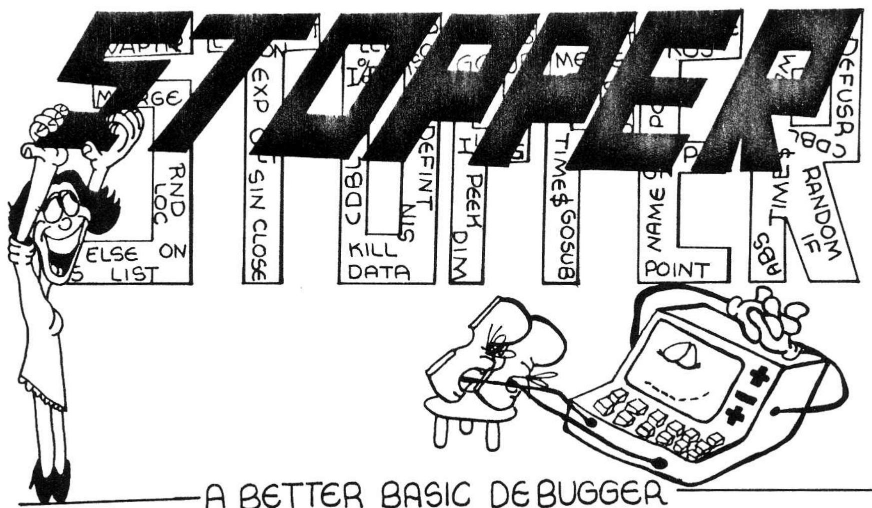
# Roxton Baker
# author of TRAKCESS
# presents:




# THE ALTERNATE SOURCE

# STOPPER

**The BASIC Breakpointer**

(c) 1981 by Roxton Baker
Box 8272, APO San Francisco 96555

This documentation assumes that you have STOPPER in a loadable form for your system (tape or disk). To transfer STOPPER from tape to disk, see the instructions near the end.

I hope that you find STOPPER to be a genuinely useful program. A great deal of time has gone into its development. If you have problems, or suggestions for improvement, please write.

## STARTING STOPPER

Note that there are two steps to both the disk and tape loading procedures!

### Disk Systems:

1) From DOS READY, type:

STOPPER <Enter>

This will cause STOPPER to determine the current setting of memory size (from location 4049H) and locate itself just under that. The value at 4049H is then reset lower to protect STOPPER. About 3.1K of high-memory RAM space is required by STOPPER.

This method of loading STOPPER will work with other high-memory utilities, provided that you load them first and that they set memory size (at 4049H) to protect themselves. There is a way around these two requirements. You may force STOPPER to load below any high memory location mmmmm by typing:

STOPPER mmmmm <Enter>

So if for example you have some other utility program for which you would normally set memory size at 60000, you would use 'STOPPER 60000' instead. Another time you might use this entry method is if you have entered and left Basic, and then tried to reload STOPPER. In this case STOPPER may detect its own previous memory size setting at 4049H, and load below that - which of course is not necessary. The solution is to use 'STOPPER 65535 <Enter>'. This forces STOPPER to load at the top of memory.

In the process of relocating itself, STOPPER briefly resides in the area 5500H-6500H, and will overwrite anything there. This should not be a problem with a disk system.

2) When STOPPER has been relocated, the program logo will be displayed. Just below that will be the program entry point (shown as "/eeeee"). Go into Disk Basic without setting memory size. It will automatically be set just below STOPPER's entry point. At the Basic READY prompt, type:

SYSTEM <Enter> /eeeee <Enter>

to activate STOPPER (you should receive the response "STOPPER <ON>"). Typing it again will deactivate STOPPER, as will STOPPER's 'Q' command. If you forget it, this "eeeee" value may be found under Basic by typing:

```
PRINT PEEK(16457) + 256 * PEEK(16458) + 2
```

| | |
|---|---|
| | precision variable value comparisons. Range of 'tt' is 0-15. |
| H | Display tolerance on SP and DP comparisons. |
| J | Reinitialize STOPPER completely. |
| Q | Turn STOPPER off. |

In addition to these direct commands, there are several keys active during the execution of your program. These are described below.

