# AT Probe

# AT Source Probe

**Last revised 5/10/88**

# AT Probe

# AT Source Probe

# Last revised 5/10/88

ATRON
Saratoga Office Center, 12950 Saratoga Ave.
Saratoga, CA 95070 * (408) 253-5933

# CHAPTER 1
# INSTALLING THE AT PROBE

# INTRODUCTION

This chapter contains the set up and installation procedures for the AT PROBE. There are three major steps in the installation process:

1. Unpacking the AT PROBE
2. Installing the AT PROBE
3. Running the AT PROBE diagnostics.

## UNPACKING THE AT PROBE

Carefully unpack your AT PROBE and inspect the AT PROBE plug and printed circuit card for damage. If they are damaged, please contact the dealer from which you purchased the AT PROBE. The AT PROBE package should contain the following components:

1. AT PROBE printed circuit card and attached umbilical cable
2. AT PROBE floppy disks
3. AT PROBE STOP/RESET switch box
4. AT PROBE 80287 extraction tool
5. Plastic card guide
6. Spare socket

## SETTING THE BASE ADDRESS OF AT PROBE MEMORY

The AT PROBE contains 1 megabyte of RAM memory which is used to store the AT PROBE software and symbol table. This memory is write protected and will prevent the AT PROBE software from being modified during a debug session from erroneous action by your program. This memory is above the lowest 1 megabyte of system memory. You can select the base address of the AT PROBE memory via the jumpers on the upper right corner of the AT PROBE. The default for this memory is 1mb at D00000. Insure that this does not conflict with other resources which are in the system. There will not normally be a resource conflict with most standard equipment. If a conflict exists, the address of AT PROBE memory can be changed to 100000 by moving the right hand jumper (shown below) from **H** to **L**. The left hand jumper must remain in the right position at all times.

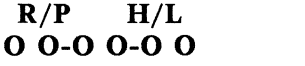



Figure 1-1. Layout of AT PROBE jumpers

## INSTALLING THE AT PROBE

To install the AT PROBE follow these steps:

1. Disconnect power and remove the top cover of your computer.
2. Set the base address of the AT PROBE as described above.
3. If required, remove boards and associated mechanical hardware in order to expose the 80287 math co-processor.
4. Find the 80287 slot and note the location of pin 1.
5. If the 80287 is installed, using the inclosed tool, carefully pry it from its socket and plug it into the AT PROBE buffer assembly. Note that pin 1 of the cpu should match the red mark on the buffer assembly.
6. Plug the buffer assembly into the co-processor socket. Insure that the cables on the buffer assembly align with the notch in the socket. (1)
7. Plug the AT PROBE hardware board into any 16 bit (dual connector) slot in the motherboard.
8. Replace previously removed boards.
9. If you want to drive AT PROBE from an external console, connect the AT PROBE serial cable to the PROBE serial port.
10. Plug the Crash Recovery Switch Box into the AT PROBE.
11. Examine the AT PROBE distribution diskette for the file README.DOC, which will describe any changes in the AT PROBE.

(1) In some systems (Compaq 286 for example), the 80287 is under other plug in cards. A card can not be plugged in over the AT PROBE because of the height. Atron supplies a special low profile plug at no charge for systems where the 80287 is under other plug-in cards. Contact Atron if you need one of these special low profile plugs.

**Other notes**

1. If you want to remove the AT PROBE hardware board from your computer without removing the buffer assembly, disconnect the cables from the buffer assembly and leave it plugged into your computer. Additional buffer assemblies can be purchased from Atron. Note that the cable is keyed to insure proper assembly.
2. As shipped from Atron, the buffer is shipped with an additional socket. This socket is provided to protect pins on the buffer assembly from accidental breakage. If any pins are broken from the buffer assembly, the entire assembly must be replaced. In some computers, limited space may constrain the use of this additional socket.

40pin chip carrier
AUGATE 540-AG70D

**FIG 1-1 REMOVING THE 80287**

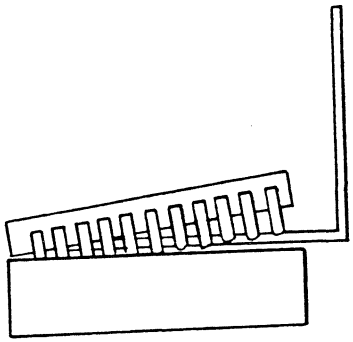
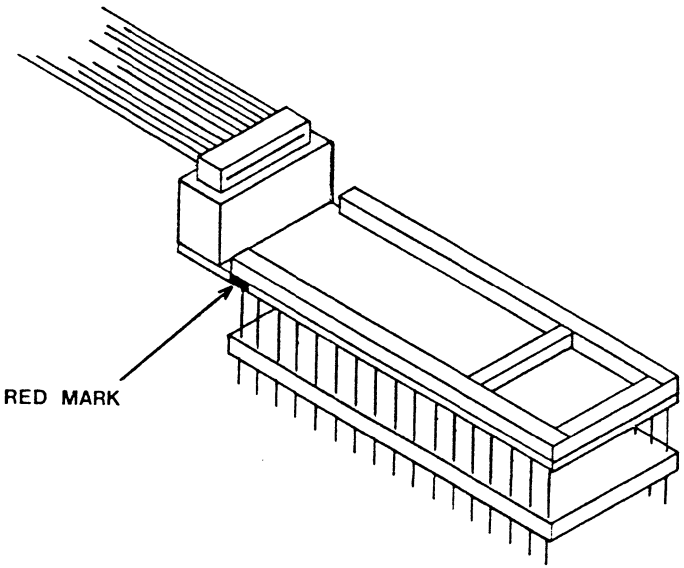


**FIG 1-2 PROBE BUFFER ASSEMBLY**



## RUNNING AT PROBE DIAGNOSTICS

The diagnostic program will test the functionality of the AT PROBE and print out a PASS/FAIL message for each diagnostic test. You can now test the AT PROBE by inserting the PROBE diskette and starting the diagnostic program with the command:

**ATPDIA**

When ATPDIA is invoked, it will output the following sign-on message and ask for the base address of the board before it executes the diagnostics.

```
Atron's AT Probe Protected Mode Diagnostics Version 03/25/86
(C)Copyright Atron Corp. 1986
Enter board base address (1 or [D]):
```

When the board base address is entered, the following tests will be performed: (2)

```
Board base address is D00000
Bank select test.....Passed.
    Total memory installed: 10H x 64K
Data RAM test (Test time approx. 1:30 mm:ss).....Passed.
Break RAM test.....Passed.
Trace RAM test.....Passed.
RAM Write Protect test.....Passed.
Press the STOP button.....Passed.
Timer test.....Passed.
Timer overflow test.....Passed.
Break register test.....Passed.
Trace register test.....Passed.
Press the RESET button to test reset
or any key to return to DOS.
```

(2) Some versions of the Phoenix (R) BIOS will cause the system to hang on the bank select test. This is because of the sequence and number of instructions used to switch from real mode to protected mode. Should this occur, use the /HP version of the AT PROBE software (atpdiahp, atprbhp, etc.).

If any of the tests fail, messages will be printed giving more detail on the nature of the failure. The AT PROBE is ready for operation only if it has passed all diagnostics. A failure in a diagnostic can be caused by several factors, including hardware conflicts. One such common conflict is cause by EGA adaptors which use "auto-switching" to adjust the video mode: these display cards trigger Non-Maskable Interrupts about 30 times a second. Some network cards can cause similar difficulties. As NMI is used by the AT PROBE breakpoint mechanism, this can cause a multitude of problems. The following are some suggestions to try for different failures.

## WHAT TO DO WHEN THE ATPDIA TEST FAILS

**Bank Select Test fails:**

Ensure that the AT PROBE memory does not conflict with memory on other boards in the system. If you do not know if there is memory on the other boards which may conflict with AT PROBE, try removing all boards which are not part of a standard AT system. A standard AT system consists of the motherboard, monochrome board and/or color graphics adapter board, and disk controller board. If AT PROBE hangs on this test, try using ATPDIAHP.

**Stop Button Fails:**

If the Stop Button Test fails, ensure that the switch box connecter was not offset when plugged in (even though it can go in either direction).

**Break Register or Trace Register Test Fails:**

If the diagnostic fails the Break Register Test or Trace Register Test, the cause may be that the buffer assembly is not properly plugged in to the socket, a pin is bent or broken, or the buffer assembly may be damaged in some other way. Speed and/or wait states are very common causes of these tests failing: AT PROBE requires a minimum of one wait state, and is rated at a speed of 8mhz. Other typical causes of the Break Register Test or Trace Register Test failing include Auto-Switching EGA boards, and the 3 1/2" floppy drive device driver which comes with DOS.

**Other Test Failures:**

If any other diagnostics fail or you can not get the AT PROBE to work, contact Atron technical support at (408) 253-5933.

**OTHER PROBLEMS:**

**System won't boot:**

Verify that the AT PROBE plug is not offset in the 80287 socket.

Verify that your system is compatible with the AT PROBE.

If any other diagnostic fails or you cannot get the AT PROBE to work, contact ATRON technical support.

# CHAPTER 2
# QUICK START

# QUICK START

This chapter shows you how to start the PROBE and describes the user interface. It describes the start up configuration file, available AT PROBE consoles and versions of AT PROBE software. It also provides a brief overview of AT PROBE commands.

# HOW TO START AT PROBE

To start the AT PROBE (1), enter:

**ATPROBE [.exefilename[.mapextension] [programparameters]]**

To start the SOURCE PROBE, enter:

**ATSOURCE [.exefilename[.mapextension][programparameters]]**

The optional [] specifications when PROBE or SOURCE are invoked have the following effects:

| Specification | Result |
|---|---|
| .exefile | Loads program from file |
| .mapextension | Loads symbol table from file (2) |
| programparameter | Passes command line to loaded program |

**Example:**

**atsource \exe\demo.exe.map**

Loads \exe\demo.exe as the program to be debugged with symbols from \exe\demo.map

**Example:**

**atsource \obj\demo.exe foo.in foo.out**

Loads \obj\demo.exe as the program to be debugged with symbols from \obj\demo.exe. The program will be passed a command line of "foo.in foo.out".

(1) The AT PROBE software must either be in the current directory or a directory referenced by the "PATH" environmental variable. If the AT PROBE loader is unable to locate P2.EXE (PROBE) or S2.EXE (source PROBE) you will be prompted for the location of that file.

(2) There must be no intervening spaces between ".exe" and ".map".

# LOADING USING INITIALIZATION FILES

The command line may also include the initialization file to be loaded, instead of a program. See the File command in chapter 5 for more information on initializations.

**Example:**

**atsource /i \exe\demo.ini**

Loads initialization information from "\exe\demo.ini."

**Example:**

**atsource \exe\demo.exe.map /i demo.ini** (3)

Will pass "/i demo.ini" as command line to demo.exe. It will not load initializations.

**Example:**

**atsource /i demo.ini \exe\demo.exe.map**

Will load initialization from "demo.ini" and will only load the program indicated in the initialization file. The \exe\demo.exe.map is ignored.

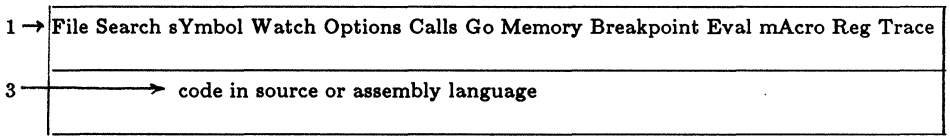
(3) The program and map file load and init file load are mutually exclusive events and cannot be combined into a single invocation.

# AT PROBE'S CONFIGURATION FILE

When invoked, AT PROBE and AT SOURCE PROBE look for a file named PROBE.CNF. If found, configuration parameters for PROBE's hardware base address, screen color, etc. are selected from this file. The default base address for AT PROBE memory is D00000 hex and is defined with the line **"ADDR=D00000"**. See Appendix D for a complete description of these configuration parameters. If PROBE.CNF does not exist, then standard defaults described in Appendix D are used. These standard defaults are usable for most systems. The file PROBE.CNF contains simple ASCII text and can be changed with a common text editor.

# THE CODE SCREEN

When AT PROBE is started, the screen looks like this:




1 The top of the screen contains the MENU BAR with a list of commands. A command is selected by typing its capital character (which is the first character for all except the sYmbol and the mAcro command).

