COVER SUITE: DIAGNOSING PROBLEM CODE

Turbo Debugging

Borland has added the missing ingredient to its product line with Turbo Debugger. It challenges Microsoft CodeView's preeminence among high-level language debuggers.

BEN MYERS

A generation of PC software developers cut their teeth on the venerable DOS DEBUG. Initially, it was the only debugging tool around, but the market soon exploded.

Today, a wide array of debuggers is available. The most significant recent development in the market is Borland's Turbo Debugger, which could attract users of not only Borland's languages, but also Microsoft's languages.

Since the early days of DEBUG, the debugger market has evolved into five broad, nonexclusive classes: assembly-language, symbolic, high-level language (into which Turbo Debugger falls), hardware-assisted, and in-circuit emulator (ICE) debuggers.

Assembly-language debuggers, of which DEBUG is the most famous and widely used example, work solely with machine code. Software developers writing in high-level languages cannot easily correlate source code with compiled assembly code. They must resort to tricks such as including program statements to write out values of critical variables during program execution.

With more complex applications, developers demand more sophisticated tools. Symbolic debuggers improve on assembly-level debuggers by using linker .MAP files or symbolic information embedded in the object-code file to relate assembly-level addresses to variables or functions. Disassembled code shows variable and function names, rather than hexadecimal operands. Microsoft's SYMDEB, introduced in 1985, is a symbolic debugger.

High-level language debuggers go one step further than symbolic debuggers by simultaneously displaying source and assembly code so that the logical relationship between them is readily apparent. The machine code is disassembled in clear relation to variables and functions, and developers can debug individual high-level language statements.

Microsoft's CodeView, introduced in late 1986, is the most popular high-level language debugger—and the one with which Turbo Debugger will compete most directly. Though primarily menu and window oriented, CodeView is downwardly compatible with the older command-line oriented debuggers, DEBUG and SYMDEB . (For a review of CodeView, see "Multilevel Debugger," Mark S. Ackerman, March 1987, p. 90.)

Rounding out the list are the hardware-assisted debuggers and the ICEs. These products are the most sophisticated and expensive PC debuggers.

Hardware-assisted debuggers typically control and monitor the debugging process with an expansion board installed on the bus. They are more flexible than their software counterparts because their hardware breakpoints do not affect the speed of a program; furthermore, they can run programs in protected memory, which can prevent the need to reboot when a piece of code hangs the system. Moreover, the software developer can usually break out of a feisty program with a break button. ICEs have the same features as hardware-assisted debuggers but are more expensive because they consist of outboard hardware that replaces the PC's processor.

MICROSCOPE COURTESY OF OPTICAL SYSTEMS DIVISION OF CAMBRIDGE INSTRUMENTS

PHOTOGRAPHY • BRUCE WELLER

TURBO DEBUGGER

The second article of this month's cover suite ("Hardware Assistance," Marty Franz, p. 58) examines hardware debuggers and looks at two in detail: Atron's 386 Source Probe and The Periscope Company's Periscope III.

TURBO ERGONOMICS

In August 1988, Borland International announced its high-level language debugging environment, Turbo Debugger 1.0, which began shipping in late September. Borland simultaneously introduced Turbo Pascal 5.0, Turbo C 2.0, and a package containing both Turbo Assembler and Turbo Debugger. In addition to other new features, both Pascal and C now have an integrated source-level debugger. Borland also offers two packages—Turbo Pascal Professional and Turbo C Professional—that bundle Turbo Assembler and Turbo Debugger with Turbo Pascal 5.0 and Turbo C 2.0, respectively.

Borland paid considerable attention to human factors when it designed Turbo Debugger's interface. The package is entirely menu driven—from the installation and customization programs to the debugger itself. By contrast, CodeView is a more cumbersome hybrid that uses both commands and pull-down menus.

Turbo Debugger has more than 200 hot keys and main- and local-menu commands. Despite this intimidating number, developers can easily navigate the program through the windows interface and its pop-up data-entry and selection boxes.

The main screen has a menu bar across the top to access primary functions, such as working with files, setting breakpoints, and running programs. The bottom line displays context-sensitive, function-key actions that vary for each window. When the user presses the Alt key, the bottom line shows additional options available using Alt with various key combinations. Pressing the Ctrl key shows the *local commands*, initiated by Ctrl combined with letter keys. Borland calls these commands local because they initiate actions in the current window.

Borland extends the window metaphor further with *panes*, which are logical subdivisions within a window. The tab key allows movement from pane to pane within any window. Progress

HELP FROM THE HARDWARE

All 80*x*86 processors have two features that help developers implement debuggers: the breakpoint instruction and single-step execution. The breakpoint instruction, INT 3, has a one-byte operand code (0CCH), not the two-byte form used by other interrupt instructions. Because an INT 3 is one byte long, it can replace the first byte of an instruction without corrupting subsequent ones (see Tech Notebook, this issue, p. 121).

When a program executes INT 3, the CPU transfers control to the breakpoint interrupt vector at low-memory location 0CH; the debugger will have previously set this vector to a debugger entry point. The CPU saves the code segment (CS) and the instruction pointer (IP) flag registers on the stack, with the IP pointing just past the INT 3 instruction.

To set an unconditional breakpoint, the debugger saves the breakpoint address and the byte of code at that address, then inserts an INT 3 there. When an INT 3 occurs, the debugger takes control and puts back the first byte of the instruction. Then, the user can issue commands to inspect or change program variables.