**BREAKPOINTS**

The most important command is '<', to set a breakpoint. This command allows you to stop at selected points and times in a Basic program without having to edit the program (and thereby lose all of your variables). Briefly, you may set a breakpoint to be taken when a certain line number has been reached, or when any variable has become equal or unequal to a specified value. Further details on these two distinct types of breakpoints are provided below. Again, note that all commands are entered at Basic's READY prompt, and all commands must be followed by <Enter>. Any active breakpoint may be turned off by 'F'. Any inactive breakpoint may be reactivated by '<'.

**LINE NUMBER BREAKPOINTS**

A line number breakpoint is set by the command '<nnnnn,ccccc', or a variation on it. The numbers nnnnn and ccccc are decimal integers with a maximum value of 65529. The lower limit on nnnnn is 0. The lower limit on the "hit count" ccccc is 1. Leading zeros need not be entered. When such a breakpoint has been set, and a Basic program is executing, STOPPER will halt the program just before the specified line (nnnnn) is executed for the specified (ccccc) time. STOPPER initializes with nnnnn = 0, ccccc=1. This is the default line number breakpoint. The 'M' command may be used to see how many "hits" remain to be taken on a line number breakpoint.

As an example, suppose that in a program you wish to break just before line 210 is executed. You would enter:

<210

and you would then RUN or CONTinue the program. If you wish instead to wait until line 210 is about to be executed for the fifth time, you would enter:

<210,5

When STOPPER hits a line number breakpoint, it restores the breakpoint hit count to its original value, but it leaves the breakpoint OFF. (You may easily reactivate any breakpoint with '<'). Then STOPPER freezes the program as if CLEAR had been pressed. The "*" hold indicator will appear in the lower right corner. Now, if you press BREAK, STOPPER will display the number of the line last executed followed by the number of the line about to be executed (the breakpoint line). This shows you just how the program arrived at the breakpoint, and can be quite useful in troubleshooting.

Line number breakpoints will not work on lines jumped to by an ON ERROR GOTO statement, because the ROM allows no opportunity to intercept such a jump. The best you can do here is to break on the second line (if any) of the error handling code. In this case, STOPPER will still show you which line caused the error.

| | |
|---|---|
| B | If a variable breakpoint has previously been defined, restore it exactly. |
| N | If a line number breakpoint has previously been defined, restore it exactly. Otherwise, activate the default line number breakpoint. |
| <variable | Define this variable to be the one of interest. This will be the variable whose value is subsequently displayed by the '?' command or is directed as a trace to screen or printer. |
| <=value | Set a breakpoint to be taken when the last specified variable (if any) becomes equal (or, on "*", unequal) to the specified value. |
| M | Show status of current breakpoint. |
| F | Fix (clear) any active breakpoint. |
| ? | Show current value of variable of interest (if any). |
| C | Delay 1 second, then CONTinue. |
| X | Same as 'C', but go into "hold" immediately, awaiting keyboard input to proceed. |
| + | Execute next statement. |
| > | Execute to beginning of next line. |
| Z | Same as '>', but go into "hold" immediately, awaiting keyboard input to proceed. |
| P | Display current line from current statement to end. This important command is also used to locate errors in a long line, as described below. |
| R | "RUN". |
| T | TRON (line number is displayed in lower right corner). |
| Y | TRON to printer in one column. |
| U | TRON to printer in eight columns. This cannot be used simultaneously with 'O'. |
| I | Trace variable value to screen. It is displayed on the bottom line. |
| O | Trace variable value to printer. This cannot be used simultaneously with 'U'. |
| G | TROFF (all traces). |
| L | "LIST". |
| Htt | Set tolerance on single and double |

**Tape Systems:**

1) From Basic READY, use the SYSTEM command to load the machine language file STOPPR. When the *? prompt reappears, type '/<Enter>'. This will cause STOPPER to determine the current setting of memory size (from location 40B1H) and locate itself just under that. The value at 40B1H is then reset lower to protect STOPPER. This means that you can use STOPPER with other high-memory utilities, provided that you load them first, and that either they set their own memory size or you set it for them. You may force STOPPER to load into the very top of memory by turning power off and then on before loading it.

While relocating itself STOPPER briefly resides in the area 5500H-6500H, and will overwrite anything there. It will not affect code below that, however. Thus you may load STOPPER even if you already have a short (less than 4K) Basic program in memory. It is best, though, to load and initialize STOPPER before loading or creating any Basic text. About 3.1K of high-memory RAM space is required by STOPPER.

