) :- X>Y, Z.
max(X, Y) :- X>Y, Y.

reverseattr(A, Z) :-
bitand(A, 1077, 112),
bitand(h11, 64, h22),
bitand(h22, 4, h22),
bitand(A, 1088, h22),
A2=bit2(h22, 63).

reversed(A2, Z) :-
bitand(A2, 1077, 112),
bitand(h11, 64, h22),
bitand(h22, 4, h22),
bitand(A2, 1088, h22),
A2=bit2(h22, 63).

/****************************
*/

PREDICATES

upc(CHAR, CHAR) :-
upc(CH, CH).

lowc(CHAR, CHAR) :-
lowc(CH, CH).

try_upper(CHAR, STRING) :-
try_upper(CHAR, STRING, STRINGLIST, ROW, ROW).

tryfirstupper(CHAR, STRINGLIST, ROW, ROW) :-
tryfirstupper(CHAR, STRINGLIST, ROW, ROW).

tryletter(CHAR, STRINGLIST, ROW) :-
tryletter(CHAR, STRINGLIST, ROW).

/****************************
*/

CLAUSES

upc(CHAR, CHAR) :-
lowc(CHAR, CHAR).

try_upper(CHAR, STRING) :-
try_upper(CHAR, STRING, STRINGLIST, ROW, ROW).

tryfirstupper(CHAR, STRINGLIST, ROW, ROW) :-
tryfirstupper(CHAR, STRINGLIST, ROW, ROW).

tryletter(CHAR, STRINGLIST, ROW) :-
tryletter(CHAR, STRINGLIST, ROW).

/****************************
*/

CLAUSES

adjframe(ATTR, ROW, COL, ROW, COL) :-
adjframe(ATTR, ROW, COL, ROW, COL).

adjustwindow(ROW, COL, ROW, COL, ROW, COL) :-
adjustwindow(ROW, COL, ROW, COL, ROW, COL).

/****************************
*/

CLAUSES

adjustwindow(L1, KOL, DOL, DKOL, ALL, AKOL) :-
L1<25-DOL, KOL<80-DKOL, ALL=LI, AKOL=KOL.

adjustwindow(L1, KOL, DOL, ALL, AKOL) :-
L1<25-DOL, ALL=LI, AKOL=KOL.

adjustwindow(L1, DOL, DKOL, ALL, AKOL) :-
KOL<80-DKOL, ALL=LI, AKOL=KOL.

adjustwindow(DOL, DKOL, ALL, AKOL) :-
AKOL=KOL.
adjframe(0,R,C,R,C):-1.
adjframe(R1,C1,R2,C2):-R2=R1+2, C2=C1+2.

/* Returns a symbolic key from the KEY domain */
readkey

Modified 2/5/88 G.Wood

Added readkey clause for symbolic key 'plus' with ASCII 43*/

readkey1 (KEY ,_,D1: - ! , readchar(T) ,char_ int(T, VAL), readkey2CKEY, VAL).
readkey2CKEY): - readchar(T) ,char_ int CT, VAL), readkey1 (KEY, T, VAL).
readkey1(tab,_,9):-!.
readkey1(esc,_, 27): - ! •
readkey1(cr ,_, 13) :-! •
readkey2(del ,83) :-! •
readkey2Cbtab, 15):-!.
readkey1(char(T), T ,_l
readkey1(plus, ,43):-1.
readkey1(ctrlbdel ,_, 127):-1.
readkey1 (bdel ,_,8) :-1.
readkey2(pgdn,81 ):- ! •
readkey2Cpgup, 73>:-!.
readkey2Cright,7S):-!.
readkey2Cdown,8D> :-! •
readkey2Cup, 72) :-1.
readkey2Cins,82):-!.
readkey2Cctrlpgdn, 118) :-! •
readkey2Cctrlend, 117):-1.
readkey2Cctr l right, 116): · ! •
readkey2Cctrlleft, 115) :- 1.
readkey2(hane, 71): - ! •
readkey2(end,79):-I.
readkey2Cfkey(N),VAL):- VAL>=84, VAL<1D4, N=11+VAL-84, I.
readkey2Cfkey(N),VAL):- VAL>58, VAL<7D, N=VAL-58, I.
readkey2Cctrlhane, 119): - !.
readkey2Cotherspec,_).
readkey2CKEY, INTEGER)
readkey1 CKEY, CHAR, INTEGER)

Modified 2/5/88 G.llood

See clauses scr

* Create the window automatically from the
* Window size predicate.
* */
trunc(Len,Str1,Str2):-str_len(Str1,L1),L1>Len, 
trunc(_,Str,Str).
settopline():-retract(not(topline)),fail.
settopline(off):-1,assert(not(topline)).
settopline(_).
oldstr(FNAME,S):-value(FNAME,S),!.
oldstr(_,n).
ass_val(FNAME,_,_):-retract(value(FNAME,_,_)),fail.
ass_val(FNAME,VAL):-VAL=\(+,,\+assert(value(FNAME,VAL)),fail.
chng_actfield(FNAME):-typeerror,\+,fail.
chng_actfield():-assert(actfield(FNAME)).
chk_found(FNAME,ROW,Col,ROW,Col,LEN,ROW,Col,Len):·
chk_found(_,ROW,Col,ROW,Col,LEN,ROW,Col,Len):·
chk_found(FNAME,ROW,Col,ROW,Col,LEN,ROW,Col,Len).
setlastfield.

CLAUSES
best_right(R0,CO,R1,C1,ROW,Col,):·
field(_,R2,C2,_),C2>CO,
better_right(R0,R1,R1,C1,R2,C2,1),
better_right(R0,CO,R2,C2,ROW,Col,).
best_right(_,R0,C0,R1,C1,C2,):·
better_right(R0,R1,R1,C1,R2,C2,1),
better_right(R0,R1,R1,R1,R2,C2,C1).
best_left(R0,CO,R1,C1,ROW,Col,):·
field(_,R2,C2,_),C2>CO,
better_left(R0,R0,R1,R1,C1,R2,C2,1),
best_left(R0,R2,C2,ROW,Col,).
best_left(_,R0,C0,R1,C1,C2,):·
better_left(R0,R1,R1,R1,R2,C2,C1),
better_left(R0,R2,C2,R2,C2,1).
best_up(R0,CO,R1,C1,C0,ROW,Col,):·
field(_,R2,C2,L2),R2>RO,
field(R0,R0,R1,R1,C1,R2,C2,1),
best_up(R0,R0,R1,R1,C1,R2,C2,1),
best_up(R0,R2,C2,R2,C2,1).
best_up(_,R0,C0,R1,C1,R1,C1,R2,C2,1),
best_up nier(R0,R2,C2,R2,C2,1).
field(R0,R0,R1,R1,C1,R2,C2,1),
field(_,R2,C2,L2),R2<RO,
best_up(R0,R0,R1,R1,C1,R2,C2,1),
best_up(R0,R2,C2,R2,C2,1),
best_up(_,R0,C0,R1,C1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,C1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
best_up(_,R0,C0,R1,R1,C1,R2,C2,1),
...
```c
nextfield(_,_,_):=-typeerror,
nextfield(R,C):=  
    field(NAME,_,ROW,COL,_,),
    chng_actfield(NAME),
    chng_actfield(NAME),!
    cursor(ROW,COL),
    /* nextfield(_,_,_):=-typeerror. 

setlastfield:=-
    field(C FNAME),
    cursor(ROW,COL).  
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    chng_actfield(ROW,C),! 
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    field(NAME,_,_,C,_,),!
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*

setlastfield:=-
    field(NAME,_,_,C,_,),
    gtf_field(ROW,R,C),
    actfield(ROW,C),
    changemode,
    cursor(ROW,C)./*
```

/* Modified 2/5/88 - G. Wood
   * Commented out nextfield(_,_,_), and replaced with indicated clause.
   * This will allow the scr(tab) clause to "wrap around" from last
   * field to first field, and changes to scr(right) to allow filling
   * last field and "wrap around" to first field.
   *******************************************************

/* Insert a new character in a field */
scr(char(T)):=-
    actfield(NAME),
    not(noinput(NAME)),
    scr(char(T)),
    nextfield(C cursor(C),C)./*

/* Delete character under cursor */
scr(del):=-
    actfield(NAME),
    not(noinput(NAME)),
    cursor(C),
    field(NAME,_,ROW,COL,LEN,0),
    field_str(ROW,COL,LEN,STR2),
    ass_val(NAME,STR1),
    cursor(ROW,COL)./*

/* Delete character before cursor and move cursor to the left */
scr(bdel):=-
    actfield(NAME),
    not(noinput(NAME)),
    cursor(C),
    field(NAME,_,ROW,COL,LEN,1),
    field_str(ROW,COL,LEN,STR1),
    ass_val(NAME,STR2),
    cursor(ROW,COL)./*

/* If there is an action - do it. Otherwise, go to next field */
scr(cr):=-
    actfield(NAME),
    field_action(NAME),
    cursor(RR,CC),cursor(RR,CC),!
    scr(tab)./*

/* Change between insertmode and overwritemode */
scr(ins):=changeo,
nextfield(R,C)./*

/* F10: end of definition */
scr(fkey(10)):=-not(typeerror)./*

scr(fkey(5)):=-not(typeerror)./*
scr(fkey(6)):=-not(typeerror)./*
scr(fkey(7)):=-not(typeerror)./*
scr(fkey(8)):=-not(typeerror)./*
scr(fkey(9)):=-not(typeerror)./*
scr(fkey(10)):=-not(typeerror)./*
scr(fkey(11)):-help. If helpsystem is used. */
```

102 TURBO TECHNIX September/October 1988
/** Predicates maintaining the top messages line */

/* UPDATE ALL FIELDS ON THE SCREEN */
************************************************************************************************************************************************
/* Shift screen */
************************************************************************************************************************************************

/** Predicates */

get_overwritestatus(STRING)
show_str(COL, LEN, STRING)
showfield(R, COL)
get_overwrite_status(INSERT): insmode,
get_overwrite_status(INSERT): overwrite.
show_str(C, L, STR): -
show_overwrite: notopl ine,
show_overwrite: -keypressed
show_overwrite: -insertmode,
show_overwrite: -keypressed

/*showfield(R, C): -keypressed,
showfield(R, C): notopl ine,
showfield(R, C): -keypressed

showcursor: notopl ine,
showcursor: -keypressed,
showcursor: -insertmode,
showcursor: -keypressed

**** ** ******* **** ****** ** ** *** ***************************
*/

/** DATABASE */

/* Database declarations used in vscrhnd */
insmode /* Global insertmode */
actfield(FNAME) /* Actual field */
screen(FNAME, DATABASE) /* Saving different screens */
valuedefine(FNAME, STRING) /* Value of a field */
txtfield(R, COL, LEN, STRING) /* Screen definition */
windowstart(R, COL, LEN)

notopl ine

/* DATABASE PREDICATES USED BY VSCRHND */

windowsize(R, COL, LEN, STRING)

notopl ine

** Database declarations used in lineinp */
lineinpstate(STRING, COL)
lineinpflag

/* Include tools */
include "xtdams.pro"
include "xtpreds.pro"
include "menu.pro"
include "status.pro"
include "lineinp.pro"
include "vscrhnd.pro" /* Or vscrhnd.pro */

/** CLAUSES */

/* Field action */
field_action(R, C): fail.

/* Field_value */
field_value(FNAME, VAL): value(FNAME, VAL), !.

/* Noninput */
noninput(R, C): fail.

GOAL clearwindow,
consult("test.scr"),
createwindow(off),
scrhnd(off, EndKey),
removewindow,
write(EndKey).

LISTING 4: TESTPROG.PRO
THE TURBO BASIC/ASSEMBLER CONNECTION

Write your procedures in Turbo Basic to make them work—then rewrite them in Turbo Assembler to make them fast.

David A. Williams

Turbo Basic is so much faster than interpreted BASIC that you might wonder if it's possible to do better. It is possible, and the way is through Borland's new Turbo Assembler. If certain key routines are coded in assembly language and called by Turbo Basic, your programs will have considerably more zip. This technique gives you the best of both worlds—the convenience of Turbo Basic, and the speed of assembly language.

TO THE METAL
Turbo Basic provides three ways to tap the power of assembly language.

CALL ABSOLUTE. The CALL ABSOLUTE statement transfers control to an assembly language routine that was loaded prior to the call at a specific memory location. Although cumbersome, this method is available in order to provide a degree of compatibility with interpreted BASIC, where this technique originated. There is no reason to recommend CALL ABSOLUTE for new programs, and I'll not discuss it further in this article.

CALL INTERRUPT. When used with the REG statement and the REG function, the CALL INTERRUPT statement provides access to all DOS and BIOS interrupt service routines. This technique has a somewhat narrow application, but it does provide a way to access certain information that is not otherwise available to a BASIC program. (For more information on CALL INTERRUPT, see "DOS Calls From Turbo Basic," TURBO TECHNIX, November/December, 1987; and "Calling BIOS Services From Turbo Basic," TURBO TECHNIX, July/August, 1988.)

Turbo Basic's most general and powerful assembly language interface method involves calls to special procedures that are called INLINE procedures. INLINE procedures may include assembly language code in the form of strings of hexadecimal constants, or code may be loaded from a machine-code binary file at compile time.

INLINE PROCEDURES
A CALL statement that is used to call an INLINE procedure is identical to a CALL statement that is used to call any ordinary Turbo Basic procedure. In fact, you can design programs with all procedures in Turbo Basic, and then replace one or more of the procedures with INLINE procedures in machine code without changing the main program.

An INLINE procedure has the following structure:

```
SUB <procedure name> INLINE
$INLINE <byte list>
$INLINE "filename"
END SUB
```

Here, `<procedure name>` is the name that is used in the CALL statement to call the procedure. The $INLINE metastatement may take either a byte list of values that represent machine code instructions, or else a file of such instructions that exists separately from the Turbo Basic source file on disk. Normally you won't use both of the two forms of the $INLINE metastatement in the same procedure (but there's no harm in doing so). A single INLINE procedure may contain any number of $INLINE metastatements that specify byte lists. However, you may load up to—but not more than—16 binary files within a single INLINE procedure by naming each file within its own $INLINE metastatement.

LISTS OF BYTES
The byte list is a series of values (usually hexadecimal) that are separated by commas. Each value represents one byte of the code that comprises a machine instruction. (Machine instructions in Intel's 8086 family of processors may be anywhere from one to six bytes in length, not counting prefixes.) You can string as many values behind the $INLINE metastatement as you wish, and there's no limit to the number of $INLINE metastatements that can be used within a single INLINE procedure.
The process of entering code as a byte list after an $INLINE metastatement is best used in very short programs that contain no jump instructions or other branches. DOS's DEBUG can perform the assembly process, but instructions have to be entered one at a time to DEBUG, and then the resulting values must be keyed into the INLINE procedure by hand. Furthermore, DEBUG cannot convert labels to addresses, and can only treat each instruction in isolation from all others. Trying to hand- or DEBUG-assemble a complex routine with lots of conditional branches is the short path to insanity, due to the maddening difficulty of calculating relative jump offsets by hand.

**ENTER TURBO ASSEMBLER**

The better method by far is to load a binary file that contains machine code that was generated with an assembler. The $INLINE metastatement can accept a filename that specifies a binary file of machine code instructions, as shown below:

```plaintext
$INLINE "MYCODE.BIN"
```

This metastatement becomes the “beef” of an INLINE procedure.

It’s beyond the scope of this article to teach assembly language programming. Although Turbo Assembler is fairly new, it’s highly compatible with MASM, and books previously published for MASM programming will help you get up to speed. Some tricks will make the assemble/link process smoother and more automatic. The simple batch file below, ASM.BAT, automates the process:

```plaintext
TASM %1;
TLINK %1;
DEL %1.OBJ
EXECBIN %1 %1.BIN
DEL %1.EXE
```

Execute ASM.BAT by typing the following command:

```plaintext
ASM <filename>
```

Here, `<filename>` is the name of the assembly language source file. Do not include the source filename extension (i.e., “.ASM”). ASM.BAT performs the assembly process, the link process, and deletes the superfluous files. ASM does leave the .MAP file on disk, however, so if you don’t intend to use the .MAP information, the .MAP file must be deleted. This step can be performed manually or by the addition of another line to ASM.BAT to delete the .MAP file.

ASM.BAT produces memory-image binary files with a .BIN extension that are ready to load through the $INLINE metastatement. These files can also be given a .COM extension; since they’re not executable, however, the .BIN extension is safer and more descriptive. Contrary to the instructions in the Turbo Basic Owner's Handbook, do not include an ORG 100 directive in the Turbo Assembler source files.

**PROBING AN ARRAY**

The rest of this article provides two useful examples of assembly language extensions to Turbo Basic, and explains how those extensions are integrated into the calling program. Future issues of TURBO TECHNIX will present additional assembly language routines, along with further discussions of specific issues such as parameter passing and the access of global resources.

MAXDEMO.BAS (Listing 1) contains the source code for a simple Turbo Basic demo program that finds the largest value in an array.
integer array. TBMAX.ASM (Listing 2) is the assembly language source code file for the routine that MAXDEMO calls to do its quick-and-dirty work. MAXDEMO first creates the integer array A with 100 elements, and then calls the Turbo Basic procedure GTMAX. This procedure executes the machine code routine TBMAX to locate the largest value in the array. TBMAX executes twice as fast as any Turbo Basic routine that you could write to perform the same function.

The stack is the key link between a Turbo Basic program and any assembly language routine. All values are passed to machine code procedures by reference rather than by value. This means that the parameter's data values themselves are not passed on the stack; instead, an address that points to the memory location where each value is stored is placed on the stack by the compiler. The assembly language routine copies the address from the stack and uses that address to read the value of the actual parameter from memory, or else to store a value into memory as a means of returning a value to the calling Turbo Basic program.

The CALL to GTMAX passes three parameters to GTMAX: MAXVAL, in which the machine code routine passes back the largest array value; A(1), which is the first element of the array; and COUNT, which is the number of array elements. Since Turbo Basic stores array elements in contiguous memory locations, the entire array can be accessed once the total number of elements, and the address of the first element, are known.

The assembly language routine must preserve the values in DS, SP, BP, and SS. Any other registers may be freely changed. In the case of TBMAX, the only critical register is BP, which is pushed onto the stack. Once BP is safely on the stack, TBMAX loads the stack pointer SP into BP. Thereafter, TBMAX accesses its parameters through offsets from BP, which now points to the top of the stack.

The stack contains a 32-bit address that points to the memory location where each parameter value is stored. Figure 1 shows the stack as it exists after the PUSH BP instruction. Each "brick" is one byte of memory, with high memory at the top of the figure. The parameters can be accessed in any order. In TBMAX, the parameter that is accessed first is COUNT, which was the last one pushed. The LES instruction was designed specifically for retrieving addresses from the stack: Given the offset of the address from BP (here, +06H, where the plus symbol means that the offset is toward high memory), LES copies the segment address from the stack into ES, and copies the offset address from the stack into BX. The MOV CX, ES[BX] instruction copies the actual parameter's value from memory into the CX register. The same technique is used to generate a pointer to the first element of array A, but it's put into DI rather than into BX. Since the array count parameter COUNT was already moved from memory into CX, the original pointer to COUNT in BX is no longer needed and can be overwritten. Hence, the last step is to generate a pointer to MAXVAL, and to place that pointer into BX until it's needed later on.

The rest of TBMAX compares the value of each array element to the value in AX. If an array element is found to be larger, that element replaces the previous value in AX. Because integers are
16 bits long, TBMAX increments DI twice for each pass through the loop. (Note: When working with long integers or floating point numbers, DI must be adjusted according to the size of the base type of the array.) After TBMAX has examined each element of array A, the value in AX is moved into MAXVAL through the pointer to MAXVAL that is now in registers ES and BX. TBMAX finishes the process by popping BP off the stack.

Some notes on TBMAX: The same routine works with a multi-dimensional array if COUNT contains the total number of individual integer elements in the array. Remember, however, that the dimension indexing system starts with zero. For example, an array dimensioned as A(2,50) has 153 elements. Also, keep in mind when using an INLINE procedure that neither the procedure nor the machine code routine may contain RETURN statements. Turbo Basic takes care of that step automatically.

PASSING STRING PARAMETERS

The process of passing string values to assembly language routines is a little more subtle. SCRNDemo.BAS (Listing 3) shows a Turbo Basic demo program that incorporates Listing 4, TBQPA.ASM. TBQPA writes the designated string parameter directly to display memory at the indicated row and column location. The parameter ATTRIB allows changes to be made to the color or to other screen attributes of the screen area that underlies the string to be written. This creates very snappy screen displays and provides a degree of color control that’s not easily achieved with standard Turbo Basic statements. To keep TBQPA simple, I did not include code to prevent video snow when using IBM-style CGA boards.

Since strings have a variable length, and are stored in a different memory segment than any other variables, a different technique is needed in order to pass string parameters. When a string is passed as a parameter, Turbo Basic pushes a full 32-bit pointer to a "string descriptor" onto the stack. The string descriptor consists of a two-byte string length counter and a two-byte offset into Turbo Basic’s string data area (or string space). An assembly language routine can access a string descriptor in the same way that the routine accesses a numeric variable. The low 16 bits contain the string length, and the high 16 bits contain the offset into string space of the first byte of string data.

The instruction LES BX, [BP+12H] sets up ES and BX to point to the first byte of the string descriptor. The subsequent MOV instruction moves the string length value into CX. A minor complication with the string length counter is solved by an AND instruction: The high bit (bit 15) of the string length counter has a special meaning to the Turbo Basic Runtime code, and should not be interpreted as part of the string length value. The AND instruction masks out bit 15 to keep it out of later comparisons and calculations. Finally, the starting offset of string data within string space is moved into SI, using the instruction MOV SI,ES:[BX+02]. Note that this offset isn’t a full 32-bit address; the segment address of string space is still needed, and can be found at DS:00, which is the first word in Turbo Basic’s data segment. A little later in TBQPA, the caller’s DS value is pushed onto the stack, and then DS is loaded with the address that is found at DS:00.

In order to move data directly into video memory, the location of video memory must be known. Video memory may be at one of two addresses (B000H or B800H) depending upon which display adapter is in use. TBQPA queries BIOS interrupt 10H to identify the video adapter, sets the address of the video buffer accordingly, and then moves the string and attribute data to the video buffer via a LOOP structure. When the data has been transferred, the routine restores the critical registers DS and BP, and returns control to the calling program.

NOT SO BASIC BASIC

TBMAX and TBQPA were kept simple to emphasize the interface between Turbo Basic and Turbo Assembler, rather than the workings of the assembly language routines themselves. Once you understand how the two languages mesh, you can build on your experience and write more advanced routines. For example, it’s not especially difficult to write assembly language routines that modify string data and then pass that data back to the calling program—just remember that you can’t change the length of the string. If your application requires you to change a string length, then set up a dummy string of an appropriate length first and pass the modified string back in the dummy string, rather than in the original string.

Numeric processing and screen handling are only two of the many areas where assembly language can improve the performance of your Turbo Basic programs. Take the time to become familiar with 86-family assembly language—you’ll find that BASIC is no longer as basic as it was when you first typed RUN.

David A. Williams is a principal staff engineer for a major aerospace company. He can be reached at 2452 Chase Circle, Clearwater, FL 34624.

Listings may be downloaded from Library 1 of CompuServe forum BPROGA, as TBTASM.ARC.

Listings begin on page 108
LISTING 1: MAXDEMO.BAS

CLS
DEFINT A-Z
DIM A(100)
RANDOMIZE C157)
FOR I=1 TO 100
   A(I)>=10000*RND(9)
NEXT
MAXVAL=O
COUNT=100
CALL GTMAX(MAXVAL,A(1),COUNT)
PRINT MAXVAL
END

SUB GTMAX INLINE
SINLINE "TBMAX.BIN"
END SUB

LISTING 2: TBMAX.ASM

;TBMAX.ASM Routine to find max value in integer array
CODE SEGMENT
ASSUME CS:CODE,DS:CODE
PUSH BP ; Save BP
MOV BP, SP ; Get stack address
; Get arguments
LES BX, [BP+06H] ; Get addr of array count
MOV CX, [ES: BX] ; Put count in CX
LES DI, [BP+0AH] ; Get addr of first element
LES BX, [BP+0EH] ; Get addr of return value
; Find the max value
MOV AX, [ES: DI] ; Get first array element
A: CMP AX, [ES: DI+2] ; Compare present with next
   JU B
   MOV AX, [ES: DI+2] ; Put new, larger value in AX
   INC DI
   INC DI
   LOOP A
   MOV ES:[BX], AX ; Store max value
; Clean up and leave
QUIT: POP BP ; Restore BP
CODE ENDS
END

LISTING 3: SCRNDEMO.BAS

CLS
DEFINT A-Z
AS=“THIS IS A TEST”
ROW=16
COL=15
ATTRIB=7
CALL WRT(AS,ROW,COL,ATTRIB)
CALL WRT(“Another test”,5,20,15)
CALL WRT(LEFT$(AS,8) + “ALSO GOOD”,10,40,7)
END

SUB WRT INLINE
SINLINE “TBQPA.BIN”
END SUB

LISTING 4: TBQPA.ASM

;TBQPA.ASM Fast screen write routine for Turbo Basic
CODE SEGMENT
ASSUME CS:CODE,DS:CODE
PUSH BP ; Save BP
MOV BP, SP ; Get stack address
; Get arguments
LES BX, [BP+06H] ; Get addr of array count
MOV DI, [ES: BX] ; Put count in DI
DEC DI ; Change DI to 0 - 79
LES BX, [BP+0EH] ; Get addr of Row variable
MOV AX, [ES: BX] ; Put Row # in AX
DEC AX ; Change to 0 - 25
LES BX, [BP+12H] ; Get addr of string pointer
MOV CX, [ES: BX] ; Put string length in CL
AND CX, 7FFFH ; Remove high bit
CMP CX, 0D ; Is it zero?
JZ QUIT ; Yes, quit
MOV SI, [ES: BX+02D] ; Put string start addr in SI
; Compute offset into video buffer
MOV DX, 0D00H ; Num of char per row
MUL DX ; # rows times 80
ADD DI, AX ; Add column number
SHL DI, 1 ; Multiply by 2
; Get video parameters
LES BX, [BP+06H] ; Get address of attribute
MOV BX, [ES: BX] ; Put attribute in BX
MOV AX, [BX+000D] ; Video buffer addr, mono
MOV ES, AX ; Put it in ES
MOV AH, 0FH ; Read video mode
INT 10H
CMP AL, 7 ; Is it mono?
JE A
MOV AX, [BX+000D] ; Video buffer addr, mono
MOV ES, AX ; Put it in ES
A: PUSH DS ; Save DS on stack
   ; Copy data to video buffer
   MOV ES, [DS: 000D] ; Get string segment
   CLI ; Clear direction flag
   ; Send 1 byte to buffer
   MOV BYTE PTR ES:[DI], BL ; Attribute byte
   INC DI ; Skip attribute byte
   LOOP B ; Loop until done
; Clean up and leave
POP DS ; Restore DS
QUIT: POP BP ; Restore BP
CODE ENDS
TB runtime handles return
END

108 TURBO TECHNIX September/October 1988
Basically speaking, there's one choice... Turbo Basic!

Turbo Basic* is the BASIC that lets even beginners write polished, professional programs almost as easily as they can write their names.

The others don't. When you really examine them, you'll find that even though they may be "quick," they make it hard to get where you're going. (Sort of like a car with an engine but no steering wheel.)

Turbo Basic takes you farther faster— in the comfort of a sleek development environment that gives you full control. Naturally it has a slick, fast compiler just like all Borland's technically superior Turbo languages. It also has a full-screen windowed editor, pull-down menus, and a trace debugging system. And innovative Borland features like binary disk files, true recursion, and more control over your compiling. Plus the ability to create programs as large as your system's memory can hold.

The critics agree. The choice is basic. Turbo Basic from Borland.

... What really makes Turbo Basic special is its blinding speed, small size, and many added commands. Programs compiled with Turbo Basic are often much faster and smaller than those produced by other compilers.

Ethan Winer, PC Magazine Best of 1987

Turbo Basic, simply put, is an incredibly good product.

William Zachman, Computerworld

Add another Basic advantage: The Turbo Basic Toolboxes

- **The Database Toolbox** gives you code to incorporate into your own programs. You don't have to reinvent the wheel every time you write new Turbo Basic database programs.
- **The Editor Toolbox** is all you need to build your own text editor or word processor, including source code for two sample editors.

60-Day Money-back Guarantee*

---

**Compare the BASIC differences!**

<table>
<thead>
<tr>
<th></th>
<th>Turbo Basic 1.1</th>
<th>QuickBASIC 4.0 Compiler</th>
<th>QuickBASIC 4.0 Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile &amp; Link to stand-alone EXE</td>
<td>3 sec.</td>
<td>7 sec.</td>
<td>---</td>
</tr>
<tr>
<td>Size of .EXE</td>
<td>28387</td>
<td>25980</td>
<td>215 sec.</td>
</tr>
<tr>
<td>Execution time w/80287</td>
<td>0.16 sec.</td>
<td>16.5 sec.</td>
<td>21.5 sec.</td>
</tr>
<tr>
<td>Execution time w/o 80287</td>
<td>0.16 sec.</td>
<td>286.3 sec.</td>
<td>292.3 sec.</td>
</tr>
</tbody>
</table>

The Elkins Optimization Benchmark program from March 1988 issue of Computer Language was used. The Program was run on an IBM PS/2 Model 60 with 80287. The benchmark tests compiler's ability to optimize loop-invariant code, unused code, expression and conditional evaluation.

---

For the dealer nearest you call (800) 543-7543
COMMAND LINE PARAMETERS IN TURBO BASIC

Divide the command line string into parameters—and conquer your Turbo Basic command line entry problems.

Duke Kamstra

The ability to read parameters that are entered on the DOS command line is a powerful feature in any application or utility program. Users have come to expect applications to read information such as filenames from the command line when a language compiler or database is invoked. The Turbo Basic Integrated Development Environment is a good example. When entered at the DOS command line, the following command invokes Turbo Basic and loads the file PARAM.BAS into the editor:

`TB PARAM`

This example only uses one command line parameter, but many programs accept two or more. The widely used ARC utility, which is sold by System Enhancement Associates (Wayne, New Jersey) uses several command line parameters. The following example shows a typical invocation of ARC51.EXE:

`ARC51 A COML.ARC *.BAS *.EXE DESCRIPT.CIS`

This example command line contains five individual parameters, which are separated from one another by spaces.

BRINGING THE COMMAND LINE HOME

Most language compilers have some means of reading the command line parameters that are used by programs written in those languages. Turbo Basic’s `COMMANDS` function returns all of the command line parameters concatenated into one string. This is a great start; however, the information in the command line string isn’t really useful until the string has been separated into its individual parameters.

When accessing command line parameters, a program needs access to two pieces of information: the number of parameters that were entered, and the values of the individual parameters themselves. While Turbo Basic provides the command line string, the process of counting and separating the parameters that make up the string must be handled with additional code. In PARAM.BAS (Listing 1), I’ve provided the Turbo Basic function `FNParamCount%()`, which returns the number of parameters; and `FNParamStr$(i)`, which returns individual parameters by number.

If you’ve done some Turbo Basic programming, PARAM.BAS should not be difficult to understand. `FNParamCount%()` handles the bulk of the work for both functions. When `FNParamCount%()` is called for the first time, it divides the command line string into individual parameters, and then stores the parameters in the global array `Parameters$()`. At the same time, `FNParamCount%()` counts the number of parameters that it stores, and saves that count value in the STATIC variable `Result%`. `Result%` becomes the value returned by `FNParamCount%()` to the calling program. After the first time it’s called, `FNParamCount%()` does not need to process the command string any further; when called again, `FNParamCount%()` simply returns the value it already stored in `Result%`.

`FNParamCount%()` separates parameters by scanning for separator characters (which may be either spaces or double quotes) in the string that is returned by `COMMANDS$.` Each time `FNParamCount%()` finds a separator, the left and right character positions of the found parameter are recorded in a two-dimensional integer array, `ParamPos%()`. `ParamPos%()` contains up to 25 pairs of integers (each integer pair consists of a left and a right character position value). This limits the number of parameters that may be extracted from the command string to 25. Since DOS limits the size of the command string to 127 characters, however, the maximum of 25 parameters should not be a crippling limitation. Once the initial scan for separators is complete, `FNParamCount%()` loops through the command string a second time, and copies the

continued on page 112
**LISTING 1: PARAH.BAS**

```
Author: Duke Kamstra

Mod. Date: 5/8/88

To use these routines in your own program, keep them in an include file. When you need to manage command line parameters in a program include these routines by inserting the next statement:

```
$INCLUDE "PARAM"
```

In your program. Be sure to set the named constant

MAXPARAMETERS appropriately for your application. If the number of parameters given on the command line is larger than MAXPARAMETERS the extras are ignored.

MAXPARAMETERS = 25 ' Maximum # of parameters that can be read by the program. Should never be larger than 64 since DOS only allows a 127 character command line.

DIM Parameters$(0:MAXPARAMETERS) ' String array used to store parameters

STYLE = 1 ' Named constant representing boolean value

DEF FNParamCountX
' Return the number of command line parameters passed to the program. Store each of the parameters in the SHARED string array Parameters$(). Note the function will only process up to MAXPARAMETERS command line parameters.

' The first time the function is called it processes the parameter list and sets a flag InitializedX to indicate that the command line doesn't need to be processed again. Any subsequent calls to the function will return the value stored in ResultX.

STATIC InitializedX ' Flag indicating parameters have been read and data structure has been initialized.

STATIC ResultX ' Store result after calling the function the first time.

SHARED Parameters$() ' Global variable to store parameter data

LOCAL IX, JX, CountX, ParmcountX, SearchCharX

XL = 0 ' Named constants used to reference ParmPos%

SR = 1

DIM ParmPos%(0:MAXPARAMETERS, XL:SR) ' Make room for positional information

IF InitializedX <> TRUE THEN ' We haven't parsed the command line yet
  ' Set flag indicating we've parsed the command line
  InitializedX = TRUE

IF COMMANDS = "" THEN ' No command line parameters specified
  FNParamCountX = 0 ' Return 0 for parameter count
  ResultX = 0 ' Save parameter count in static variable
  EXIT DEF ' Leave the function

ELSE ' At least one command line parameter was specified
  ' First we need to determine the number of parameters
  IX = 1

  WHILE (IX <= LEN(COMMANDS )) AND (CountX < MAXPARAMETERS)
    CountX = CountX + 1 ' Increment parameter count
    ParmPos%(CountX, XL) = IX ' Store left position of parameter
    ParmPos%(CountX, XR) = LEN(COMMANDS ) + 1 ' Store right position of parameter
    IF MIDS(COMMANDS, IX, 1) = CHR$(34) THEN ' Parameter is enclosed in double quotes
      SearchCharX = CHR$(34)
      ParmPos%(CountX, XL) = ParmPos%(CountX, XL) + 1 ' we don't want the "
    END IF
    IX = IX + 1 ' Increment parameter counter
  WEND

  ' Now we need to store the parameters in our SHARED string array

  FOR IX = 1 TO ParmcountX
    PARAMETERS(IX) = MIDS(COMMANDS, ParmPos%(IX, XL), ParmPos%(IX, XR) - ParmPos%(IX, XL))
  NEXT IX

  IF CountX <= FNParamCountX THEN ' Check to make sure parameter exists it is read from the global SHARED array
    ParmPos% = Parameters$(CountX)
  ELSE ' The function has already been called once
    ParmPos% = NIL
  END IF

  EXIT DEF ' FNParamCountX

ELSE ' The function has already been called once
  EXIT DEF ' FNParamCountX
```

**LISTING 2: PARMDEMO.BAS**

```
Author: Duke Kamstra

Mod. Date: 5/8/88

This program demonstrates the subroutines FNParamCount() and FNParamStr().

Compilation instructions:
1. In the Turbo Basic Integrated Development Environment:
   a. Load the program into the Turbo Basic editor.
   b. In the Options|Parameter line menu define a command line parameter list. For example:
      this is a "test parameter list"
   c. Press ALT-R to run the program in memory.
2. From an .EXE file:
   a. Load the program into the Turbo Basic editor.
   b. In the Options|Compile to menu select EXE file.
   c. Press ALT-C to compile PARAM.BAS to PARAM.EXE.
   d. Press ALT-F 0 to leave the Turbo Basic Integrated Development Environment.
3. At the DOS command line type:
   PARAM this is a "test parameter list"

$INCLUDE "PARAM" ' Include the command line parameter routines

CLS
PRINT FNParamCount$; "parameters were passed to PARMDEMO"
PRINT "The parameters are:
FOR IX = 1 TO FNParamCount$ PRINT "Parameter# "; IX, FNParamStr$(IX) NEXT IX
```

Mod. Date: 5/8/88

This program demonstrates the subroutines FNParamCount() and FNParamStr().

Compilation instructions:
1. In the Turbo Basic Integrated Development Environment:
   a. Load the program into the Turbo Basic editor.
   b. In the Options|Parameter line menu define a command line parameter list. For example:
      this is a "test parameter list"
   c. Press ALT-R to run the program in memory.
2. From an .EXE file:
   a. Load the program into the Turbo Basic editor.
   b. In the Options|Compile to menu select EXE file.
   c. Press ALT-C to compile PARAM.BAS to PARAM.EXE.
   d. Press ALT-F 0 to leave the Turbo Basic Integrated Development Environment.
3. At the DOS command line type:
   PARAM this is a "test parameter list"

$INCLUDE "PARAM" ' Include the command line parameter routines

CLS
PRINT FNParamCount$; "parameters were passed to PARMDEMO"
PRINT "The parameters are:
FOR IX = 1 TO FNParamCount$ PRINT "Parameter# "; IX, FNParamStr$(IX) NEXT IX
```

September/October 1988 TURBO TECHNIX 111
The PARMDEMO program immediately summarizes the parameters, as shown in the following sample output:

4 parameters were passed to PARMDEMO
The parameters are:
Parameter# 1 fee
Parameter# 2 fie
Parameter# 3 foo
Parameter# 4 fum

The PARMDEMO program calls FNParamCount%() to determine how many parameters were passed to PARDEMO, and then calls FNParamStr$() to read each of the individual parameters.

Note that your program may call either of the functions in either order, and as often as necessary. The call to FNParamCount%() in FNParamStr$() assures that if FNParamStr$() is called first, the command line parameters are still processed and stored in Parameters$. Either way, the parameters will be there when you need them.

Duke Kamstra is a quality assurance coordinator for Borland International, Inc.

Listings may be downloaded from Library 1 of CompuServe forum BPROGA, as TBCOML.ARC.
GETTING IN THE LOOP

LOOP is the key to repeating blocks of statements without using GOTO.

Tom Wrona

One of the key facets of structured programming is the art of making loops. While structured programming is permitted by the syntax of BASIC, and encouraged by certain Turbo Basic features, BASIC (unlike Pascal) does not require structured programming. Thus, if your first programming language is BASIC, you might not fully appreciate the significance of loops in structured programming.

When using interpreted BASIC, it's all too easy to produce what professional programmers call "spaghetti code": meandering, unstructured code that's hard to understand and hard to debug. The two prime spaghetti code influences in BASIC are the language's reliance upon line numbers, and its primitive looping abilities. Turbo Basic, however, corrects both problems. First of all, line numbers aren't required in Turbo Basic; in fact, you should never use them. Period. Second, Turbo Basic's looping facilities are much more sophisticated than interpreted BASIC's good old FOR..NEXT, as I'll explain in this article.

BEYOND FOR..NEXT

FOR..NEXT only permits a block of statements to be repeated some number of times. Listing 1 is a minimal program that illustrates how FOR..NEXT works, and shows the loop's use of the STEP keyword to increment the loop counter by a number other than one. Run this listing and watch what it does. While FOR..NEXT is useful, more powerful looping constructs are needed for writing commercial-quality software.

When you first start programming, it's a little difficult to see what your modest efforts have in common with commercial programs such as WordStar or Lotus 1-2-3. You begin by learning that a program is a list of instructions that are executed sequentially by the computer; your own programs contain sequential lists of Turbo Basic commands. However, when you start up an advanced application such as MicroCalc (the spreadsheet program that is included with Turbo Basic), you notice that its commands don't seem to be very sequential—the program is just there, on the screen, all at once.

All programs, MicroCalc included, are thoroughly sequential—this becomes apparent when you look closely at the nature of the sequence. Listing 2 shows a short program that is very similar to programs written by most BASIC programmers while they're getting their feet wet. The program begins, executes some statements, and stops, producing the output shown in Figure 1. The text lines shown in Figure 1 appear on the screen, one after the other, as the program executes each program line.

Figure 2 is a screen "snapshot" of the MicroCalc screen that appears when MicroCalc executes. Compare Figure 1 with Figure 2. Rather than appearing to be the result of a sequence of instructions, MicroCalc seems to be just "there" all at once, awaiting input.

The operative word here is "awaiting." By the time MicroCalc has drawn the spreadsheet grid and begins waiting for our input (in this case, a number, a letter, a cursor movement key, or a slash command), the program has already done a lot of preparatory work and is in the middle of a loop. Examine Listing 3, which shows the source code for MicroCalc's main program. We can pinpoint the exact location in the code when the program seemingly pops up on the screen all at once. (I've added numbers to the printed listing for reference purposes; these numbers are continued on page 114)

Let's play with numbers!
Pick a number and I'll tell you facts about it.
What's your number? 42
The square root of your number is 6.48074069840786.
Want to know something else (Y/N)? Y
A circle with a diameter of 42 would have a circumference of 131.88.
That's all! Thanks for playing!

Figure 1. Programs written by newcomers often present a simple, linear question-and-answer session such as the one shown here. A repeating command loop offers a great deal more sophistication with respect to how a program communicates with the user.
not present in the actual MC.BAS file.) Line 67 is the comment line
shown below:

• set up a LOOP UNTIL '/Q' command
is chosen

Immediately after this line, a
DO..LOOP begins that deter­
mines which key has been
pressed by the user. (I'll discuss
DO..LOOPs in more detail
shortly.) This DO..LOOP, which
is the main body of the program,
shunts the flow of the program to
the subroutine that is invoked by
the keypress. Everything above
line 67 in the program is prepara­
tion for the DO..LOOP. Lines 58-
65 check if a filename (of a pre­
viously saved spreadsheet) has
been typed in after the "MC" on
the command line; if the filename
was entered, then the subroutine
Load is CALLed to load that
sheet; otherwise (ELSE), a blank
spreadsheet is drawn by CALLing
the Grid subroutine. Subroutines
such as Grid are contained in var­
ious include files, which are part
of MicroCalc.

PSEUDO-CODE

One way to understand a pro­
gramming problem is to think in
terms of "pseudo-code." Pseudo-
code is an English-language
"sketch" of a program that you
create before you get down to the
job of coding in your actual pro­
gramming language. (For more on
pseudo-code, see "Binary Engi­
neering," TURBO TECHNIX, No­
vember/December, 1987.)

Pseudo-code is useful not only
for creating a program, but also
for analyzing an existing program.
One good way to increase your
understanding of program struc­
ture is to reverse-engineer a pro­
gram's source code back to
pseudo-code. For example, the
pseudo-code equivalent of the
code from the beginning of the
program to the start of the main
loop at line 68 is shown below:

Initialize all variables
and arrays (CALL Init)
IF a filename was typed in...
CALL the spreadsheet file
loading subroutine.
No filename? (ELSE)
CALL Grid to draw
a blank spreadsheet.
That's all, go on. (END IF)

The main loop extends from
the DO keyword in line 68 to
LOOP UNTIL CalcExit% in line
93. This main loop, called a
DO..LOOP, is one kind of control
structure.

CONTROL STRUCTURES

Control structures such as
DO..LOOPs are a language's
method for determining which
instructions get executed, based
upon the value of a variable or
the occurrence of an event. Not
all control structures are loops.
For example, an IF..THEN con­
ditional test is used to determine if a
spreadsheet file should be loaded.

The logic of such a test is very
English-like: IF the filename ex­
ists, THEN CALL Load to load it,
ELSE draw a blank spreadsheet.
Although multiple tests can be
performed using IF..THEN, each
test occurs only once. Thus,
IF..THEN is a one-way action, not
a loop.

The DO..LOOP statement. While
both IF..THEN statements and
DO..LOOP statements always in­
volve testing, DO..LOOPs are
used more as processing tools, in­
stead of testing tools. Again, the
name DO..LOOP reflects the
function of these keywords in an
English-like fashion: The program
will DO some process UNTIL or
WHILE some expression is true
or false.

What MicroCalc's main
DO..LOOP does (and, therefore,
what the program spends most of
its time doing) is nothing more
than waiting for keyboard input.
When such input appears, the
program processes the keyboard
input to see what should be done
next. MicroCalc keeps on process­
ing keyboard input until a "/Q" is
entered to terminate the pro­
gram.

Calling ReadKBD. The main
loop's first action (at line 69) is to
CALL a little subroutine called
ReadKBD. ReadKBD, which is
reproduced in Listing 4, tells its
caller which key has been pressed.
What is ReadKBD? Another loop,
of course. ReadKBD's loop is the
Time is money. And coding a DBMS application like Accounting or Order Entry takes a lot of both. Simply because hacking out mountains of code with your RDBMS or 4GL is too slow. Not to mention the time to rewrite if you make a mistake or change the design.

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

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

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

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

OR PAY NOW AT NO RISK
Pay when you order and we'll waive the $19.95 restocking fee so you have absolutely no risk.

SPECIAL OFFER $695
$199
Magic LAN multi-user — $399
Magic RUN — call for price

Order Now Call:
800-345-MAGIC
In CA 714-250-1718

ATTENTION BTRIEVE® USERS
Now you can quickly enhance your BTRIEVE® based applications beyond the capabilities of XTREIVE® and RTRIEVE®. Use Magic PC as a turn-key BTRIEVE® Application Generator to customize your applications without even changing your existing code.
Paradox* is both the first family in DBMS and the top-rated relational database. Software Digest has ranked Paradox #1 for the past 2 years; PC Magazine gave Paradox its "Editor's Choice" award and InfoWorld named it 1987 "Product of the Year" for Database Systems.

Now there's OS/2

Paradox OS/2 is the newest member of the Paradox family—more are on the way and they're all 100% compatible with each other.

Paradox OS/2 allows you to take advantage of powerful OS/2 features such as addressing up to 16 megabytes of memory and running concurrent sessions. And Paradox OS/2 even lets you start new OS/2 sessions from within Paradox.

Harness the power of 386

Paradox 386 is powerful new DOS software for your powerful new hardware and it's designed exclusively for 80386-based systems. It also lets you ignore the old 640K limits and races through your data 32 bits at a time instead of just 16. It's a perfect solution for anyone faced with very large tables (tens of thousands of records or more) and/or large applications.

As proof of Borland's commitment to delivering compatibility across diverse hardware and software environments, Paradox 386 and Paradox 2.0 can share the same databases and applications on a network.

Giovanni Perrone, PC Week

"Paradox ... it's the PC database-management system equivalent to turbo-charging an M-series BMW."