To resume execution at breakpoint, the debugger subtracts a value of one from the IP on the stack and does an IRET to execute the original instruction. The program runs under its own control until the next breakpoint occurs. Because the debugger executes only when a breakpoint is reached, the mechanism for unconditional breakpoints does not degrade program execution time significantly.

The second feature that aids tracing on Intel processors is single-step execution. The debugger enables this feature by setting the trap flag (TF) in the flags register. Whenever the TF is set, the processor transfers control to the address in the INT 1 vector (at location 4 in low memory) after executing each instruction. Single-step execution is slower than normal execution, often by a factor of 100.

Implementing conditional breakpoints using only these two features is laborious. If the user wants to break program execution based on the change of a variable in memory, the debugger must execute the program entirely in single-step mode. With each single-step interrupt, the program checks the variable being monitored for a change, and, if no change has occurred, executes the next instruction.

For conditional breakpoints set at source level, a well-designed debugger can run somewhat faster by inserting breakpoints at the first assembly instruction generated for each line of source code. A breakpoint-handling procedure tests the variable and continues program execution if no change is found. Conditional breakpoints based on the value of the expression are handled in much the same way. When a debugging interrupt occurs, the debugger evaluates the parsed expression to see if it is true and acts accordingly.

Additional debugging help comes from the 386 processor's ability to set four hardware-monitored breakpoint addresses. Debugging software can set a breakpoint for instruction execution, data writes, or data reads and writes; data breakpoints can be one, two, or four bytes wide.

Unlike the INT 3 instruction, 386 breakpoints do not modify the code to set an instruction breakpoint, and they can be set on data accesses. Breakpoint interrupts for this feature occur on the INT 1 vector; the single-step interrupt is still supported by the 386 on this vector, so the CPU provides a status register to give the reason for the interrupt.

Although the 386 debug registers are a welcome addition to the tools for a debugger, they still leave a great deal of work for the debugger designer. This is especially true in multitasking environments such as Unix and OS/2. For example, the debug registers are not stored in the task-state segment; therefore, the software must explicitly save and restore them when more than one task is being debugged.

A major element in the design of a 386 debugger is how to make best use of the four sets of debug registers. Watching data entirely with software is many times slower than with the 386 debug registers. Thus, the design of a 386 debugger favors watching data through the debug registers whenever possible. If the debugger allows more than four concurrent breakpoints, a combination of INT 3 and single-step interrupt handlers must suffice for the excess that cannot be accommodated by the 386 debug registers.

—Ben Myers

through windows already opened is even faster using *history lists*, which store the last sequence of choices made in the current session. As a short cut, pressing the Alt key and the window number selects an open window.

DEBUGGING ENVIRONMENTS

Unlike CodeView, Turbo Debugger has built-in support for a variety of hardware. It supports four different debugging environments: 8086 mode, 386 virtual mode, remote, and hardware-assisted debugging. In all modes, the Turbo Debugger user can step through statement execution, with or without dropping down into function calls, at both source-code and assembly levels.

To use the virtual-memory features when debugging on a 386, the user must install the TDH386.SYS driver in CONFIG.SYS and then run the 386 virtual debugger, TD386. TD386 runs entirely in extended memory, allowing the program being debugged (target program) to load and run at the virtual memory address it would use in actual conditions. The 386 device driver permits the developer to set hardware breakpoints for instruction fetches, memory reads, and read/write memory accesses at specified addresses (see the sidebar at left, "Help from the Hardware").

The target program can use 80286 and most 386 instructions, except for those that operate in protected mode. A toggle changes the display between the 32-bit 386 extended registers or 16-bit registers.

The virtual-8086 mode and memory management make the system practically immune to crashes, even if the target program destroys memory contents within its own address space. These same features, however, slow the program down and cause timing problems for some applications.

When a 386-based PC is not available or memory is at a premium, Turbo Debugger's remote debugging interface program, TDREMOTE, is a useful alternative. Turbo Debugger runs on the first machine, and TDREMOTE and the target program run on the second, effectively insulating the debugger from crashes. TDREMOTE requires only 15KB of memory, permitting the developer to debug large programs (see table 1 for a comparison of Turbo Debugger and CodeView target-program sizes). Setting up to debug a program in a remote PC is a relatively simple process. TDREMOTE and Turbo Debugger communicate through serial ports interconnected by a null modem cable.

TABLE 1: ***Turbo Debugger and CodeView Comparison***

	BORLAND	MICROSOFT
PRODUCT	Turbo Debugger	CodeView
VERSION	1.0	2.2
DEBUGGING PROGRAMS		
Compiled with /Zi (TDCONVRT)	●	●
With MAP files (TDMAP)	●	○
Supports other languages with .MAP files	●	○
OS/2	○	●[a]
Microsoft Windows	○	●
Turbo C 2.0 and Turbo Pascal 5.0	●	○
OTHER DEBUGGING FEATURES		
Views 32-bit 80386 registers	●	●
Uses 80386 hardware debugger features	●	○
Documented hardware debugger support	●	○[b]
Has remote debugging	●	○
Uses command macros or files	Macros[c]	Files
Logs output	●	●[d]
Views data structures in source format	●	○
Integrates use of mouse for commands	○	●
Uses expanded memory	●	●
Reports own and target's EMS usage	●	○
Searches for instructions in assembly code	●	○
MAXIMUM SIZE OF PROGRAM AND DOS (KB)		
8088, 8086, or 80286	411	410
80386	640	410
Remote PC	625	N/A

● = *Yes* ○ = *No* *N/A = Not applicable*

[a] *CodeView for Windows applications is available with Windows 2.1 Software Development Kit.*
[b] *The CodeView file format is available to developers who enter into a contractual agreement with Microsoft.*
[c] *Command macros cannot be saved between debugging sessions.*
[d] *Screen output is redirected to a file and is not visible to the user.*

Turbo Debugger offers users of Borland languages the functionality of Microsoft's CodeView, with many added features including Borland's use of the 386's memory-management and debugging features and innovative remote debugging.