2 Once a command from the MENU BAR is selected, a MENU BOX hangs down from the MENU BAR to let you select a subcommand. The subcommand is also selected by typing a single key or using the down arrow key to move a highlight to the selected subcommand and then typing the <Enter> key.

3 This part of the screen is called the CODE SCREEN and it appears immediately below the MENU BAR. It is the default screen when no other AT PROBE command has been invoked. The CODE SCREEN displays the program if it has been loaded with the [initfile] when AT PROBE was invoked, with the File Program-Load command or the File Initializations command. If no application has been loaded, code is displayed at the current CS:IP. The CODE SCREEN displays code in either assembly language or Source depending upon state of F3 key. The instruction at the current cs:ip is marked with "=" after the address and is dark blue on a color monitor. While in the Code Screen, use the paging and cursor keys to move the inverse video block around the code in assembly or source. Functions keys as described next operate in the Code Screen.

# FUNCTION KEYS

The following function keys operate with the CODE SCREEN as well as several other screens. They closely resemble the use of Function keys found in the Microsoft CodeView (R) debugger.

F1 Help key. Brings up a hierarchy of Help Screens. At each command and subcommand, pressing F1 will bring up a screen of text about that command.

F2 Toggles register window off and on. Register window appears vertically on right side of screen and operates like a Watch Window (described in the Watch command).

F3 Switches between source and assembly mode in Code Screen and Calls command.

F4 Displays applications screen. Press any key to return to the AT PROBE screen. (This does not apply to any console except the local console).

F5 Start program execution and set a sticky breakpoint on the instruction at the highlighted line, adding an execute breakpoint to next available breakpoint in breakpoint screen and activating it. Shows breakpoint number in highlight to the right edge of the screen on the line of the instruction. When the instruction is displayed in the future, the breakpoint is displayed. All other active sticky breakpoints are still active. When breakpoint is detected, control returns to the Command Menu.

F6 When data is being entered into a DIALOG BOX, F6 toggles the effect of the cursor keys between two states:

1. The cursor is moved left or right between data entry fields in the DIALOG BOX (default), or the inverse block code screen.

2. Moves the cursor left or right within the current DIALOG BOX data entry field.

F7 Sets a temporary breakpoint on the line with the highlight, executes to that line and then removes the breakpoint. All other active sticky breakpoints are still active. When breakpoint is detected, control returns to the Command Menu.

F8 Executes a single step of the next instruction, steps into procedures and interrupts. In order to single step in source code, first select source to be displayed on the Code Screen by pressing the F3 key.

F9 Sets or clears an execution breakpoint on instruction at highlight. Setting a breakpoint adds and activates a sticky breakpoint with the next available breakpoint number. Clearing deletes the breakpoint from breakpoint screen. Shows breakpoint number in highlight to the right edge of the screen on the line of the instruction. When the instruction is displayed in the future, the breakpoint is displayed.

F10 Executes a Program step. If the source line or instruction contains a function, procedure, or interrupt, the AT PROBE runs real time until it returns to the instruction after the currently highlighted instruction.

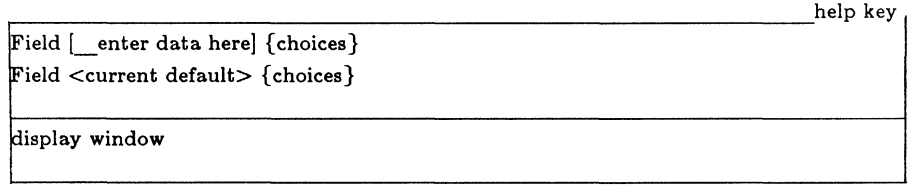
Alt-F10 This key lets you start a Copy Paste process. Press this key and the cursor becomes a solid block to the left center of the screen. Move the block with the cursor keys and press <Alt-F10> again to anchor the block. Use the cursor keys again to highlight a block of text. The block of text can cover more than one line. Press any other Alt-Key which stores the highlighted text as an ASCII string in an Alt-Key which becomes a macro. If the macro is already defined, then the new entry replaces the old definition. Pressing this Alt-key will recall the ASCII string it stores.

Note: The Function keys F5, F7, and F9 operate in the Code Screen, File View, Unassemble, sYmbol display change, Search, and Calls commands. F3 works in the main Calls command.

# DIALOG BOXES

## Entering parameters into DIALOG BOXES

When a command is invoked from the MENU BOX, a Dialog Box appears to provide further input to the command.




1 A blinking underscore cursor appears in the DIALOG BOX in a [field]. AT PROBE waits for your data entry into this field or you can type the <enter> key to take the default.

2 The other <fields> in the dialog box show you their current or default settings. A field may have a list of {choices} from which you can select by typing the first capital letter of the choice.

3 If there is a highlight field in the DISPLAY WINDOW while the blinking cursor is in the DIALOG BOX, then the contents of the DIALOG BOX are transferred to the highlight field when you type <enter>. If an expression is typed into the DIALOG BOX, it is evaluated before the value is put into the highlighted field. This lets you construct and edit expressions in the DIALOG BOX and have the results go to the highlight field in the DISPLAY WINDOW. A maximum of 255 characters can be entered into the DIALOG BOX (the characters scroll horizontally within the DIALOG BOX). The expressions inside the DIALOG BOX can be edited with the EDIT KEYS. See EXAMPLES OF USING EDIT KEYS later in this chapter.

4 Some commands prompt you several times with DIALOG BOXES. In this case, after typing <enter> for each DIALOG BOX, the blinking cursor is transferred to the next DIALOG BOX. When all necessary data is entered for the command to operate, the command executes. Execution can mean:

Put data into the DISPLAY WINDOW
Put or get information from the system memory or disk files

5 While in a command, you can move to another DIALOG BOX manually by typing the <TAB> key. This lets you make new entries into these fields. Also, in some commands, you are not automatically prompted through all DIALOG BOXES. This is because some of the DIALOG BOXES are so rarely used, it would be annoying to be prompted for them during each command. These fields are, however, displayed in the formatted screen of the command, and you may reach them by typing the <TAB> key until the blinking cursor gets to them. These are sometimes called <TAB> TO fields.

6 For commands which have both a DIALOG BOX and a display area below the DIALOG BOX, there is typically only one "main" field in the DIALOG BOX. You may <TAB> to other fields and change the choices there, but the display area will not be updated until you <TAB> back to the "main" field. Also, if the command allows a highlighted cursor line to be moved up and down in the display, this cursor will only be active at the "main" DIALOG BOX field. The "main" fields varies from command to command.

# TERMINATING A COMMAND

Once in a command, you stay in the command until the <ESC> key is typed. This lets you make changes in the DIALOG BOX in a command for re-execution of the same command. ESC typed at any point will terminate a command and return control back to the previous screen. Any command execution which is in process but which has not already executed is canceled with the <ESC> key. Note however, that you stay in a command and may execute it several times before terminating it with the <ESC>. These changes which have already occurred are not undone with the <ESC>.

# EDITING COMMAND PARAMETERS

If the syntax entered into the DIALOG BOX field is not recognized, an error appears in an MESSAGE BOX describing the problem. In addition, the blinking cursor is placed just after the point in the DIALOG BOX where the syntax could not be recognized by AT PROBE. The data in the DIALOG BOX may be edited with the editing keys. These keys are described in the section titled WHAT THE EDIT KEYS DO.

# WHAT THE EDIT KEYS DO

**Keys affecting the display window**

*Cursor Keys (Up, Dn)*

Move the highlight field in the DISPLAY WINDOW, the MENU BAR, or move in the DIALOG BOX if not in the display window or menu bar.

*Cursor Keys (Left, Right)*

If the cursor is in a drop down menu, moves to the next drop down menu to the left/right. If the cursor is in a dialog box, moves the highlight field in the DISPLAY WINDOW or the cursor within the current field (depending upon state of F6 key).

*Ctrl Left and Ctrl Right*

While inputting data in a Dialog Box, deletes all characters from the cursor to the beginning or the end of the line.

*PgUp and PgDn*

For commands which can display additional screens of information in the DISPLAY WINDOW, these keys show the previous or following information associated with the command in process.

*Ctrl PgUp and Ctrl PgDn*

For commands which display files, real time trace data, or source files during single step in the DISPLAY WINDOW, these keys move to the start and end of the file. Also to first and last of what is being displayed (eg. symbols).

*<TAB>*

Move blinking cursor to the next DIALOG BOX for the command in process.

**Keys for editing data in the dialog box**

*<BACKSPACE>*

Move blinking cursor left one character and blank this character.

*HOME*

Move blinking cursor one space toward the beginning of DIALOG BOX but do not delete the characters. The cursor keys cannot be used for moving the blinking cursor in the DIALOG BOX since they are reserved for moving the highlight field in the DISPLAY WINDOW. If the code display is currently active, HOME will cause the screen to show line pointed to by the present CS:IP.

*END*

Move the cursor one space toward the end of the DIALOG BOX. The cursor keys cannot be used for moving the blinking cursor in the DIALOG BOX since they are reserved for moving the highlight field in the DISPLAY WINDOW.

*Ctrl HOME*

Move blinking cursor to the beginning of DIALOG BOX.

*Ctrl END*

Move the cursor to the end of the DIALOG BOX.

*DEL*

Delete the character above the blinking cursor.

**Keys terminating commands**

*ESC*

Terminates a command which is in the process of executing and returns to the previous screen.

*Ctrl Break*

Terminates current command in process which does not stop with the <ESC> key. Returns to the code screen.

# EXAMPLES OF EDITING COMMAND PARAMETERS

The EDIT KEYS can be used to make changes in the DIALOG BOX. The definition of these keys was described previously. Here are some examples of using the edit keys.

Start by using the Memory Display command. Type:

**M D**

The command prompts for startaddress in the DIALOG BOX. Type:

**MAINxMODULE.PROCEDURENAME**

Since no symbol matches this, an error message pops up. Type any key to pop down the error message. The blinking cursor is in the DIALOG BOX. Use the HOME and END keys to move the blinking cursor just under the x. Then type the DEL key. Now, to re-execute this command type <enter>. Alternately, the cursor keys can be used if the F6 toggle is set to move the cursor within the current field (4).

(4) See Function Keys, F6.

# TAB FIELDS

The AT PROBE commands typically have several options which are not normally changed. They may be changed occasionally, however, and therefore must be available when needed. These options are displayed on the screen as:

*Label:< >*

These fields contain default values in the < >. You may get to these fields to change the defaults by typing <TAB>. The default stays in the < > until you select then change it. The <TAB> fields are described for each command in Chapter 5.

# COMMAND SUMMARY

**File**

| | |
|---|---|
| Program Load | Load program, symbols with options |
| View | Display up to 10 files simultaneously |
| Initializations | Load PROBE initialization conditions |
| Quit | Return to DOS or run in the background |
| Log file | Log a debug session for later review |
| Revision | Display the current PROBE software version. |

**Search**

Search the file corresponding to the current cs:ip

**sYmbol**

sYmbols display/change
: Display, change or delete symbols

Default modulename
: Sets default modulename for symbols

Load module selections
: Load symbols from only from selection selected modules

Step source screen modules
: Source step only in selected modules

Module to file assignments
: Display/change source files assigned to program modulenames

**Watch**

| | |
|---|---|
| Define-edit | Define or edit a pop up Watch Window |
| Remove | Delete currently defined window |
| Load | Load a file of predefined windows |
| Save | Save current windows definitions to disk |

**Options**

Screen
: Selects screen switch and remote consoles

View operands during step
: Display operand contents during step

Mix source during step
: Show source code with assembly during step

| | |
|---|---|
| sYmbols displayed with code | Show symbols with code during step |
| Case sensitivity | Include case in symbol interpretation |
| sTep count | Number of steps to take during step |
| Interrupts | Controls interrupt system while in PROBE |
| Read after write verification | Verify changes made by PROBE memory commands |
| Function call linkage style | Specify call entry protocol |

**Calls** Display stack variables, return addresses

**Go** Start program and set non sticky breakpoints

**Memory**

| | |
|---|---|
| Display change | All non-float memory types |
| Io port | IO devices |
| Float display | All floating point types |
| Unassemble/assem | Unassemble or assemble code into memory |
| Block operation | Operate on memory in blocks |
| Variable | Supports complex C data types |

**Breakpoint**

| | |
|---|---|
| Define | Define breakpoints |
| Activate | Activate defined breakpoints |
| Inactivate | Inactivate defined breakpoints |
| Clear | Clear all breakpoint fields to default state |
| Set pass counter | Set breakpoint pass counter |

**Evaluate** Calculate expression and display in several bases

**Macro**

| | |
|---|---|
| Define | Define a new macro and assign it to an Alt-key |
| Edit | Change the definition of a current macro |
| Remove | Delete a currently defined macro. |
| Load | Load previously defined macros from file |
| Save | Save all currently defined macros to a disk file |

**Register**

Display 80286 processor registers, flags, and 80287 floating point registers and flags (if co-processor is present)

**Trace**

| | |
|---|---|
| Instructions | Display trace data with prefetch filtered |
| Unprocessed ins | Display trace with prefetch not filtered |
| Save-to-disk | Save data to disk |
| Raw data | Display trace data in hex format |
| DMA cycles | Toggle DMA cycles on/off in trace display |

# COPY AND PASTE

AT PROBE has a copy and paste feature which eliminates typing in the long addresses and symbolnames required during program debugging. When you are looking at information on one screen which you want to pick up and deposit into fields on another screen, you can use PROBE's copy and paste keys.