Giovanni Perrone, PC WEEK

The Paradox Network really works

Network users, you need Paradox's multiuser capabilities. The network runs smoothly, intelligently and so transparently that multiusers can access the same data at the same time—without getting in each other's way. (But safeguards prevent multiple users from altering the same data at the same time.) And with screen refresh you get real-time data updates on your screen.

[Paradox is] a true network application, a program that can actually take advantage of a network to provide more features and functions, things that can't be done with a standalone PC.

Aaron Brenner, LAN Magazine

[Paradox elegantly handles all the chores of a multiuser database system with little or no effort by network users.]

Mark Cook and Steve King
Data Based Advisor

*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.

(1) Borland products are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks of their respective holders. Copyright ©1988 Borland International, Inc.
relational database, has now OS/2 versions!

“Query-by-Example” gives you the right answer, right now

Our “Query-by-Example” (QBE) technique is just one illustration of the technological leadership offered by Paradox for the past 2 years.

QBE is fast and simple to use. Simply call up a form and check off the information you want.

Without having to write a line of code, you can, for example, get answers to queries like: Find all the items we sold for more than $1000 and tell me who ordered them.

An artificial intelligence technique called “heuristic query optimization” gives Paradox’s QBE the ability to figure out not just the right answer, but also the fastest way to get the right answer.

QBE makes high-speed links between one piece of data and another and quickly sees the relationships your question calls for.

PAL: A powerful programming language

PAL, the Paradox Application Language, is a full-featured, high-level, structured database programming language that lets you write sophisticated Paradox programs (scripts) and applications. It includes such powerful features as looping constructs, arrays, branching, procedures, and a full set of functions.

Most people we meet who give Paradox a try, end up switching to it . . .

Mark Cook and Steve King
Data Based Advisor

There’s a Paradox 2.0 version for you

Whether you’re a DOS or OS/2 user, there’s a Paradox version for you.

60-Day Money-back Guarantee*
For a brochure or the dealer nearest you, call (800) 543-7543

Data Based Advisor
GETTING IN THE LOOP
continued from page 114

In the main loop, the test condition is LOOP UNTIL CalcExit%. Although CalcExit% is an integer (this is indicated by the presence of the percent sign), it’s used here as a Boolean variable that can be interpreted as either True or False. In the Command subroutine, which interprets slash commands, there is a SELECT CASE statement that assigns CalcExit% with a value of True if “/Q” has been pressed. (Just as DO..LOOP is FOR..NEXT’s big brother, SELECT CASE is IF..THEN’s big brother. For more on SELECT CASE, see “SELECT CASE: Choosing One From the Many,” TURBO TECHNIX, March/April, 1988.)

Where to test? In both ReadKBD and the main loop, the condition is tested at the bottom of the loop. However, testing can be performed at the top of the loop, at the top and the bottom of the loop, or at neither. If testing is not done at either the top or the bottom of the loop, the loop is then endless and repeats forever, unless an exit is performed somewhere in the middle of the loop via a GOTO statement (bad practice), or else via the EXIT LOOP statement (infinitely better) as shown below:

```
DO
  GetSomeinput(InputPresent%) IF NOT InputPresent% THEN EXIT LOOP
  ProcessInput
LOOP
```

Testing at the top of the loop is simple to perform, as demonstrated by the following code:

```
QuitProcess% = 0
DO UNTIL QuitProcess%
  DoSomeWork
  AreWeDoneYet(QuitProcess%) LOOP
```

The logical opposite of a DO..UNTIL loop is a DO..WHILE loop. In a DO..WHILE loop, the processing operation is repeated WHILE the condition is true. The operation of a DO..WHILE loop is shown below:

```
GetSomeInput(InputPresent%) DO
  ProcessInput
  GetSomeinput(InputPresent%) WHILE InputPresent%
Notice that the presence of the keyword WHILE at the bottom of the loop makes the LOOP keyword unnecessary.
```

Another syntax for DO..WHILE is called WHILE..WEND; this syntax is borrowed from older versions of BASIC. WHILE..WEND tests at the top of a loop, as demonstrated in the following code:

```
GetSomeInput(InputPresent%) WHILE InputPresent%
  ProcessInput
GetMoreInput(InputPresent%) WEND
WHILE..WEND is completely equivalent to DO..WHILE; whether you use it or not is strictly a matter of taste.
```

ONWARD
To learn how to write commercial-quality software, you have to understand how it differs from the toy programs that we all write when starting out. With Turbo Basic, an important first step is to understand structured programming and control structures such as DO..LOOP statements. Where do you go from here? Try studying the source code for MicroCalc. Print it out and follow the program flow into the various include files and their procedures and functions. Rewrite MC.BAS as pseudo-code to get some more insights into how large programs are put together. Identify useful subroutines like ReadKBD that you can use and reuse in your own projects. Obtain public domain programs that include source code, and study the code with a critical eye. Is the code sloppy or tight? Is it spaghetti code or well-commented structured code?

The more source code that you study, and the more that you write yourself, the better you’ll become at programming in Turbo Basic. And someday, perhaps, the commercial program that pops up on my screen will be yours.

---

Tom Wrona is a writer, consultant, and the author of How to Run a Hard Disk PC, published in March by Scott, Foresman & Company. Reach Tom via CompuServe (76137,3363) or MCI Mail.

Listings may be downloaded from Library 1 of CompuServe forum BPROGA, as LOOPS.ARC.

118 TURBO TECHNIX September/October 1988
LISTING 1: FORTEST.BAS
'A simple program demonstrating FOR...NEXT with the STEP modifier:
FOR i = 2 to 8 STEP 2
PRINT i
NEXT i
PRINT "who do we appreciate?"

LISTING 2: NUMBERS.BAS
'Toy program by Tom Wrona
CLS
PRINT "let's play with numbers!"
PRINT "pick a number and I'll tell you facts about it."
INPUT "what is your number?" ; number
PRINT "the square root of your number is " SQR(number) ."
INPUT "want to know something else (Y/N)" ; answers
IF UCASES(answers) = "N" GOTO DONE
PRINT "a circle with a diameter of number" would have
PRINT "a circumference of " 3.14 * number ."
DONE:
PRINT "that's all! thanks for playing!"

LISTING 3: MC.BAS
MC.BAS
VERSION 1.0
Turbo Basic
(C) Copyright 1987 by Borland International
System Requirements:
DOS Version 2.0 or later
- 320K
This program is a simple spreadsheet program that is provided
as an example of a simple application that can be done in
Turbo Basic. You are encouraged to study this program and
make any enhancements and modifications that you might want.

LISTING 4: READKBD.BAS
'ReadKBD, a subroutine contained in MC1.INC
SUB ReadKBD(RetChar$)
' this function reads a keystroke from the keyboard
' and returns 1-2 character string.
DO
RetChar$ = INKEY$
LOOP UNTIL RetChar$<>"
END SUB
TURBO ASSEMBLER: CIVILIZING MACHINE LANGUAGE

If you’ve never tackled the 86-family’s own language, this may be the Ideal time to start.

Tom Swan

You’ve probably heard the famous truths about assembly language—"Programming in assembly language is more difficult than teaching buffaloes to pirouette;" "An assembly language program can trash memory faster than Oliver North can shred a sensitive document;" and, "Only 13-year-old software prodigies can understand assembly language mnemonics!"

These are bad raps. Assembly language is not a great deal more difficult to learn and to use than any other computer language. This is especially true now with the availability of new features such as Ideal mode, local labels, and improved command-line options in Turbo Assembler—Borland International’s newest Turbo language and the partner of Turbo Debugger. If you’re eager to learn assembly language, you couldn’t have picked a better time to begin.

Turbo Assembler is not just for beginners, though. If you’re an experienced assembly language programmer, you’ll be happy to know that Turbo Assembler is fully compatible with the Microsoft Macro Assembler (MASM). Turbo Assembler recognizes all MASM macros, conditional assembly and other directives, plus simplified segment models. If you have existing assembly language programs to maintain, Turbo Assembler can almost certainly assemble them.

Of course, Turbo Assembler carries the famous Borland mark of the gazelle—it assembles a 2000-line test file in less than four seconds on a 16-mHz 80386 system (about twice as fast as MASM 5.1). And, like MASM, Turbo Assembler supports all typical PC processors (8088, 8086, 80186, 80286, 80386) and math coprocessors (8087, 80287, 80387).

Other features make Turbo Assembler friendly to use. For example, the following command assembles all of the .ASM files in a directory:

TASM *.ASM

Turbo Assembler’s most intriguing new feature, called Ideal mode, is a logical refinement to standard MASM syntax. If you’re new to assembly language programming, Ideal mode will help you get up to speed without getting bogged down in minor syntactical quirks that plague other assemblers (especially MASM). If you’re an old pro (or a young pro!), you’ll appreciate Ideal mode’s many improvements to MASM syntax, plus the ability to switch back to full MASM compatibility at any time and assemble existing modules written in the standard syntax. I’ll cover Ideal mode in more detail shortly.

First, however, a note to beginners: If assembly language is still gobbledygook to you, skim over the specific examples in this introduction. I’ve tried to provide general information for those of you with little or no assembly language experience, but there isn’t enough room here for a complete tutorial. For help with learning assembly language, refer to the Turbo Assembler manual, other TURBO TECHNIX articles, and forthcoming books on Turbo Assembler. [Editor’s note: Including one by the estimable Mr. Swan.]

USING TURBO ASSEMBLER

Unlike other Turbo languages, Turbo Assembler is not an integrated development environment with a text editor and pull-down menus. Instead, Turbo Assembler
operates from the DOS command line, similar to the way MASM runs. Turbo Assembler requires the use of a separate editor for typing programs, and most people probably will use the editor in Turbo C, Turbo Pascal, Turbo Basic, or Turbo Prolog. Other good choices are the MicroStar editor in the Turbo Pascal Editor Toolbox, or the notepads in SideKick and SideKick Plus. You can also use an editor such as Brief (my favorite), or any word processor that edits plain ASCII text.

Many people will use Turbo Assembler with one or more high-level Turbo languages to convert selected BASIC subroutines, Pascal procedures, or C functions to assembly language in order to gain the extra speed that only pure machine code can give. In fact, experts estimate that most programs spend about 90 percent of their time executing only 10 percent of their code. In theory, therefore, the conversion of the critical 10 percent of any program to assembly language potentially increases program speed by almost as much as could be done by rewriting the entire program.

To help you mix and match Turbo Assembler with other Borland languages, individual chapters in the Turbo Assembler manual explain how to interface assembly language to Turbo Pascal, Turbo C, Turbo Basic, and Turbo Prolog. Turbo C can even call Turbo Assembler directly to assemble inline assembly language statements embedded in Turbo C source text.

Of course, standalone programs can also be assembled with Turbo Assembler. Programs can be located in one file, or else divided into modules, assembled separately, and then linked with other modules to create the final code file on disk. Since Turbo Assembler is fully compatible with MASM, you can take advantage of the thousands of lines of published assembly language source code available on bulletin boards, in magazines and books, and elsewhere.

**QUIRKS MODE**

Through a special command, Turbo Assembler can even reproduce known MASM bugs and quirks. To use this command, type QUIRKS into your source text to throw Turbo Assembler into quirks mode for near 100-percent MASM source code compatibility, warts, bugs, and all.

The only MASM programs that Turbo Assembler cannot digest are a few rare (and poorly written) examples that rely on MASM's two-pass nature. Turbo Assembler is a one-pass assembler—it reads a program text file a single time in order to generate an object file that contains the assembled program code. MASM reads a program text file twice—once to identify labels, and once again to generate the object code. With respect to speed, one pass is obviously better than two. Besides, you're better off not using—and never writing—finicky two-pass-dependent programs in the first place.

**IDEAL MODE—ENTER STAGE RIGHT**

Besides MASM compatibility (with or without quirks), Turbo Assembler introduces Ideal mode—this departure from MASM syntax is a subject that's bound to be controversial among bit-twiddlers everywhere. Ideal mode is to assembly language what Hamlet and other Shakespearean plays were (and are) to English—the sensible and inventive force that civilizes an existing language. Shakespeare didn't create English. He improved and expanded the language in ways that have lasted until today and that will no doubt endure for as long as English itself. Similarly, Turbo Assembler's Ideal mode improves MASM syntax in ways that are likely to have long-lasting effects on PC assembly language programming. Ideal mode is not just a new assembly language syntax—Ideal mode has refined, reformed, and civilized MASM.

**New and improved syntax.** Ideal mode improves MASM syntax in two fundamental areas: consistency

__continued on page 122__
An excellent example of how Ideal mode’s stronger Turbo Assembler never lets addresses be confused on program speed. checking helps prevent bugs by restricting assignment. Type-checking. In Ideal mode, most keywords begin an instruction, rather than appearing in the apparently random fashion that they do in MASM. Table 1 compares several Ideal mode keywords to their MASM equivalents. Notice that **ENDP** and **ENDS** are optionally followed by the name of the procedure or segment that was previously used in a matching **PROC** or **SEGMENT** directive. (In MASM, the name precedes the keyword and, therefore, must be used in both places.)

<table>
<thead>
<tr>
<th>MASM Mode</th>
<th>Ideal Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>name ENDP</td>
<td>ENDP [name]</td>
</tr>
<tr>
<td>name ENDS</td>
<td>ENDS [name]</td>
</tr>
<tr>
<td>name GROUP segs</td>
<td>GROUP name segs</td>
</tr>
<tr>
<td>name LABEL type</td>
<td>LABEL name type</td>
</tr>
<tr>
<td>name MACRO args</td>
<td>MACRO name args</td>
</tr>
<tr>
<td>name PROC type</td>
<td>PROC name type</td>
</tr>
<tr>
<td>name RECORD args</td>
<td>RECORD name args</td>
</tr>
<tr>
<td>name SEGMENT args</td>
<td>SEGMENT name args</td>
</tr>
<tr>
<td>name STRUC</td>
<td>STRUC name</td>
</tr>
<tr>
<td>name UNION</td>
<td>UNION name</td>
</tr>
</tbody>
</table>

**Table 1. Ideal mode versus MASM keywords. Bracketed items are optional.**

**Type-checking.** Ideal mode’s stronger type-checking rules help you write programs that have fewer bugs and make more sense both to you and to the assembler. When assembling in Ideal mode, for example, Turbo Assembler never lets addresses be confused with values stored in memory (this is a prime source of bugs even with experienced programmers). Ideal mode also eliminates MASM’s annoying tendency to calculate some offsets relative to individual segments that are collected by the **GROUP** command. In Ideal mode, items in grouped segments are always accessed relative to the **group**, not to the segment in which the items reside. Pascal and C programmers know that strong type-checking helps prevent bugs by restricting assignments and other operations to variables of compatible types. With Turbo Assembler’s Ideal mode, assembly language programmers can now enjoy similar benefits with no loss of capability and no penalty on program speed.

**IDEAL MODE AND BRACKETS**

An excellent example of how Ideal mode’s stronger type-checking rules help prevent bugs is the way that square brackets (e.g., [ ]) are required in order to obtain the contents of a memory location. For example, **[MyVar]** with brackets refers to the contents stored in memory at the location marked by the label, **MyVar.** This rule has important consequences in constructions such as the following:

```
Count dw 0
mov ax, [Count]
```

Here, **Count** is a label (a pointer) that locates a two-byte word in memory, which is initialized to zero. (The **dw** stands for “define word.”) The second line moves the contents of **Count** into register **ax**. Because of the brackets, there’s no question that **[Count]** refers to the contents of the memory location and not to the value of the **Count** label itself. Contrast this with the following:

```
mov ax, Count
```

MASM allows this ambiguous construction. (So does Turbo Assembler in MASM mode, of course.) The instruction seems to be loading **Count** into **ax.** But that’s silly. **Count** is a label, a 32-bit address composed of 16-bit segment and offset values—and 32-bit labels cannot be loaded into 16-bit registers. Only 16-bit values can be loaded into 16-bit registers, and only 8-bit values can be loaded into 8-bit registers. Since it knows that this instruction is senseless, MASM assumes that you must be trying to load **ax** with the contents stored at the address of **Count** and, therefore, happily assembles the program as though you had written **mov ax, [Count]** with brackets!

Turbo Assembler in Ideal mode properly warns that you probably forgot the brackets around **Count.** Ideal mode can do this because it checks that the type of the destination (**ax**) is compatible with the source (**Count**).

When you do want to load the value of a label into a register, you must specify which type-compatible part of the label is to be used. To assign the 16-bit offset value of the label **Count** to **ax**, relative to the segment that declares the label, you must write:

```
mov ax, OFFSET Count
```

Both Turbo Assembler (in all modes) and MASM correctly assemble this instruction. When the program runs, the 16-bit offset address of **Count** is moved (copied) into **ax.** The danger here—and the reason that Ideal mode rejects the bracketless construction—is that you might easily forget to type the **OFFSET** keyword when referring to the label’s address. If you do this in MASM, the assembled code mistakenly refers to the contents stored at this address, and you won’t know something is wrong until the program begins to misbehave. Turbo Assembler’s Ideal mode spots this and other subtle mistakes during assembly, thus helping you to write programs that run as you intend. Unlike MASM, Ideal mode never tries to decide what you “really” mean!

**OTHER IDEAL-MODE FEATURES**

Another important Ideal-mode feature is a new job description for a useful assembly language employee—the lonely dot. In MASM, dots have many jobs. Dots begin some directives (**.LIST** and **.RADIX**), but not others (**INCLUDE** and **COMM**). Dots separate structures, as in **CUSTOMER.ADDRESS**. Dots are used in floating point numbers (5.2) and in some commands (**.386**) that look like numbers, but aren’t. It’s enough to drive you batty, if not dotty.
The Ideal dot. In Turbo Assembler's Ideal mode, dots never begin keywords. Period. Dots always separate identifiers in structures and unions, and mark the decimal places in floating point numbers.

Since no Ideal-mode keyword begins with a dot, some MASM directives are necessarily different, as shown in Tables 2 and 3. For instance, the MASM command .286 (which enables 80286-processor instructions) is P286 in Ideal mode. Ideal mode commands that begin with percent signs, such as %LIST and %NOCREF, affect program listings. These changes help clarify programs and make them easier to read. In Ideal mode, you always know a command when you see one. Even better, you don’t have to hunt through the manual to find out whether a command requires a leading dot.

<table>
<thead>
<tr>
<th>MASM Mode</th>
<th>Ideal Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>.CREF</td>
<td>%CREF</td>
</tr>
<tr>
<td>.CALL</td>
<td>%MACS</td>
</tr>
<tr>
<td>.LFCOND</td>
<td>%CONDNS</td>
</tr>
<tr>
<td>.LIST</td>
<td>%LIST</td>
</tr>
<tr>
<td>.SFCOND</td>
<td>%NOCONDNS</td>
</tr>
<tr>
<td>.XALL</td>
<td>%NOMACS</td>
</tr>
<tr>
<td>.XCREF</td>
<td>%NOCREF</td>
</tr>
<tr>
<td>.XLIST</td>
<td>%NOLIST</td>
</tr>
</tbody>
</table>

Table 2. Ideal mode versus MASM listing controls.

<table>
<thead>
<tr>
<th>MASM Mode</th>
<th>Ideal Mode</th>
<th>MASM Mode</th>
<th>Ideal Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>.186</td>
<td>P186</td>
<td>.ERR2</td>
<td>ERRIF2</td>
</tr>
<tr>
<td>.286</td>
<td>P286N</td>
<td>.ERRB</td>
<td>ERRIFB</td>
</tr>
<tr>
<td>.286C</td>
<td>P286N</td>
<td>.ERRDEF</td>
<td>ERRIFDEF</td>
</tr>
<tr>
<td>.286P</td>
<td>P286</td>
<td>.ERRDIF</td>
<td>ERRIFDIF</td>
</tr>
<tr>
<td>.287</td>
<td>P287</td>
<td>.ERRDIFI</td>
<td>ERRIFDIFI</td>
</tr>
<tr>
<td>.386</td>
<td>P386N</td>
<td>.ERE</td>
<td>ERRIFE</td>
</tr>
<tr>
<td>.386C</td>
<td>P386N</td>
<td>.ERRIDN</td>
<td>ERRIFIDN</td>
</tr>
<tr>
<td>.386P</td>
<td>P386</td>
<td>.ERRIDNI</td>
<td>ERRIFIDNI</td>
</tr>
<tr>
<td>.387</td>
<td>P387</td>
<td>.ERRNB</td>
<td>ERRIFNB</td>
</tr>
<tr>
<td>.8086</td>
<td>P8086</td>
<td>.ERRNDEF</td>
<td>ERRIFNDEF</td>
</tr>
<tr>
<td>.8087</td>
<td>P8087</td>
<td>.ERRNZ</td>
<td>ERRIF</td>
</tr>
<tr>
<td>.FARDATA</td>
<td>FARDATA</td>
<td>.CODE</td>
<td>CODESEG</td>
</tr>
<tr>
<td>.FARDATA?</td>
<td>UFARDATA</td>
<td>.CONST</td>
<td>CONST</td>
</tr>
<tr>
<td>.MODEL</td>
<td>MODEL</td>
<td>.DATA</td>
<td>DATABASE</td>
</tr>
<tr>
<td>.RADIX</td>
<td>RADIX</td>
<td>.DATA?</td>
<td>DATABASE?</td>
</tr>
<tr>
<td>.ERR</td>
<td>ERR</td>
<td>.STACK</td>
<td>STACK</td>
</tr>
<tr>
<td>.ERR1</td>
<td>ERRIF1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Ideal mode versus MASM dot commands.

Nesting and field names. Ideal mode structures and unions can also be nested (this is an illegal operation in MASM). In addition, field names that are inside one structure can be the same as the field names that are inside another structure. The ability for two or more structures to have the same field names is especially helpful during the manipulation of linked lists with many structures, where all link fields in various records are named something like NextRec and PrevRec. MASM requires unique names to be invented for fields in all records, even when the fields have identical purposes.

PROGRAMMING IN IDEAL MODE

Other major differences between MASM and Ideal modes are best described by example. Listing 1 is an Ideal mode program that displays a disk directory. This program incorporates a single directory "search engine" that is similar to the search engines for Turbo Pascal and Turbo C presented elsewhere in this issue.

To create and run DR.EXE, use the following commands:

TASM DR
TLINK DR
DR

After an initial comment line in the listing, the keyword IDEAL initiates Ideal mode. Although not shown here, the keyword MASM can be used to switch back to MASM compatibility. This lets you alternate between the two modes in the same listing as often as you like.

Because the %TITLE directive begins with a percent sign, you know that this command affects listing output. Notice that a comment line (the text that follows the semicolon) is allowed because the title string in Ideal mode must be enclosed in quotes. To create a listing file, assemble the program with the following command:

TASM /L DR

To generate a cross-referenced symbol table at the end of the listing, use this command instead:

TASM /C/L DR

Table 4 lists other command-line options that can be used during assembly.

continued on page 124
TURBO ASSEMBLER

continued from page 123

be used in both Ideal and MASM modes.
choice for most standalone assembly language pro-
ciate constant values with the identifiers: Attribute,
FileName, Cr, and U. During assembly in Ideal
sions are not evaluated until the program uses the
equated identifier. At that time, the associated text re-
places the identifier in a process similar to the oper-
ation of a macro. In the sample listing, the equates
are simple numbers. Suppose, however, that you
have the following equates:

C = 4;
Value EQU C+10;
C = 9;

In MASM mode, Value equals 14 because Turbo As-
sembler evaluates the expression C+10 when reading
the EQU declaration. In Ideal mode, Turbo As-
sembler evaluates C+10 at the place where the Value
identifier later appears in a program statement. The
difference is important. Because the second equate
redefines C to 9. Value in Ideal mode equals 19, not
14 as it would in MASM mode. (Here, an equals sign
is the same as EQU, but allows the value associated
with an identifier to be changed.) In Ideal mode, you
can be certain that C+10 uses the value of C because
C exists at the place in the program where the
equated identifier appears.