TDRF, a program running the machine used for the debugger, supports file transfer between Turbo Debugger and the target PC. TDRF transfers data at 9,600, 40,000, or 115,000 bits per second (bps) and permits normal DOS functions such as file deletion and renaming, directory creation, deletion, and listing on the remote computer.

Borland provides documentation for writing device drivers for interfacing Turbo Debugger to hardware debuggers so that memory- and I/O-access breakpoints produced by the board are handled by Turbo Debugger. No vendors are yet shipping a compatible hardware driver, although several, including Atron and Periscope, are considering such a product. Borland says the interface eventually will support instruction trace-back and extra onboard memory for a symbol table.

If vendors of hardware debuggers provide their own drivers to meet the Borland debugger driver specification, the promise of well-integrated hardware and software debugging will be realized. Several hardware debugger manufacturers support CodeView format files under license agreements with Microsoft.

Turbo Debugger can use EMS 3.2 or 4.0 to store its symbol tables, and it keeps track of expanded memory used by the program being debugged. Expanded memory is used for symbol tables only; no executable code is loaded in EMS. On 386-based PCs, Turbo Debugger runs the target program in 1MB of virtual 8086 space (640KB of program space) and uses the 386 memory-protection hardware to keep the target program from contaminating the rest of the environment. Through 386 microprocessor capabilities, Borland has given the developer much of the functionality of a hardware-assisted debugger. CodeView users can get 386 functionality with MagicCV, an add-in product from Nu-Mega.

While Turbo Debugger supports Borland's language compilers and assembler, developers also can debug programs written in Microsoft languages at the source level, using the TDCONVRT facility included with the debugger. The program converts Microsoft .EXE files compiled for CodeView into the Borland .EXE format.

Turbo Debugger works with most compilers and linkers that produce detailed .MAP files by using the included TDMAP utility to append Turbo Debugger information to the .EXE file. Turbo Debugger, however, operates only under DOS, not OS/2, and cannot debug programs compiled for the Microsoft Windows environment. Microsoft, on the other hand, has added OS/2-specific features to CodeView (see the sidebar, "CodeView Under OS/2: Nice Threads").

GETTING UNDERWAY

Installation of the three Turbo Debugger diskettes takes only a few minutes with the menu-driven INSTALL program, which automatically unpacks compressed files. The TDINST program controls several customization parameters; the user can change many while Turbo Debugger is running.

Customization options include choosing window colors; specifying editor, source, and debugger directories; enabling remote debugging and use of EMS; selecting the language syntax used for expression evaluation; and selecting how display video pages are managed between the debugger and the user screen.

TDINST has several options for handling screen swapping between the debugger and the executing program's (user) display screen image. If the system has multiple display pages, as in the CGA, EGA, or VGA, the Turbo Debugger screen will be maintained on a separate display page. The user can also swap screens in software—a slower, but less disruptive method.

A second monitor can display Turbo Debugger while the first displays the user screen. TDINST permits the user to update screens continuously, when a change occurs, or not at all.

The commands TD or TD386 followed by command-line options and the target program name starts Turbo Debugger. Commands follow either Unix style (preceded by a hyphen) or the DOS convention (preceded by a forward slash). An −h or −? displays all command-line options.

Source-level debugging requires programs compiled or assembled with the following options. For Borland's Turbo languages, the /v command-line option or its menu equivalent instructs the compiler to include debugging information in the .OBJ file. Microsoft languages that support CodeView require the /Zi option as if CodeView is the debugger. TDCONVRT then converts the resulting .EXE files. For properly linked and compiled programs, Turbo Debugger presents a source-code window, which displays the first executable lines of source code. Otherwise, the package displays disassembled machine language.

CODEVIEW UNDER OS/2: NICE THREADS

While Borland was busy creating Turbo Debugger, Microsoft was working to add OS/2-specific features to its existing CodeView debugger. In most respects, CodeView is unchanged since we last reviewed it (see "Multilevel Debugger," Mark Ackerman, March 1987, p. 90). However, Microsoft C 5.1 includes a major enhancement to CodeView that facilitates OS/2 debugging—support for multithreaded applications.

The OS/2 environment refines multitasking within a process to a *thread* level, OS/2's fundamental unit of scheduling. Unlike an individual process that has its own data space, an OS/2 thread shares the process environment of its parent. Starting a program under OS/2 actually invokes an instance of the program as thread 1. Thereafter, thread 1 can start other threads (for example, threads 2, 3, and so on).

The behavior of CodeView's standard commands is affected by these threads. For example, a breakpoint that is set with the BP command will stop when any thread reaches the breakpoint. Other commands, such as Trace, Step, and Execute, apply to the current thread, but also allow other threads to run concurrently. Thus, an Execute command will run the current thread in slow motion, but OS/2 can schedule other threads that could preempt that thread. Similarly, a Trace command will execute a single instruction in the current thread, but OS/2 may also run many instructions in other threads before it returns to CodeView.