2) When STOPPER has relocated itself, the program logo will be displayed. Just below that will be the program entry point (shown as "/eeeee"). Now type:

```
SYSTEM <Enter> /eeeee <Enter>
```

to activate STOPPER (you should receive the response "STOPPER <ON>"). Typing it again will deactivate STOPPER, as will STOPPER's 'Q' command. If you forget it, this "eeeee" value may be found by typing:

```
PRINT PEEK(16561) + 256 * PEEK(16562) + 4
```

## STOPPER'S COMMANDS

With STOPPER active, you have a number of commands available to be entered at Basic's READY prompt. You must press <Enter> after any command. These commands include:

| Command | Meaning |
|---|---|
| <nnnnn | Set a breakpoint at line nnnnn, to be taken the first time the line is hit. The program will halt BEFORE any of the line is executed. |
| <nnnnn,ccccc | Set a breakpoint in line nnnnn, to be taken when it is hit ccccc times. See below for more information on these numbers. |
| <nnnnn, | Set a breakpoint at line nnnnn, keeping the previous hit count. |
| <,ccccc | Set a breakpoint at the same line as was previously used, but change the hit count to this new ccccc value. |
| <variable=value | Set a breakpoint to be taken when the specified variable becomes equal to the specified value. |
| <variable*value | Set a breakpoint to be taken when the specified variable becomes unequal to the specified value. |
| < | Restore previous breakpoint exactly, whether a line number or a variable value breakpoint. |
| <, | Restore previous line breakpoint, but with a hit count of one. |

for display. If no variable has been named, the '?' key will just pause the program, without affecting the display. Since pressing '?' requires the use of the shift key, any tracing to printer will be temporarily disabled (see below).

Pressing BREAK during program execution will cause the program to break and STOPPER to print out the current line number.

## SINGLE STEPPING A PROGRAM

From any point at which you have pressed BREAK you may continue via 'C', 'X', '+', '>', or 'Z', as listed earlier. The latter three commands will single-step through your program, stopping completely between each statement or line. These commands differ from CLEAR key stepping in that they force the program to break, rather than just freezing its execution. Note that the '+' and '>' commands may be entered conveniently by holding down the left shift key while pressing the command key and then <Enter> on the right.

While stepping in this manner, whenever STOPPER reaches a line end (as distinct from a statement end) it will print out the next line to be executed. If a variable value breakpoint is active, and you step through it using '>', it will be taken as if it had been hit at full speed; that is, the program will freeze with the "*" hold indicator on. This will not occur with line number breakpoints, or with any statement stepping using '+'.

Another point to remember in the use of '>' is that stepping will continue until the start of the next line IN PROGRAM FLOW ORDER. Thus if you use '>' to step through Line 10 in this example:

```
10 GOSUB 30 : GOTO 100
```

the program will halt next at Line 30.

Statements using Basic's INKEY$ command will require some care when single-stepping. In order to handle them, the 'X' and 'Z' commands have been provided. The 'X' command is identical to 'C', except that you are immediately put into a "hold" state with the "*" indicator showing - just as if you had hit a breakpoint or had pressed CLEAR. Similarly, the 'Z' key is identical to '>', except that here too you immediately enter the "hold" state.

Thus if you wished to step through the following code, and later watch what happens when '2' is the INKEY$ value, you might use '<110' to break cleanly at a recognizable place:

```
100 PRINT"ENTER NUMBER OF TORPEDOS!!!!"
110 I$=INKEY$ : IF I$="" THEN 110
120 T=VAL(I$) : GOSUB 2100
       etc.
```

(It is an exclusive and valuable feature of STOPPER to ignore all misspellings in Star Trek programs.)

Next you will want to input a value into the INKEY$ loop. The command '+' will step through the INKEY$ statement, but will not input a value to it. Therefore you would instead use 'Z' which, after you press <Enter>, will put the system into "hold" awaiting further keyboard entry. The "*" hold indicator will appear to confirm this. Then you would press '2', which is how many "torpedos" you wanted to fire. Or phazers". Line 110 will now be executed in full, and the program will stop again at line 120. You may now continue to single-step via '+' or '>'. Had you wished to continue normal execution after pressing '2', you could in this case have used 'X' rather than 'Z'.

