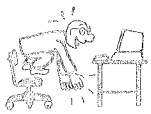
## AT A GLANCE...

T–DebugPLUS v. 2.0
Turbo Pascal source code debugger

TurboPower Software
3109 Scotts Valley Drive
Scotts Valley, CA 95066
[408,196438–8608]



Price
$60 list; $49 from the Programmer's Connection

System Requirements
IBM PC, XT, AT, or close compatible; 256KB RAM; two diskette drives; DOS 2.0 or later; Turbo Pascal 3.00 or later for the IBM PC; a fixed disk is recommended.

Reviewer
James T. Smith
[415–469–2251]

Pro
Well designed and implemented; easy to use; executes the Turbo compiler itself; supports Turbo EXTENDER; inexpensive.

Con
May require a separate object code debugger.

The GSI Pascal debugger
Turbo Pascal Source Code Debugger

GSI Transcomm
1380 Old Freeport Road
Pittsburgh, PA 15238
[412–963–7270]

Price
$49.95 list

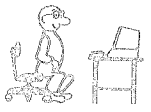
System Requirements
IBM PC, XT, AT, or 100% compatible; 256KB RAM; DOS 2.0 or later; a fixed disk and much more memory is recommended.

Pro
Flashy window manipulation; reasonable memory requirement; inexpensive.

Con
Poorly designed; hard to use; slow; built–in compiler differs from Borland's and doesn't produce .Com files; can't handle low level constructs.

TURBOsmith v. 2.0b
Turbo Pascal Source Code Debugger

Visual Age
642 North Larchmont Boulevard
Los Angeles, CA 90004
[213–534–4202]



Price
$69 list; $59 mailorder

System Requirements
IBM PC, XT, AT, or close compatible; two diskette drives; a fixed disk is recommended; DOS 2.0 or later; Turbo Pascal 3.00 or later for the IBM PC. I needed 640KB memory for DOS plus TURBOsmith operation, with my virtual disk, etc., pushed up into expanded memory.

Pro
Well designed and easy to use; executes Turbo Pascal compiler itself; includes an object code debugger and a rudimentary assembler; inexpensive.

Con
Requires a vast amount of memory; still under development and has too many bugs.

In this review article, I shall consider in detail two IBM PC Turbo Pascal source code debuggers — T–DebugPLUS and The GSI Pascal debugger. I will write about the third, TURBOsmith, briefly.

A debugger is a software development tool that executes a test program on test data. To this extent, a debugger is an interpreter for the program development language. With a debugger, however, you can pause at intermediate steps to inspect the state of your machine. At these breakpoints you can alter its status

— change a variable or perhaps even an instruction — to answer questions such as "how would the next part of my program function if this variable had a different value"? For convenience, the debugger should input test program and data and display intermediate status reports in formats similar to those in which you conceived them.

A major debugger design problem is separating test program input/output (i/o) from your dialogue with the debugger itself. If the test program used only file i/o, or if you could dedicate separate consoles to test and debug i/o, this would not be difficult. However, you usually have only one console. You would use the debugger to develop software that itself assumes total console control. The products reviewed here handle these conflicts differently.

Object code debuggers, like the DOS Debug utility, accept test programs only in machine code. They give intermediate status reports in primitive assembly language. Thus, an object code debugger includes a disassembler. To permit altering the test program at intermediate steps, these debuggers must include a rudimentary assembler as well.

Source code debuggers accept test programs in a higher level language. If the language is to be interpreted, then the debugger is merely an enhanced interpreter. For compiled languages, debugger design is more complex. This review is about compiled language debuggers. In order to give intermediate status reports in source language format, these debuggers require detailed information from the compiler. They need memory locations, type of variables, and the correspondence of machine language to source code. If the compiler produces object code modules (Turbo Pascal does not), the linker will need additional information — the utility that builds executable machine code files from the modules.

There are two alternatives for providing compiler information to the debugger. The products under review use both. First, the debugger can include a compiler. Second, it can accept the compiler itself as input.