To reflect the multiple-thread nature of OS/2, CodeView uses a command prompt that displays the number of the *current thread*, which is the thread currently selected for debugging (for example, **001>**). Because every thread has its own stack and register set, the display changes to reflect new values any time the current thread is changed. Note that CodeView controls and monitors just the threads in the program being debugged; OS/2 schedules other threads in the system in the normal way.

For detecting elusive bugs in multithreaded programs, developers need precise control over threads. Help comes in the form of the thread command, ~ (the tilde character), to control execution of specific threads. The command has two fields. The first field specifies the thread to be operated upon:

- ***n*** **thread number *n*;**
- **#** **the last thread executed;**
- ***** **all threads in the program;**
- **.** **the current thread.**

The thread command's second field is a subset of the CodeView commands: BreakPoint, Execute, Go, Program Step, and Trace. Three other commands are also recognized. The Select command changes the current thread to the one specified. The Freeze command disables threads so they will not run in the background. Freezing all but the current thread, for example, ensures that only the thread being debugged runs during a specific section of code. The Unfreeze command reverses the effect of the Freeze uncommand.

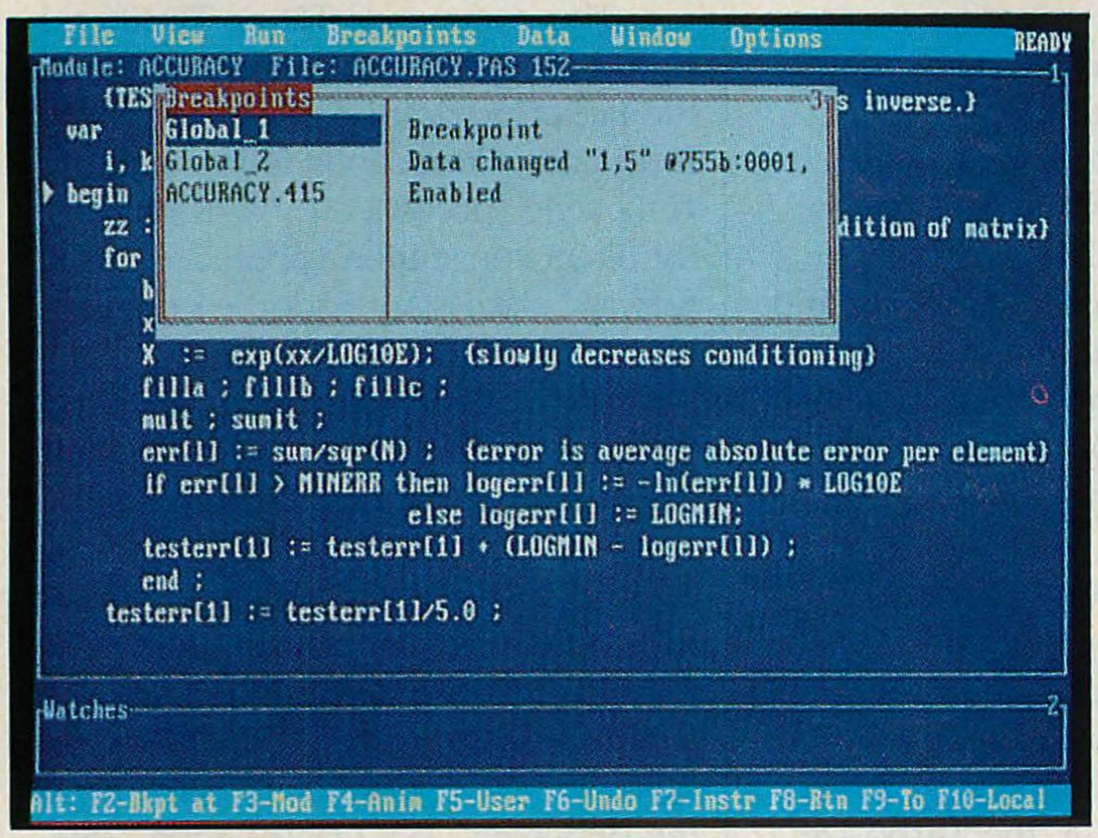
The thread command is powerful, but painfully cryptic. For example,

~	**Show status of all threads**
~*F	**Freeze all threads**
~3U	**Unfreeze thread 3**
~*G	**Run all unfrozen threads**
~2S	**Make thread 2 current**
~2BP .53	**Set a breakpoint for thread 2 at line 53**

CodeView has strong debugging abilities for OS/2, but they certainly look clumsy when compared with Borland's Turbo Debugger. If Microsoft intends for CodeView to reign supreme under OS/2, the company needs to send its debugger in for an overhaul before Borland hauls Turbo Debugger over to OS/2.