Long lines with FOR-NEXT loops in them can cause problems for both breakpointing and single-stepping. Consider this line, where we assume that N has been defined as an integer:

compared exactly). When you say "break if A!=1.234567", do you mean EXACTLY 1.234567, or close to it? If the requirement is for exact equality, then use the 'Htt' command (in this case 'H0') to set a zero tolerance (STOPPER initializes with tolerance tt=4). However, if the value you are concerned with has been calculated in the program - which is the most likely case - rather than having been read in as a data item or stated explicitly, then an "exact" comparison may fail, due to round-off errors in the machine arithmetic. What you really want to do is break when A! is approximately 1.234567. Since there is no way for STOPPER to know beforehand just how loose this approximation may be, the 'Htt' command will let you specify the required tolerance on the breakpoint comparison. The syntax of this command was shown in the command list. The tolerance value 'tt' may range from 0 through 15. The effect of this value is to cause STOPPER to ignore the last tt bits in its comparisons of SP and DP variable values. Ignoring zero bits means that the values must match exactly. Ignoring fifteen bits means that they may differ by a great deal (the actual effect varies between SP and DP numbers). You may never need the 'Htt' command, and should only be concerned with it if you set an SP or DP breakpoint that should have been taken, but wasn't, or if the program is breaking on values close to, but not quite, the one you wanted. If you change the tolerance 'tt' it will remain at its new value until STOPPER is reinitialized. You may use the 'H' command to display the current value of 'tt', and it will be shown whenever the status of an SP or DP breakpoint is displayed.

## THE "P" COMMAND

Whenever you press BREAK while your program is running, STOPPER will print out the current line number. The 'P' command may then be used to show you exactly where in the line you stopped. Normally, 'P' will display the remainder of the current line, beginning with the next statement to be executed. However, if the program has just hit a variable value breakpoint or if it has halted on an error, then the 'P' command will display THE STATEMENT THAT CAUSED THE HALT (followed by the rest of the line). This will be the statement LAST executed, not the one next to be executed. This allows you to instantly determine, even in long, multiple-statement lines, exactly where an error occurred or a value changed.

## DURING PROGRAM EXECUTION: CLEAR, ?, >

When your program is running you may slow it or single-step it using the CLEAR key (or shift/CLEAR, if your DOS does not prevent it). Pressing and releasing CLEAR will freeze the program at the current statement. To confirm this action, the "*" hold indicator will appear in the bottom right corner of the screen. Pressing and releasing CLEAR again will allow the next statement to be executed. Holding CLEAR down will allow the program to continue at about three statements (not lines!) per second. You can tell when each statement executes by the flicker of the "*" indicator. Pressing <Enter> or most any other key will start the program running at full speed again.

Whenever the "*" hold indicator is showing, you may press '<' to reset the most recent breakpoint. This is generally just a convenience, but it is useful when you do not wish to disturb the video display. Consider the case where a breakpoint is to be enabled only after a certain point in the program has been reached. Define the breakpoint, disable it using 'F', and RUN the program. At the point of interest, press CLEAR to stop the program and display the "*" hold indicator. Then press '<' and this same "<" symbol will appear in place of the "*". This tells you that the last breakpoint has been reset. It is exactly the same as using the '<' command at the READY prompt.

If a variable has been defined via '<variable', then you may, while your program is running, press '?' to observe the value of the variable. This value will appear on the bottom line, just as if the 'I' key were active. The value will remain, with program execution paused, as long as '?' is held down. If the total length of the variable name and value exceeds 22 characters, it will be truncated

## VARIABLE VALUE BREAKPOINTS

A variable value breakpoint is set by the command '<variable=value' or a variation on it. You may specify any type of variable (integer, single or double precision, or string) and the variable may be subscripted. The only limitation is that the variable's name must be no more than fourteen characters long. You may specify any reasonable 'break' value for this variable, as long as the value is no more than 23 characters long. You must take care to have a decimal point in any floating point value, and double precision (DP) numbers must either have more than seven digits or must end with the 'D' exponent identifier. Also, you may specify that you wish the program to halt when the named variable becomes unequal to the given value. For this, simply use "*" instead of "=". Example variable value breakpoints are:

```
<X!=21.                     (note decimal point for SP)
<L%=724                     (note no decimal point for integer)
<VK#=43.17833069212
<WP$(8)="BAD MESSAGE"
<QQ$*"GOOD MESSAGE"
<NS!(7,6)=-0.03274          (note negative value)
<J2#(4,4)*+29.1145D0        (note optional "+")
```

It is always safest to use %,!,#, or $ to specify the variable type explicitly, as was done in these examples. Making a practice of this will avoid the possibility of having to reenter the breakpoint spec, as well as the chance of setting a breakpoint on a nonexistent variable. As shown in the command list earlier, you may change the break value on a previously specified variable by entering only the value. Thus, if TMT$(6,1,9) had been the most recent variable specified, you could modify its break value with:

<="BREAK HERE INSTEAD"

or with:

<*"BOUND TO BE!"