In Listing 1, DATASEG defines the program’s data
segment, which is the memory storage area for vari-
ables. Two of these variables are strings. FileSpec,
which is an ASCIIZ string that ends with a zero byte,
holds the directory search wildcard (identical to wild-
card expressions such as * .PAS or TEST.* in DOS
DIR commands). The program uses CrLf (a peculiar,
although common, kind of DOS string that ends with
a dollar sign) to display blank lines. The third vari-
able, DTA, reserves 128 bytes for the DOS directory
search functions.

The program’s code segment begins at the key-
word CODESEG. The comments to the right of each
line describe the assembly language instructions. No-
tice how OFFSET keywords specify label addresses.

The DATASEG and CODESEG keywords demon-
strate Turbo Assembler’s simplified memory seg-
ments. (Similar keywords are available in MASM
mode.) You can always define segments the hard way
by using SEGMENT directives, as required in early
versions of MASM. Most times, however, you can use
the simplified directives and select an appropriate
memory model from Table 5.

PROCEDURES AND LOCAL LABELS
Assembly language procedures, which are optionally
delimited by the PROC and ENDP directives, resem-
ble BASIC subroutines more than Pascal procedures
or C functions. As Listing 1 shows, the name follows
the PROC directive in Ideal mode; MASM reverses
this order.

Notice the labels @@tlO:, @@t20:, and @@t30:
inside DirSearch. Farther down, two of these same
labels appear again. After a LOCALS directive (not
required in Ideal mode), labels that begin with @@
are local to the portion of the program that is sepa-
rated from the rest of the program by nonlocal labels.

Local labels, which can be used in both Ideal and
MASM modes, have two main purposes. A local label
can define a temporary destination for a jump, such as
the jmp @@tl0 instruction in procedure Dir-
Search. More importantly, a local label can also elimi-
nate the worry that you may have used the same
label in another part of the program. The use of lo-
cal labels avoids the annoying MASM error “Symbol
already defined,” because unique labels no longer
have to be invented for every last destination in your
program.
Local labels are not merely convenient, however. They can also help prevent serious bugs by restricting short jumps to small sections of code. For example, if you misspell or forget to define the @@t10: label in procedure DirSearch, the jmp @@t10 instruction cannot accidentally jump into the middle of the next procedure, which also contains a label @@t10:. The bug is prevented because the nonlocal label ListDir lies between the local label @@t10: and the jmp @@t10 instruction.

AN ASSEMBLY LANGUAGE SEARCH ENGINE
Listing 1 contains a procedure, DirSearch, that searches the current directory for a given file specification (which may contain wildcard characters) and a file attribute byte. DirSearch uses the DOS Find First and Find Next functions (as described in "A Directory Search Engine in Turbo C" on page 74 of this issue). To use DirSearch, extract the procedure DirSearch from the program and include it into your own program. Call DirSearch with dsdx addressing a null-terminated file specification string. If you desire, assign a set of attributes to ex that limits directory entries to those entries that are marked with the Archive, Hidden, or other flags. Otherwise, set cx to zero to ignore file attribute settings.

Assign to bx the code-segment offset of a procedure to be called by DirSearch each time a matching file is found. The corresponding procedure in Listing 1 is ListDir, which simply transfers one filename, a character at a time, to the standard output through a character stream. Or, you could search for two different filename endings and list all *.EXE and *.COM files (a fancy pattern-matching scheme that DOS cannot provide from its command line). The choices are limited only by your imagination.

After all, isn’t that the reason why you’ve decided to learn— or why you’re already using—assembly language? Like no other programming language, assembly language offers the most flexibility for the implementation of your software dreams.

If you’ve been meaning to learn assembly language, or if you’re tired of fighting MASM’s crock of quirks, take a look at Turbo Assembler and try a few examples in Ideal mode. I think you’ll be pleased. Undoubtedly, some MASM fans will hear about Ideal mode and say, “If it ain’t broke, why fix it?” I say, “It’s been broke all along, and the repair truck has finally arrived.”

Tom Swan is the author of Mastering Turbo Pascal 4.0, Second Edition (Howard W. Sams). Barring World War III or, even worse, a coffee bean shortage, Tom’s new book, Mastering Turbo Assembler, will be available early in 1989.

Listings may be downloaded from Library 1 of CompuServe forum BPROGB, as TADIR.ARC.
A reliable tool for parsing strings is needed to split Paradox fields into subfields—MATCH fills the bill.

Bill Cusano

Sooner or later it's going to happen: Your database needs will eventually grow to the point where you need to restructure a table to provide more detail. A 5000-record name and address table that contains city, state, and zip code information in a single field is a perfect example. If you want to restructure the table so that city, state, and zip code each have their own field, you have a serious problem.

The solution is a parsing tool that splits the address field into its three components: city, state, and zip code. In the grammatical sense, parsing means to split a sentence into its grammatical components (i.e., subject, verb, object, and so forth). What's needed here is a variation on that theme—a method of splitting a string along some logical boundary, such as a comma, a space, or some other combination of characters.

THREE ON A MATCH

A PAL function, called MATCH, provides a way to match a string against a pattern. The syntax of MATCH is:

MATCH(String, Pattern, [Vars])

String is the text string to be tested, Pattern is a string template against which the string is matched, and [Vars] is an optional list of variables used in segmenting the string on the first occurrence of a pattern match. To see how this type of parsing can help solve our problem, let's look at a few typical strings that contain city, state, and zip code data:

"Scotts Valley, CA 95066"
"Los Angeles, CA 90066"
"Redmond, WA 98073"
"East Hartford, CT 06108"

The four lines shown above contain similar logical boundaries between the separate data elements that are to be extracted. All of the lines contain at least a comma between the city and state information, and at least one space between the state and zip code information. This pattern is consistent through all of the strings.

Defined concisely, the pattern consists of a variable number of characters for the city, followed by a comma and one or more spaces, followed by two or more characters that represent the state, and ending with one or more spaces followed by the zip code.

In PAL, the double period (..) in a pattern string represents any number of alpha, numeric, or special characters. The double period can be used to build the pattern just described. The first part of the pattern string contains a double period to represent the variable number of characters (including spaces) that comprise the city information. This double period is then followed by a comma and a space (..) to indicate the logical break between the city and the state.

Another double period followed by a space (..) represents the pattern for the state information and its separator from the zip code. Although each state is represented by only two characters, we can't be sure how many spaces will precede the state information, so the double period is used just to be safe.

A final double period represents the zip code and the complete pattern string becomes ".., ..,". A variable name must be present to receive each segment of the information (i.e., each portion represented by a double period); the variable names City, State, and Zip are used in this example. A MATCH function can then be stated for each of the example strings, as shown in Figure 1.

Tidy as they seem, these MATCH invocations won't do the job in all cases. In all except the first example string, in fact, the value of the variable State is set to a single space character because multiple spaces are present between the comma and the state code. In such a case, the middle double period (..) in the match pattern picks up the second blank space after the comma and considers that blank space to

continued on page 128
In Paradox, you create many tables and often quite a bit of confusion. How many times have you asked yourself:

- I have many tables in several directories, how can I keep track of them?
- Are my Field Types, Image Formats and Validity Checks consistent across my tables?
- I renamed my "Staff" table to "Employee", where do I have to make Tablelookup changes?
- How much disk space is used by a table and its entire family?
- What settings have I placed in reports (length, width, setup, etc.)?
- How can I tell when my tables need to be restructured to remove wasted space?
- Which of my tables are encrypted, write protected, or corrupted?
- What rights have I assigned to each field for password protected tables?

The 12 Paralex reports answer all of these questions, and many more. Paralex creates Data, Table and Password dictionaries that gather extensive information from your Paradox tables. The reports may be sent to printer, screen, or disk file. Dictionaries are Paradox tables, so you have total flexibility. Paralex is menu driven, so it's easy to use. We can't imagine Paradox applications without Paralex.

Although Paralex list price is $149.95, you may order for only $99.95 + $5.00 shipping and handling.

You may order by credit card, by calling 800-336-6644. Checks and Purchase orders may be sent directly to:

Zenreich Systems
78 Fifth Avenue, New York, NY 10011 212-691-0170

Paralex requires Paradox 2.0 (or higher) and a 256kb machine. Paralex was reviewed by the Paradox Users Journal and DataBased Advisor. Paralex is written in proceduralized PAL code. Registered Paralex users will receive a disk with several useful procedures that went into the building of Paralex.

---

Enhance the power of Paradox.

With PlayRight. The first ASCII text editor designed especially for Paradox. PlayRight's multi-file editing, extensive block operation, script formatting, custom configuration and spool-printing capabilities bring speed and efficiency to your writing—and debugging. And, because it looks and feels like your built-in PAL editor, it's easy to use. PlayRight snaps right into Paradox, instantly replacing the simple PAL editor. Instantly enhancing the power of Paradox.

"PlayRight is great—I can hardly imagine anyone who writes scripts being without this program."
—Doug Cobb
Paradox Users Journal
October 1987

$129.95 30-day money back guarantee
To place your credit card order:
800-262-8069
For a free brochure, call
704-552-9875

The Burgiss Group
3332 Eastburn Road
Charlotte North Carolina 28210

Paradox is a registered trademark of Ansa Software; Ansa is a Borland International Company.
LISTING 1: CSZ.SC

; SCRIPT: CSZ.SC
; AUTHOR: Bill Cusano (516) 293-6846
; FUNCTION: Demonstrate using MATCH to parse a string

PROC CSZSplit(CSZ)
PRIVATE x4, x5

The IF statement below tests whether the string in CSZ matches a given pattern. The MATCH function performs this test, and if the test passes, variables City and State are assigned the values of their corresponding patterns within the string. The double dot (...) pattern used here accepts any number of characters or numbers in the position.

IF MATCH(CSZ, ".. ...", City, State) THEN

The WHILE command below tests, in each pass through the loop, that the string value of the variable State matches the quoted pattern. Here, if the string contains a leading space, the loop continues. The MATCH function performs a logical test for a match and, upon a match, it fills the variable x4 with all characters to the right of the leading space.

WHILE MATCH(State, " ..", x4)

Each pass through the loop causes the variable State to be reassigned to the value of x4. Thus the string loses its leading blank space.

State = x4
ENDWHILE

The WHILE loop above would only be run if there are leading spaces in the string. If it does not run, we need to assign the value of State to the variable x4, which is tested below.

x4 = State

Here, we're using MATCH again to separate out the State and ZIP data from the remains of the string once City has been removed.

IF MATCH(x4, "...", State, Zip) THEN

This WHILE statement removes leading spaces from Zip:

WHILE MATCH(Zip, " ..", x5)

Zip = x5
ENDWHILE
ENIF
ENDPROC

Below is a test program for procedure CSZSplit:

City = ""
State = ""
Zip = ""

@ 2,4 ? "Enter String: " ; Enter a string to split
ACCEPT "AZ5" TO CSZ
CSZSplit(CSZ) ; Split city, state, and zip from CSZ
@ 6,4 ?? City ; Show the three fields split from string CSZ
@ 7,4 ?? State
@ 8,4 ?? Zip
sleep 3000

PARSING STRINGS
continued from page 126

MATCH
("Scotts Valley, CA 95066"," ..", City, State, Zip)
MATCH
("Los Angeles, CA 90066"," ..", City, State, Zip)
MATCH
("Redmond, WA 98073"," ..", City, State, Zip)
MATCH
("East Hartford, CT 06108"," ..", City, State, Zip)

Figure 1. A first cut at using MATCH to parse city, state, and zip information from a single string. This won't work correctly because there may be multiple spaces between the components, and there's no way to match on multiple spaces.

be the state information. The Zip variable then contains all of the remaining information in the string, which includes both the state and zip code.

To allow for extra spaces, the string must be split into two stages. In the first stage, the string is split into two pieces, rather than three. As a result of the following MATCH statement, the variable City contains the city information, and the variable State contains both the state and the zip code:

MATCH("Redmond, WA 98073"," ..", City, State)

Any leading spaces in the string in State can be trimmed by using another MATCH statement within a simple loop test, as shown in Listing 1.

Once the city data has been parsed out, the same process is repeated in order to split the State string that now contains the state and zip code information. After copying State into a temporary variable named x4, the following invocation of MATCH performs the second split:

MATCH(x4, " ..", State, Zip)

Again, a WHILE loop should be used after the split to remove any leading space characters from the Zip string.

LET'S SPLIT
The CSZSplit procedure in Listing 1 demonstrates the versatility of the MATCH function in parsing the address string to produce separate city, state, and zip code strings. Once you add three new fields to the original table to house these values, you can loop through the expanded table record by record and store the values of the variables City, State, and Zip into the new fields. PAL can do it—problem solved!

Bill Cusano is the owner of Cusano Marketing, a consulting group that offers training and developer support marketed under the name "Sable Solutions."

Listings may be downloaded from Library 1 of CompuServe forum BORAPP, as PMATCH.ARC.
CAPTURING DIRECTORIES WITH SPRINT

Sprint’s gateway to DOS—the call command—lets you consider much of DOS’s power as an extension of Sprint.

Bruce F. Webster

Quite apart from the expected text-processing features, Sprint’s macro language offers considerable low-level access to DOS and to the computer itself. This is exemplified by Sprint’s call command, which lets you execute DOS commands or external programs through DOS’s Exec function. In this article, I’ll present GetDirectory, a Sprint macro that illustrates how a technical documentation specialist might take advantage of these features to customize Sprint for special needs—in this case, to easily input a file directory summary without leaving Sprint. GetDirectory uses call to execute a DOS DIR <filespec> command, redirect DOS’s output to a disk file, and then read the disk file into a document at the cursor position.

DISSECTING A MACRO

The Sprint source code for GetDirectory is given in Listing 1. Because of the terseness of the Sprint macro language—which resembles a cross between Forth and C—GetDirectory isn’t very big. Let’s dissect it, line by line, to see just how it works.

The first line of code establishes the macro’s name (GetDirectory). Macro names are not case-sensitive—GetDirectory and getdirectory are seen as the same by the macro compiler. When beginning a new macro, you must specify the macro name, followed by a colon. This signals the end of the previous macro (if any) and the start of the new macro.

The command set QF "DIR.LST" on the next line copies the string "DIR.LST" into the variable QF. QF is one of Sprint’s 26 predefined string variables, which are named Q0..Q9 and QA..QP. DIR.LST is the name of the temporary disk file that will hold the directory listing.

A file specification for the directory listing is entered via the following line:

message "Enter filespec: " set Q0

This is a standard method for printing a prompting message on the status line and then reading in a response from the keyboard. In this case, the response

continued on page 130
DIRECTORY CAPTURE

continued from page 129

is copied into variable Q0. In general terms, the command message <string> prints <string> on the status line. The command set Q<n> may take an optional string value (as shown earlier). When the string value is present in a set command, the value is copied into the named variable. Since no string value is contained in the set command in GetDirectory, set waits for string input from the keyboard. The string data read from the keyboard is then assigned to Q<n>.

The mark command in the next line is somewhat tricky. The syntax mark [...] saves your place in the edit buffer, executes the commands within the curly braces, and then returns you to your position in the edit buffer. The command to Q0 states that you are now editing the contents of Q0. The command string delete past iswhite deletes any leading white-space (blanks, tabs, and so forth) in Q0. The aim is to remove any leading blanks that you might have entered in the file specification.

The rest of GetDirectory is contained in a single if statement. Sprint's if statement general format is:

if <expression> <command>

Since each separate Sprint macro command can be considered an expression, combinations of commands that act as one expression must be enclosed in parentheses. Likewise, in order for the if statement to execute more than one command, the commands to be executed must be within curly braces.

The expression (0 subchar Q0) tests to see whether or not Q0 contains a file specification. The literal effect of (0 subchar Q0) is to return the character stored in Q0[0]. If Q0 is nonempty, then the returned character is nonzero, which is equivalent to TRUE. If the expression resolves to TRUE, then the rest of the if statement is executed. If Q0 is empty, then the returned character is NULL (ASCII 0, the standard C end-of-string character), which is equivalent to FALSE—this means that the rest of the if statement is then skipped.

The first line of the if statement's block calls message twice. The first invocation of message clears the message bar (by virtue of the leading \n, which prints a new line) and displays the string "Looking for...". The second invocation of message prints the file specification contained in Q0. The two displays comprise a status message that's shown to the user while the directory is being read.

The next line contains the DOS Exec command call. The number (32) that precedes call is a command code that tells call not to switch to the DOS screen; as a result, the DOS operation happens invisibly. All of the strings that follow the call keyword are concatenated together and then passed to DOS through the Exec function. In this case, COMMAND.COM is executed by using the /c directive to pass a command line to COMMAND.COM that consists of three items: "DIR," the file spec in Q0, and the redirection command ">DIR.LST." In effect, the following DOS command is being executed from within Sprint:

```
DIR <filespec> > DIR.LST
```

The output from the DIR listing is redirected to the file DIR.LST.

Once created, DIR.LST must be read into the file that you're editing, by the read QF command. QF, if you remember, contains the string "DIR.LST," which is the name of the file that contains the directory data. The read QF command automatically displays the message "Reading in DIR.LST..." and starts the reading process.

The status command on the next line is very much like the message command, except that status's message is automatically erased as soon as another status or message command is executed. The status command is actually executed before Sprint finishes reading DIR.LST, so that status's message replaces the message displayed by the read command.

The macro's last line uses the call command to execute COMMAND.COM again. This time, COMMAND.COM is passed the command string "ERASE DIR.LST," which deletes the temporary file DIR.LST once DIR.LST is no longer needed.

FETCHING A DIRECTORY

To use GetDirectory, run Sprint, key in the source code, and save the source code as DIR.SPM in your Sprint directory. Close that file and open a document file. Bring up the Utilities menu by pressing Alt-U. Select the Macros submenu and then choose the Load command. You'll be shown a list of all of the SPM files in your Sprint directory; select DIR.SPM and press Enter. You've now loaded the macro and compiled it into Sprint.

To use the macro, select the Utilities/Macros/Enter command. When the status line prompts you for the macro name, enter GetDirectory. Sprint then asks if you wish to Execute the macro or to Assign the macro to a key sequence. Press "A" (for Assign), then press Alt-H to assign the macro to the Alt-H hotkey. (I present Alt-H as an example because it isn't used by the standard Sprint interface.)

Now, move the cursor to some point in your file and press Alt-H. When the status line prompts you for a file specification, type "*.*" and press Enter. The message "Looking for *.*" appears on the status line, followed by the message "Reading in results...". The standard directory information for the specified files is then inserted at that point in your file.

THE call OF THE WILD

The techniques embodied in GetDirectory can easily be adapted to other purposes. For example, you could reproduce the Utilities/DOS command function for use within a custom user interface. The call command could be set up to run another program instead of COMMAND.COM. call is your gateway to DOS—let your imagination go to work.

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 Library I of the BORAPP forum on CompuServe, as SPDIR.ARC.
Introducing Sprint—the professional, programmable word processor!

SPECIAL OFFER: ONLY $99.95!
The race into the Age of Customization is on—led by Sprint.* You can use Sprint as is and be very happy with the way everything works for you—or you can easily customize Sprint to do everything your way.

It's a completely customizable word processor that, for example, lets you re-define keys, delete menu items, make your own shortcuts, invent your own menus, and use Sprint's online facility to create your own quick reference cards.

You're given the customizing power to avoid pop-up menus altogether—if that's the way you like to work. Sprint can be completely function-key-driven, and while Sprint's function key assignments are logically defined, they're easy to alter.

Nothing goes slow when you Sprint!

Sprint is fast. It scrolls fast, edits fast, switches between files fast, offers fast shortcuts and proves that the slow way is no way.

You can work on up to 24 files at once, divide your screen into as many as six windows, and never miss a beat because Sprint remembers which files you were working on last.

Because Sprint brings you the speed you're used to with Turbo Pascal® and Turbo C®, it never wastes your time and true Turbo-performance is finally available in a text editor.

To see just how much faster Sprint works for you, check out the comparative time tests.

Sprint gives you six optional interfaces including EMACS

The customizing you choose to do is one variation on Sprint's theme and there are six others.

We give you free (for a limited time) Alternative User Interfaces for:

- EMACS
- WordStar®
- Microsoft® Word
- MultiMate®
- WordPerfect®
- SideKick®

And you also get file conversions for:

- WordStar
- Microsoft Word
- WordPerfect
- MultiMate
- DisplayWrite® 4

Prices and specifications subject to change without notice.

Minimum System Requirements:

For the IBM PS/2 and the IBM family of personal computers and all 100% compatibles. Requires PC-DOS® (MS-DOS® 2.0 or later), 256K memory (384K recommended), and two floppy drives or a hard disk.

*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.
you can **Sprint**?

Sprint lets you use EGA and VGA cards for 43- or 50-line displays; it directly reads ASCII files without conversion and saves files with hard carriage returns for electronic mail.

You're given a built-in compiler with a syntax similar to C; separate source files; an extensive macro language; the ability to call DOS functions and much, much more.

"Auto-Save" means you'll never lose your work when you **Sprint**!

Forgetting to "Save" is a fact of life as are power outages, and it used to be that fact of life as are power outages, and it used to be that a power outage could wipe out everything you've done. Not any more. Your work is always safe when you **Sprint**.

**Sprint's** "Auto-Save" automatically saves your words as you type, so if the lights do go out, you may be in deep dark-ness but not deep trouble. Sprint's Auto-Save is more than "insurance," it's also invisible. You know it's there, but it does its job without interrupting yours.

---

### See how fast you can **Sprint**!

<table>
<thead>
<tr>
<th></th>
<th>Save File</th>
<th>Top to Bottom</th>
<th>Go To Line 1500</th>
<th>Search &amp; Replace</th>
<th>Find Unique Word</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sprint 1.0</strong></td>
<td>5.9</td>
<td>.1</td>
<td>.1</td>
<td>1.6</td>
<td>3.3</td>
</tr>
<tr>
<td>WordPerfect 4.2</td>
<td>41.1</td>
<td>5.3</td>
<td>5.4</td>
<td>6.6</td>
<td>6.2</td>
</tr>
<tr>
<td>WordStar 4.0</td>
<td>4.4</td>
<td>4.6</td>
<td>4.7</td>
<td>17.1</td>
<td>13.8</td>
</tr>
<tr>
<td>MS Word 4.0</td>
<td>9.7</td>
<td>.1</td>
<td>N/A</td>
<td>4.6</td>
<td>7.0</td>
</tr>
</tbody>
</table>

Tests were performed on a Multitech 286 AT (8 MHz), 640K RAM, file size 103K, 1636 lines. 14 occurrences. Times shown are in seconds. (Benchmark details available upon request.)
You have a head start when you *Sprint!*

<table>
<thead>
<tr>
<th>Feature</th>
<th>Sprint 1.0</th>
<th>WordPerfect 4.2</th>
<th>MS Word 4.0</th>
<th>WordStar 4.0</th>
<th>MultiMate Adv. II, 1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum file size</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
<td>Disk</td>
<td>128K</td>
</tr>
<tr>
<td>Mail Merge</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Thesaurus (integrated)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Windows Open (maximum)</td>
<td>6</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Files Open (maximum)</td>
<td>24</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cross-Reference (dynamic)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Indexing Options</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>No</td>
</tr>
<tr>
<td>Snaking Columns (chg # on same page)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Not same pg.</td>
<td>Yes</td>
</tr>
<tr>
<td>Parallel Columns</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>H-P LaserJet Support</td>
<td>Full</td>
<td>Full</td>
<td>Full</td>
<td>Partial</td>
<td>Full</td>
</tr>
<tr>
<td>PostScript Support</td>
<td>Full</td>
<td>Text</td>
<td>Full</td>
<td>No</td>
<td>Text</td>
</tr>
<tr>
<td>Mouse Support (integrated)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>AutoSave (without interruption)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>User Interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Define Shortcuts Dynamically</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Run Alternative User Interface</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Verify spelling as you type</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Fully programmable macro language</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**What you get when you *Sprint!***

- Includes Auto-Save that saves your work without interrupting it
- Sprint supports 350 popular printers including HP LaserJet,* other laser printers and typesetters plus has PostScript* support
- Supports multiple fonts, including downloadable fonts, in all sizes including scaled sizes
- Includes file conversions for Microsoft Word, WordPerfect, MultiMate, WordStar, and DisplayWrite 4 (DCA RFT)
- Includes Alternative User Interfaces for EMACS.