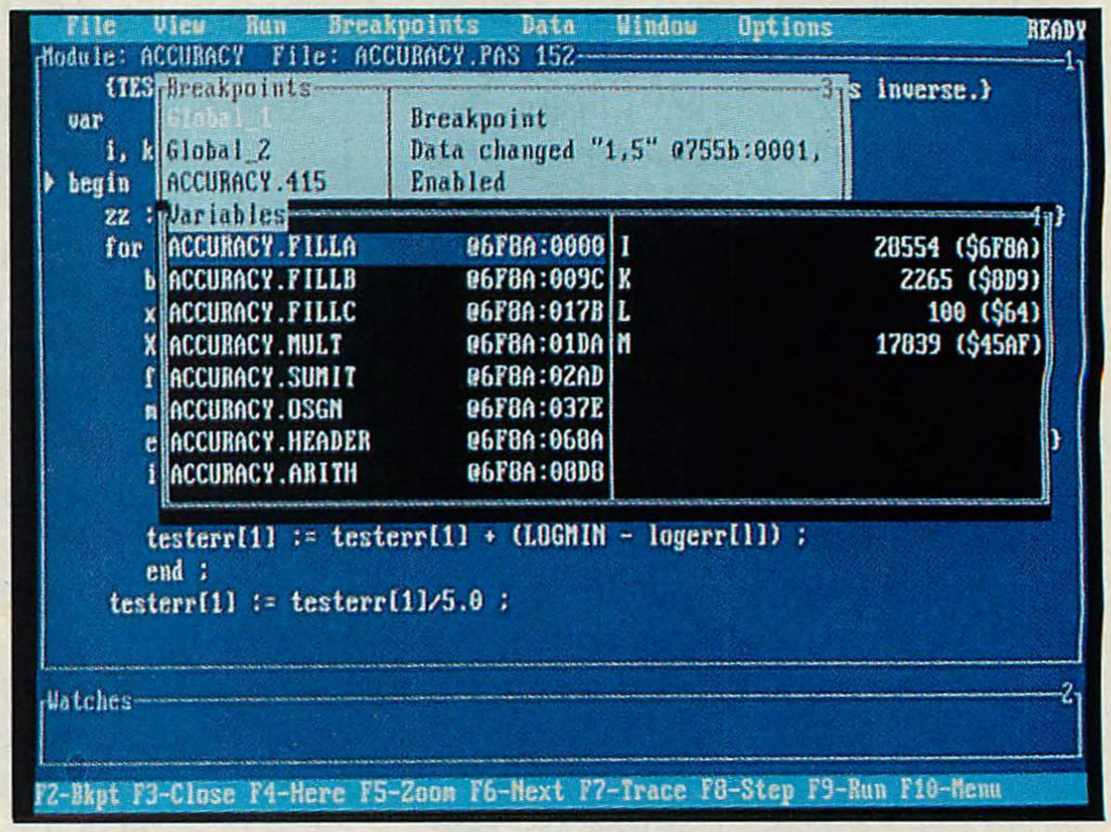
—David Methvin

PHOTO 1: *Setting Breakpoints*



A few keystrokes can set a breakpoint, run the program, trace the execution at the breakpoint, add another conditional breakpoint, and review current status in seconds.

PHOTO 2: *Viewing Variables*



A listing of all the variables in the program ACCURACY.PAS is only two keystrokes away—Alt-V, V. The right pane shows global variables; the left pane shows local variables.

Turbo Debugger has many options for running target programs. One possibility is to execute a single instruction, what Borland calls the *trace-into* function. This selection executes one instruction at a time, including calls. When traced, a call is executed and the window displays the code within the called function or procedure. Turbo Debugger also has a *step-over* function that executes the call and returns to the instruction following the call, treating the call as one logical instruction.

Another alternative is *animation*, which runs the program in slow motion, highlighting each instruction as it executes. Animation continues until the program encounters a breakpoint, terminates, or is interrupted by the Ctrl-Break keystroke combination. The user can set the time interval between animated instructions (0.3 of a second is the default).

Manipulating breakpoints is an easy proposition. The user sets breakpoints with either a single keystroke or from the breakpoint window (see photo 1). While the breakpoint window is active, the user can enable, disable, remove, and add breakpoints. To determine where to set a breakpoint, the user can examine local and global variables from the view-variables window (see photo 2).

Breakpoints can trigger when the target program reaches a specified line of source or assembly code, a variable has a stated value, or an expression is true. In Turbo Debugger, the term breakpoint encompasses the CodeView concepts of *breakpoint*, *watchpoint*, and *tracepoint*.

An *unconditional* breakpoint is a specific place in the program code where execution is to stop. A *conditional* breakpoint stops program execution only when a certain condition is true, such as when a variable has a given value or the program changes a value in memory. Turbo Debugger's conditional-breakpoint expressions are stated in the syntax of the source language (C, Pascal, or assembly) and can be conditioned on either an expression being true or a change in a variable.

A watchpoint evaluates a value of an expression and stops the program when the expression is true. A tracepoint checks all specified program variables or memory-referencing expressions for changes after each instruction executes.

The CPU window provides an all-in-one machine-level view of the target program. It has separate panes for disassembled instructions, registers, flags, stack values, and data. The user can also create multiple windows, each displaying a separate code or data area.

The dump window displays data in memory, as referenced by the current data-segment register of the target program. The user can search for and change a value in memory from the dump window. Data formats include byte, word, long, comp, float, real, double, and extended. In byte format, each byte is accompanied by its ASCII character representation. Turbo Debugger also displays data in any floating-point format supported by Borland compilers (including Turbo Pascal real) and any IEEE format supported by an Intel 80*x*87 math coprocessor.

The developer selects a source-code module (including the main module and any source files) for viewing and debugging through the module window. From this window, the developer can easily see the assembly code that corresponds to the source code by accessing the CPU window. This feature is a distinct advantage over CodeView, where the user has to scroll through assembly code to find the corresponding source code.