When STOPPER hits a variable value breakpoint, it turns the breakpoint OFF. Then it freezes the program just as if CLEAR had been pressed. The "*" hold indicator will appear in the lower right corner. Now, if you press BREAK, STOPPER will print out the number of the line that caused the variable value to change. If the line contains many statements, you may display exactly the one that caused the break by using the 'P' command, as described below.

Variable breakpoints require a bit of care in their use. First consider the problem of breaking when A% becomes unequal to 12, by entering the command '<A%*12'. Obviously, when the first statement of the program is about to be executed, A% will be 0 - causing a break. To avoid this you must not activate the breakpoint until the program has set A% to 12. The simplest way to accomplish this is to first set a breakpoint at A%=12. When this is hit, Break and type '<*12' to reset the breakpoint for the inequality. Then use 'C' to continue.

Another place to be careful is in the use of subscripted variables. If the number of subscripts is three or less, and if each of the subscripts you specify is 10 or less, then it is possible to set up the breakpoint before running the program. The only remaining problem will be if, somewhere in the program, the array in question is used in a DIM statement. This will cause a "Redimensioned Array" error when hit with the breakpoint active. The solution here is to set a line breakpoint right after the DIM statement; when the "*" hold indicator appears, BREAK and then type 'B' to reactivate the variable break, and 'C' to continue. Incidentally, you will also get this "Redimensioned Array error if you are tracing such a variable when the DIM statement is hit.

If the number of subscripts is more than three, or if any of them exceed 10, you will have to wait until the array has been DIMensioned by the program before specifying your variable break. Use a line number breakpoint to accomplish this.

Finally there is the question of exactness in single and double precision variable value comparisons (integers and strings are always

```
100 FOR N=1 TO 50 : NEXT N : X=SIN(V) : etc.
```

You may want to break on the X=SIN(V) statement, but can't because it doesn't have a line number. And it is not practical to step (with '+') through the fifty FOR-NEXT loops that precede this statement. One solution would be to set a line breakpoint at 100 with '<100'. When it's hit, step into the first execution of the FOR-NEXT loop with '+'. Now BREAK, set a breakpoint at N=51 with '<N%=51', and continue with 'C'. At the break you will be positioned just before X=SIN(V), as desired.

## TRACE FACILITIES

STOPPER provides some enhancements to Basic's program trace capability. You may now trace line numbers or variable values, and you may direct either or both traces to screen or printer. The T, Y, U, I, O, and G commands are used for this, as mentioned in the command list above. 'T' is just like TRON, except that the line numbers are shown in the bottom right-hand corner of the screen. 'Y' sends these numbers to the printer, in one column. If you want a more compact printout, perhaps for a long trace, 'U' causes a line number printout in eight columns. If a variable has been defined via '<variable', you may direct a continual trace of its value to the bottom line of the screen (with 'I') or to the printer (with 'O'). This screen display will be limited to 22 characters total. The printer output will comprise the full name, and up to 24 characters of the variable's value. Note that the 'O' and 'U' commands may not be used simultaneously. Also, you should not use 'Y', 'U', or 'O' if you don't have a printer! To maintain compatibility with the widest possible variety of printers, some of which do not provide a "ready" signal, STOPPER does not check for printer availability. Just as with an LLIST command, your system will hang if you use these commands without a printer.

Sometimes, especially within delay loops, you will want to temporarily disable all printer tracing. This may be done by pressing SHIFT. When you release SHIFT, any printer tracing will resume. This key was chosen for convenience; obviously, you must press SHIFT to use the '?' function during execution, meaning that you will briefly lose any printer traces. This should not be a problem.

The 'G' command clears all traces.

## STOPPER AND SPEED

Whenever STOPPER is ON, it will affect to some degree the speed of your program. If no breakpoint is active, the slowdown is about 5%. If a line number breakpoint is set, the slowdown increases to about 10%. Variable value breakpoints will consume more time, especially if the current value of the variable is close to the specified break value. Figure on a 25% slowdown, roughly, except in the case of double precision. DP math is so slow to begin with that the additional delay due to STOPPER is not perceptible. If you are directing a trace of variable value to the screen (the 'I' command), an overall slowdown of about 50% will result. Of course, if you direct any trace to your printer ('Y', 'U', or 'O' commands), program execution will be reduced to printer speed.