To copy information from a screen, start by typing the <Alt-F10> key. The highlight field in the DISPLAY WINDOW will shrink to the size of one character and will be positioned on the left side of the screen about half way down from the top. As usual, use the cursor keys to move this small highlight field to the start of the information you want to copy. Next, type <Alt-F10> again which anchors the highlight field at this location. Use the cursor keys again to spread the highlight field over the characters on the screen you want to copy. Next, you can deposit the highlighted information into any Alt-key.

The information stored in these function keys can now be pasted into any other screen. The location on any screen which has the blinking cursor can receive the information stored in the Alt-key by typing the key. The ASCII text stored in the function key is available each time the key is typed until the information in the key is changed.

# ERROR MESSAGES

When a command or command parameter is not recognized by AT PROBE, an error message "pops up" on the screen in a MESSAGE BOX to indicate the problem. Appendix A contains a summary of these error messages. To remove an error message from the display press any key.

# REMOTE CONSOLES

When AT PROBE is first started, command entry is done through the resident console. However, you have several choices for console IO with Options Screen. The simplest choice creates two virtual screens and isolates the AT PROBE screen from the application screen.

If you have two video controller boards in your system, you can move the AT PROBE display to the screen "Other" than the one which is the current default.

A third choice is to switch to an entirely separate CRT which you connect to the "Remote" or Com1 or Com2.

The description on how to connect a remote console and set up its configuration parameters is shown in Appendix E.

# FILES ON YOUR PROBE DISKETTES

There are several files on your PROBE product diskettes which may or may not be needed depending upon what you are doing. A list of these files and a description is given in Appendix B for each version of PROBE software. Only those used for "RUNNING" are required for the actual execution of PROBE software.

# VERSIONS OF PROBE SOFTWARE

This manual describes the standard versions of the AT PROBE and SOURCE PROBE software. Other versions of these software products are available which are optimized for specific applications. These versions are described by inserts to this manual.

# USING WILDCARD CHARACTERS WHEN ACCESSING FILENAMES

PROBE interprets file specifications in the same manner as DOS on the computer. A filespec is defined as:

**[drive] [path] [name of file]**

If not specified, the default drive/path is used. AT PROBE lets you use wildcard characters in any command which prompts you for a file name. If you cannot remember the name of the file you want to specify or you can only remember part of the file name, then use the wildcard character *. If you are familiar with DOS, the * works in exactly the same way as it does in the DIR command.

**Examples:**

To tell the AT PROBE command to display all filenames in the current directory, type:

***.***

To specify all files in the directory \MAIN\DEMOS type:

**\MAIN\DEMOS*.***

To specify all files in the current directory with a .HEX extension type:

***.hex**

To specify all files in the current directory which start with the letter A type:

**A*.***

When you are prompted for a file name in a Dialog Box, you can simply type "*" and a summary of files from the specified or default drive and directory is painted on the screen. Wildcards can be used to limit the number of displayed files. The highlight field in the Display Window can be positioned on and file name. Typing enter will select the highlighted file name.

# CHAPTER 3
# SYMBOLS AND VALUES

# CHAPTER 3
# SYMBOLS AND VALUES

This chapter defines values, addresses and how they are used in AT PROBE commands. Other command definitions which apply to all AT PROBE commands are also described. It then describes how to generate symbolic debugging information and how modulenames and symbolnames are interpreted as values and addresses in AT PROBE commands.

# VALUE

A value is a 32 bit quantity that can be represented by any of the items shown in table 3-1.

**Table 3-1**
**Examples of value definitions**

| Value represented as: | Examples |
|---|---|
| a symbol name (it's address) | MAIN |
| a 32 bit numeric hex constant | 12345678 |
| a 32 bit numeric decimal constant | 12345678T |
| a register name (it's contents) | AH (1) |
| an ASCII character in quotes | 'A' |
| a dereferenced memory location | (described later) |

(1) A value which matches a register name is interpreted as the register name. Register names are specified in the R command. If you want a hex value instead, precede the value with a 0.

# ADDRESS

An address is represented by:

segment expression:offset expression

# EXPRESSIONS

An expression is a value calculated by combining a series of values with operators.

**+ , - , * , / , ~, & , | , % ^**

Normal precedence of operators as defined in the C language is assumed: (*, / ,% ,&) are higher than (+, -, |) and evaluation proceeds left to right on operators with equal precedence. Precedence may be overridden by the use of parenthesis. Table 3-2 explains each operator and lists them in order or precedence.

**Table 3-2**
**Definition and precedence of operators**

| Operator | Definition |
|---|---|
| Highest precedence | |
| - | 2's complement |
| ~ | bitwise negation |
| Next highest | |
| * | multiplication |
| / | division |
| % | modulus (remainder) |
| & | bitwise and |
| Lowest precedence | |
| + | addition |
| - | subtraction |
| \| | bitwise inclusive or |
| ^ | bitwise exclusive or |

**Examples:**

Assuming the following values in memory, here are several expressions and their resulting values.

The symbol lvar represents a long (four byte wide) integer. The address of lvar is 1000:0 and its value is 20000000

Note that the symbol "lvar" when enclosed by brackets is dereferenced. See the section on "Dereferenced Memory" later in this chapter.

| Expression | Value |
|---|---|
| lvar | 00010000 |
| [lvar] | 20000000 |
| -[lvar] | E0000000 |
| ~[lvar] | DFFFFFFF |
| [lvar]*2 | 40000000 |
| [lvar]/2 | 10000000 |
| [lvar]%10 | 0 |
| [lvar]&30000000 | 20000000 |
| [lvar]+5 | 20000005 |
| [lvar]-5 | 1FFFFFFB |
| [lvar]\|FF | 2000000FF |
| [lvar]^F0000000 | D00000000 |

# RADIX

When you enter values into AT PROBE commands, they are interpreted as hex unless you specify another numeric base. Subscript the value with t for a base of ten (i.e. decimal). Put quotes around the value to make it an ASCII string.

**Example:**

```
10      ; ten hex or 16 decimal
10t     ; 10 decimal or A hex
'10'    ; ASCII string
```

# BOOLEAN EXPRESSIONS

Boolean expressions use boolean operators and result in one of two boolean values. The AT PROBE boolean operators result in a value of FFFFFFFF (or non zero) if the result is TRUE and a value of 00000000 if the result is FALSE. The boolean operators may be joined with the '&' and '|' operators for the boolean AND and OR functions.

**Table 3-3**
**Definition and precedence of boolean operators**

| Operator | Definition |
|---|---|
| < | less than (unsigned) |
| <s | less than (signed) |
| <= | less than or equal (unsigned) |
| <=s | less than or equal (signed) |
| = | is equal to |
| == | is equal to |
| <> | is not equal to |
| != | is not equal to |
| >= | greater than or equal (unsigned) |
| >=s | greater than or equal (signed) |
| > | greater than (unsigned) |
| >s | greater than (signed) |

**Examples:**

Here are some boolean expressions and a description of how the operators work.

| Boolean Expression | Description |
|---|---|
| AX>BX | If the contents of register AX is greater than register BX, then the result is true. |
| [123]=[456] | If the contents of memory at address DS:123 are equal to the contents of |

| | |
|---|---|
| | memory at address DS:456, then the result is true. |
| offset(ProcedureA) <IP | If the offset of the symbol ProcedureA is less than the register IP, then the result is true. (See "Seg & Offset Operators in this chapter) |
| [Celsius] <10t | If the contents of the variable Celsius is less than decimal 10, then the result is true. |

# DEREFERENCED MEMORY

The contents of a memory location may be used as a value in an expression. This is commonly referred to as dereferencing memory. The value is pointed to by an address expression which is defined as follows:

[address]size

The size may be omitted or may be:

| Size | Definition |
|---|---|
| B | use the byte at the specified address. |
| W | use the word (16 bits) at address (Default). |
| D | use double word (32 bits) at the specified address. |
| P | use 32 bit pointer at specified address |
| E | use 48 bit pointer at specified address |
| S | sign extend a memory access to double word "S" may be used om combination with "B", "W", or "D". "BS" means sign-extended byte to double word; "WS" means sign-extended word to double; "DS" is the same as "D". |

**Examples:**

Assuming the following values in memory, here are some address expressions and their resulting values as interpreted by AT PROBE.

The symbol svar is a short (16 bit) integer
The address of svar is 2000:0
The bytes of memory at 2000:0 are AA, BB, CC, DD
The bytes of memory at 2000:5 are 81, 22, 33, 44
The bytes of memory at 4433:2281 are EE, FF, 77, 88
The bytes of memory at 4433:2291 are 99, 00, 55, 66
The bytes of memory at DS:2281 are 00, 22, 44, 88

| Address | Expression value (in hex) |
|---|---|
| svar+5 | 20005 |
| [svar+5]b | 0081 |
| [svar+5]bs | ffffff81 |
| [svar+5] | 2281 |
| [svar+5]d | 44332281 |
| [[svar+5]p]b | EE |
| [[svar+5]p] | FFEE |
| [[svar+5]p]d | 8877FFEE |
| [[svar+5]p+10]b | 99 |
| [[svar+5]p+10] | 0099 |
| [[svar+5]p+10]d | 66550099 |
| [[svar+5]]d | 88443300 (2) |

(2) The result of the first memory dereference is a 16-bit number (2211), not a complete address. To dereference that, the AT PROBE will assume a default segment of DS to make the address it needs. The only time a memory dereference yields an address is when the expression inside is of type "far pointer" or if the "[]p" or "[]e" notation is used.

## SEG & OFFSET OPERATORS

The following operators let you extract the segment and offset from an address (3). Note that the () are required.

**seg**(address)
**offset**(address)

**Examples:**

The address of "x" is 4ECB:FFFF

**seg**(x) is 4ECB
**offset**(x) is FFFF

(3) When using the seg operator in a boolean expression, precede it with a space so that the "s" of "seg" will not be construed as a "signed" boolean operator (See "Definition and Precedence of Boolean Operators).

## END-OF-FUNCTION OPERATOR

When you want to set a breakpoint over the entire range of a function, it is useful to know the end address of the function. If a function was compiled and loaded into the AT PROBE with debug information, the End expression operator will return the address of the last byte in the function. The address expression can be a function name or any expression which evaluates as an address within the range of a function. The operator has the following form:

**end**(address expression) (4)

The End operator can also be used as a convenient way of executing to the end of the current function. If the current CS:IP is within a function with debug information, then a temporary breakpoint of "end(cs:ip)" or just "end(ip)" can be entered in the DIALOG BOX of the GO command.

### Examples of using the end operator

To set a Fetch breakpoint over the whole of function MAIN, use the following range in the Breakpoint DIALOG BOX:

Address: <main > to <**end**(main) >

To execute to the end of the current function, type:

**end**(cs:ip) (5)

(4) Note the use of "()"
(5) This assumes that the last byte of the function is a "RET" instruction.