Turbo Debugger lists all variables in a program that have global scope, including function or procedure names, in the left-hand pane of the variables window. The right-hand pane displays all variables local to the current function or procedure. The window displays the current values of all variables in the format known by the source program. The value associated with a function name is a pointer, for example, and the value of a Turbo Pascal string variable is ASCII text surrounded by single quotes.

The package displays Turbo Pascal Boolean and enumerated data types with their source-code values as well as their integer equivalents; the values of Boolean variables are either true or false. Because the debugger displays data structures as aggregates of simple variables, the developer cannot change an entire data structure. However, elements of an array or data structure can be modified individually.

To see the status of and set and reset unconditional or conditional breakpoints, the developer accesses the breakpoint window. The F2 key is a quick way to toggle an unconditional

breakpoint anytime at a source- or assembly-code line where the cursor is positioned. The user can set conditional breakpoints based on either a change in a variable or an expression being true. Either makes the program run slowly because the debugger evaluates the breakpoint condition after single-stepping through each instruction and should be used sparingly, generally after stopping the program in a presumed problem area with an unconditional breakpoint.

The Turbo Debugger mechanism used for managing breakpoints is simple, is menu driven, and, unlike CodeView, does not use a command syntax for setting conditional breakpoints. The developer enters the expression that controls a conditional breakpoint into a pop-up window in the syntax of the language selected from the debugger option pull-down menu. Conditional breakpoints based on changes in variables also are entered into a pop-up window.

Turbo Debugger users can monitor simple variables or data structures in the watch window. The value of a watch variable is updated within the watch window while a program runs under control of the debugger. If the target program is running on a system with an 80*x*87 math coprocessor, the user can inspect, clear, and alter as many as eight 80-bit floating-point stack registers and 19 flags through the math coprocessor window.

The helpful user-screen function permits the user to see the actual screen the executing program would display at any given moment. Pressing Alt-F5 toggles between the user screen and the debugging screen.

LEARN FROM EXPERIENCE

Finding the bugs and examining the Turbo Pascal runtime library and DOS programs are easy tasks. For source-level debugging, the relationship between source and assembly code is readily apparent, particularly with the Turbo Debugger options to view either one or both. Turbo Debugger was installed on a PC Designs 8-MHz Turbo AT with an Atronics EGA+ video controller, a 16-MHz IBM PS/2 Model 80, and a Compaq 386/20. The package performed well when used to debug programs with several usual and some not-so-common problems.

Uninitialized variables. A source-code analyzer that executed correctly when compiled with Turbo Pascal 3.01 produced unexpected results with Turbo Pascal 5.0. Within 10 minutes, Turbo Debugger pinpointed an uninitialized variable that caused the problem. An unconditional breakpoint was set in a procedure where the problem was suspected, and the code was traced, watching the values of variables related to the symptom. The .COM file format of Turbo Pascal 3.01 had initialized the variable to zero, masking the logic error in the program. Without a debugger, a developer would have to analyze the source-code structure, which is less likely to pinpoint the problem and can be difficult and time-consuming.

Incorrect variable types. Julian date-translation procedures converted from Turbo Pascal 3.01 reals to 5.0 long integers failed when called by a production program, but worked when called by a test program. Turbo Debugger's two views of the variables revealed that the production program called the functions by passing integer variables rather than long-integer variables.

The Turbo Debugger mechanism for managing breakpoints is simple, menu driven, and does not use a command syntax.

Cooling hot spots. Turbo Debugger helped optimize the hot spot (a processor-intensive area) of a statistical analysis package. To view the generated assembly code, the .EXE program was loaded under control of Turbo Debugger; the source file for the procedure being optimized was selected. The cursor was then moved to the source lines being optimized, the CPU window was accessed, and the code generated by the compiler was inspected. As changes were made to optimize a major loop in the source code, the debugger displayed the corresponding assembly code. This comparison allowed examination of the clock cycles required for the generated code until a practical minimum was reached.

PC Tech Journal's OPTZTEST.PAS program, designed to test common code optimizations, was speeded up using Turbo Debugger to see what code Turbo Pascal generates (see photo 3). OPTZTEST (available for downloading on PCTECHline) uses the standard Turbo Pascal Move procedure to move information in memory quickly. The code in the Move procedure within the runtimes linked into the application was viewed. The Move procedure always does slower 8-bit moves (REP MOVSB) rather than fast 16-bit moves (REP MOVSW) for all but the odd byte being moved. A faster procedure was substituted for Move, resulting in a 50-percent reduction in processor time for moving data and a nominal increase in .EXE file size (see photo 4).

Dissecting FORMAT. The statistical analysis software mentioned above can monitor and count INT 13H disk BIOS calls; however, it failed to show disk activity for the DOS 3.3 FORMAT command when formatting a 3.5-inch diskette. Turbo Debugger verified that FORMAT does not use the INT 13H disk BIOS in this case. FORMAT was loaded under the control of Turbo Debugger, and unconditional breakpoints were set on INT 13H instructions.

On a 5.25-inch diskette, one breakpoint for each track was formatted with an INT 13H, AH = 05H (format track). When a 3.5-inch diskette was formatted, no INT 13H breakpoints were encountered. Turbo Debugger's Search function found the INT 13H instructions quickly; this could not be done with CodeView, because it cannot search for assembly-level instructions.