Suggested List Price  
- $199.95
- $495.00
- $450.00
- $495.00
- $565.00

60-Day Money-back Guarantee*

To order now,  
Call (800) 543-7543
Special offer: Sprint for only $99.95!

For registered Borland customers and for a limited time only (offer ends September 30, 1988), Sprint is all yours for only $99.95 direct from Borland!

The suggested retail price for Sprint is $199.95. We think $100.00 off is the best way we can show our appreciation for your loyalty and support. (When you consider that many word processors are in the $500 to $600 range, that $99.95, including 6 alternative user interfaces, should start looking even better!)

Sprint works with today's hardware and will work with tomorrow's. Anywhere from an 8088 PC through a 386.

It's already a major success story in Europe; it's the #1 selling word processor in France (and everyone knows, 50 million Frenchmen can't be wrong!)

Sprint. It's the word processor you'd expect from Borland: the value, technical excellence and programmability you'd expect from Borland. Sprint, for your eyes only, $99.95.

*Plus shipping and tax if applicable.

---

Section 5

SALES AND MARKETING PLAN

5.1 New Pricing for Video 2

Video 2 will be priced competitively. Its price will be $50 lower than the top selling competitor, Super Video, but slightly more than two other similar products. Video 2 is clearly a better product than all of its competition. The following charts compare competitive prices.¹

<table>
<thead>
<tr>
<th>Brand</th>
<th>Minimum Base Price</th>
<th>Average Base Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprint</td>
<td>$99.95</td>
<td>$129.95</td>
</tr>
<tr>
<td>Super Video</td>
<td>$149.95</td>
<td>$179.95</td>
</tr>
<tr>
<td>Generic</td>
<td>$159.95</td>
<td>$189.95</td>
</tr>
</tbody>
</table>

We took a random sampling of 100 consumers in each of our four sales regions. We then averaged the number of each region to get the maximum price point (see Table 5.1). The suggested price for Video 2 is below all maximum price points.

5.2 Marketing Tools

In order to maintain an increasing market share with Video 2, we need to arrange the following sales promotion:

- **New Product Literature.** We will write new sales literature about Video 2 to all the major music chains and music stores along with four demo tapes. The literature should include the following:

  1. Identification of the new improvements.
  2. Use for the product. The recommended uses and benefits of Video 2 need to be spelled out clearly. (This will also be incorporated into a featured piece for the Monthly Bulletin in Music Stores.)
  3. In-store marketing of product. Show how Video 2's new display rack will fit neatly as the end of an aisle or as part of an existing display. Dealers will be able to order the product on extended terms and return the product for stock balancing if their sales of Video 2 do not exceed their sales of the original Video 1.

- **New Product Announcement.** We will write a new product announcement to all dealers as a "Take the Video 2 Test." Offer

- **New Co-op Advertising Program.** This new program will emphasize the new collaborative policy of the company, and the new product line.

Table 5.1: Survey Results

<table>
<thead>
<tr>
<th>Region</th>
<th>Maximum Base Price</th>
<th>Average Base Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>North</td>
<td>$129.95</td>
<td>$149.95</td>
</tr>
<tr>
<td>South</td>
<td>$129.95</td>
<td>$149.95</td>
</tr>
<tr>
<td>Midwest</td>
<td>$129.95</td>
<td>$149.95</td>
</tr>
<tr>
<td>West</td>
<td>$129.95</td>
<td>$149.95</td>
</tr>
</tbody>
</table>

¹The average results are all manufacturer's suggested retail prices as of June 1, 1988.
²The poll was conducted by our Market Research Group. The complete results of the poll are listed in Appendix A.

National Oceanic Music Corporation Report
Last issue, I talked about a number of the basic data types and structures, and how to design them. In this issue, I'll explore another category of data types and will discuss some guidelines for data structure design.

**Enumerated Types**

An enumerated data type (or EDT) is a user-defined data type. In defining an EDT, you list (or enumerate) its possible values, which are identifiers enclosed within parentheses and separated by commas. Any of the enumerated values can then be assigned to variables that are declared to be of that data type. Figure 1 gives examples of a few EDTs in both Pascal and C, and shows how you might use them.

Enumerated data types are actually disguised integer constants. In Pascal, they're strongly disguised—you can't use these integer constants directly in integer expressions unless they're converted first into an integer value via the `Ord` transfer function. In C, you can treat enumerated data types exactly like `int` values. In both languages, the first identifier has a default value of 0, the next has a value of 1, and so on. Thus, N identifiers map onto the integer range 0..N-1. As shown in Figure 1, C gives you the additional ability to explicitly assign values to given identifiers. In fact, since EDTs in C are really just integers, most of the following discussion focuses on EDTs in Pascal.

What issues are involved in the design of enumerated data types? The first issue is this: Why use enumerated data types at all? Answer: To help document your program. When used properly, EDTs make your programs easier to read and modify. Compare the code shown in Figure 1 with that in Figure 2, which shows the Pascal code from Figure 1 with all EDT values converted to their corresponding integer values. It is not at all clear from the code what the values 9, 1, 5, and 3 represent; in fact, the first value is misleading, while the last value bears no apparent relation to what it represents.

This brings up the second issue in EDT design: mapping. As defined in Figure 1, the values `January` through `December` in the EDT `Months` correspond to the integers 0..11. However, you may want these values to correspond to the integers 1..12 for calculation or other purposes. To implement this change

```
Listing 1: TESTDAYS.PAS

program TestDays;
uses CRT; { for ClrScr, GoToXY, ClrEol }
type
  Days = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat, endDay);
const
var
  Today, Tomorrow : Days;

function ToUpper(S : string) : string;
var
  I, Len : word;
  Ch : char;
begin
  Len := Length(S);
  for I := 1 to Len do begin
    Ch := S[I];
    if ('a' <= Ch) and (Ch <= 'Z') then S[I] := Chr(Ord(Ch)-32)
  end;
  ToUpper := S
end;

function GetDay(Prompt : string) : Days;
{ writes out Prompt at (1,1) -- continues to prompt until
  the user enters a valid day name (Sun..Sat); case doesn't
  matter -- returns the day value entered }
var
  Ans : string[5];
  Day : Days;
begin
  repeat
    GoToXY(1,1); ClrEol;
    Write(Prompt);
    Readln(Ans);
    Ans := ToUpper(Ans);
    Day := Sun;
    while (Day <= endDay) and (Ans <> ToUpper(DayName[Day])) do
      Day := Succ(Day)
  until Day <> endDay;
  GetDay := Day
end;

begin (main body)
  ClrScr;
  Today := GetDay('Which day of the week is today? ');
  Tomorrow := Succ(Today);
  if Tomorrow = endDay
    then Tomorrow := Sun;
  Write('Tomorrow is: ', DayName[Tomorrow])
end. (of program TestDays)
```
**Turbo Pascal:**

```pascal
type
  Days  = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
  Months = (January, February, March, April, May, June, July, August, September, October, November, December);
  Coins = (penny, nickel, dime, quarter, halfdollar, dollar);

var
  Today    : Days;
  ThisMonth: Months;
  Coin     : Coins;

begin
  ThisMonth := October;
  for Today := Mon to Fri do begin
    end;
  Coin := quarter;
  end.
```

**Turbo C:**

```c
typedef enum { sun, mon, tues, wed, thurs, fri, sat } days;
typedef enum { january, february, march, april, may, june, july, august, september, october, november, december } months;
typedef enum { penny=1, nickel=5, dime=10, quarter=25, halfdollar=50, dollar=100 } coins;

main {
  days    today;
  months  thisMonth;
  coins   coin;

  thisMonth = october;
  for(today = mon; today <= fri; today++) {
    ...
  }
  coin := quarter;
  ...
}
```

In Pascal, you must actually expand the EDT definition and give it a new dummy initial value to map to 0. In C, another solution is available: Just assign the value of 1 to the identifier `January` and the other months will follow suit, so that `February` will have a value of 2, and so on. Figure 3 shows examples for both languages.

Note that the `noMonth` solution in Figure 3 has another advantage: It can act as a "null" value. When assigned to a variable, a null value indicates that the variable doesn’t currently hold any particular month. This allows you to distinguish between variables that have actually been assigned a given value, and those which aren’t currently being used.

Another reason why you might want to “pad” the beginning or the end of an enumerated type with extra values (especially in Pascal) is range checking. Consider the code in Figure 4, which sets up a `while .. do` loop to cycle through the days of the week, and increments `Day` at the end of the loop. The problem is this: If range checking is enabled and `Day = Sat`, then the statement `Day := Succ(Day)` causes a run-time error. Why? Because the variable `Day` is only allowed to have the values Sun through Sat. Thus, `Succ(Sat)` is out of the range of the EDT, and therefore is undefined. One solution, shown in Figure 5, is to pad the EDT with an extra value at the end so that `Day` holds the value `endDay` after the last call to `Succ`. (Another solution, of course, is to turn off range checking, but that decision may return to haunt you later.)

There’s one last problem with EDT’s: text I/O. Pascal doesn’t support the reading of EDT values directly into an EDT variable, nor does it allow an EDT value to be put into a `Write` or `WriteLn` statement. Your only real option with Pascal is to typecast the EDT value to some type (such as `Char` or `Integer`) that’s compatible with text I/O. C does let you treat `enum` variables just like any other integer variable, but (as with typecasting EDT values in Pascal) doesn’t help you if you want to enter or display an actual EDT value, such as `January` or `penny`.

**continued on page 138**

---

*Figure 1. Some examples of enumerated data types and how they are used.*

*Figure 2. The Pascal code from Figure 1, with all EDT values converted to their corresponding integer values.*
Turbo Pascal:

```pascal
type
  Months = (noMonth, January, February, March, April, May, June, July, August, September, October, November, December);
```

Turbo C:

```c
typedef enum { noMonth, january, february, march, april, may, june, july, august, september, october, november, december } months;
```

or

```c
typedef enum { january=1, february, march, april, may, june, july, august, september, october, november, december } months;
```

Figure 3. Pascal and C adjustments to make an enumerated type line up with a given range of integers.

```pascal
type
  Days = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat);
var
  Day : Days;
begin
  ...  
  Day := Sun;
  while Day <= Sat do begin
    
    Day := Succ(Day)
  end;
  ...
end.
```

Figure 4. Code that generates a runtime error if range checking is enabled.

```pascal
type
  Days = (Sun, Mon, Tues, Wed, Thurs, Fri, Sat, endDay);
```

Figure 5. One solution to the problem in Figure 4.

**BINARY ENGINEERING**

continued from page 137

The general solution here is to construct an array of strings that contain text equivalents of the EDT values, and then index the array by the enumerated type. For input, read in a string from the console, compare that string to each of the strings in the array, and use the corresponding EDT value when a matching string is found. To make the process case-insensitive, convert both strings to upper- or lowercase before the comparison. Similarly, the same array can be used to display or print the string that represents each EDT value as needed. I've provided the Pascal implementation of this solution in Listing 1; the C implementation is (as they say) left as an exercise for the reader.

**OUTPUT AND INPUT**

In this and my previous columns, I've talked a fair amount about some of the different data types and structures that you can have in a program. The question still remains: How do you go about designing them? The first step is to look at the output that your program requires. The ultimate purpose of a program is to produce output of some sort: text, data, graphics, electronic signals to hardware, or (in the case of many benchmarks) duration of execution. The information that a program generates determines what information it must track during execution—and the latter is the information that goes into your data structures.

Anticipated input also affects the design of your data structures, since you need some way to hold whatever data the program might receive during execution, whether that be from a file, from the user, or from some other device. The input itself is ultimately determined by the nature of the desired output.

**CALCULATION VERSUS STORAGE**

Just because your program outputs certain information doesn't mean that your data structures must hold that information. For example, a program that prints a multiplication table doesn't need to hold the entire table in an array. Instead, the program can generate a value in the table and then print the value, based upon a few pieces of information. Likewise, a program that draws a circle on the screen only needs to know the circle's center coordinates, its radius, the line width, and the line color; the program doesn't have to keep an actual copy of the screen image.

The same principle holds true for the internal workings of the program. Suppose you write a spreadsheet program where each cell is represented by a record. You could choose to implement the spreadsheet as a linked list of those records, traversing the list to find the cells as you need them. With this method, records are created only for the nonempty cells. While this minimizes memory usage, it does so at the cost of performance. As an alternative approach, you could implement the spreadsheet as a two-dimensional array of the same records. In this case, you can access any cell directly, and perform operations on the spreadsheet very quickly. Regardless of how small the actual spreadsheet is, however, a large amount of memory is used with this method, and the size of the spreadsheet is quickly limited.

The above example represents a classic tradeoff in computer programming: speed versus memory.
You can often make a program run faster if more explicit information is stored, but to do so requires additional memory. Likewise, less memory is used if the minimal information is stored and the rest of the information is calculated as needed; however, the program will run more slowly.

Which way should you go? That depends, of course, upon which resource is more limited: time or space. If you're trying to make the program run as fast as possible, then you can use memory to hold more explicit information. For example, I once wrote a graphics package that drew certain geometric figures by calling a few trig functions (sin and cosine). Unfortunately, that process slowed things down. My solution was to create an array to hold all of the required values, and then to index into that array to get the values. The result was a significant improvement in drawing speed, at the cost of the memory that was used to hold the array.

On the other hand, if memory is limited, then you can afford to take the necessary time to calculate information as it's needed, rather than to calculate the information once and then store it. For example, if an application requires your program to check if a number is prime, standard numerical checks can be used to determine "prime-ness," rather than using the method of storing a table of prime numbers.

GUIDELINES
What are the criteria to use when deciding which information will be stored and which will be calculated? Here are a few:

**How often do you need this information?** If it's used at only one or two spots in the program, you may be better off just calculating the information at those spots. On the other hand, if the information is used repeatedly—either within a loop or in many different places in the program—store the information in memory so that it can be quickly retrieved.

**How much space does the information use in relation to other data?** If you're writing a payroll program, the paycheck amount can be calculated as needed and may not need to be stored. But in this case, you're only looking at a few extra bytes per record, so why not just store the paycheck amount? It may come in handy elsewhere. On the other hand, storing an entire multiplication table takes up far more space than the upper and lower limits that are needed to generate it.

**How much time does it take to calculate the information?** The geometric graphics package mentioned above was written for the Apple II, which has a fairly complex relationship between memory locations and pixels on the screen. My program could make the necessary calculations to generate the correct byte and bit mask values that turn a given pixel on or off. However, this process slowed things down too much. My solution was to set aside several hundred bytes of memory for two lookup tables that gave me the byte-and-bit information for every X,Y pixel location on the screen.

**Do you really need the information?** It's easy to get into the habit of adding more fields to a record than you really need. When writing a payroll program, for example, you might be tempted to put a lot of extraneous information into each record (date of birth, sex, height, weight). Such information normally has no bearing on the determination of how much the person gets paid, so why store it? Remember: If it doesn't affect output, then you probably don't need it.

**Do you need to access "subsections" of the information?** If you write a program that prints a list of numbers in a series, then your best bet may be to simply generate the numbers as they are printed. However, if you need to reference specific numbers in that series, then storing the series in an array may be the better answer.

**How much storage space do you have to spare?** If you've got a lot of space to work with, then go ahead and use it. The use of more space will improve performance and reduce program size and complexity. Likewise, if you're tight on memory, then look for ways to swap speed for space. Also, disk space can become critical as well, especially if you're using diskettes.

How many instances of the data structure will you need? There's a difference between maintaining an address list of a few friends, and writing a database program to track 25,000 students. In the first case, you can store a lot of information for each friend, since you're unlikely to run out of memory. In the second case, every byte in a record definition adds 25K to the database size, and memory disappears in a hurry.

**How critical is performance?** In the case of the graphics package, performance was extremely critical, so I elected to use a large amount of memory for the lookup tables, rather than to use a small, simple subroutine to perform the byte-and-bit calculations. Even though memory was also very tight, I chose to go for performance and look for memory savings elsewhere.

**DESIGNS ON DATA**
Data structure design, like algorithm design, is both a science and an art. There are rules, guidelines, and even formulae that you can apply in order to figure out the best solution to a given problem. At the same time, an instinct for quickly narrowing down the choices comes with time and practice—practice that includes a fair amount of trial and error. The trick is to be aware of the possibilities, and to look for new and different solutions, rather than to always adhere to the same old methods. It may turn out that the old approach was the better one, but that's valuable information as well. As always, the best way to hone your programming skills is to sit down and write programs.

In the next issue of TURBO TECHNIX, I'll continue with part three of this discussion, where I'll compare linked lists and arrays. In the meantime, happy coding.

Bruce Webster is a computer mercenary living in California. He can be reached on MCI Mail (as Bruce Webster), and on BIX (as bwebster).

Listings may be downloaded from Library 1 of CompuServe forum BPROGA, as EDT.ARC.
Because of its early association with the Japanese fifth-generation supercomputer project of the 1990s, Prolog has been called a fifth-generation language. Assembly language, on the other hand, gets as close to the nuts and bolts of the computer as is possible without using binary. One wonders, then, why the Turbo Prolog programmer would ever consider "dirtying" his/her hands with something as low-level as assembly language. Actually, there are a number of very good reasons to link your Turbo Prolog routines with Turbo Assembler. Two reasons that immediately come to mind are the resulting increase in speed and the reduction in code size. Another reason to use Turbo Assembler is to perform some of the low-level functions that are difficult to handle in Turbo Prolog. In this way, you can develop new predicates to further extend the capabilities of Turbo Prolog.

This article will take you step-by-step through the Turbo Prolog/Turbo Assembler connection. In the process, I'll examine how a Turbo Assembler predicate is created, and will show how to pass simple data types between the two languages. Ultimately, you'll walk away from this article with two new predicates—open and read—to open and read binary files.

READING BINARY FILES

Turbo Prolog provides the filemode predicate to open files in either text or binary mode. In text mode, Turbo Prolog translates certain characters so that the file is more readable. For instance, the sequence 0D0AH is interpreted as a carriage return/line feed, which generates a new line. Binary mode, on the other hand, allows your program to read a file without making any such conversions. Since this process is particularly useful when reading binary files, this mode is called binary mode.

Unfortunately, a couple of characters are special in Turbo Prolog. One such character, 1AH, marks the end of a file. Therefore, whenever a file is read in binary mode and the character 1AH is encountered, Turbo Prolog thinks the job is done and ignores the rest of the file. One solution to this dilemma is to write an assembler routine to read the file.

DUMP.PRO (Listing 1) shows a simple Turbo Prolog program that calls two assembly language modules. This program opens a file, reads each byte from the file, and then displays the bytes on the screen. DUMP reads any file, including binary files, and shows how to pass simple data types from Turbo Prolog to Turbo Assembler.

Starting with the run predicate in Listing 1, the program creates a window, prompts the user for a filename, and then passes the filename to the readFILE predicate:

```
readFILE(filename):-
  open(filename,FileHandle),
  FileHandle <> 255,
  repeat,
  read(FileHandle, NumberBYTESread, ReadBuf),
  processBYTE(NumberBYTESread, ReadBuf),
  NumberBYTESread = 0, !.
```

`readFILE` takes the filename that is passed to it, and immediately calls `open` (which is written in Turbo Assembler).

The first argument of `open` contains the name of the file to be opened (this filename is declared as a string). The second argument returns an integer value that corresponds to the file handle created when `open` opens the specified file. If the file can't be opened, then `open` returns the value of 255. The statement that follows the call to `open` checks to make sure that `FileHANDLE` is not equal to 255. If `FileHANDLE` is set to 255, this statement fails, causing Turbo Prolog to backtrack to the second `readFILE` clause:

```
readFILE(filename):- 1,
  removeWindow,
  write("Sorry, unable to open ", filename, ",\n", nl,
exit.
```

In the case where `open` returns a valid file handle, `readFILE` enters the `repeat` loop (for more on `repeat` loops, see "Failing With Grace," TURBO TECHNIX, July/August, 1988). The `repeat` loop continually calls `read` (also written in Turbo Assembler) to read a byte from the file that was just opened, and calls `processBYTE` to write that byte on the screen.
read takes FileName as its first argument, and returns two integer arguments: NumberBytesRead and ReadBuf. Normally, NumberBytesRead is set to 1 to indicate that a single byte has been read from the file. ReadBuf contains an integer that represents the ASCII code for the byte that was read from the file. Next, NumberBytesRead and ReadBuf are passed to processByte, which displays on the screen the ASCII code of the character in ReadBuf. (Of course, this routine could also process the input byte in many other ways.)

The terminating condition, NumberBytesRead = 0, returns true when the end of the file is reached, and terminates the repeat loop. Once the condition NumberBytesRead = 0 succeeds, readFILE also succeeds and control returns to run. Since there are no more subgoals to execute, the program ends.

INTERFACING CONSIDERATIONS

When passing an argument to an assembler routine, Turbo Prolog pushes that argument onto the stack. Turbo Prolog may push either the actual value of the argument or the address of the argument, depending upon the argument's data type. If the argument is an integer, for instance, the actual value is placed on the stack. On the other hand, if the argument is a string, then the address goes on the stack. Table 1 summarizes the manner in which arguments are passed onto the stack.

<table>
<thead>
<tr>
<th>IF THE ARGUMENT IS:</th>
<th>AND THE VARIABLE IS:</th>
<th>WHAT IS PUSHED ONTO THE STACK IS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>An output argument</td>
<td>Any data type</td>
<td>4-byte address of the output argument</td>
</tr>
<tr>
<td>An input argument</td>
<td>String, symbol, or compound object</td>
<td>4-byte address</td>
</tr>
<tr>
<td>An input argument</td>
<td>Integer</td>
<td>2-byte value</td>
</tr>
<tr>
<td>An input argument</td>
<td>Real</td>
<td>8-byte value (IEEE format)</td>
</tr>
<tr>
<td>An input argument</td>
<td>Char</td>
<td>2-byte value</td>
</tr>
</tbody>
</table>

Table 1. A summary of how various data types are pushed onto the stack.

There's no need to worry about the data type of return arguments, since Turbo Prolog always passes return values by reference.

In addition to passing arguments to the assembler routine, Turbo Prolog pushes a four-byte return address onto the stack. This address is the location of the next instruction in the Turbo Prolog program where execution is to continue after the assembly language predicate finishes its work. For instance, consider the call to open, which passes two arguments:

open(FileName, FileHandle)

Recall that FileName is a string variable. According to Table 1, Turbo Prolog pushes the address of a string on the stack. Thus, the first argument on the stack is a four-byte address. Since FileHandle is a return value (as designated by the output flow pattern in the global declaration), FileHandle is also passed as a four-byte address. Figure 1 shows what the stack looks like when open (in Listing 2) first starts its work. These initial conditions are called the activation record for the Turbo Assembler module.