## SIZEOF OPERATOR

The Sizeof operator returns the storage size for a symbol or function length if the symbol is in a file compiled with debug information. The default length is two bytes for non-typed data. The operator has the following form (Note the use of "()"):

**sizeof**(symbol)

### Example of using the sizeof operator

To return the size of the data structure D_STRUCT enter from the Eval dialog box:

**sizeof**(D_STRUCT)

To return the size of the function main() enter:

**sizeof**(main)

# USING SYMBOLS

AT PROBE allows you to use the symbolic information from your program during debugging instead of absolute numbers. The symbolic debugging information which is generated depends upon the compiler vendor, version number, compiler controls, and linker controls. Appendix C lists these items for different compiler vendors. Refer to this appendix for more details on how to generate symbolic debugging information. The symbol table information is loaded into AT PROBE from the .exe file or through the .map file. AT PROBE puts the symbol table into its on-board memory where it is hidden and write protected. Other parts of this manual describe important information regarding symbols. This information is not duplicated in this Chapter but here is a summary of where to find this information.

| Description: | Location |
|---|---|
| Single step source code | Function keys: Ch 2 |
| Loading symbols into PROBE | File Program load: Ch 5 |
| Selectively loading symbols | sYmbol Source modules: Ch 5 |
| Defining symbols on line | sYmbol Display/change: Ch 5 |
| Deleting symbols | sYmbol Display/change: Ch 5 |
| Ignoring case in symbols | Option Case sensitive: Ch 5 |
| Default modulename prefix | sYmbol Default modulename: Ch 5 |
| Indirectly referencing symbols | This Chapter |

## SCOPE OF SYMBOLS

A symbol is a value which can have both a segment and an offset. A symbol can be used in an expression at any place a value is expected. Symbols can have a scope. AT PROBE interprets the scope of symbols in the same way as the C language (6). Some definitions are shown here for reference:

| Symbol scope | Description |
|---|---|
| EXTERNAL or PUBLIC | Visible to all modules. This symbol has a fixed address when the program is loaded. |
| LOCAL | Visible only to the declaring procedure. This symbol is stack based. |
| STATIC | Visible to module after point of declaration. This symbol has a fixed address when the program is loaded. |

Note that AT PROBE provides a more complete means of referencing symbols than is available to the programming language itself. This is necessitated by the need to refer to symbols with different scopes but identical names. For example, if the current function contains the LOCAL variable **foo** which has the same name as the EXTERNAL variable **foo**, referencing **foo** will return the LOCAL variable. In this circumstance, to reference the EXTERNAL variable **foo**, it is necessary to use a leading backslash '\' (**\foo**). If the LOCAL variable **foo** did not exist, an attempt to reference **foo** would cause the EXTERNAL variable to be returned. A partial list of syntax which can be used to reference various symbols follows (7):

(6) A symbol is searched for in the current function first, and then, if the variable is not local it is searched for globally.

(7) This list does not consider such complex combinations as referencing several STATIC variables which have the same name but are in different modules.

If the symbol is stored in AT PROBE's symbol table is EXTERNAL, and there is no LOCAL variable with the same name, it is referenced as:

**symbolname or**
**.symbolname or**
**\symbolname or**
**\.symbolname or**
**..modulename.symbolname**

If the symbol is stored in AT PROBE's symbol table is EXTERNAL, and a LOCAL variable with the same name exists, the EXTERNAL symbol can be referenced as:

**\symbolname or**
**\.symbolname or**
**..modulename.symbolname**

If the symbol is LOCAL to the current function it is referenced as:

**symbolname or**
**.symbolname or**
**functionname.symbolname or**
**.functionname.symbolname or**
**\functionname.symbolname or**
**\.functionname.symbolname or**
**..modulename.functionname.symbolname**

If the symbol is LOCAL to an EXTERNAL function other than the current function it is referenced as:

**functionname.symbolname or**
**.functionname.symbolname or**
**\functionname.symbolname or**
**\.functionname.symbolname or**
**..modulename.functionname.symbolname**

If the symbol is LOCAL to a STATIC function other than the current function but it is in either the current module or the default module, it is referenced as:

**functionname.symbolname or**
**.functionname.symbolname or**
**..modulename.functionname.symbolname**

If the symbol is LOCAL to a STATIC function other than the current function and it is not in either the current module or the default module, it is referenced as:

**..modulename.functionname.symbolname**

If the symbol is STATIC and LOCAL to the current module or the default module it is referenced as:

**symbolname or**
**.symbolname or**
**..modulename.symbolname**

If the symbol is STATIC and local to a different module it is referenced as:

**..modulename.symbolname**

If the symbol is a linenumber for a high level language executable statement, which is in the current module then it is referenced as:

**..modulename#linenumber or**
**#linenumber**

If the symbol is a linenumber contained in a different module for a high level language executable statement, then it is referenced as:

**..modulename#linenumber**

Function names are treated the same as symbol names. A module is a single unit of compilation and has a modulename assigned to it by the compiler.

## REFERENCING SYMBOLS IN COMMANDS

When a symbol is used in an AT PROBE command, it is interpreted as an address. In this manual, most examples will use symbols instead of absolute numbers. Here are some notes regarding symbols used in commands. The use of "[]" and "." operators in an address are the same as defined in the language C. Note that the "[]" indicates dereferencing memory if the "[]" is not preceded by an arrayname. A symbol in C source code is interpreted by the compiler as a value. A symbol from the AT PROBE symbol table is interpreted as an address. Therefore, the @ operator in C is not needed in expressing an AT PROBE symbol as an address. The * operator in C is expressed in AT PROBE as dereferenced memory. The table below shows how these are used in an address expression.

## USING COMPLEX SYMBOL TYPES IN COMMANDS

| AT PROBE expression | C interpretation |
|---|---|
| arrayname [index] | array of values |
| arrayname [index].membername.membername.... | array of structure |
| structurename.membername.membername.... | structure member |
| structure.membername[index] | structure of arrays |
| [pointer] | pointername in C |
| [pointer]d | *pointer type in C |
| pointer | @pointer in C |
| [pointer].membername.membername.... | structure member |

## EXAMPLES OF USING SYMBOLS

In Chapter 5, the AT PROBE commands will use symbols in many examples. Here are a few examples of how symbols are used:

This expression references the PUBLIC symbol MAIN

```
MAIN
```

This expression references the procedure named IO assuming it is in the module named IOROUTINES.

```
..IOROUTINES.IO
```

This expression references the byte pointed to by the 32 bit pointer MEDIUMPOINTER.

```
[[MEDIUMPOINTER]P]B
```

## DEFAULT SYMBOL PREFIX

If no modulename is specified when you specify symbol or linenumber, then the current default prefix is used. After the symbol table is loaded, the default prefix is set to the first module encountered during symbol loading, provided that there was no default module already set. The default module is used as a last resort in symbol look ups: AT PROBE looks for the "current" module first. You can define a new default module with the sYmbol Default modulename command (See Chapter 5).

**Example:**

First, assume the default prefix is ..MAIN. The table below shows a symbol as you would specify it. AT PROBE exhaustively creates combinations of the default prefix and the symbol you type and tries to match them to a valid symbol in the symbol table.

| Symbol | AT PROBE searches for |
|---|---|
| FOO | local function .FOO |
| | global .FOO |
| | static ..<current module> .FOO |
| | static ..MAIN.FOO |
| PROC.FOO | global .PROC.FOO |
| | static ..<current module> .PROC.FOO |
| | static ..MAIN.PROC.FOO |
| ..<module>PROC.FOO | static ..<module>PROC.FOO |

# C SOURCE DEBUGGING

If you are running the SOURCE version of the AT PROBE software, then there are additional features available in AT PROBE.

1. The F3 function key lets you switch the CODE SCREEN between a display of assembly language and C source code.
2. The source code is intermixed with Assembly language when code is unassembled or the real time Trace data is displayed, providing that Options Mix source is set to on.

In order for the AT PROBE to get access to the source code, it must know where it is stored. The AT PROBE uses the modulenames found in the symbol table and assigns the source code in a file named:

modulename.ext (8)

You can view the source file name assignments to modulenames with the sYmbol Module-to-file-assignment command. If the source code is in a different file or a different path, you must change the assignments on this screen.

(8) "ext" is the source file extension specified in the .exe debug information, or the prefix specified in the File Program Load dialog box if .exe debug information is not used.

# CHAPTER 4 DEBUGGING APPLICATIONS

# INTRODUCTION

The commands are listed alphabetically in Chapter 5, COMMAND REFERENCE, and no attempt is made to duplicate the complete explanation of each command as it is being used in these examples. If the short explanation of the command is not sufficient in the example, please turn to Chapter 5, COMMAND REFERENCE, for more information.

This chapter contains two sections of application examples for using the AT PROBE:

**A SAMPLE DEBUGGING SESSION**
**ADVANCED DEBUGGING TECHNIQUES.**

The first section exercises many of the AT PROBE commands on an example program which has been included on the AT PROBE diskettes. The second section is more advanced and contains many real world debugging scenarios which have been used by previous AT PROBE users. The second section assumes a thorough understanding of the AT PROBE commands.

# A SAMPLE DEBUGGING SESSION

## OVERVIEW

The following is a brief tutorial designed to quickly familiarize you with the use of the AT PROBE. In the tutorial you will learn to:

**Load a file and its associated symbolic information.**

**Automatically initialize the AT PROBE using an init file.**

**Go to a function in a program.**

**Single step through a program.**

**Establish sequential and non-sequential breakpoints.**

**Trap on read/writes to local variables by other functions.**

**Use Trace to display run time information.**

**Examine the stack, registers, memory, variables, data structures, pointers, and arrays.**

**Step through a chain of pointers, and examine multiple instances of a local variable created through recursion.**

As this tutorial progresses, it will be necessary to input some lengthy sequences of keystrokes. For the maximum benefit it is suggested that you take your time, examining the result of each keystroke before entering the next one.

## THE PROGRAM

The program to be debugged uses four modules, three of C code linked to one of assembly language. The source, object and executable files for the program are included on your disk, so you can actually try the example in real time. This example is written to demonstrate various features of the Source PROBE. If you are using Pascal, Fortran, or some other language in your application, you will still find this tutorial useful from a procedural point of view. The program counts from 0 to 250 five times, and outputs the current value of the count variable in binary, octal, decimal, and hexadecimal. Next, it creates a binary tree which sorts an array of data structures, then outputs the contents of the array in sorted order by means of recursion. As delivered, the program has several bugs, one of which will prevent it from running to completion. As the tutorial progresses, these bugs will be identified, using the AT PROBE, and eliminated. The program source code is listed below:

```
/*    demo.c
*
*     A demo program for the AT PROBE
*/

#include <stdio.h>

  void print_nums(void), other_fng(void);
  extern void print(short *), demo3(int), demo4(void);
  short *zeroptr = (short *) 10;


main(argc,argv)
  int argc;
  char *argv[];
{
  short i;

  demo3(7);     /* clear the display, using attribute 7 */

  if (!--argc)
    for (i = 0; i < 5; ++i) {
      print_nums();
      other_fng();
    }

  demo4();
  exit(0);
}
```

```
void print__nums()
{
  short count;

  for (count = 0; count < 250; ++count) {
    if (count == 50)
      *zeroptr = 0xff;
    print (&count);
  }
  return;
}

void other__fng()
{
  puts("\nResetting counter\a\n");
}
```

---

```
/*    demo2.c
*
*     A second module to be linked in with demo.c
*/

#include <stdio.h>
#include <string.h>

  struct TYPES {
    short value;
    char hex[32];
    char decimal[32];
    char octal[32];
    char binary[32];
  };

  struct TYPES types;


/*  Print "val" in binary, octal, decimal and hexadecimal format
*/

void print(val)
  short *val;
{
  struct TYPES *tptr = &types;

  types.value = *val;
  itoa(*val,types.hex,16);
  itoa(*val,types.decimal,10);
  itoa(*val,types.octal,8);
  itoa(*val,types.binary,2);

  printf("%8sy =%4sq =%4st = %sh\n",
    types.binary,types.octal,types.decimal,types.hex);

  if (strlen(types.decimal) > 2)
```

```
    *val = 1;

  return;
}
```

---

```
;     demo3.asm
;
;     An example of mixed language programming.
;     This assembly module uses BIOS to clear the screen.
;
;     The object module links to demo.obj, demo2.obj, and demo4.obj
;
;     requires MASM 5.0
;

.radix 10
.model small
.data

if @codesize
  ATTRIBUTE equ 6 ;large code model uses a 2 word return address
else
  ATTRIBUTE equ 4 ;small code model uses a 1 word return address
endif

  PUBLIC __demo3

.code

__demo3  PROC

  PUSH   BP       ;save the old base pointer
  MOV    BP,SP    ;use to access passed parameters
  PUSH   SI       ;save register variables
  PUSH   DI

  MOV    AH,07h  ;function 7, scroll screen down
  MOV    AL,0    ;scroll the entire display
  MOV    CX,0    ;the upper left corner of the window is 0,0
  MOV    DH,24   ;the lower right row is 24
  MOV    DL,79    ;the lower right column is 79

  MOV    BH,BYTE PTR [BP+ATTRIBUTE]   ;get the passed attribute

  INT    10h      ;clear the display

  MOV    AH,02h  ;function 2, reset cursor position
  MOV    BH,0    ;select page 0
  MOV    DX,0    ;row 0, column 0

  INT    10h      ;reset the cursor

  POP    DI
  POP    SI
```

```
  POP   BP

  RET

__demo3   ENDP
end
```