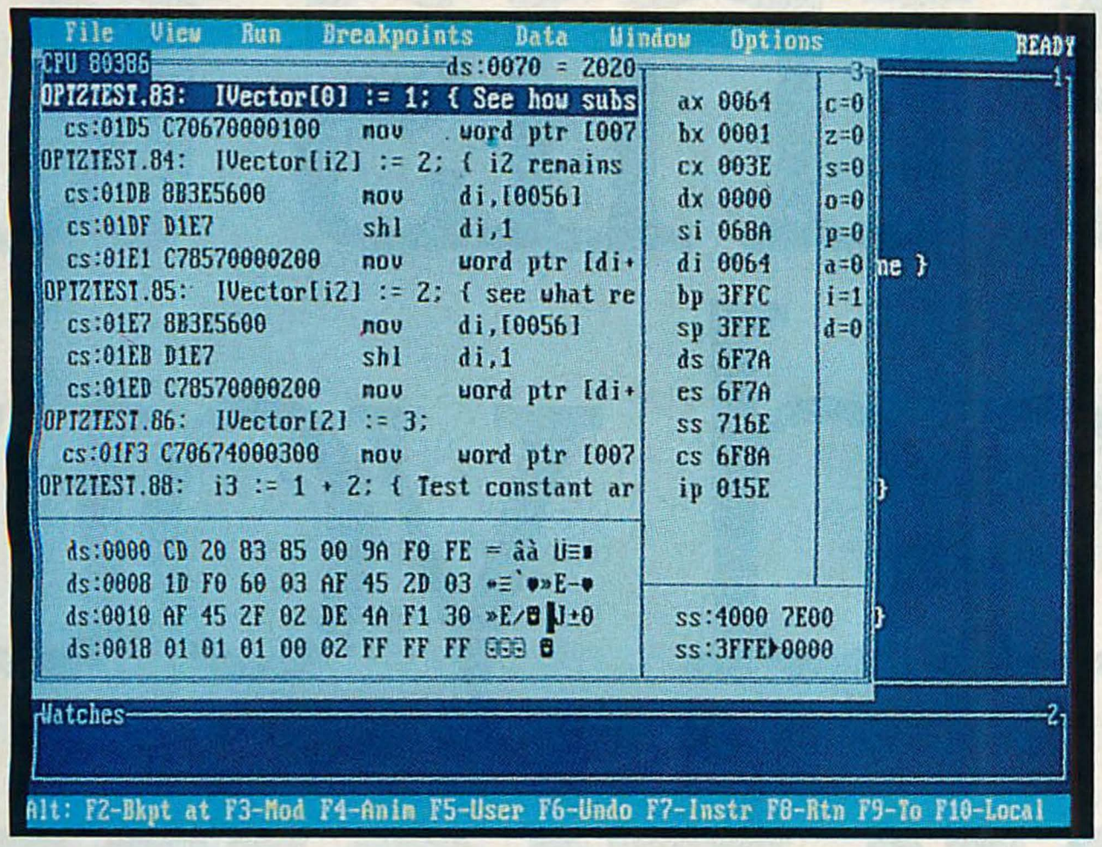
The arguments must be exact when the Turbo Debugger search function is used to find code in memory. A useful addition to Turbo Debugger would be the ability to search for occurrences of a symbolic operand (op) code, regardless of its hexadecimal value. This would make it easy to search FORMAT.COM for all OUT instructions, with any value from the set 0E6H, 0E7H, 0EEH, or 0EFH.

Bugs in the debugger. A minor malfunction occurred during testing. When TDH386 was run with the command-line arguments FORMAT B:, FORMAT did not find the B: argument, indicating that the 386 version of the debugger may not be setting up the program segment prefix of the target program correctly. The non-386 Turbo Debugger performed correctly under the same conditions. Borland technical support said the problem will be corrected, but would not commit to a definite date.

SUPPORTING PLAYERS

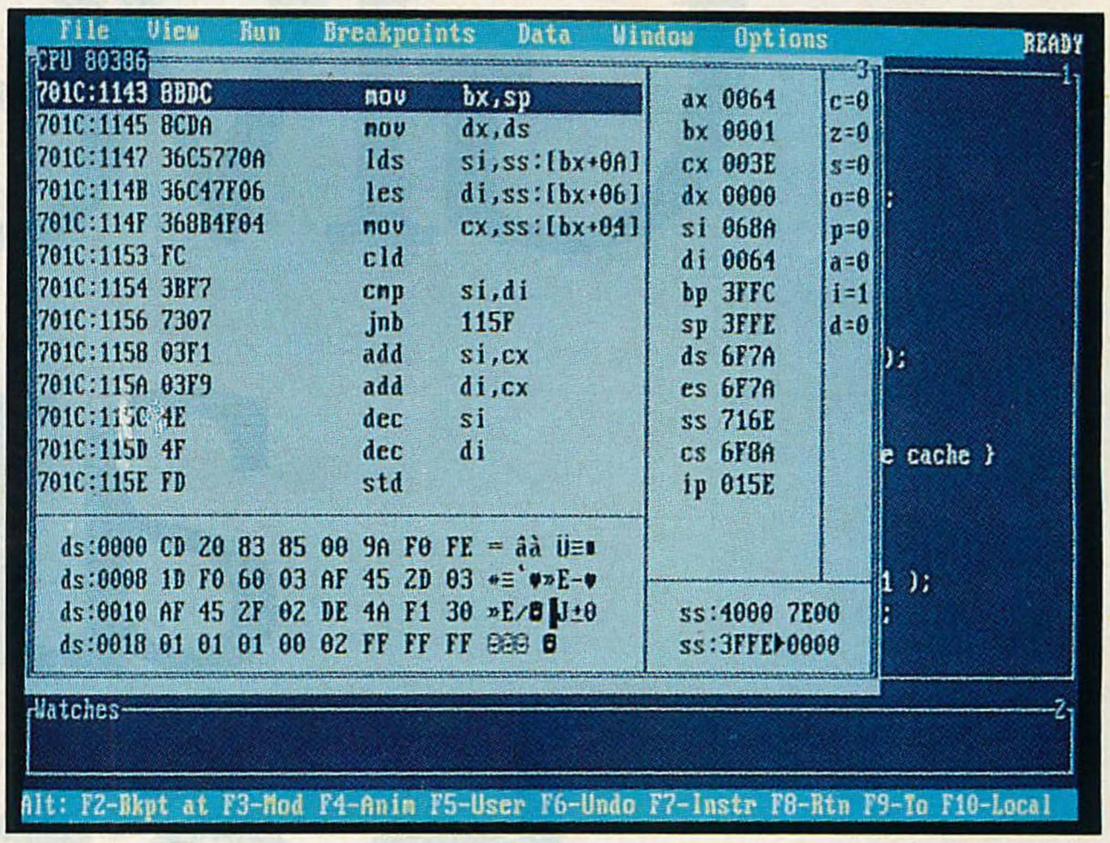
Several utilities play important supporting roles for Turbo Debugger. After debugging is complete, the TDSTRIP utility removes symbol-table information from .EXE files, which eliminates the need to recompile a program to get a finished product.