<table>
<thead>
<tr>
<th>BOTTOM</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of FileName</td>
<td>4 bytes</td>
<td></td>
</tr>
<tr>
<td>Address of FileHandle</td>
<td>4 bytes</td>
<td></td>
</tr>
<tr>
<td>Return address to Turbo Prolog routine</td>
<td>4 bytes</td>
<td></td>
</tr>
</tbody>
</table>

TOP

Figure 1. The activation record for OPEN.ASM.

It's good practice to push the contents of the base pointer (BP) onto the stack in order to keep the base pointer out of harm's way. That's because the calling program needs the contents of BP to be preserved. This step is performed as follows:

PUSH BP
MOV BP, SP

Now BP, instead of the stack pointer (SP), can be used to access the stack. Keep in mind that whenever something is pushed onto the stack, the location of all of the items on the stack is changed. Figure 2 shows what the stack looks like after BP is pushed onto it.

<table>
<thead>
<tr>
<th>BOTTOM</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of FileName</td>
<td>4 bytes</td>
<td></td>
</tr>
<tr>
<td>Address of FileHandle</td>
<td>4 bytes</td>
<td></td>
</tr>
<tr>
<td>Return address to Turbo Prolog routine</td>
<td>4 bytes</td>
<td></td>
</tr>
<tr>
<td>BP</td>
<td>2 bytes</td>
<td></td>
</tr>
</tbody>
</table>

TOP

Figure 2. The status of the stack after pushing the base pointer.

DOS interrupt 21H (function 3DH) is used to open the file specified in DS:DX. This function requires that the string that represents the filename be an ASCII (null-terminated) string. Since Turbo Prolog stores its strings as ASCII strings, there's no need to convert the string.

To get the address of FileName, the segment of the address must be moved into DS, and the offset of the address must be moved into DX. As you can see from Figure 2, the offset is at the position BP + 10, and the segment is at BP + 12.

MOV DX, [BP] + 10
MOV DS, [BP] + 12

continued on page 142
LISTING 1: DUMP.PRO

/*

NOTE: This program links in two external object files: open.obj and read.obj. Assemble these files using Turbo Assembler. Be sure to compile this program as a project.

If compiling as a project, create a project file called "dump.prj" with the following text:

simple
open
read

To compile, choose "Project" from the "Compile" pulldown menu, and select the appropriate project file.

PROJECT "SIMPLE"

global DOMAINS
file=infile
STRINGLIST = STRING*
INTEGERLIST = INTEGER*

GLOBAL PREDICATES
open(STRING,INTEGER) - Ci,0) language asm
read(INTEGER,INTEGER,INTEGER) - Ci,o,o) language asm

PREDICATES
run
readfile(STRING)
repeat
processBYTE(INTEGER,INTEGER)

GOAL
run.

CLAUSES

/**************************** run ***************************/
run if
makewindow(1,7,7,7,"Test of OPEN and READ routines ",0,0,25,80),
cursor (5,10),
write("Enter filename :
11 "),
read ln(Filename),
clearwindow,nl,nl,
readFILE(Filename).

/**************************** readFILE ***************************/
readFILE(Filename) if
open(Filename,FileHANDLE),
FileHANDLE <> 255, /*Check for error in opening file */
repeat, /* we backtrack to here if not EOF */
read(FIELDHANDLE,NumberBYTESread,ReadBUF),
processBYTE(NumberBYTESread,ReadBUF),
NumberBYTESread = 0,1, /* Check for end of file. */
readFILE(Filename) if 1,
removewindow,
write("Sorry, unable to open ",Filename,"."),nl,
exit.

/**************************** processBYTE ***********************/
processBYTE(NumberBYTESread,ReadBUF) if
NumberBYTESread < 0, 0,
write(ReadBUF," "), 1.
processBYTE(0,) if 1,
nl,
write("End of file.
"),
nl.

/**************************** END OF DUMP.PRO ***********************/

LANGUAGE CONNECTIONS

continued from page 141

Before calling function 3DH, we need to take care of a few other registers. First, set AL to zero to specify that we want to open a file for reading. Since AH specifies the function call, set AH to 3DH.

Once the registers have been set, make the call to int 21. If no problems arise, then interrupt 21H opens the file and returns the file handle in the AX register. If DOS cannot open the file, the carry flag is turned on. The following instruction lets us easily check if the carry flag is set:

JC FAILURE

This instruction says, "If the carry flag is set, jump to FAILURE." If the carry flag is not set, the address of the file handle is assigned to DS:DI via the LDS instruction:

LDS D1,DWORD PTR [BP] + 6

Next, move the contents of AX to the output argument, and restore the DS and BP registers. There is one final bit of cleanup to do before control is returned to Turbo Prolog. Eight bytes must be popped off the stack for FileName and FileHANDLE (four bytes each).

Listing 4 shows the read routine that actually reads the file that was just opened. The module is straightforward and well commented, so I'll leave this module as an exercise for the reader. The overall connection is similar to the open routine. Be sure to save the original base pointer, and to move the stack pointer to BP. Also, don't forget to save Turbo Prolog's data segment. When returning to Turbo Prolog, remember to restore the original base pointer, along with the data segment. Finally, pop all "pushed" parameters off the stack.

ONE STEP BEYOND

This example demonstrates how to pass simple data types, such as strings and integers, between Turbo Prolog and Turbo Assembler. A number of unanswered questions still remain. For instance, there's the question of how to handle complex structures, such as lists or compound objects. In addition, you'll need to know how to allocate memory for structures in Turbo Assembler in a way that's acceptable to Turbo Prolog. Finally, you may want to call a Turbo Prolog predicate from Turbo Assembler. The answers to these questions will be the subject of a future "Language Connections." •

Philip Seyer is a composer, writer, and microcomputer analyst. He is coauthor of Turbo Prolog Advanced Programming Techniques, TAB Books, Inc.

Listings may be downloaded from Library 1 of CompuServe forum BPROGB, as PROASM.ARC.
LISTING 2: OPEN.ASM

This program receives the address of an ASCII string (a string that ends with a binary zero). The string contains the name of a file that is to be opened. The program opens the file and returns an integer that acts as a file reference (also called a file handle). We return the file handle by placing its address on the stack.

10 <--- FIRST PARM (INPUT PARM)
9
8
7
6
5
4
3
2
1
0

TOP OF STACK

A_PROG SEGMENT BYTE
ASSUME CS: A_PROG
PUBLIC open_O
open_O PROC FAR
MOV SI,DS ;save SI on stack
PUSH BP ;save BP on stack
MOV BP,SP ;BP = SP
MOV DX,[BP]+10 ;get offset address of filename
MOV DS,[BP]+12 ;get segment addr. of filename
SUB AL,AL ;set AL to 0 for read access
MOV AH,3Dh ;open function
INT 21h ;Invoke the interrupt
JC FAILURE ;Jumps to FAILURE if carry flag is set
MOV AX,OFFh ;Make AX 0
INC AX ;Increment AX
MOV BYTE PTR [AX],OOh ;Move zero to hibyte
AX shows the number of bytes read. So now we return this information to Prolog.

FAILURE:
MOV AX,OFFh ;Offh will be error flag
MOV [DI],AX ;Move DI point to output parm
POP BP ;Restore BP
MOV DS,SI ;Restore DS
RET 8 ;pop the parms off the stack

INT 21h ;If carry flag is set
MOV DS,[BP]+10 ;Get Handle
MOV AX,OFFh ;Offh is hex for -1
LDS DI,DWORD PTR [BP]+10 ;Get Handle
MOV [DI],AX ;Move AX to ByteArrayInputStream
POP BP ;Restore BP
MOV DS,SI ;Restore DS
RET 10 ;Pop the parms off the stack

FAILURE:
MOV AX,OFFh ;Offh is hex for -1
LDS DI,DWORD PTR [BP]+10 ;Get Handle
MOV [DI],AX ;Move AX to ByteArrayInputStream
POP BP ;Restore BP
MOV DS,SI ;Restore DS
RET 10 ;Pop the parms off the stack

The 10 after RET is critical. Without it, the return to Turbo Prolog will be fouled up.

We put 10 after RET here because we need to pop the parameters that were pushed on the stack when this routine was called.

This routine receives 3 arguments on the stack:
- FileHANDLE (2 bytes)
- ByteArrayInputStream (4 byte pointer)
- 2*4 + 4 + 4 = 10, hence the 10 after RET

read_O ENDP
A_PROG ENDS
END

LISTING 3: READ.ASM

This program receives an integer on the stack that refers to an open file. This number is called a file handle. The Turbo Prolog program gets the file handle by calling the assembly language predicate "open." FileHANDLE appears at offset 14 on the stack.

BOTTOM OF STACK

15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0

TOP OF STACK
The fundamental purpose of the Runtime is to provide a common set of building blocks with which you can more easily and quickly construct applications. Starting with this column, we turn for a while to the addition of significant and useful new building blocks.

Our first building block is a basic command parser and handler. Many applications use command lines. Even if you’re not interested in building such an application, this example shows how to integrate a completely new building block into the Runtime.

**THE TALE**

A command handler insulates you from the messy details of reading and parsing a command line. It handles leading spaces, ignores blank lines, lets the user delete characters, and searches for a command match. The command handler presented in this article also includes a routine, cmd_file, that redirects input from the standard input file (stdin) to another file.

The command handler is driven with a list of command keywords. The list should contain one or more entries of the structure type cmd_key (see CMD.H, in Listing 1). Each entry in the list has three components: a command keyword, a routine to call for that command, and a pointer to the next entry. The “next entry” pointer for the last entry must be null.

The command handler is fairly straightforward to use. First, declare a list of command entries. (You must write the routines for each of the commands. Fortunately, those routines are the functions that your application provides.) The command prompt is established by calling cmd_init with a prompt string. Next, call cmd_read to get a line of input from the user. Finally, call the command parser, cmd_parse. cmd_parse returns either a pointer to the command entry that matches the user’s entry, or else a null pointer. cmd_parse also returns the rest of the command line if the user entered a valid command. A null pointer indicates that the user’s input did not match a command. In this case, you can call cmd_error to print a standard error message.

The command handler also handles lines that contain several commands in a row. For example, consider the following command line:

```plaintext
SET FOO ON
```

In this case, cmd_parse returns the command SET, and moves the start of the input line to the “F” in FOO. Call cmd_parse again to parse FOO. Finally, call cmd_parse a third time to parse ON.

It’s important to note that, even though the command entry structure (cmd_key) contains pointers to routines, the command handler does not use those pointers. The programmer must call the routine for the command that cmd_key returns. This design lets your program ignore those routines in some cases, and possibly to not provide these routines for some entries. This flexibility is useful, for example, when cmd_parse must get both a command (for which you probably would call a routine) and its arguments (for which you would not call a routine).

The complete command handler is in two files: CMD.H (Listing 1) and CMD.C (Listing 2). We suggest that you place CMD.H into your standard Turbo C include directory (usually \TURBOC\INCLUDE), since you need to include CMD.H in order to use the command handler.

**THE RUNTIME**

This article assumes that you’ve already loaded the Runtime source and built all of its libraries. If you have not, use the Runtime’s install procedure to build the directories and load the source files (see “Tales From the Runtime,” TURBO TECHNIX, July/August, 1988). Then change directories to the Runtime’s CLIB directory (usually \TURBOC\LIBRARY\CLIB), and use the following two batch file commands to build all of the objects and libraries:

```plaintext
CLIB all -1\turbo\include
CLIBRLIB
```

In addition, you must tell Turbo C to use these libraries rather than the standard libraries. To do so, run TC and select Options / Directories / Library directories. Assuming that you’ve used the standard directory structure from our earlier columns, enter the following directories:

```plaintext
C:\TURBOC\LIBRARY\CLIB;
C:\TURBOC\LIB
```

Turbo C now checks the modified libraries first, and then looks into its standard LIB directory for any-
thing that's not in those libraries. Choose Options/Store Options, and be sure to answer "Y" when Turbo C prompts you to overwrite TCCONFIG.TC.

The next step is to add the new routines to the libraries. Change directories to the CLIB Runtime directory, copy CMD.C to that directory, and then enter the following batch file command:

```bash
CLIBREPL all cmd,c -1\turbo\include
```

This batch file command updates all of the memory model libraries. You'll get the following warning message when you run this batch file for the first time:

```
Warning: 'CMDS' not found in the library
```

This warning message appears because CLIBREPL removes each old object from the library before it inserts the new object. Since our routines are not yet in those libraries, CLIBREPL cannot remove them; thus, it gives the warning and then adds our routines. If you repeat this process later, you won't get a warning.

**TAKING ORDERS**

CMDTEST.C (Listing 3) tests the command handler. Since we've heavily commented CMDTEST.C, we'll only touch on a few key points that may not be obvious.

To avoid problems with forward references, declare the routines for each command (exit, prompt, and execute) first. The problem of forward references occurs when you use a routine in a file before you present its code, as we do here. Without these declarations, the compiler complains that it doesn't know these routines when it encounters calls to them. An alternate solution to this problem is to change the order of the routines, so that each routine appears before it's used.

Place the commands into an array, `main_cmd[]`, of `cmd_key` structures. Initialize each element of the array to point to the next element (with the exception of the last element, which must be a null pointer). You don't have to worry about declaring the size of the array; the use of brackets ([ ]) forces the compiler to handle that for you. While this example uses an array, that's not a requirement. Also, the command entries must be in a linked list, but their order doesn't matter.

You can also use the same routine for more than one command, as shown by "exit" and "quit." Both of these commands use the standard C exit routine.

The main program of CMDTEST.C uses `cmd_init` to set its prompt. The program continues to get commands until the user enters "exit" or "quit" on the command line. The loop first calls `cmd_read` to read a line, and then calls `cmd_parse` on that line to get a command (along with any arguments). If `cmd_parse` returns zero, the user did not enter a legal command, so `cmd_error` is called. If the user entered a legal command, the loop calls the routine associated with that command, which is located in the `main_cmd[]` array. That routine is then passed with the rest of the command line. The command's routine could use `cmd_parse` to parse the rest of the command line, although we do not do so in our example.

`execute` uses `cmd_file` to redirect the command handler's input to the file whose name the user entered. `prompt` uses `cmd_init` to change the prompt to the string that the user entered after "prompt."

**INSIDE THE COMMAND HANDLER**

As with CMDTEST.C, CMD.H and CMD.C are heavily commented, so we suggest that you read them for a detailed explanation of the command handler. We will only touch on the high points here.

The main contribution of CMD.H is the command entry data structure, `cmd_key`, which is described above. CMD.H also contains constants that limit the maximum line and prompt sizes, plus templates for all of the command handler's routines.

CMD.C contains the source code for the routines. All of those routines share common variables that contain the input line, the prompt string, and pointers to the input and output files.

To avoid problems encountered when writing `prompt` in CMDTEST.C, `cmd_init` copies the prompt string to the command handler's internal prompt variable. We initially left the new prompt in the input buffer, which is where the new prompt is entered by the user. Since we saved only a pointer to that prompt, however, new text typed by the user appeared as the prompt the next time the command line was displayed. To avoid that problem, we maintained our own copy of the prompt string.

When a command does not match the command line, `cmd_error` simply displays an error message. You could place more sophisticated error handling into the program at this point, if you chose to do so.

`cmd_file` changes the command handler's input file. After the caller opens the input file, `cmd_read` reads that file, closes it, and returns to `stdin` for input. This approach precludes nested command files, a loss that we decided was not significant for the basic command handler.

`cmd_parse` parses the command line. The arguments to `cmd_parse` are the input line to be parsed and the linked list of possible commands. `cmd_parse` uses `cmd_token` to get the first token, and then sequentially searches the list of possible commands, using `cmd_compare` to compare the token with each command key. `cmd_parse` handles exact matches, one or more partial matches, and no match. If an exact match or only one partial match occurs, then `cmd_parse` returns the matching key. Otherwise, it returns zero.

`cmd_compare` compares the user's entry with the possible command keywords. `cmd_compare` returns `CMD_MATCH` for an exact match; otherwise, it returns `CMD_NOMATCH`. The constant

*continued on page 147*
LISTING 1: CMD.H

/* This file contains the information that we include in the command handler, CMD.C. */

LISTING 2: CMD.C

/* First we include several standard C include files, and then we include our command handler's include file, CMD.H. */
#include <stdio.h> /* For standard I/O functions */
#include <string.h> /* For string manipulation functions */
#include <conio.h> /* Our command handler's include file */

/* Now we define our command handler's variables. Note that by defining every function before we use it we avoid all forward reference problems. */
char cmd_buffer[ CMD_MAX ]; /* Input line buffer */
char cmd_prompt( CMD_PR_MAX ); /* Storage for the prompt */
FILE *cmd_ifp, *cmd_ofp; /* Input and output file pointers */
FILE *cmd_inp, *cmd_outp; /* Standard input and output device */

/* Now we define the command handler's functions. Note that by defining every function before we use it we avoid all forward reference problems. */

/* The first function, cmd_init(), sets the command handler's prompt, it does so by copying the prompt string from the container in which the caller passed it to our internal storage. It uses the standard Turbo C string copy function, and it copies up to the maximum prompt size. It does not return a value. */
void cmd_init( char *prompt ) /* Note that it returns nothing. */
    { strcpy( cmd_prompt, prompt, CMD_PR_MAX ); } /* end routine cmd_init() */

/* The cmd_error() routine is more of a placeholder than anything really useful. It just prints a generic error message. It has no arguments and does not return a value. */
void cmd_error( ) /* Note that it returns nothing. */
    { printf( "No command matches what you entered.\n" ); } /* end routine cmd_error() */

/* The next routine, cmd_file(), lets the caller re-direct the command handler's input from its default (or previous) source to a file. It takes the file pointer of that file as its only argument. It does not return a value. */
int cmd_file( FILE *ifp ) /* Pointer to the file that is to be the new input source */
    { /* Remember that the command handler gets its input from the file associated with the file pointer cmd_ifp. */
        cmd_ifp = ifp; } /* end routine cmd_file() */

/* cmd_compare compare a keyword with a token that cmd_parse has extracted from the user's input. */
void cmd_compare( char *key, char *token ) /* The comparison function */
    { int match; /* To get the comparison result from strcmp() */
        /* We use the standard library function strcmp() to do the comparison. It does a case-insensitive comparison. */
        match = strcmp( key, token );
        if ( match == 0 ) /* If they match exactly, or CMD_NOMATCH if they do not exactly match. */
            { /* This is the routine where we would add partial keyword matching. */
                return( CMD_MATCH );
            } else return( CMD_NOMATCH ); } /* end routine cmd_compare() */

/* cmd_token() finds the first "token" in a line of user input. We define a token to be a string of characters that contains no spaces and that is followed by either a space or an end-of-string null (\0). */
/* The command handler should not contain any non-printable characters (such as carriage return or line feed), and it must be terminated by a null. It is up to the caller to make sure that this condition is true. Note that cmd_read() always returns a string terminated by a null, so this condition should not be a problem. */
/* This routine has one argument, a pointer to a pointer to a line of input text. */
/* After it finds the first token, it moves the input line pointer to point to the first character after the space after that token. (It replaces that trailing space with a null.) If the token ends with a null rather than a space, this routine leaves the input line pointer pointing at the null. */
int cmd_token( char **tmp_ptr ) /* Pointer to a pointer to a line of input text */
    { /* pointer to the start of the input line */
        char *save_ptr, *tmp_ptr; /* pointer to the start of the input line */
        /* Start both of our work pointers at the beginning of the line */
        /* Listings continue on page 148 */

LISTINGS 1: CMD.H

/* This file contains the information that we include in the command handler, CMD.C. */

LISTING 2: CMD.C

/* First we include several standard C include files, and then we include our command handler's include file, CMD.H. */
#include <stdio.h> /* For standard I/O functions */
#include <string.h> /* For string manipulation functions */
#include <conio.h> /* Our command handler's include file */

/* Now we define several constants that we use throughout the command handler. These are result codes that cmd_compare() can return as it is comparing what the user entered to the set of legal keywords. */
#define CMD_PART 2
#define CMD_NOMATCH 0
#define CMD_MATCH 1
#define CMD_PART 2

/* Now we define several variables that we use throughout the command handler. */
char cmd_buffer[ CMD_MAX ]; /* Input line buffer */
char cmd_prompt( CMD_PR_MAX ); /* Storage for the prompt */
FILE *cmd_ifp, *cmd_ofp; /* Input and output file pointers */
FILE *cmd_inp, *cmd_outp; /* Standard input and output device */

/* Now we define the command handler's functions. Note that by defining every function before we use it we avoid all forward reference problems. */

/* The first function, cmd_init(), sets the command handler's prompt. It does so by copying the prompt string from the container in which the caller passed it to our internal storage. It uses the standard Turbo C string copy function, and it copies up to the maximum prompt size. It does not return a value. */
CMD_PART is defined for the case where the user's input token is a substring of the keyword. cmd_parse is designed to handle that case, but cmd_compare does not yet check for partial matches. This is a potentially important enhancement.

cmd_compare uses the standard library routine strcmpi to do a case-insensitive comparison. strcmpi returns 0 for a match. If no match occurs, strcmpi returns a negative or positive number to indicate which string is lexically "greater."

cmd_token finds and returns the first token in the input line that it receives as an argument. cmd_token also updates the pointer to that line to point to the first character after the space that follows the token. A token is defined in the program as a string of non-space characters followed by a space, or by a null character that marks the end of the string. The caller of cmd_token must ensure that the input line ends with a null. (The string returned by cmd_read is so terminated.)

cmd_token accepts all characters up to a space or a null. If cmd_token encounters a space, it replaces that space with a null, moves the input pointer past the space, and returns the token. If cmd_token encounters a null, it returns the token that it found, if any. cmd_token ignores all spaces at the beginning of the line.

cmd_read gets a line of input either from the console (stdin) or from a command file that is set by cmd_file. cmd_read also handles I/O chores such as echoing characters, ignoring leading spaces and empty lines, and deleting characters.

cmd_read uses getch to read from the console. getc is used to read from a file because getc only returns input when the user terminates the line by pressing Enter. Our program must examine every character, not just the entire line, so that we can provide command editing features.

One consequence of this choice is that input redirection at the DOS level will not work with our command handler. Fortunately, this is generally not a problem for an interactive command line parser.

cmd_read puts most of the characters that it encounters into the input buffer. Newline characters and carriage returns are treated in a special way, since they mark the end of a line. cmd_read also handles an end-of-file character, which causes cmd_read to close the input file and then reset the command handler to read from the console. Backspaces receive special treatment as well, because they are used for character deletion. Spaces are handled differently because they must be echoed on the screen, but ignored at the beginning of a line.

This design has two small flaws. First, because cmd_read ignores leading spaces and does not put them in the input buffer, they cannot be deleted. Second, if the user enters any other nonprintable characters, cmd_read puts them in the input buffer and passes them along. Remedies to these two flaws would improve the command handler.

ANOTHER TALE

We've already noted several useful improvements that you could make to this basic command handler. Many others are also possible, such as support for more than keyword parsing, with special codes for such things as numbers, filenames, and dates. You could also improve the basic data structure, perhaps by making the function field a union of a function, a pointer, or a number. With this modification, the caller could do more with the parsed command than just call a routine.