---

```
/*   demo4.c
*
*    A fourth module to be linked in with demo.c
*
*    This module demonstrates the advanced use of the
*    variables command.
*/

#include <stdio.h>
#include <string.h>


  enum INTERESTS { Bicycling = 1,
                   Break_dancing,
                   Computers,
                   Classical_music,
                   Dancing,
                   Exercise,
                   Fine_art,
                   Flying,
                   Games,
                   Hang_gliding,
                   Hot_tubbing,
                   Literature,
                   Magic,
                   Motorcycle_racing,
                   Opera,
                   Philosophy,
                   Rock_n_roll,
                   Running,
                   Scuba_diving,
                   Skiing,
                   Theater,
                   Travel };


  typedef struct __NAMES {
    char first[25];
    char last[25];
  } NAMES;

  typedef struct __PERSON {
    NAMES names;
    char phone[16];
    enum INTERESTS hobbies[5];
  } PERSON;
```

```
typedef struct __TREE {
  struct __TREE *left;
  struct __TREE *right;
  PERSON *entry;
} TREENODE;


PERSON phone__book[] = {
      {
        "John","Cannon","(206) 524-4521",
          Scuba__diving,Exercise
      },{
        "Frank","Adkins","967-8186",
          Motorcycle__racing,Opera,Magic
      },{
        "Cathy","Mann","415 996-9898",
          Dancing,Running,Exercise,Fine__art
      },{
        "","Atron","(408) 253-5933",
      },{
        "H.D.","Graves","253-5433",
          Philosophy,Bicycling,Classical__music
      },{
        "Don","Hennings","(312) 121-3435",
          Rock__n__roll,Running,Skiing
      },{
        "","Information","??? 555-1212"
      },{
        "Stacy","Harkala","882-8715",
          Travel,Flying,Scuba__diving,Hot__tubbing
      },{
        "Otto","Marsh","494-3658",
          Games,Literature,Philosophy,Classical__music
      },{
        NULL
      }};

TREENODE __root, *root = &__root;


/*    Create a binary tree, sorting on last name, then output the contents
*     of that tree in sorted order.
*/

demo4()
{
  creat__tree();
  display__tree(root);
}


/*    Generate a binary tree which sorts "phone__book[]"
*     by first and last names.
*/
```

```
creat_tree()
{
  TREENODE *node = root, *tmpnode;
  int i;

  root->entry = &phone_book[0];

  for (i = 1; *phone_book[i].phone; ++i) {
    tmpnode = (TREENODE *) malloc(sizeof(TREENODE));
    memset(tmpnode,'\0',sizeof(TREENODE));
    node = root;
    for (;;)
      if (stricmp(phone_book[i].names.last,node->entry->names.last) < 0)
        if (!node->left) {
          node->left = tmpnode;
          break;
        } else
          node = node->left;
      else
        if (!node->right) {
          node->right = tmpnode;
          break;
        } else
          node = node->right;

    tmpnode->entry = &phone_book[i];
  }

}


/*   Display the array phone_book in alphabetical order by
*    means of a left prefix transversal of the tree.
*/

display_tree(node)
  TREENODE *node;
{
  if (node->left)
    display_tree(node->left);

  printf("\n%s %s\n%s\n",
       node->entry->names.first,
       node->entry->names.last,
       node->entry->phone);

  if (node->right)
     display_tree(node->right);
}
```

## COMPILING AND LINKING

AT PROBE supports Codeview symbolic information generated by the Microsoft C compiler version 4.0 or greater and the Microsoft Macro Assembler version 5.0 or greater. To include this symbolic information in the executable file, compile with the /Zi command line option and link using the /CO flag. For example, if you have the Microsoft C compiler version 5.0 and Masm version 5.0 you would want to compile and link our demonstration program with the following commands:

**masm /Zi /Mx demo3;**

**cl -Zi demo.c demo2.c demo3.obj demo4.c**

We have included an executable version of "demo.exe" so that you will not have to assemble, compile, and link the example program.

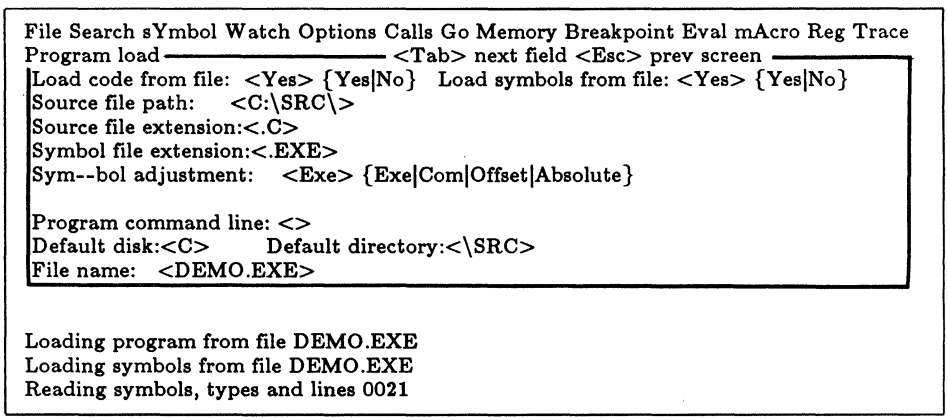
## STARTING AT PROBE

The AT PROBE can be started and loaded three different ways: by specifying the executable file name (with an optional map file parameter) at the command line (ATPROBE demo.exe.map); by loading the executable file from within the AT PROBE, or by loading AT PROBE with an initialization file (ATPROBE /i initfil.ini). Note that some machines with the Phoenix (R) BIOS will require the use of AT PROBE software with the "HP" suffix: ATPRBHP and ATSRCHP. If it was necessary to use the ATPDIAHP PROBE diagnostics, then "HP" suffix software is required.

Start the AT PROBE with the following command line:

**atsource demo.exe**

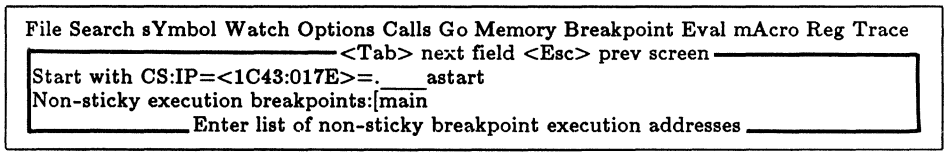
You will see the following screen:




To see a source level display, press the **<F3>** key.

## GO

It is often useful to execute past the run-time initialization code to the first C instruction. If you are looking at a source level display, press **<F8>** to single step into the main() procedure. An alternate method would be to **Go** to main(): this is accomplished with the keystrokes **G main <CR>**, where **G** brings down the Go menu, main enters the address of main() as non-sticky breakpoint, and **<CR>** represents the Carriage-Return (enter) key.



## SINGLE STEPPING

Press **<F3>** to switch to assembly mode. You will see the following display:

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                                  PgUp/Dn, Arrows move highlight
14.   int argc;
1C13:0010=PUSH   BP
 ^ ^ ^ ^        0000
1C13:0011 MOV    BP,SP
1C13:0013 MOV    AX,0002
1C13:0016 CALL   04CC            ; .___chkstk
19.   demo3(7);   /* clear the display, using attribute 7 */
1C13:0019 MOV    AX,0007
1C13:001C PUSH   AX
1C13:001D CALL   0136            ; .__demo3
1C13:0020 ADD    SP,0002
21.   if (!--argc)
1C13:0023 DEC    WORD PTR [BP+0004]  ; ..DEMO.main.argc
1C13:0026 JNZ    0040            ; ..DEMO#27
22.       for (i = 0; i < 3; ++i) {
1C13:0028 MOV    WORD PTR [BP+FFFE],0000 ; ..DEMO.main.i
1C13:002D JMP    0032            ; ..DEMO#22+0A
1C13:002F INC    WORD PTR [BP+FFFE]  ; ..DEMO.main.i
1C13:0032 CMP    WORD PTR [BP+FFFE],0003 ; ..DEMO.main.i
1C13:0036 JGE    0040            ; ..DEMO#27
23.           print__nums();
1C13:0038 CALL   0050            ; ..DEMO.print__nums
```

AT PROBE has set the CS:IP to the first instruction in main(), before the stack initialization procedure ___chkstk. It will be necessary to step beyond ___chkstk before setting a breakpoint on the local variable *count*. Pressing **<F8>** three times will move the highlight to the line with the instruction:

**CALL xxxx ;.___chkstk.**

At this point, pressing **<F8>** a forth time would cause the AT PROBE to step into the function ___chkstk. Instead, press **<F10>** to run in real time through ___chkstk, stopping immediately after returning from that function call. Press **<F3>** again to return to source display mode.

## DISPLAYING REGISTERS

If it is desirable to watch the current register values as code is stepped though, press **<F2>**. Pressing **<F2>** a second time will cause the register display to be removed from the screen. To load the watch window included with the demo programs, use the following sequence of keystrokes:

**W L demo.wch <CR>**

Now press **<Alt-S>** to bring up a window display of the stack, and set the registers display **<F2>** to on. When you wish to toggle the stack display off, press **<Alt-S>** a second time. Using **<F8>**, single step through the assembly language module demo3.

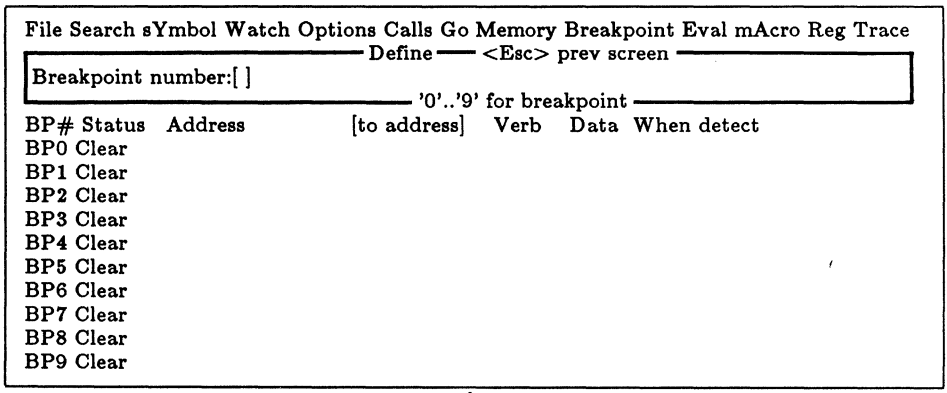
```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                          <Tab> next field <Esc> prev screen
 Processor:<Main> {Float|Main}
 Enter new value:[
                                  Arrows move highlight
AX=0007  CS=12FD  SS=14AD  DS=14AD  ES=14AD  GDTR=000000,FFFF
BX=1314  IP=0136  SP=1310  SI=0082  DI=132D  IDTR=000000,FFFF
TR=0000
CX=0019           BP=1316           MSW=FFF0=TS0 EM0 MP0 PE0
DX=000E           FL=0206=O0 D0 I1 T0 S0 Z0 A0 P1 C0
```

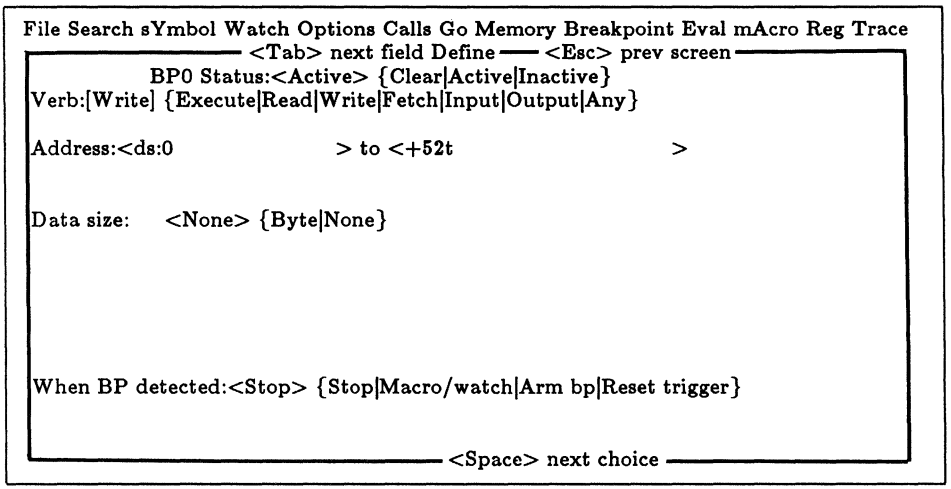
If at any time you wish to examine the complete 80286 register set and/or change the value of a register, use the Registers command. **R**, for registers, will display the screen shown above.

## INTRODUCTION TO BREAKPOINTS

Next, a simple breakpoint will be set. Before continuing, a bit of background is in order. When a program overwrites a value in the range of DS:0 to DS:34h (DS:52 decimal), the Microsoft C (R) compiler will issue a null pointer error on exiting from the program. Test.c is deliberately written so that when *count* is equal to 50 a value of 0xff will be written to DS:10. This first breakpoint will trap on the range of memory being tested by the Microsoft compiler. To create the breakpoint, enter the following sequence of keystrokes:

**B D 0 W DS:0 <CR> +52T <CR> <CR> <CR> A <ESC> <ESC>**



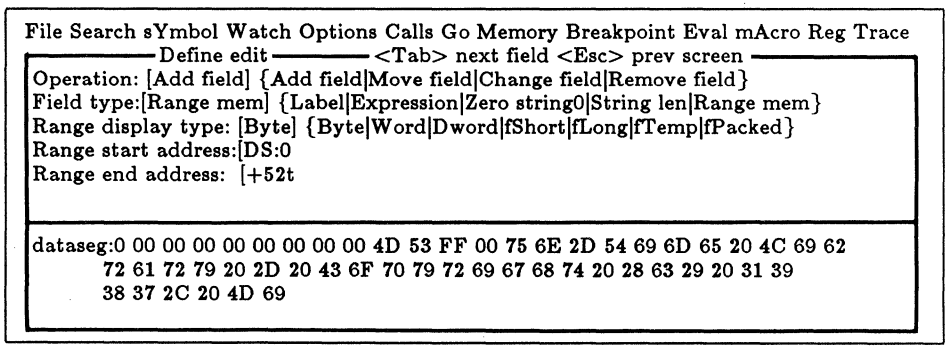



Now enter **G <CR>**. The program will send to the display a series of numbers from 0 to 49 and then execution will be suspended.

## DEFINING A WINDOW

Since the program was stopped by memory overwrite breakpoint, it would typically be desirable to examine the block of memory that was written to. To do so, enter the following sequence:

**W D <Alt-W> <CR> <CR> <CR> data seg:0 <CR> <CR> R <CR> DS:0 <CR> +52T <CR> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CursorRight> <CR> <ESC> <ESC>**

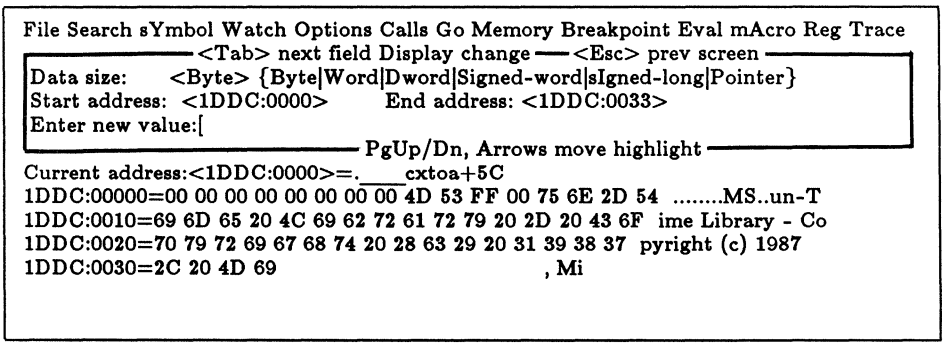



The watch window shown above will appear containing a display of memory over a range of from DS:0 to DS:52. To open the window, press **<Alt-W>**. To close the window, press **<Alt-W>** a second time.

## MACROS

While the display of memory in the watch window contains all the information necessary to examine the overwrite, sometimes a simple hex dump fails to fully convey the meaning contained in a block of memory. This dilemma is best dealt with by means of the Memory Display command. To simplify future use of the Memory Display, a mAcro will be created containing the necessary sequence of keystrokes to re-open that display at any time. To create the macro, enter the following series of keystrokes:

**A D <Alt-D> <CR> a display of DS:0 to DS:52 <CR> <CR> Y M D <CR> DS:0 <CR> +52T <CR> <Alt-D>**




Pressing **<ESC>** will restore the console to the source lines display. To active the macro, press **<Alt-D>**.

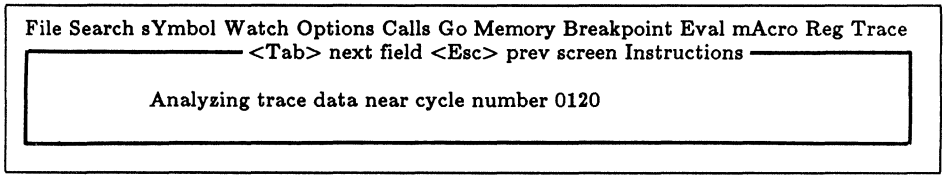
If your keyboard has function keys across the top, which can be difficult to use, you might create other simple macros to substitute for those keys. For example, Alt-S might be used to single step, and Alt-C might substitute for F7, going to the cursor line.

The beauty of macros is the labor they save. Nevertheless, if it is necessary to re-define the macro each time the debugger is started, the extent of that savings of labor would be slight. Macros (and watch windows) can be saved to disk, then automatically reloaded at will.

**A S demo.mac <CR>** will save the macro to the file "demo.mac". Likewise, **W S demo.wch <CR>** will save all watch windows to the file "demo.wch".

## DEFINE REAL TIME TRACE

For an in depth analysis of the sequence of instructions leading up to the break, press **T I**. The following display will appear for several seconds while the AT PROBE analyzes the trace:




Once the trace information has been analyzed, the following display of source lines, assembly mnemonics and bus cycles will be shown (1).

(1) Because of the way the 80286 "pipeline" works, by the time memory has been overwritten and the breakpoint detected, the program may have executed some additional instructions. These instructions will appear below the line marked "B" on the trace display.

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                <Tab> next field <Esc> prev screen Instructions
Search addr:<Any>          to <>
Verb:      <Any> {Read|Write|Input|Output|Any}
Data size: <Any> {Byte|Word|Dword|Any}   Data: <>
Begin search of trace:[No] {Yes|No}
                          PgUp/PgDn/Arrows move within memory

29.       if (count == 50)
 0001C479 CMP    WORD PTR [BP+FFFE],0032
 0001ED9A   READ    -    0032   .__end+07DA
 0001C47D JZ     $+0005

30.           *zeroptr = 0xff;
 0001C482 MOV    BX,WORD PTR [0042]
 0001DDD2   READ    -    000A   .__zeroptr
 0001C486 MOV    WORD PTR [BX],00FF
B 0001DD9A   WRITE   -    00FF   .____cxtoa+6C

31.       print (&count);
 0001C48A LEA    AX,[BP+FFFE]
 0001C48D PUSH   AX
 0001ED94   WRITE   -    100A   .__end+07D4
 0001C48E CALL   $+0038
 0001ED92   WRITE   -    0071   .__end+07D2
```

The line with a leading 'B' marks the point where the break actually occurred. After the "B" on that line the absolute address accessed is displayed, the type of memory access (a write), and the value accessed. In this example, the value FF hex is being written to the absolute address 1DD9A, which is offset 6C hex from the symbol ____cxtoa. The previous line shows the assembly code which causes that bus access to occur. That line shows the absolute address of the instruction (1C486 hex) and the mnemonics (MOV WORD PTR [BX],00FF) which construe the actual code.

The trace display can be scrolled through with the **<PgUp>**, **<PgDn>**, **<CursorUp>**, **<CursorDown>**, **<Ctrl-PgUp>**, and **<Ctrl-PgDn>** keys. To exit the trace, press **<ESC>**.

## INIT FILES

Saving and restoring macros and watch windows is a useful capability. A more powerful feature of the AT PROBE is its ability to automatically load a program, macros, watch windows, and symbols. To save the current AT PROBE environment, enter **F I <CR> <CR> <CR> demo.ini <CR>**. Exit the AT PROBE by entering **F Q <CR>**. Now restart the AT PROBE by entering **atsource /i demo.ini <CR>**. As the various files are auto-loaded, the following screen will be displayed:

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
Initializations ──────────── <Tab> next field <Esc> prev screen ──────────
 Load/save initialization information:<Load> {Load|Save}
 Default disk:<C>       Default directory:<\SRC>
 File name:  <DEMO.INI>


Loading option...
Loading module...
Loading program...
Loading program from file C:\SRC\DEMO.EXE
Loading symbols from file C:\SRC\DEMO.EXE
Reading symbols, types and lines 0198
Loading macro...
C:\SRC\DEMO.MAC
Loading watch...
C:\SRC\DEMO.WCH
```

Auto loading an init file containing a "Startup" macro used to prepare AT PROBE for a debugging session can streamline and simplify the debugging process.

## CALLING MACROS FROM BREAKPOINTS

Press **<F3> <F8>**. This will bring you to main(). Now, reset breakpoint 0 with the sequence:

**B D 0 W DS:0 <CR> +52T <CR> <CR> M <Alt-D> <CR> A <ESC> <ESC>**

Run the program by typing **G <CR>**. As you see, there is a difference between the breakpoint just entered and the similar instructions used to set the breakpoint earlier. Previously, after the memory write, the breakpoint had simply stopped program execution. This time, when the program breaks, the macro created earlier to display the 52 bytes of memory at DS:0 will be automatically executed.

## DEFINE SEQUENTIAL BREAKPOINTS

Next, using sequential breakpoints, a trap will be put on a write to a local variable in such a manner that the program will break only when a write to that variable occurs from a function *other* than that which owns the variable. Press **<ESC>** to clear the display. Next, put the display into assembly mode and set execution breakpoints on the line which calls print() and on the line immediately following that line.

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                    PgUp/Dn, Arrows move highlight
38.     print (&count);
12FD:0078=      LEA     AX,[BP+FFFE]    ; ..DEMO.print__nums.count
 ^ ^ ^ ^        0002
12FD:007B       PUSH    AX
12FD:007C       CALL    00A0            ; ..DEMO2.print             BP1
12FD:007F       ADD     SP,0002                                     BP2
12FD:0082       JMP     0060            ; ..DEMO#35+07
40.   return;
12FD:0084       JMP     0086            ; ..DEMO#41
41. }
12FD:0086       MOV     SP,BP
12FD:0088       POP     BP
12FD:0089       RET
44. {
12FD:008A       PUSH    BP
12FD:008B       MOV     BP,SP
12FD:008D       XOR     AX,AX
12FD:008F       CALL    04CC            ; .____chkstk
45.   puts("\nResetting counter\a\n");
12FD:0092       MOV     AX,0566
12FD:0095       PUSH    AX
12FD:0096       CALL    1992            ; .__puts
12FD:0099       ADD     SP,0002
```

Enter **B D**. You should see a display similar to the one shown below:

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                          Define ── <Esc> prev screen ──
Breakpoint number:[ ]
                          '0'..'9' for breakpoint ──
BP#     Status  Address          [to address]  Verb    Data  When detect
BP0     Inactive                   +52t        Fetch   None  <AltD>
BP1     Active  ..DEMO#38+04                   Execute       Stop
BP2     Active  ..DEMO#38+07                   Execute       Stop
BP3     Clear
BP4     Clear
BP5     Clear
BP6     Clear
BP7     Clear
BP8     Clear
BP9     Clear
```

Breakpoint 0 will now have to be changed to break on a write to the variable count which is local to the function print_nums(), and breakpoints 1 and 2 will be modified to arm and disarm breakpoint 0. Enter the following sequence of keystrokes:

**0 W <Ctrl-CursorLeft> print_nums.count <CR>**
**<Ctrl-CursorLeft> <CR> <CR> S A <ESC>**
**1 <CR> <CR> A 0 <space> 2 <CR> <ESC>**
**2 <CR> <CR> R <ESC> <ESC>**

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                <Tab> next field Define ── <Esc> prev screen ──
          BP0 Status:<Active> {Clear|Active|Inactive}
Verb:[Write] {Execute|Read|Write|Fetch|Input|Output|Any}

Address:<print_nums.count          > to <                          >


Data size:    <None> {Word|Byte|None}




When BP detected:<Stop> {Stop|Macro/watch|Arm bp|Reset trigger}

                                   <Space> next choice ──
```