PHOTO 3: *CPU Status*



Disassembled machine code generated by Turbo Pascal overlays the related source code in the CPU window, whose five panes include stack, registers, flags, code, and a data pane of raw data in the area of memory selected.

PHOTO 4: *Assembly Code Implementation*



Another view of the CPU window shows the underlying assembly-code implementation of a Move procedure call in Turbo Pascal. This kind of analysis by Turbo Debugger can help in making decisions optimizing performance.

If a program that was compiled with a non-Borland compiler is linked with .MAP files, the TDMAP program appends the .MAP information to the .EXE file in Turbo format. The user can then debug these programs with Turbo Debugger. With TDMAP, developers also can use Turbo Debugger on Turbo Pascal 4.0 programs. The steps to do this are as follows:

- Compile Turbo Pascal 4.0 program with /$T+ command-line option to create a .TPM file.
- Create a .MAP file from the .TPM file using the Turbo Pascal 4.0 utility, TDMAP.
- Run TDMAP to combine the .EXE file with information from the .MAP file into Turbo Debugger format.

Another worthwhile utility, TDUMP (which Borland calls a module disassembler), is actually a file analyzer that breaks down the structures of programs, object files, and libraries. TDUMP does not disassemble code, but relies on the debugger to do so. For programs, TDUMP decodes the .EXE file header and shows the initial stack segment address, the program entry point, and all addresses requiring loader relocation.

For object files, TDUMP formats the segment definition information, PUBLIC symbols, and locations requiring linker fix-ups. The utility also displays the contents of all data and code segments in hexadecimal format. For libraries, which are collections of object files, TDUMP shows the same .OBJ information and some library-specific information.

If a file is not in one of these three formats, TDUMP simply prints a file dump in hexadecimal, ASCII, or both. It does not analyze the structure of Turbo Pascal 5.0 .TPU or .TPL files, which are the equivalent of libraries in other languages. No Turbo Debugger utility is provided to accomplish this task, but TDUMP does give a straight hexadecimal readout of the files.

The TDPACK utility reduces the size of the debugging information appended to the executable code of an .EXE file. For example, a 55KB file compiled with debugging information grew to 115KB. TDPACK eliminated about 9,000 bytes of duplicate information, such as strings and data-type information. If Turbo Debugger runs out of memory while debugging a large program, running TDPACK could help. All the utilities display a list of options when started without a command-line argument.

The *Turbo Debugger User's Guide* is well written and detailed, covering installation and overall operation. The bulk of the guide explains operation details, such as examining and modifying files and data, setting breakpoints, and evaluating breakpoint and watch expressions in the target program's source language. Several chapters explain the nuances of debugging programs at the assembly level and debugging with an 80*x*87 coprocessor.

AN EXCELLENT VALUE

Finding bugs in software is a complex task. While no software can replace careful analysis, high-level language debuggers provide an invaluable aid. With Turbo Debugger, the developer can work in the familiar high-level source-code environment, but can easily drop down to compiler-generated machine instructions.

Turbo Debugger is a solid addition to any software developer's toolbox and rounds out Borland's programming product line. It has many attractive features including an easy-to-use interface; close integration with Turbo C and Turbo Pascal; the ability to debug CodeView-compatible .EXE files; and the capacity to debug very large programs, either on a 386-based PC or remotely, using two PCs.

Moreover, the package is a bargain. Borland sells Turbo Assembler and Turbo Debugger bundled together for $149.95. Turbo Debugger is included with Borland's Professional Turbo C and Professional Turbo Pascal; both packages sell for $250.00 each. Current users of Turbo C or Turbo Pascal can upgrade to the Professional packages for $99.95 each.

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001
408/438-5300
Turbo Debugger 1.0

CIRCLE 331 ON READER SERVICE CARD

Ben Myers, owner of Spirit of Performance Inc. in Harvard, Massachusetts, specializes in languages and other software. His last article for PC Tech Journal *was a review of Turbo Pascal 4.0 in April 1988.*