The first alternative is satisfactory only if the included compiler agrees with the details of the one you will use for the production version of your test program. The only way to insure this is for them to coincide. This is probably the way of the future: production compilers will be components of source code debuggers. At present, this is not the case. So with a product of the first type, you run the risk of debugging a test program essentially different from your production version.

The second alternative, inputting the compiler, requires a detailed understanding between compiler and debugger designer. The information needed by the debugger must be in a standard location within the compiler. Without this understanding, the debugger must change in step with the versions of the compiler.

I considered the following questions for each product:

- How is it organized?
- What are its limitations?
- How does it separate debugger commands and reports from test i/o?
- How do you observe and control the sequence of test program execution steps?
- How do you observe and modify test program variables at intermediate steps?
- Can you observe and modify execution and variables in the machine language version of the test program?
- How effective is the user interface design and implementation?
- Does the debugger sport some special features?

T–DebugPLUS

T–DebugPLUS is a descendant of the public domain program T–Debug. It inputs the compiler and executes it as a child program. Since all debugging is done in the compiler Memory mode, T–DebugPLUS cannot handle the Turbo Chain and Execute procedures, nor features that require .COM files. It can debug programs that use overlay files, but not the overlays themselves. To analyze the machine code version of a test program, you will probably need an object code debugger too.

T–DebugPLUS includes utilities for generating .MAP and line numbered source code files for use with symbolic object code debuggers. Officially, T–DebugPLUS requires Turbo Pascal IBM PC version

3.00b, 3.01a, or 3.02a. Its stated memory requirement is 192KB plus that of the test program. T–DebugPLUS produces one temporary file, which you may place on a virtual disk for maximum speed.

T–DebugPLUS permits use of two monitors. With just one, while executing part of the test program, it displays the test output screen. It then replaces that by the debugger screen when it regains control. Then, <F10> toggles between screens. T–DebugPLUS handles graphics mode output from the CGA, EGA, and Hercules display controllers — <F10> switches between text and graphics modes when necessary. (T–DebugPLUS automatically determines which controller is installed!)

Since test program and debugger are not simultaneously in control, keyboard input is handled well except in two minor ways. First, response to <Ctrl–Break> is somewhat unpredictable. When the test program responds, it passes control back to the debugger. T–DebugPLUS cannot handle compiler directive $U+. Second, function KeyPressed works differently under the debugger. These are symptoms of faulty design in the compiler, not necessarily the debugger.

T–Debug PLUS displays source code in the top part of the debugger screen, with its file identified. It numbers statements and emphasizes the next instruction scheduled for execution. You can view source code by specifying a line number, a routine name, a code segment address, the name of an included file, or by requesting display of the routine that called the current one. It also permits vertical and horizontal scrolling.

You can install temporary and permanent breakpoints, and execute with these pauses or return to the next procedure. Conditional breakpoints are possible too. These can take effect if a specified variable or location in memory changes value or if a variable assumes a given value. You can execute one statement at a time, and you can execute a procedure without pause. You cannot change the test program execution sequence.

When the debugger is in control, you can exit to the Turbo menu to edit, compile, or set options. This is an abnormal exit from the test program, and may leave files, etc., in disarray.

During a debugger pause, you can view virtually any constant, variable, location in memory, or 8086 register. You can request specific information during the current pause, or set up a screen window to display constantly updated values of several variables. These data may be scalars, strings, sets, array or record entries, or one dimensional ranges of array entries such as A[1,10..15]. You can view them in ASCII, hex, or decimal format. A user defined scalar is displayed as spelled in its declaration. You can display pointers as addresses, as well as the values of the variables they refer to. T–DebugPLUS does not handle the Turbo Mem and Port arrays like ordinary ones, but you won't need the former because you can display arbitrary memory locations.

During debugger pauses, you can also change the value of any variable or region of memory and inspect the heap structure. A rudimentary integer arithmetic calculator is also available.