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                  <Tab> next field Define ── <Esc> prev screen
            BP1 Status:<Active> {Clear|Active|Inactive}
Verb:[Execute] {Execute|Read|Write|Fetch|Input|Output|Any}

Address:<..DEMO#38+04                                        >




When BP detected:<Arm bp> {Stop|Macro/watch|Arm bp|Reset trigger}
Arm bp:<0 2>

                                  <Space> next choice
```

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                  <Tab> next field Define ── <Esc> prev screen
            BP2 Status:<Active> {Clear|Active|Inactive}
Verb:[Execute] {Execute|Read|Write|Fetch|Input|Output|Any}

Address:<..DEMO#38+07                                        >




When BP detected:<Reset trigger> {Stop|Macro/watch|Arm bp|Reset trigger}

                                  <Space> next choice
```

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                              Define ── <Esc> prev screen
Breakpoint number:[ ]
                              '0'..'9' for breakpoint
BP# Status  Address              [to address]  Verb    Data  When detect
BP0 Active  print_nums.count                   Write   None  Stop
BP1 Active  ..DEMO#38+04                       Execute       Arm bp 0 2
BP2 Active  ..DEMO#38+07                       Execute       Reset trigger
BP3 Clear
BP4 Clear
BP5 Clear
BP6 Clear
BP7 Clear
BP8 Clear
BP9 Clear
```

Breakpoint #1 will cause breakpoint 0 to be armed each time the function print is called, and breakpoint number three will reset the entire breakpoint chain each time it is triggered.

As the breakpoint definitions are entered, the display should be the same as shown above. Having entered the breakpoint definitions, press **G <CR>**.(2) After running for a short time, the program will break on one of the following lines: (3)

```
    *val = 1;
  return;
}
```

(2) As AT PROBE sequential breakpoints do not occur in realtime, a loss of performance on the part of the program being debugged may be observed.

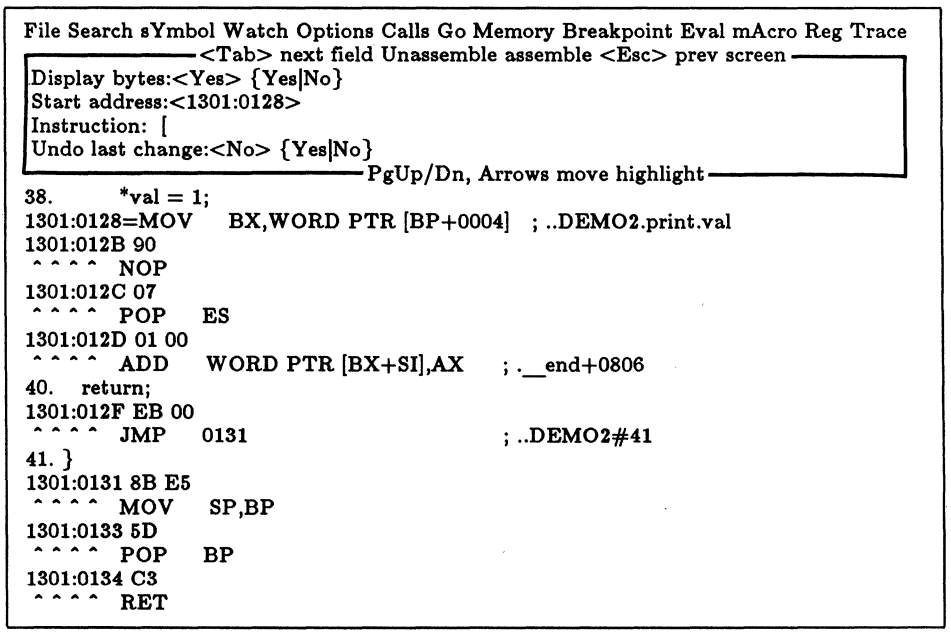
(3) Because of processor queueing, the actual line where program execution ends may vary.

## ASSEMBLE/UNASSEMBLE

Leave the trace mode and enter **M U**. Enter the address to be unassembled as follows:

**#38 <CR>**

Press **<TAB> Y <CR>** followed by pressing **<CursorDown>** one time. At this point the high lighted line should be the same as the line which caused the unwanted memory write. The display will show the instruction coded as bytes of hex. In order to progress beyond the memory write which is causing an endless loop, it will be necessary to delete the write instruction. This will be accomplished by changing the four bytes comprising the write instruction to a series of four NOP instructions (see screen display below). To delete the write instruction enter **NOP <CR>** four times, followed by **<ESC>** to return to the source line display. Once the source of the memory overwrite has been eliminated, the program will no longer break at this point.

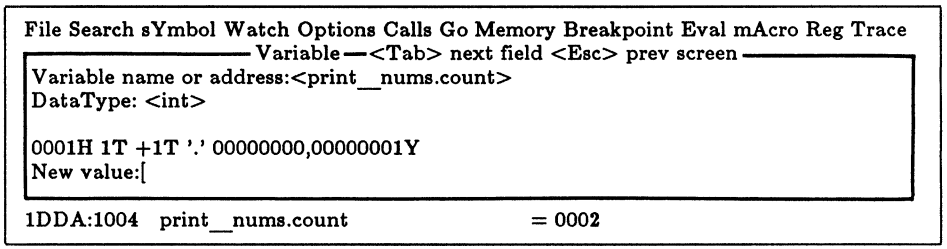


## THE CALLS COMMAND

Having eliminated the memory overwrite it may be valuable to be able to examine the current state of the stack. This can be done with the Calls command. Calls is activated by pressing **C**. The Calls command will display the variables and functions on the stack as follows:

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                          <Tab> next field <Esc> prev screen
Show arguments:<Yes> {Yes|No}
Show locals:  <Yes> {Yes|No}
Operation:    [Find] {Code|Find|Next|Previous|Up|Down|Variable}
Function/variable name:<.___stbuf+05><>
                          <Space> next choice; PgUp/Dn, Arrows move highlight
CS:IP is 1C43:09ED at .___stbuf+0005

Function at .___stbuf+0005 with BP = 0FD4:
Returns to 1C43:082B at .__printf+0015

Function at .__printf+0015 with BP = 0FE6:
Returns to 1C43:012C at #36+0017 in function .print

Function .print with BP = 0FFA:
Returns to 1C43:0071 at #31+0007 in function .print__nums
1DDA:0FFE  print.val                    = 1004
1DDA:0FF8  print.tptr                   = 07A0

Function .print__nums with BP = 1006:
Returns to 1C43:0026 at #19 in function .main
1DDA:1004  print__nums.count                 = 0002

Function .main with BP = 1010:
```

As the previous memory write caused print__nums.count to be inadvertently changed, it is desirable to restore the proper value to that variable before continuing program execution. To do so, enter **F count <CR> V <CR>**. The first instruction will locate the variable *count* on the stack and the second activate the Variable function, which will display the screen below:




Press **<F4>** to examine the last value output to the display. Having noted that value (which in this case is 100T), press **<F4>** again. Enter the value which was last on the screen, followed by **<CR>** **<ESC>.**

Before continuing, press **<CursorUp>** five times to high-light the address to return to print__nums, followed by **<F7>** to go to that address. This will cause the program to move one level up the stack.

## THE FILE VIEW COMMAND

It is possible to examine and set breakpoints or go to a location in a file other than the current one by using the File View command. Enter the following keystrokes:

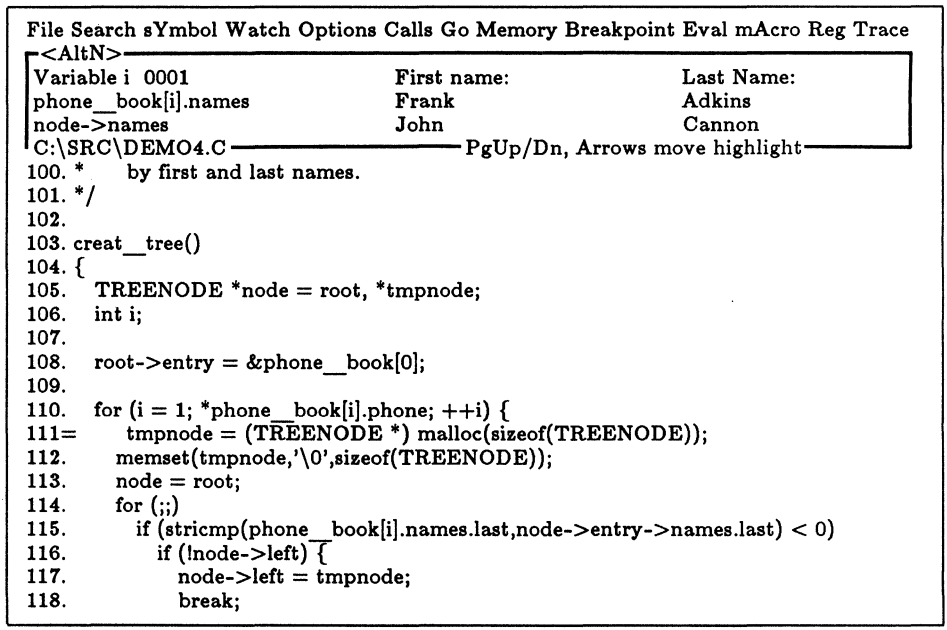
**F V demo4.c <CR> L 111 <CR> <F7>**

This causes the source file "demo4.c" to be examined, and in this case line number 111 is located, then using <F7> the applications program is executed up to that line.

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
View ──────────────────── <Tab> next field <Esc> prev screen ──────────────────┐
│Operation:[Find] {Find|Next|Previous|Line}                                    │
│Find:<>                                                                       │
└C:\SRC\DEMO4.C ──────────────────── PgUp/Dn, Arrows move highlight ────────────┘
101. */
102.
103. creat__tree()
104. {
105.    TREENODE *node = root, *tmpnode;
106.    int i;
107.
108.    root->entry = &phone__book[0];
109.
110.    for (i = 1; *phone__book[i].phone; ++i) {
111.       tmpnode = (TREENODE *) malloc(sizeof(TREENODE));                BP0
112.       memset(tmpnode,'\0',sizeof(TREENODE));
113.       node = root;
114.       for (;;)
115.          if (stricmp(phone__book[i].names.last,node->entry->names.last) < 0)
116.             if (!node->left) {
117.                node->left = tmpnode;
118.                break;
119.             } else
120.                node = node->left;
```

## THE VARIABLE COMMAND

At this point the demo program's execution has stopped in function creat_tree(). You may wish to single step through this function as the binary tree is created in order to gain an understanding of the form of that data structure. A watch window has been provided which was loaded along with the stack (<Alt-S>) watch window. The watch window will show the current state of the variables *phone_book[i].names* and *node->names*. This watch window can be opened with <Alt-N>.



To examine the data structure from which the binary tree is being created, that being the array phone_book[], enter **M V phone_book <CR>**. The following display will appear:

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
          <Tab> next field Variable <Esc> prev screen
Variable name or address:<phone_book>
DataType: <struct _PERSON [10]>
Member:  <[0].names.first>
"John"
New value:[

1DC3:0044    [0].names.first          = "John"
1DC3:005D    [0].names.last           = "Cannon"
1DC3:0076    [0].phone                = "(206) 524-4521"
1DC3:0086    [0].hobbies[0]           = Scuba_diving
1DC3:0088    [0].hobbies[1]           = Exercise
1DC3:008A    [0].hobbies[2]           = 0000
1DC3:008C    [0].hobbies[3]           = 0000
1DC3:008E    [0].hobbies[4]           = 0000
1DC3:0090    [1].names.first          = "Frank"
1DC3:00A9    [1].names.last           = "Adkins"
1DC3:00C2    [1].phone                = "967-8186"
1DC3:00D2    [1].hobbies[0]           = Motorcycle_racing
1DC3:00D4    [1].hobbies[1]           = Opera
1DC3:00D6    [1].hobbies[2]           = Magic
1DC3:00D8    [1].hobbies[3]           = 0000
1DC3:00DA    [1].hobbies[4]           = 0000
1DC3:00DC    [2].names.first          = "Cathy"
```

From within the Variable command complex data structures such as phone_book[] may be examined and manipulated. Using the cursor keys, highlight different substructures within phone_book[]. To alter a member element of the array, highlight that member and enter the new value to the "New Value: [" prompt. Note that strings must be contained in quotes, and that enumerated data types can be entered as either integers or the enumerated types as were defined in the appropriate ENUM declaration.