Our next column will focus on one "enhancement" that is, in itself, a useful building block: improvements to cmd_read. A number of improvements can be made, including the ability to support the DOS command-line editing functions. But, that's another tale.
/* cmd_parse() gets the first token from the input line and finds a command that the token matches. It also moves the input line position pointers, and then continues in the loop to get the next character. */

int match;

/* Now check to see if that token matches any of the command keys. Loop through the list of key entries and compare each key entry to that token. */

while ( *t"_ptr != '0' )
{
    /* Loop until we hit a null. */
    while ( *tmp_ptr != '0' )
    {
        /* If we are on a space, we must process it. If not, we just increment the line pointer and move on. */
        if ( *tmp_ptr == ' ' )
        {
            /* If the space is at the beginning of the line, we ignore it by incrementing both our work and initial line position pointers, and then continuing in the loop to get the next character. */
            if ( tmp_ptr == save_ptr )
            {
                tmp_ptr++;
                save_ptr++;
                continue;
            }
            /* If we are not at the beginning of the line, we replace the space with a null, move the pointer to the next character, and exit the loop. */
            *tmp_ptr++ = '0';
            break;
        }
        else
            tmp_ptr++;
        *cmd_ptr = *tmp_ptr; /* Update input line pointer */
        return( save_ptr ); /* Return pointer to token */
    }
    /* end routine cmd_token() */
    return( OL ); /* return code from cmd_parse() */
}

/* cmd_token() gets the first token from the input line and tries to find a command that the token matches. It also moves the input line pointer past the token. It returns either the command entry that the token matched or, if there was no match, zero. */

It has two arguments: a pointer to a pointer to a line of input text, and a linked list of command entries. */

/* Now handle the input character. */
switch ( c )
{
    case '
':
        /* If the input character is an end-of-file indicator, we must have just finished processing a command file. Close that file. Then reset our global input file pointer to point to standard input so that we can get more input interactively from the user. */
        if ( cmd_ifp == stdin )
        {
            cmd_ifp = stdin;
            c = getch();
            else
                c = getc( cmd_ifp );
        }
        /* Now handle the input character. */
        switch ( c )
        {
            case '
':
                /* If the input character is a backspace, use it to erase the previous character. If we are at the start of the line, ignore it, because there is nothing to delete. Otherwise, print backspace, space, backspace to erase the previous character. Then backup the buffer pointer. */
                if ( part_key == OL )
                {
                    printf( "\010", cmd_prompt );
                    cmd_ptr--;
                    break;
                }
                /* If the input character is either a carriage return or a line feed, we may be done. Echo a newline. Then, if we are at the beginning of the line, re-type the prompt and continue. Otherwise, we really are done with this line. Put a null in the buffer to terminate that string, and return a pointer to that buffer. */
                case 'n':
                case 'r':
                    putc( '\r', cmd_ofp );
                    if ( cmd_ptr == cmd_buffer )
                    {
                        printf( "\010", cmd_prompt );
                        break;
                    }
                    *cmd_ptr = 'A';
                    return( cmd_buffer );
                }
                else
                {
                    printf( "%s", cmd_prompt );
                    break;
                }
            else
                return( OL );
        }
        break;
    
/* end routine cmd_parse() */
/* We assume all other characters are printable ones that could be part of a command. We just copy them to the buffer and echo them on the screen. */
default:
  *cmd_ptr++ = c;
putc( c, cmd_ofp );
bang;
} /* end the switch statement */
} /* end character processing */

LISTING 3: CMDTEST.C

/* This file contains a simple program that tests our command handler.
We include in it both the standard Turbo C I/O routines and our command handler include file. */
#include <stdio.h> /* standard I/O routines */
#include <cmd.h>  /* Command parser definitions */

/* We define several of the routines in this file to avoid forward reference errors. */
void exitO;
void prarptO;
void execute();

/* main_cmd is an array of key entries that define our test's legal commands. We use the cmd_key structure from CMD.H. To show that more than one command can use the same action function, we make the exit and quit commands synonyms. We link each entry to the next one by using the & operator to get the address of that entry. To make the last entry point to nothing else, we use a 0 pointer. (We coerce the 0 to long to make it 32 bits. */
cmd_key main_cmd[ ] = {  
  "execute", execute, &main_cmd[ 1 ],
  "exit", exit, &main_cmd[ 2 ].
  "prompt", prompt, &main_cmd[ 3 ],
  "quit", exit, (cmd_key *) OL
};

/* The main program is a simple test that uses the command handler's functions. */
main ()
{  
char *arguments; /* Pointer to the arguments that the command handler returns */
cmd_key main_ans; /* The command entry it returns */
  
/* First we initialize our command prompt. */
cmd_init( "Cmd test" ) ;

/* Then we loop forever, processing commands. We exit the loop when the user enters exit or quit. */
while ( 1 )
{
  
/* First we get a line of input from the user. The string pointer that cmd_read returns points to the first command line that the user typed that contained something other than blanks. */
arguments = cmd_read();

/* We then call the command parser to parse that command line. We pass it the address of the address of the string it should parse, so that it can move the arguments pointer past the one command that it processes. In this way we can process a string of several commands by calling it with the same argument line repeatedly. We also pass it the address of the first entry in the linked list of command entries.
   cmd_parse returns the command entry from that table that matched the command the user entered. */

/* main_ans = cmd_parse( &arguments, &main_cmd[ 0 ] );
  */
  
/* cmd_parse returned a null command entry, then the user entered a command that did not match any of the legal options. In that case we call the command handler's error routine.
  */
if ( main_ans == (cmd_key *) OL )
  cmd_error();
else
  (*main_ans->function) ( arguments );
} /* end the command processing */

/* end the main program */

/* Now we provide the routines for the commands in our main_cmd structure.
main_cmd[ 1 ] and main_cmd[ 3 ] both use the standard C function exit().
main_cmd[ 0 ] uses the routine execute(), which we define below.
It executes commands from a command file whose name the user gives as an argument. It opens the file and passes its file pointer to the command handler's cmd_file() routine. */
void execute file() {  
  FILE *ifp; /* File pointer for that file */
  if ( fopen( file, "r" ) ) == NULL )
    printf( "Cannot open the file %5
", file );

// end routine execute */

/* main_cmd[ 2 ] uses the routine prompt(), which we define below.
It changes the command handler's prompt to the prompt string that the user entered by calling cmd_init() with that new prompt. */
void prompt(prompt_str) char *prompt_str; /* The new prompt string */
{  
  cmd_init( prompt_str );
} /* end routine prompt */
Right up there on the list of Great Unsolved Mysteries, along with Amelia Earhart’s disappearance and the Loch Ness Monster, are the questions, “Why didn’t they put interrupt-driven communications into the PC’s ROM BIOS?”, and “Why doesn’t interrupt 14H buffer the characters that come in from the communications line?”

If you wish to roll your own telecommunications code in Turbo Pascal, this little enigma means that you can poll interrupt 14H and content yourself with 300-baud operation. Otherwise, you can handle interrupts directly by diving into the arcane lore of the 8259A PIC and 8250 UART and then, before giving up, use somebody else’s library.

Designed for use with Turbo Pascal 4.0, Blaise Computing’s Turbo Asynch Plus is a library that effortlessly handles the messy details of interrupt-driven asynchronous communications on PC, AT, PS/2, and compatible computers. Turbo Asynch Plus’ three diskettes contain the units (.TPU files), .OBJ files, Pascal and assembler source code, project files for re-compilation, and sample programs.

**A text file device driver (TFDD) allows communications ports to be treated as files via the use of Readln and Writeln statements.**

The sample programs include a file transfer program that uses XMODEM protocol, a checkout program that tests and demonstrates the various Asynch Plus functions, and a (somewhat limited) terminate-and-stay-resident (TSR) program that performs background communications. Source code for the units is provided, but an assembler must be used in order to change any of the assembly language code.

Turbo Asynch Plus is organized into three levels for easier maintenance and comprehension. The first level, which is called “Level 0,” is written entirely in assembly language. This level handles the details of interrupt-driven communications, supports multiple ports, hardware and software flow control (DTR, RTS, and XON/XOFF in combination), baud rates up to 19,200, and the data word formats that are normally available through the BIOS. Level 0 also handles the buffering of characters to and from the communications ports, and the use of fixed-size circular queues that are provided by the caller. Level 0 may be linked into the rest of your program, or else loaded separately as a TSR utility.

The second of the three levels, “Level 1,” uses Level 0’s functions in order to provide a basic Turbo Pascal interface to the communications ports. Level 1 calls Level 0’s functions with the Turbo Pascal Intr() procedure. Level 1 allows you to set and read a port’s transmission options, read and write characters, and open and close the port completely from Turbo Pascal without worrying about assembly language or hardware arcana.

The last of the three levels, “Level 2,” extends Level 1 by adding such niceties such as the automatic management of the buffers on the heap, and a record structure for setting the communication port options.

Along with this three-level set of basic functions, Turbo Asynch Plus contains four other support units. One of these units accesses the PC’s normal BIOS services through interrupt 14H. (This unit is probably more useful on a PS/2, which has a more complete set of communications services, including 19,200 baud support, built into its BIOS.) The second support unit facilitates the process of sending commands and receiving responses from a Hayes-compatible modem. The third unit is an XMODEM unit that handles the process of transmitting and receiving files using XMODEM protocol with either checksum or CRC. Finally, a text device driver
allows communications ports to be treated as files via the use of ReadLn and WriteLn statements.

Turbo Asynch Plus’ documentation is good. The three-ring, PC-style slipcase binder includes 198 laser-printed pages and a complete index. One shortcoming for unsophisticated programmers, however, is that the fairly technical Level 0 functions are discussed before the more frequently used functions in Levels 1 and 2. Fortunately, numerous examples and an introductory section on asynchronous communications soften the blow. All of the example programs are extremely well-written and should be no trouble to work with. The .DOC files on the diskettes also provide additional information beyond the material in the manual.

If Turbo Asynch Plus has a drawback, it’s that the product’s scope is too narrow. In order for your program to perform terminal emulation, for example, you have to write the screen and control sequence handlers yourself. Similarly, the modem support provided in the modem unit consists simply of sending and receiving modem commands with error checking, and doesn’t include higher-level functions such as initialization, dialing, or hangup. (You can borrow routines that do some of these things from the FILEMOVE example program, however.) Support for other file-transfer protocols, such as Kermit, would also be useful.

But not everyone needs these extra functions, and Blaise has probably made a wise decision to concentrate on the basics of moving data over wires. If you need to do asynch communications with Turbo Pascal—especially over multiple lines simultaneously with industrial-strength error checking and recovery—then Turbo Asynch Plus is definitely worth looking into.

— Marty Franz

**The manual was written by someone with a clear and concise command of the English language and an obvious understanding of programming.**

TurboPower’s Turbo Professional 4.0 for Turbo Pascal contains numerous unit files that comprise a broad-ranging collection of more than 300 Pascal routines. These routines support long strings (up to 65,520 characters), random access text files, interrupt service routines, terminate-and-stay-resident programming, the use of extended and expanded memory, runtime error recovery, huge arrays (up to 32MB), automatic heap compression (when exiting to DOS or executing a child program), sorting, keyboard macros, BCD arithmetic, and, of course, screen handling. Full source code is supplied for all routines, including those in assembly language.

It appears that as much time was spent documenting Turbo Professional as was spent programming it. The spiral-bound manual exceeds 400 pages, and was written by someone with a clear and concise command of the English language and an obvious understanding of programming (this is a rare combination). Each chapter of the manual is dedicated to one Turbo Professional unit, with one or two routines on each page. Pascal routines for each unit are listed in alphabetical order—this is infinitely preferable to a canonical alphabetical listing of every routine in one huge section, because related routines can be found without the need to sift through quantities of irrelevant material. Each description contains a subprogram declaration, a statement of purpose, comments, cross references (“see also ...”), and—nearly always—an example. Frequently, examples of what not to do, as well as what to do, are presented. Several working example programs demonstrate the use of Turbo Professional’s units. Documentation also includes a complete index, plus an appendix that contains all unit dependencies.

Another plus for Turbo Professional is its overall approach. TurboPower did not assume that a programmer would buy this toolbox for the purpose of building a single application around it. As a result, most of Turbo Professional’s routines don’t force a programmer to make decisions about hardware configurations while writing the program. Instead, the routines let the program itself query the hardware to determine the best configuration at runtime.

As an example, consider Turbo Professional’s large arrays. Many library products force the programmer to determine the size of the array at compile time. This approach requires a least common denominator approach, where the array is sized to work on a computer that has the least amount of memory and the least number of

*continued on page 152*
hardware features. As long as the programmer is writing the program for one person on a known computer, this situation is perfectly tolerable. However, it's not tolerable for the person who writes general application software where the end user and the computer hardware are both unknown. Realizing this, Turbo Professional uses pointers and untyped variables to allow each routine to manipulate arrays of varying sizes.

In general, any Turbo Professional routine can be inserted into an existing application without the need to restructure the entire application to conform to Turbo Professional. Bravo!

It's impossible for a single person to thoroughly test the more than 300 Pascal procedures and functions in Turbo Professional. However, the routines themselves appear to be bug-free—during more than six months of use in programs ranging in length from 8000 to 15,000 lines, no bugs manifested themselves.

As with all good things, there are limits to what can be done with Turbo Professional 4.0. The screen-handling routines work only in text mode. Also, even though the manual claims that novice programmers with a passing knowledge of interrupt service routines and terminate-and-stay-resident programs can use the toolbox for writing TSRs and ISRs, there isn't really enough information in the manual to spare a novice some lengthily, and perhaps painful, trial and error experiences in these two areas. In addition, nothing indicates how much generated code a given Turbo Professional routine adds to your application (a programmer who knew the object code size for each procedure and function could assess the tradeoff between the power of a routine and the memory that the routine uses). Finally, Turbo Professional often takes a routine that would normally be a procedure and makes the routine return its own error code by declaring the routine as a function. This process often requires the use of dummy variable assignments or do-nothing program statements just to call the function, as in the following example from the manual:

```
if not
SaveWindow(1,1,CurrentWidth,
   Succ(CurrentHeight),
   False,P)
then
(can't fail;
 buffer already allocated);
```

These flaws are, at best, quibbles. Turbo Professional 4.0 is a well-executed Turbo Pascal toolbox with procedures and functions that are usable by programmers at all experience levels. The product is well thought out, reasonably priced, powerful, and immediately useful. If you're looking for a good toolbox for Turbo Pascal 4.0, I suggest you give Turbo Professional 4.0 some serious consideration.

—Rick Ryall

### 386MAX occupies only 3K of DOS memory, and relocates the bulk of its 58K of code to the high end of extended memory.

**386MAX**

Qualitas

8314 Thoreau Drive

Bethesda, MD 20817-3164

(301) 469-8848

$74.95

Like money, memory isn't everything—but it certainly makes many things easier. The key in either case is to make the most of what you have. While the 80386 CPU contains the machinery for putting memory wherever you need it, there's more to memory management than simply throwing addresses around. Qualitas' 386MAX is a utility designed to make the most of 386 memory management in as many different ways as possible.

386MAX began as a means to fill a hole. The first Intel 386AT motherboards had 512K of fast 32-bit real-mode RAM, and special slots for an additional 4MB of fast 32-bit extended RAM starting at the 1MB mark. However, no alternative existed between 512K and 640K other than to use slow, AT-style 16-bit RAM, which (at 16MHz) devours machine performance in a torrent of wait states the moment program execution wanders into it. 386MAX fills that 128K hole with fast 32-bit extended memory, using the 386's built-in memory management. Any empty space left in systems with MDA or CGA display adapters is also filled with DOS memory, up to the 704K mark. Furthermore, on systems that allow it, 386MAX fills the empty space between the high end of display memory and the low end of ROM with 32-bit RAM, and makes that RAM available to DOS as well.

These are the obvious tricks that can be played with 386 page remapping. 386MAX pulls quite a few others as well, such as the following:

- **386MAX** emulates EMS RAM (including LIM 4.0 functions), using 32-bit extended RAM;
- **386MAX** moves slow ROM-based BIOS code into fast 32-bit RAM (resulting in a 40 percent improvement in BIOS performance);
- On 16-bit systems equipped with 386 accelerator boards, **386MAX** swaps the slow 16-bit RAM on the motherboard with fast 32-bit extended RAM on the accelerator board;
- **386MAX** can move TSR utilities into high DOS memory between the display adapter and ROM, thus freeing up contiguous low DOS RAM for normal applications;
- **386MAX** can locate blocks of ROM, and time memory access for all types of memory throughout the system.
In essence, 386\textsuperscript{MAX} is "glue" for pulling a system together under the 386 and DOS. The product consists of a DOS device driver that contains the actual memory management machinery, plus a standalone utility that identifies blocks of memory, times memory performance, and loads TSRs into high memory. The driver occupies only 3K of low DOS memory, and relocates the bulk of its 58K of code to the high end of extended memory.

I've successfully used 386\textsuperscript{MAX} in a number of configurations in my system, which contains 512K on the motherboard and 2MB of fast 32-bit extended memory. Most frequently, I backfill the 128K "hole" mentioned earlier, and then divide extended memory into two portions. A 1200K section is treated as EMS memory that contains SideKick Plus overlays and a Turbo Pascal 5.0 edit buffer, and the balance is left as extended memory that contains a RAM disk for use with the Turbo Pascal compiler and Turbo Assembler. When working with graphics, I skip the RAM disk and use all of the extended memory for EMS, which is then divided between SideKick Plus and Tall Tree Systems' JLaser SA (Standalone). JLaser SA is a small board that allows a bimapped image stored in EMS memory to be converted into video. This video image is fed directly to the laser controller of a standard Canon-based laser printer. As a result, a full-page 300 dpi image prints from memory to paper in about 10 seconds.

The only important limitation of 386\textsuperscript{MAX} is due to the nature of the 386 itself. 386\textsuperscript{MAX} must be the memory management "boss"—and other bosses, such as Windows 386 or PC MOS/386, cannot peacefully coexist with 386\textsuperscript{MAX}. For the same reason, 386\textsuperscript{MAX} conflicts with Paradox 386. However, Qualitas and Quarterdeck have cooperated to allow 386\textsuperscript{MAX} to work with DESQview as a functional substitute for Quarterdeck's own QEMM memory manager. Qualitas states plainly that ill-behaved TSRs and programs that try to exploit the 386 may conflict with 386\textsuperscript{MAX}. In my experience, however, all important TSRs have functioned correctly, both in low memory and in high memory.

The documentation is terse, but unambiguous and complete. Any programmer who has worked successfully with a command-line compiler or assembler will be comfortable with the command-switch complexity of 386\textsuperscript{MAX}. The product does what it says it will do, and has not failed under my testing.

Much of the magic of the 386 remains dormant because of the lack of software to bring the magic into play. 386\textsuperscript{MAX} turns the magic loose—I recommend it highly.

—Jeff Duntemann

---

**CBTREE, the easiest to use, most flexible B+tree file manager for fast and reliable record access**

**CBTREE... Includes over 8,000 lines of ‘C’ source code FREE!**

Since 1984, thousands of ‘C’ programmers have benefited from using CBTree.

**Save programming time and effort.** You can develop your applications quickly and easily since CBTree’s interface is so simple. You’ll cut weeks off your development time. Use part or all of our complete sample programs to get your applications going FAST!

**Portable ‘C’ code.** The ‘C’ source code can be compiled with all popular C compilers for the IBM PC including Microsoft C, Quick C, Turbo C, Lattice C, Aztec C and others. Also works under Unix, Xenix, AmigaDos, U//trix, VAX/CMS, and others. CBTree includes record locking calls for multi-user and network applications. The CBTree object module is only 22K and is easy to link into your programs. You don’t even pay runtime fees or royalties on your CBTree applications!

**Reduce costs with system design flexibility.** CBTree allows unrestricted and unlimited designs. Reduce your development costs. You define your keys the way you want. Supports any number of keys, variable key lengths, concatenated keys, variable length data records, and data record size. Includes crash recovery utilities and more.

**Use the most efficient search techniques.** CBTree is a full function implementation of the industry standard B+ tree access method, providing the fastest keyed file access performance.

**Database Calls:**

- Get first
- Get last
- Get previous
- Get next
- Get less than
- Get less than or equal
- Get greater than
- Get greater than or equal
- Get sequential block
- Get partial key match
- Get all partial matches
- Get all keys and locations
- Insert key
- Insert key and record
- Delete key
- Delete key and record
- Change record location

**CBTree is only $159 plus shipping - a money-saving price!**

We provide free telephone support and an unconditional 90-day money back guarantee!

To order or for additional information on any of our products, call TOLL FREE 1-800-346-8038 or (703) 847-1743 or write using the address below.

**NEW CBTree add-on products to make your programming easier. AVAILABLE NOW!**

- Adhoc Query Reporting System
- SQL Interpreter

If you program in ‘C’, sooner or later you’re going to need a B+ tree. Don’t delay until you’re in a crunch, plan ahead, place your order for CBTree now. Orders shipped within 24 hours!

---

**PEACOCK SYSTEMS, INC.**

2108-C GALLOWS ROAD

VIENNA, VA 22180
Whether you’re a computer communications user or a programmer, if you want to learn more about computer communications, then this book is required reading. It’s really two books in one: An introduction to serial communications, and a communications guide for C programmers.

The first of the book’s two sections covers the basic topics that you need to understand before you attempt any serious serial communications programming. This introductory and background material, although somewhat technical, is clearly written and independent of any programming language. The discussion concentrates on defining serial communications, describing how it works, and providing insight as to why it often appears to be such an arcane field. This section contains no program code at all.

The second section gets down to the business of programming serial ports in C (along with a little help from assembly language), and includes program listings for IBM PC and Kaypro computers. The C code is standard enough to be widely applicable across hardware environments and C compilers. The ubiquitous Hayes modem (and compatible modems) is presented as the basis of the program interface to data communications hardware.

If you are not already a moderately experienced C programmer, you’re apt to have trouble with the material in this section. The author advises relative newcomers to C to gain appropriate experience first, and then come back to the book—this is good advice.

**COVERAGE**

As the title states, the book’s coverage is limited to serial communications, and further restricted to the asynchronous realm. Even those restrictions leave a lot of ground to cover.

Campbell presents an entertaining history lesson on the evolution of serial communications since the late 1800s. Although short in human terms, the history of communications is a lengthy one in the technological sense, and it’s interesting to see the effect of the legacy of mechanical contrivances upon today’s solid-state technology and terminology.

An examination of standards describes and explains the ASCII character set, including the codes that represent letters, numbers, punctuation marks, and control characters. The EIA RS-232 serial specification—the electrical, mechanical, and functional specification to which our computer’s serial ports and modems should adhere—is discussed as well.

The author categorizes ASCII characters into six sets: graphics characters; physical device control; logical communications control; physical communications control; information separators; and code extension controls. Characters in each category are fully described and clearly explained. Extensive use of tables and illustrations helps clarify and organize this standards information, which is far more accessible than it is in any of the standards documents themselves.

Campbell takes the time to explain the many conventions of serial communications, such as the uses of control codes. His book is one of only a small number of books that accurately define the BREAK signal and its purpose, and delve into the inner workings of the cyclical-redundancy check (CRC) method of error detection.

In his description of the infamous RS-232-C standard and its application, Campbell is careful to point out how the interface is reliable for its intended purpose, which is to connect data terminal equipment (DTE), such as terminals and computers, to data communication equipment (DCE), such as modems. The fog that surrounds the RS-232-C has been the result of poorly written standards...
documents, and the frequent application of the standards to situations far outside of their intended scope.

With all of the current interest in public-access bulletin-board systems and information utilities, file transfer protocols are a hot topic. Campbell presents technical and operational aspects of the Kermit and XMODEM file transfer protocols to show how both binary and ASCII text files can be easily moved from one computer system to another.

The subject of error detection and correction is an important one, and Campbell gives it a significant amount of coverage. He describes the use of simple parity, checksums, and CRC methods to detect errors.

Campbell approaches the universal asynchronous receiver/transmitter (UART)—the heart of the computer's serial port—from two directions. He first describes a virtual UART, which demonstrates all of the tasks that a UART must do in order to convert parallel data to an asynchronous serial form and vice versa. (These tasks are not trivial because of exacting timing considerations and other factors.) The process of designing a virtual UART helps programmers appreciate the benefits of using a packaged UART, and gives them an understanding of the complex programming requirements of a general-purpose UART such as the National 8250.