To debug machine language code, you must use a machine language debugger to execute T–DebugPLUS. In T–DebugPLUS, execute debugger command X to determine addresses corresponding to procedures or statements. Then use command DG to enter the machine language debugger to disassemble or single step the code. The T–DebugPLUS package includes utilities TMap and TDList for generating .MAP files and line numbered source code listings for use with symbolic machine code debuggers. These are compatible with those produced by the Microsoft linker. A symbolic debugger, such as Symdeb, can read them. I have not tested this T–DebugPLUS feature.

Overall, the T–DebugPLUS user interface design is very good. In particular, its separation of test and debugger i/o is virtually flawless. I do have a few complaints:

- You can't view an include file that contains only declarations.
- To display a record entry, you must enter the complete name. This is inconvenient and prevents debugging with statements.
- The rules governing display and alteration of variables and memory regions are a bit chaotic, due to features added with Version 2.0.
- You must use a separate object code debugger for most low level work.

The implementation of the design is superb. The debugger command codes are natural, there are few

unnecessary keystrokes, and response is fast. The on-line help screen and the manual provide just what's needed, lean and clean.

There is one special feature of this product that I find amazing: it is written entirely in Turbo Pascal, and its source code — about 11000 lines — is included! You can change many T-DebugPLUS defaults — for example, the maximum number of include files permitted — by making obvious minor source code changes and recompiling.

Two other special features are noteworthy. First, T-DebugPLUS includes a special provision for debugging programs that use TurboPower's Turbo Extender package for program modularization. Second, the company provides consultation via CompuServe. I have not tested either of these features.

### The GSI Pascal debugger

Unlike the product just reviewed, the GSI Pascal debugger, also known as SymPas, includes a compiler supposedly equivalent to Turbo Pascal, but which cannot produce .COM files. You are expected to use SymPas for program development, then turn to Turbo for final compilation. This is unlikely for three reasons:

- The two compilers aren't equivalent. (In particular, absolute memory addressing doesn't work in SymPas, and its keyboard input differs from Turbo's.)
- The Turbo development environment is more convenient.
- The SymPas debugging facility is too primitive.

SymPas requires a "100% compatible" IBM PC and recommends 512KB memory. It consists of several files, and builds many temporary files. For reasonable speed, you should have enough memory to put all these files on a virtual disk. (My 640KB setup is not enough.) Since SymPas compiles only to p-code, you cannot debug any Turbo construct that depends on machine language or .COM files.

You can run a test program in execution or debugging mode. In the former, SymPas merely interprets p-code. <Ctrl-Break> is always effective, in spite of Turbo conventions. In the latter mode, test output is obscured by debugger report windows. You can drag these around, however, to uncover your vitals.

SymPas allows you great flexibility in requesting which part of your source code to view. A particularly nice feature is its display of the procedure tree. However, the window is too small, so you must constantly scroll both vertically and horizontally. It is easy to forget what you just saw.

In debugging mode you can execute the test program using either temporary or permanent (but removable) breakpoints. You can single step it or execute it with automatic pauses between statements. The manual doesn't explain how to use this last feature. (Use trace without "automatic break".) Single stepping requires two keystrokes per step, and distracting screen rearrangements occur every time. Permanent breakpoint installation uses addressing different from the displayed line numbers. You have to activate a breakpoint window, switch to the source code window, move the cursor to the desired breakpoint, switch back to the breakpoint window, and then accept the setting. You can change the test program execution sequence by resetting the p-code equivalent of the IP register.

During debugger pauses you can view all constants and variables accessible from the current routine. This is awkward and in primitive format. It requires action in three different windows and familiarity with the Turbo memory formats for variables of various types. Then you have to pop the window stack three times to get back to debugging. It is not clear which references to absolute memory locations work and which do not. References to the Mem array do not work.