## MISCELLANEOUS

Some of the commands provided (such as 'C', 'R', and 'L') are for convenience only, and are not really important to the use of STOPPER. They simply invoke their associated Basic functions.

STOPPER will respond to invalid entries or impossible requests with the "** NO GOOD! **" message.

The commands '<=' and '<*' are not valid unless followed by a value.

Whenever NEW, LOAD, or CLOAD is entered, STOPPER is reinitialized - just as if the command 'J' had been used.

STOPPER talks to the printer via the standard DCB. Any printer that works with Basic should work with STOPPER.

If you press BREAK while waiting at an INPUT statement, STOPPER will think that the program halted on an error. This means that stepping via '+' and '>' will not be allowed.

For those of you who wish to create custom copies of STOPPER once it has been relocated: the Start address is the same as the 'eeeee' Entry address mentioned earlier. The Stop address will vary with the version of STOPPER that you have, but may always be determined in this way: load STOPPER at the very top of memory, and subtract the Entry address that results from 65536. This will give you STOPPER's total length, including all workspaces, once relocated. Add this to the current Start address to get the current Stop address.

## STOPPER AND THE SYSTEM

Because it does not patch into the keyboard DCB, STOPPER will not interfere with standard keyboard utilities. And it should be compatible with any DOS. I have found that under DOSPLUS Basic, the TRON single-stepping feature of that Basic will not work when STOPPER is on. It will work if you turn STOPPER off. STOPPER's command keys have been carefully chosen to avoid conflict with this and other new DOSs. That is why so many of these commands had to be shifted keys. Remember that if you are entering a shifted key command, you do not need to release SHIFT before pressing ENTER.

STOPPER appears to be compatible with many of the aftermarket Basic enhancement packages, but this cannot be guaranteed. I would appreciate hearing of your experiences with STOPPER in combination with any other products.

## TRANSFERRING STOPPER TO DISK

If you received STOPPER on tape, and wish to run it from a disk system, you must create a /CMD file on disk. You may do this directly using the TRSDOS standard TAPEDISK utility, Apparat's LMOFFSET, Misosys' CMDFILE, Acorn's FLEXL, or any of a number of monitor programs. On tape, STOPPER is a plain-vanilla SYSTEM file with filename 'STOPPR'. One way to create a /CMD file is to use TAPEDISK as follows:

From DOS READY, execute TAPEDISK. Make sure that drive 0 has at least 4 grans of free space on its disk. At the Tapedisk '?' prompt, enter 'C' to start loading the tape. When this is complete, enter:

```
F STOPPER/CMD:0 5500 6500 5500
```

Note the required blanks following 'F' and the filespec!

## CREDITS

I could not have begun writing this program without the guidance of David Cornell's excellent article on Label Basic (80-Micro, Dec. 1980, pp160-184). I would have been unable to finish without continual reference to James Farvour's superb book "Microsoft Basic Decoded". Bruce Hansen's powerful TASMON monitor was used extensively in the program debugging, and STOPPER is relocatable thanks to Jack Decker's articles on that subject (TAS Issue Nos. 6 and 10). All programming was done under Misosys' EDAS Editor Assembler, an indispensible tool for large files. Thanks to all these people.

# STOPPER

## The BASIC Breakpointer

### by Roxton Baker, author of "TRAKCESS"

Now: debug your BASIC programs with ease. NO MORE editing STOP statements in and out, and NO MORE losing your variables! NO MORE wondering how a variable got changed, or where in the line an error occurred. STOPPER will save you hours of mental frustration by adding commands like these to BASIC:

SET BREAKPOINTS. Specify which line number and how many times it should be executed before breaking.

RESTORE BREAKPOINTS. Restore previous breakpoints with the same parameters, or specify new hit counts.

DEFINE VARIABLES OF INTEREST. Specify certain variables whose value will be displayed by the ? command. Also permits tracing to screen or printer.

SHOW STATUS of breakpoints.

CLEAR active breakpoints.

SLOW STEP or SINGLE STEP with spacebar.

TRON to printer in one or eight columns.

And there's much more! STOPPER is the most powerful BASIC programming tool available. Completely relocatable; requires just 3.1 K of memory. Works under Level II BASIC or Disk BASIC.