The final chapter of the book describes interrupts in the IBM PC family of computers and the Kaypro machine. This material shows how to implement interrupt-driven, rather than polled, communications programs. The inclusion of the Kaypro information provides an important comparison of serial communications in both the DOS and CP/M operating system environments. Because the Kaypro has no internal timing facilities, the programmer faces a much more difficult task in generating precise timing intervals, delays, and "tick" marks. The relatively small amount of assembly language code in this book is confined to low-level tasks such as timing functions, checking keyboard status, and other hardware-dependent functions.

**STRENGTHS AND WEAKNESSES**

The *C Programmer's Guide to Serial Communications* provides excellent, in-depth coverage of crucial topics of serial communications programming that are often neglected. This very readable book is a good blend of theory and practice, and is carefully crafted.

Physically, the book is both too big and too small. I would prefer to see this 650-page book divided into two separate volumes—each of the two sections is effectively a complete book in itself. (Well, almost—the second book could discuss such additional topics as background communications and RS-232 networks.)

Diskettes of source code are available from the author. The source requires the use of an assembler in addition to a C compiler. Some minor modifications to the C and assembler source files are needed in order for these files to work with certain compiler and memory-model combinations. Appendix C contains instructions continued on page 156.
to help you with the assembly language interface under PC-DOS/MS-DOS.

Joe Campbell's *C Programmer's Guide to Serial Communications* has taken a position on my reference shelf next to the dictionary, thesaurus, C compiler manuals, and other frequently used reference volumes that must be within three feet of my operating position.

Campbell calls serial communications programming "doing battle with the serial port"—this is an apt description of the process to those of us who have done it. This book gives you the tools and techniques that you need to have a fighting chance in that battle—a programmer who wanders off into serial communications without this handy guidebook is taking inordinately high risks. — Reid Collins

**FILE FORMATS FOR POPULAR PC SOFTWARE**


**MORE FILE FORMATS FOR POPULAR PC SOFTWARE**


As one who has written books, I sometimes review books with a bit of envy, wishing that I had been given the contract instead, and wondering how I could have made the book better. But the writing of these two books is a job I wouldn't wish for. *File Formats for Popular PC Software and More File Formats for Popular PC Software* present file formats that are used by popular PC application programs. This is hardly a subject to stir the blood, but it's required reading for anyone who programs a PC for a living.

Each spiral-bound book has sturdy, glossy stock covers. With each book you get very little snappy patter, no tutorial information, and three parts. The first part of each volume describes each file format in excruciating detail. No-nonsense tables present information about byte offsets, contents, and their purpose. While a few comments have been added, explanations are kept to a minimum. A notable exception is the Framework II format that is presented in *More File Formats*—with 68 pages of description, it's easily the most complicated file described in either of the two books. The second section of each book contains dumps of sample files with expanded control characters and added offsets (presented in what author Jeff Walden terms "music staff" style). The final part of each volume consists of the source listing of the Turbo Pascal program that produced the dumps.

The first book contains the file formats for Lotus 1-2-3, Symphony, Ability, dBASE II and III, DIF, Multimate, Microsoft Multiplan (SYLK), SuperCalc 3, VisiCalc, WordStar, and WordStar 2000. The second book contains formats for Framework II, Reflex, Microsoft Rich Text Format, SuperCalc 4, SuperProject Plus, Volkswriter, and WordPerfect. Most of the formats were obtained with the direct cooperation of the various vendors, so the information can be assumed to be accurate. The author scrupulously identifies the versions of the software that correspond to the file formats described in the text. As new versions of software are released, we can only hope that these books will keep up with the leading edge.

The value of this file format information cannot be overstated. Just glancing at the layout for Lotus 1-2-3 .WKS files, for example, is enough to convince me that there's no way I'd reverse-engineer it even if I had a river of Jolt Cola and the fastest 386 system my employer could buy. But thanks to Mr. Walden's efforts and Lotus' cooperation, the necessary information is all there in the book. Although the fiendishly intricate structure of the .WKS files requires fancy coding in order to create a routine to read or write Lotus files, you now have a fighting chance.

Not only is the information in these books valuable, it's useful in a practical sense. If you develop software programs for a large corporation that uses one or more of the programs discussed by Mr. Walden, you will need to read or write at least one of these files eventually. While most major applications perform ASCII file translation, the ability to read and write files in the files' native format offers a cleaner and more direct way to interface to those files. In addition, some programs, such as WordPerfect, support more features in native file format than in ASCII format, so direct access becomes a requirement.

In short, since the ability to exchange data between applications is an important consideration of PC programming these days, the file layout information provided in these two books is a necessity. But don't plan on buying these books, reading them just once, and then coming away from the experience with detailed knowledge about the internal workings of the covered products. These are reference books—you will need knowledge of both a specific application and a programming language in order to make good use of the books' information. While programming to proprietary file formats isn't easy, Mr. Walden's two indispensable books make it at least possible. — Marty Franz
**COMPUSERVE**

The best online information about the Borland languages can be found on CompuServe's three Borland forums. Quite apart from providing the listings appearing in TURBO TECHNIX, the Borland forums contain many megabytes of useful utilities and source code in all Borland languages. Furthermore, some of the most interesting and knowledgeable people in the programming subculture hang out on CompuServe, providing an informal, online user group that is always in session. If you have a question, leave a message in the appropriate forum, and in almost every case someone will jump in with an answer.

Subscribing to CompuServe can be done through the coupon enclosed with every Borland product (which also includes $15 worth of online time for your first month) or by calling CompuServe at (800) 848-8199. You'll need a modem and some sort of communications software that supports the XMODEM file transfer protocol.

**How to access the Borland Forums on CompuServe:**

**TURBO TECHNIX** listings for Turbo Pascal and Turbo Basic are available in Library 1 of the BPROGA Borland Programming Forum (GO BPROGA).

Turbo C, Turbo Prolog, and Turbo Assembler listings are stored in Library 1 of the BPROGB Forum (GO BPROGB). Listings for Business Language articles are also available in Library 1 of the Borland Applications Forum (GO BORAPP).

From the initial CompuServe prompt, type **GO <forum name>** or follow the menus. If you're not already a member of a forum, you must join by following the menus before you can download the listing files.

**How to download TURBO TECHNIX code listings from CompuServe:**

At the forum menu, type: Library 1. This will take you to the TURBO TECHNIX data library, where all listing files are stored. Listing files are archived using the ARC32 archiving scheme. You will need the ARC-E.COM program (available in Library 0 of BPROGA, BPROGB, and BORAPP) or one compatible with it to extract listing files from downloaded archives.

Magazine archive files are organized two ways: by article and by issue. In other words, there will be one .ARC file for every article that includes listings; and a single, larger .ARC file for each issue that contains all of the individual .ARC files for that issue. You can therefore download listings for individual articles, or download the entire issue's listings in one operation.

The all-issue files follow a naming convention such that NVDC87.ARC contains all listing archives from the November/December, 1987 issue; JNFB88.ARC contains the listings from the January/February, 1988 issue; and so on. The name of an article's individual listings archive file is given at the end of the article.

To download an archive file, bring up the Library 1 prompt and type: **DOW <filename>/PROTO: XMO**

After pressing Enter, start your own communications program's XMODEM receive function. After you have completely received the file, you must press Enter once to inform CompuServe that the download has been completed. Once you have downloaded an archive file, you can "extract" its component files by invoking ARC-E.COM at the DOS prompt with:

**C>ARC-E <filename>**

---

**CHANGE OF ADDRESS**

If you've moved or changed your name since you began receiving TURBO TECHNIX, please let us know so we can make sure your copies go to the right person in the right place. Send us a letter providing both your old and your new name and address, and attach an existing mailing label from TURBO TECHNIX if possible. Send the letter to:

**TURBO TECHNIX**


1800 Green Hills Road

Scotts Valley, CA 95066-0001

---

**ONLINE AND BETWEEN COVERS**

The following information describes two sources where you can learn more about Borland language products: On the Borland CompuServe forums, and in books now or soon to be in print. This issue, the CompuServe highlights are from the Turbo Pascal/Turbo Basic Forum, BPROGA. The files shown in this section are not related to articles in TURBO TECHNIX, but are of general interest to Turbo programmers. All files for Turbo Basic are stored in Library 9; all Turbo Pascal files are stored in Library 2. The books presented here are only a sampling. (We can't possibly list all published Borland-related books. Also, this listing reflects no judgment about the quality of any book.) For more information on these and other Borland-related books, contact the publishers or your local bookstore.

**TURBO PASCAL: (Library 2)**

**PIBMD0.ARC**

Uploaded: 6/6/88

Size: 20,935 bytes

This archive contains routines for interfacing Turbo Pascal 4.0 programs to several popular DOS-based multitaskers: TaskView, OmniView, DESQview, DoubleDos, and TopView.

continued on page 158
This program illustrates how to change the values of typed constants within a Turbo Pascal 4.0 program. It is a common technique that can be used to create installable software.

EGASAV.ARC
Size: 3,072 bytes
This archive contains a unit that interfaces two routines for saving and restoring EGA (640 x 350) 16-color graphics screens to and from RAM. Provides a good example of accessing and programming the EGA's internal registers.

AUTO12.ARC
Size: 10,112 bytes
TPHRT is a high-resolution timer/profiler for Turbo Pascal 4.0. The user places simple calls to TPHRT routines in the source code under study, compiles and runs the code, and a complete report of all TPHRT timer activity is generated. Up to 100 different timers may be active. Resolution is one microsecond and is self-calibrating.

STDERR.ARC
Size: 1,662 bytes
This archive contains a unit that in one microsecond and is self-calibrating.

NWDMO2.ARC
Size: 7,964 bytes
Latest demo in source code of window routines. This file demonstrates some windowing capabilities of the Turbo Basic Toolboxes. The .EXE file demonstrates the windowing capabilities of a Toolbox for those who don't own one. The source code is in the file NWDMO2.ARC for the additional routines that call the Toolbox routines.

VARSTR.ARC
Size: 1,408 bytes
This file demonstrates how you can find the address at which Turbo Basic has stored your strings.

CPLARC
Size: 19,072 bytes
CPI (Communication Program Interface) is a TSR device driver for buffered data input, baud-rate selection from 75 to 115,200 baud, background communication, signing on data input, and more. Easy to use with any language or directly from DOS without programming the comm chip.

TURBO C BOOKS

* Turbo C for Beginners; Steve Burnap; Compute! Books
* Turbo C: The Essentials of Programming; Ira Pohl/Al Kelly; Benjamin/Cummings

Using Turbo C, Herbert Schildt; Osborne/McGraw-Hill
Advanced Turbo C, Herbert Schildt; Osborne/McGraw-Hill
Turbo C: Memory Resident Utilities, Sreen I/O and Programming Techniques; Al Stephens; MIS Press
Turbo C, The Art of Program Design, Optimization and Debugging; Stephen Randi Davis; M&T Books
Turbo C Programmer's Library; Kris Jamsa; Osborne/McGraw-Hill
Turbo C: The Complete Reference, Stephen O'Brien; Osborne/McGraw-Hill
The Waite Group's Turbo C Bible; Naba Barkabati; Howard W. Sams & Co.
Turbo C Programming for the IBM; Robert LaFore; Howard W. Sams & Co.

Complete Turbo C; Strawberry Software; Scott, Foresman & Co.
Programming with Turbo C; Beverley and Scott Zimmerman; Scott, Foresman & Co.
Mastering Turbo C; Stan Kelly-Bootle; Sybex Inc.
Systems Programming in Turbo C; Michael Young; Sybex Inc.
Turbo C Programmer's Resource Book; Frederick Hultz; Tab Books, Inc.
Turbo C DOS Utilities; Robert Alonso; John Wiley & Sons, Inc.
Turbo C Survival Guide; Larry Miller/Alex Quilici; John Wiley & Sons, Inc.
Turbo C At Any Speed; Richard Wiener; John Wiley & Sons, Inc.

TUG
The national user group for Turbo languages is TUG, the Turbo User Group. TUG publishes a bimonthly journal called Tug Lines that contains bug reports, programming how-to's, and product reviews. Extensive public-domain utility and source code libraries are available to members. An optional multi-user BBS with file uploading/downloading, messaging, and teleconferencing is available to the public. Membership dues are $24.00 US/year (including Washington State); $28.00 Canada and Mexico; $39.00 overseas.

TUG
P.O. Box 1510
Poulsbo, WA 98370
BBS: (206) 679-1151

LOCAL USER GROUPS
One of the best places to look for advice and face-to-face assistance with your programming problems is at a local user group meeting. Most user groups in the larger cities have special interest groups (SIGs) devoted to the most popular programming languages, usually with strong Turbo presence. We will be listing some of the largest and most active user groups in major urban areas across the country; obviously, there are thousands of user groups that we cannot list due to space limitations. If no listed group is convenient to you, ask about local user groups at a local computer store or check with a faculty member at a high school or college with a computer curriculum.
<table>
<thead>
<tr>
<th>ADVERTISERS’ INDEX</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aker Corp.</td>
<td>115</td>
</tr>
<tr>
<td>American Cybernetics</td>
<td>43</td>
</tr>
<tr>
<td>ASCII</td>
<td>159</td>
</tr>
<tr>
<td>Austin Code Works, The</td>
<td>47</td>
</tr>
<tr>
<td>Black &amp; White International</td>
<td>37</td>
</tr>
<tr>
<td>Blaise Computing</td>
<td>5</td>
</tr>
<tr>
<td>Borland International, Inc.</td>
<td>29-32,</td>
</tr>
<tr>
<td></td>
<td>44-45, 89, 93, 97, 109, 116-117, 131-135</td>
</tr>
<tr>
<td>Burgiss Group, The</td>
<td>127</td>
</tr>
<tr>
<td>Chen &amp; Associates, Inc.</td>
<td>159</td>
</tr>
<tr>
<td>CHANCElogic</td>
<td>51</td>
</tr>
<tr>
<td>Computer Solutions</td>
<td>159</td>
</tr>
<tr>
<td>Disk Software</td>
<td>36</td>
</tr>
<tr>
<td>English Knowledge Systems, Inc.</td>
<td>159</td>
</tr>
<tr>
<td>Entelekon Software Systems</td>
<td>61</td>
</tr>
<tr>
<td>Ithaca Street Software, Inc.</td>
<td>159</td>
</tr>
<tr>
<td>Lahey Computer Systems, Inc.</td>
<td>39</td>
</tr>
<tr>
<td>Matrix Software</td>
<td>25</td>
</tr>
<tr>
<td>Max Software Consultants, Inc.</td>
<td>147</td>
</tr>
<tr>
<td>Microway</td>
<td>41</td>
</tr>
<tr>
<td>Nostradamus</td>
<td>23, C3</td>
</tr>
</tbody>
</table>

C:>CLASS.ADS

C:>CLASS.ADS is TURBO TECHNOLOGY magazine’s display classified advertising section. Special sizes and rates are available for C:>CLASS.ADS—$150 per column inch, with a 2-inch minimum. (A minimum ad, for example, measures exactly 2 1/16” wide by 2” long.) All C:>CLASS.ADS must be prepaid and submitted in camera-ready form (black and white PMT or Velox) to:

C:>CLASS.ADS TURBO TECHNOLOGY 1800 Green Hills Road P.O. Box 660001 Scotts Valley, CA 95066-0001

For information, please contact the Advertising Department at (408) 438-9921.

T Pascal, Turbo C Microsoft C Complete data base code in just 10 minutes!

Draw & paint your screens, point out indexes & that’s it! Generator has B-tree file manager, Automatic indexing, Context sensitive help, Automatic programmer documentation. Unlimited technical support

T Pascal $389 / C Versions: $499

30 day money-back guarantee

Turbo Programmer ASCII - (800) 227-7681

<table>
<thead>
<tr>
<th>OPT-TECH SORT</th>
<th>THE HIGH PERFORMANCE SORT/MERGE UTILITY</th>
<th>USE STAND-ALONE OR CALL AS A SUBROUTINE. UNLIMITED FILE SIZE, MULTIPLE KEYS, RECORD SELECTION &amp; MUCH MORE!</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICON-TOOLS™</td>
<td>ICON EDITOR, ICON FILES, FUNCTION SOURCE CODE, TURBO C, QUICK C, METAL/IND, ESL, AND GFX GRAPHICS</td>
<td>$69.95</td>
</tr>
<tr>
<td>TURBO SOFTWARE</td>
<td>WE HAVE A LARGE COLLECTION OF THE BEST SHAREWARE &amp; PUBLIC DOMAIN FOR THE TURBO LANGUAGES!</td>
<td></td>
</tr>
</tbody>
</table>
Academic types will tell you that with proper design techniques, it's possible to write programs that are correct from the word GO. In the real world, with real programs that do real work, it never works out that way. In fact, if a program under development never crashes, it can't be much of a program!

Even the best jugglers drop some balls when they work on new tricks! As a matter of fact, it is well known that the best jugglers are probably the ones that have dropped a lot of balls. There's nothing wrong with crashing during development. What's wrong is crashing and not understanding why.

**Error Is Human, But...**

In 44 B.C. (Before Computers), Cicero said: "Any man can make mistakes, but only an idiot persists in his error." In Latin, it became more radical over time: "Errare humanum est, sed perseverare diabolicum: Error is human, but repeating it is an act of the devil!"

Of course, you have to spend time and energy in careful software design, or the product will never even be finished, much less work. The primary factor is the project's overall architecture. That's the foundation on which work will be done. It has to be solid and very well thought out. Then you start with a design, and I guarantee you that there will be some major changes to that design by the time the product ships. It's natural. Otherwise, you can bet that it won't be much of a product. Too many people show more interest in a quick profit than in quality work. Products are rushed to market even though the programs have more bugs than a tropical island.

But even with the best design humanly possible, there will be bugs. There will be crashes. The key is to have good people using debugging tools that they understand, so that even the worst crash is a learning experience.

**The Right Tools**

Your tools should match your problem. You can't debug a leading-edge program with a trailing-edge debugger. If you're going to program for the 386, your debugger had better understand the 386. If your program is going to use EMS, your debugger had better understand EMS. Otherwise, you're over-driving your headlights...

Turbo Debugger helps you to see what's going on. Look at everything. If your program brings the system down, look carefully at all the side effects as well. That can be quite an education right there: You can learn more about DOS when it goes down in flames than you can when it works! But you don't learn anything unless you can watch what happens right up to the explosion.

It helps to have a safe place to watch from. Powering down is a waste of time. That's why at Borland we built 386 support into Turbo Debugger. We put Turbo Debugger in one protected virtual-86 partition and left the test program in another, so that even if the test program crashes its partition, nothing touches Turbo Debugger.

And then there are the dangerous ideas you get while implementing a cutting-edge design. They may work fine, or they may blow you away every time, but you won't know until you try, and with Turbo Debugger you can make an informed decision. Without Turbo Debugger, you can only do what's safe. With Turbo Debugger, you can explore new territory and make it safe.

**Watch What Goes Wrong**

This is the importance of debugging: To watch what goes wrong so that you can not only fix that bug but recognize that whole class of bugs. Little by little, you fine-tune your design so that it becomes crashproof.

It's OK to crash. A program can die a thousand deaths, and come back every time. And each time it'll be a little better, if you really work at learning from your mistakes. Remember, someone who never makes a mistake doesn't usually make anything, and like the ancient Chinese saying goes: "The first thousand times don't count!" ■
Turbo-Plus 5.0

$99.95

SOFTWARE AHEAD OF ITS TIME

Turbo-Plus™ Features:
- Screen Painter
- Unit Libraries
- Assembler Efficiency
- I/O Code Generation
- Window Code Generation
- Window Management
- Special Effects
- Screen Support
- System Integration
- User Color Selection
- Run-Time Dynamic Menus
- Universal Menus
- Window & Menu Compression
- Transparencies and Shadows
- Keyboard Support
- Cursor Support
- Field I/O Routines
- Reentrant Routines
- Diagnostic Tools
- File Handling
- System Resources
- Sound Effects
- Critical Error Handlers
- Automatic Directories
- Sample Programs
- Complete Pop-Up Help
- 280 Page Illustrated Manual

The Language Standard is TURBO PASCAL 4.0
The Enhancement Standard is TURBO PLUS 5.0

"Turbo Plus 5.0 gives every Program the professional touch . . . saves hours of coding. A must in my programming."

Mike Cushman • Former Editor, "", PC World

"After spending hundreds upon hundreds of dollars searching through many utilities and libraries, I must say that Nostradamus is my choice!"

Mr. Paul Mayer • ZPAY Payroll Systems • Franklin Park, IL

"I've tried most similar products on the market, Turbo-Plus with Screen Genie is clearly superior."

Dr. David Williamson • Chiropractic Health Services • Durnam, NC

"This is, without a doubt, the most powerful and easy to use programming toolbox that I have seen for the PC environment, and Turbo Pascal in particular."

Mr. John Drabik • Geotron International, Inc. • Salt Lake City, UT

"Your products are first rate. Your Turbo-Plus products give my humble efforts a touch of class and speed that I would never have achieved otherwise. Obviously your products make Turbo Pascal a much better product."

Mr. L.M. Johnson • Saguaro Technical Services • Cave Creek, AZ
SPECIAL TURBO SALE
Get $5.00 Off Every Turbo Pascal^4 Book
Get $3.00 Off Every Turbo C^ & Turbo Basic Book

Using Turbo C^ by Herbert Schildt
For all C programmers, beginners to pros, this excellent guide helps you write Turbo C programs that get professional results.
Borland-Osborne/McGraw-Hill Programming Series

Advanced Turbo C^ by Herbert Schildt
Unveils Turbo C power programming techniques to serious programmers. Covers Turbo Pascal conversion to Turbo C and Turbo C graphics.
$22.95 Paperback, ISBN: 0-07-881280-1, 397 pp., 7 3/4 x 9 1/4
Borland-Osborne/McGraw-Hill Programming Series

Turbo C^: THE COMPLETE REFERENCE
By Herbert Schildt
Covers Version 1.5
Programmers at every level of Turbo C expertise can quickly locate information on Turbo C functions, commands, codes, and applications—all in this handy encyclopedia.
$24.95 Paperback, ISBN: 0-07-881346-9, 850 pp., 7 3/4 x 9 1/4
Borland-Osborne/McGraw-Hill Programming Series
$21.95

Turbo Pascal^ VERSION 4
by Steve Wood
Build the skills you need to become a productive Turbo Pascal 4 programmer. Covers beginning concepts to full-scale applications.
Borland-Osborne/McGraw-Hill Programming Series

Advanced Turbo Pascal^ VERSION 4
by Herbert Schildt
The power of Turbo Pascal 4 will be at your fingertips when you learn the top-performance techniques from expert Herb Schildt.
Borland-Osborne/McGraw-Hill Programming Series
$16.95

Turbo Pascal^ PROGRAMMER'S LIBRARY, SECOND EDITION
by Kris Jamsa and Steven Nameroff
Take full advantage of Turbo Pascal, and the newest versions of Turbo Pascal, with this outstanding collection of programming routines. Includes routines for the Turbo Pascal toolboxes.
$24.95 Paperback, ISBN: 0-07-881368-9, 600 pp., 7 3/4 x 9 1/4
Borland-Osborne/McGraw-Hill Programming Series
$17.95

Using Turbo Basic^ by Frederick E. Mosher and David I. Schneider
Introduces Turbo Basic to novices and seasoned pros alike. Learn about the Turbo Basic operating environment and the interactive editor.
Borland-Osborne/McGraw-Hill Programming Series
$16.95

For A Limited Time Only
ORDER TODAY! CALL TOLL-FREE 800-227-0900 Use Your Visa, MasterCard, or American Express

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