Some special features are noteworthy. <F2> activates a decimal/hex calculator with cut/paste capability. You can request a directory tree display. (This may take a long time for SymPas to construct if you have many directories.) Finally, you can inspect the contents of an arbitrary file.

Overall, the SymPas design is terrible — so inconvenient that you will never try it a second time. In some respects, the implementation of the design is bad, too. For example, accessing all the SymPas and temporary files takes far too long, and several keystroke conventions differ from Borland's. In one aspect, though, the implementation is very impressive. Its window handling is fast, flashy, and reasonably intuitive, even though it interferes seriously with the business at hand: debugging.

### TURBOsmith

I had originally intended to review TURBOsmith in detail. With its current version that proved impossible.

This product comes from a family of debuggers named CODEsmith. It has a structure like T–DebugPLUS, but promises to be a much more powerful package. It includes an object code debugger, disassembler, and a limited assembler. TURBOsmith's user interface is as well designed as T–DebugPLUS's, with even more sophisticated windowing features.

For these added features you must pay a severe price in memory requirements. To trace its own tiny demonstration program, TURBOsmith needed about 477KB available memory. This exceeded what I normally have on a 640KB PC with DOS 3.1, a 90KB virtual disk, a 32KB print buffer, and only one resident utility (PopAlarm). The manual warns that serious software development should only be attempted on a larger machine. In principle, I agree and will soon upgrade to 1.6MB with an AST Rampage! board. By installing my virtual disk and print buffer in expanded memory, about 120KB will become available for TURBOsmith. But most PC systems — for example, those owned by universities or students — do not have this capacity. This product, at least for a couple of years until expanded memory becomes standard, will have a rather small market.

A second reason for my inability to consider TURBOsmith in detail is that it is still under development. By some accident, my copy of Version 1.0 had the very first serial number. It also had so many bugs that it was unusable. Visual Age has released some interim versions, including the current 2.0b. However, some serious bugs or else design errors must remain, because it refuses to trace one of the short, simple string handling demonstration programs used for the other products reviewed. This shouldn't be due to the lack of memory, because I tried it with 583KB available. It crashes ungracefully, claiming that it can find no source code!

### Conclusion

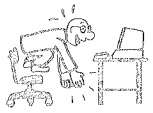
I believe that once it stabilizes, TURBOsmith will be the debugger for professional IBM PC Turbo Pascal software developers. Until then, and for ordinary folk, T–DebugPLUS is the one to buy. It is inexpensive, wonderfully designed and implemented (though it lacks machine language capability), and it works. The first time I used T–DebugPLUS, it saved me a day's debugging time on a production numerical simulation program. I wouldn't try to work without it, and I recommend it to my students and colleagues. &

## Software Review

### *AT A GLANCE...*

TAPCIS
Telecommunications
Program for Accessing CompuServe

by Howard Benner



Support Group, Inc.
P.O. Box 1577
Baltimore, MD 21203
[800–872–4768]

Price
$79

A shareware product that can be downloaded from the SF PC BBS or CompuServe.

Reviewer
Rodney Paul [415–386–4991]

Pro
Saves large amounts of time (and money) by automating the use of CompuServe forums and EASYPLEX E–mail; includes a versatile text editor for composing messages offline and the "Quick–B" data transfer protocol.

Con
Automated functions may bomb out from excessive line noise; inhibits novice users from learning CompuServe commands.

In two previous articles, I touted the scripting capabilities of Qmodem to automate such telecommunications activities as BBS log–ons and navigation. Qmodem lends itself well to a wide variety of BBS and other online systems. But when I recently signed up for CompuServe, it was clear that Qmodem had reached its limits. Although I believe that the program's script language can probably accommodate the system's complexity and vastness, I had neither the time, the inclination, nor the patience to experiment with the clock ticking to the tune of $12.50 per hour. Qmodem has no facility for writing and editing messages offline and lacks the CompuServe "Quick–B" data transfer protocol. To truly get the most from CompuServe, it was clear I would need to try something else.