Once you have conceptualized how the tree is constructed, close the watch window with **<Alt-N>** and go to line number 141 by entering **G 141 <CR>**. This will cause execution to stop in the function which outputs the contents of the binary tree in sorted order. Press <F9> to set a breakpoint at this line. Now, set the break point counter to 6 with **B S 6 <CR>**. At this point, enter G <CR> to cause the function display_tree() to execute five times past the breakpoint set at line

141, stopping on the sixth instance of that breakpoint. At this point the function display_tree() has called itself six times and returned twice, leaving five instances (including the initial call) of display_tree() on the stack.

Before examining the stack, it is helpful to load a special macro file created for this example. Enter **A L DEMO.MAC <CR>**. Now bring up the Calls command by entering **C**. With the cursor keys, move the highlight to the second instance of the variable *display_tree.node*. Press **V** for Variable. You should see the display below:

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                    Variable—<Tab> next field <Esc> prev screen
Variable name or address:<1DC3:12EC>
DataType: <struct __TREE *>

142AH 5162T +5162T '.*' 00010100,00101010Y
New value:[

1DC3:12EC  1DC3:12EC                  = 142A
```

In this case the address of the second instance of the local variable *node* was automatically entered into the "Variable name or address" dialog box rather than the variable name itself in order to distinguish the particular instance of the variable being examined. The bottom line displays the address of the variable and the value contained within that address. Next, since this variable is a pointer, it will be dereferenced and cast from a pointer to type __TREE to type __TREE. Enter the following sequence:

**<CR> <TAB> <CTRL-HOME> [ <CTRL-END> ] <TAB> <TAB> C [node]**

At this point you should have the following display:

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                    Variable—<Tab> next field <Esc> prev screen
Variable name or address:<[1DC3:12EC]>
DataType: <>  Use same type as symbol:[[node]

143AH 5178T +5178T '.:' 00010100,00111010Y
New value:<>

1DC3:142A  [1DC3:12EC]                = 143A
```

The address in "Variable name or address" has been dereferenced by the use of a set of square brackets. The DataType is about to be changed to the same type, that being the type of *node* dereferenced, or the structure _TREE. Enter **<CR>**.

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                    Variable—<Tab> next field <Esc> prev screen
Variable name or address:<[1DC3:12EC]>
DataType: <struct _TREE>
Member:  <left>
143AH 5178T +5178T '.:' 00010100,00111010Y
New value:[

1DC3:142A  left                          = 143A
1DC3:142C  right                         = 1432
1DC3:142E  entry                         = 01C0
```

The structure _TREE is a struct containing three pointers. The first two are pointers to other _TREE nodes, while the third points to an element of the array being sorted. If, for example, it is desirable to automatically chain through a tree or a list of this sort, macros can be used to this purpose. We have provided two such macros in the file DEMO.MAC which you loaded earlier. Enter **<ALT-L>** followed by Y to move to the left node. Now enter the following sequence of keystrokes:

**<TAB> <CTRL-CursorLeft> 258 <CR> <TAB> <TAB> C [[node].entry] <CR>**

```
File Search sYmbol Watch Options Calls Go Memory Breakpoint Eval mAcro Reg Trace
                Variable—<Tab> next field <Esc> prev screen
Variable name or address:<258>
DataType: <struct __PERSON>
Member:  <names.first>
"Stacy"
New value:[

1DC3:0258       names.first                 = "Stacy"
1DC3:0271       names.last                  = "Harkala"
1DC3:028A       phone                       = "882-8715"
1DC3:029A       hobbies[0]                  = Travel
1DC3:029C       hobbies[1]                  = Flying
1DC3:029E       hobbies[2]                  = Scuba__diving
1DC3:02A0       hobbies[3]                  = Hot__tubbing
1DC3:02A2       hobbies[4]                  = 0000
```

This process was very similar to that which previously dereferenced a *node* pointer into the _TREE structure, but this time the *node* member *entry*, which points to a structure of type _PERSON (which is an element of the array *phone_book[]*) was dereferenced. You might note that since a variable of type _PEOPLE was not available, it was necessary to dereference *node->entry with the expression **[[node].entry]**, to gain the proper type cast.

After experimenting with the Variable command, you may wish to Go to the end of the demo program.

## EXITING AT PROBE

Once the program has completed execution, it must be reloaded using the **Files Program Load** instruction, before any more debugging can be performed. To return to DOS use the **Files Quit** command.

# ADVANCED DEBUGGING TECHNIQUES

This section shows more advanced examples of using AT PROBE. These debugging procedures have been arrived at by previous users in real applications.

## DEBUGGING A BOOT LOAD SEQUENCE

If you are developing a boot loader, it will be necessary to make some additions to the loader before using the AT PROBE for debugging. A primary concern in using AT PROBE to debug a boot load sequence is to replace the AT PROBE vectors at locations 0:4, 0:8, and 0:C after they have been modified by the boot loader, and to load the 8k AT PROBE vector table from the AT PROBE protected mode memory into a suitable safe block of memory in the real mode address space. As this is a complex and involved process, it will not be described in detail here: rather, the source and make files for the PROBE.SYS device driver are included on the distribution diskette to provide an example of the necessary code. The best place to start in your study of this code is the function **Initialize** in drinit.asm. If you wish, you may use this code in your application.

Once the application has been modified to deal with the issues discussed in the previous paragraph, AT PROBE can be loaded. Next, the symbol table can be loaded as discussed in LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM. Finally, the system is rebooted by using Go with a starting address of ffff:0 hex (4).

If the AT PROBE loader is not overwritten by memory test of by the boot loader itself, it will be sufficient to just have the boot loader set the INT 1, 2, and 3 vectors back to their previous values; the boot loader doesn't need to do all the work that probe.sys does.

If a jump to self loop was used to stop AT PROBE at a specific address, then wait a few seconds for the system to execute up to that point and return control to AT PROBE by pressing the STOP button (5). If it is desired to trap somewhere in the boot loaded program,

(4) If you wish to avoid the system memory tests, write a value of 1234 hex to 0:472 before jumping to BIOS.

(5) It may be necessary to switch to an external console before rebooting the system. This should be done if the system locks up when the STOP button is pressed.

rather than simply using the stop button at the point of an infinite loop, then it is advisable to set a *hardware* breakpoint. An execute breakpoint will not work because the boot loaded program will write over the top of the breakpoint trap. Thus, if you want to stop at an instruction somewhere in the boot loader, use the Fetch verb in the hardware breakpoint. Another option would be to hardcode an INT 3 instruction directly into the boot loader, so that it will come off the disk with a breakpoint set.

Since DOS does not know that AT PROBE is running at this point, AT PROBE commands File,Macro and Windows commands that load, and save to disk,will not work. Additionally Source Step cannot be used as it will perform disk a access to get source file information. If you want symbols attached to your boot loader, see the section entitled "LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM" later in this chapter.

## DEBUGGING A DEVICE DRIVER WHICH INSTALLS ITSELF

When debugging a device driver which is installed from config.sys, as with the debugging of a boot load sequence it is necessary to reset vectors 1,2, and 3 before using the AT PROBE. Atron has provided a device driver called PROBE.SYS (6) to simplify this task. PROBE.SYS can be used to set vectors 1, 2, and 3 to values passed as parameters. These parameters can be passed in two forms:

1. As a single digit value for the address of the megabyte of AT PROBE protected memory followed by the segment address of 16k block of real memory used by AT PROBE. For example, if ADDR = D, D is set in PROBE.CNF the line in config.sys would be:

   **device=PROBE.SYS D, D000**

   This will set vectors 1,2, and 3 to values of D000:0000, D000:0003, and D000:0006 respectively.

2. Using the complete segment address of the megabyte of AT PROBE protected memory followed by the segment address of the 8k block of real memory used by AT PROBE. For example, if ADDR = D, D is set in PROBE.CNF the line in config.sys would be:

   **device=PROBE.SYS D00000, D000**

Upon installation, PROBE.SYS will display the new values of the three vectors being altered. Note that PROBE.SYS must be installed before the device driver to be debugged. It is also important to provide some sort of a means to regain control at a specific point in the device driver. If the symbol table is loaded before the device driver is loaded, then a hardware breakpoint can be used; other options include hardcoding in an INT 3 instruction, which will automatically return control to the AT PROBE, or writing a jump to self into the device driver then using the STOP button to return control to the AT PROBE.

(6) On some systems using the Phoenix (R) Bios it will be necessary to use PROBEHP.SYS. This applies if the PROBE software being used has the suffix "HP".

Having installed PROBE.SYS in the config.sys file which references a device driver to be debugged, you can now load and debug that device driver by using the following procedure:

1. Load AT PROBE.

2. Insert the diskette with the new boot loader to be debugged.

3. Using Go, begin execution at ffff:0 hex. At this point, the system will reboot.

4. Wait a few seconds for the system to execute up to the jump to self. Return control to AT PROBE by pressing the STOP button.

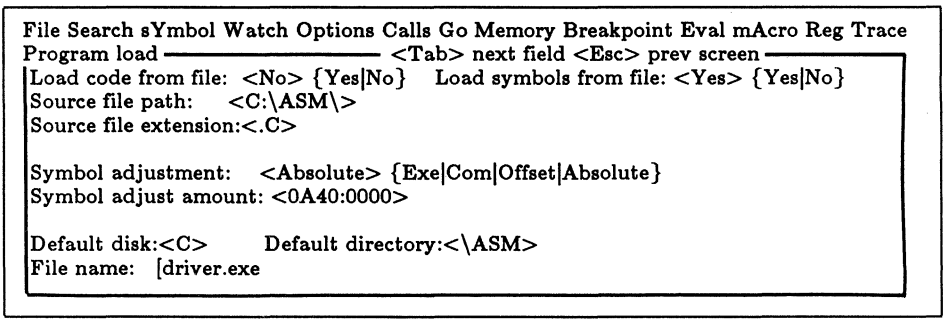
5. If it is desired to have the symbols for the device driversymbols, then make a note of the current code segment (7). Reboot the system without the device driver, and using this value follow the instructions in the section Loading the symbol table when AT PROBE does not load the program later in this chapter. Repeat steps 1 through 4.

**Example:**

If a device driver named "driver.asm" is assembled with Microsoft MASM 5.0 with an ORG of 0 (as required by MS-DOS) and using the /Zi option, then converted to binary format with exe2bin, the symbolic information would be loaded with the following sequence:

**F P <TAB> N <CR> <CR> <CR> A SegVal:0 <CR> <CR> <CR> driver.exe <CR>**

(7) This value will remain constant so long as the same version of MS-DOS and the same device drivers are being loaded before the device driver being debugged. Accordingly, it should be necessary to load the device driver without symbols to obtain this value one time only.




This will adjust symbol values to match the absolute value (in this case 0xA40:0) of the loaded program. Now a jump to BIOS may be made to load the device driver, as described earlier.

5. The device driver is now installed and can be debugged by AT PROBE. If a jump to self loop was used to stop program execution at a certain point, it will be necessary exit that loop before continuing with the debugging process.

## DEBUGGING A DEVICE DRIVER INVOKED FROM COMMAND.COM OR A TERMINATE AND STAY RESIDENT PROGRAM

If a device driver or quit and stay resident program is invoked from DOS, then use the following procedure.

1. Load AT PROBE. You may want to load the program's symbol table at this point. If this is the case, see LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM later in this chapter. If you load the symbols at this time, you may wish to set a hardware breakpoint(s).

2. Execute a quit and stay resident command:

   **F Q <TAB> Y <CR>**

3. You are now in DOS. Now invoke the device driver or TSR program to be debugged from the command line.

4. If a jump to self loop was used (see the section entitled DEBUGGING A DEVICE DRIVER WHICH INSTALLS ITSELF), press the STOP button to return control to AT PROBE. If the symbol table was not loaded before the program, you may want to load symbols at this time.

## LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM

When AT PROBE does not load a program, some extra steps are necessary in order to load the program's symbol table. Most importantly, it is necessary to know the absolute address where the program is to be loaded. Once this value is known, it is possible to load the symbol table either before or after the program itself is loaded. There are advantages and disadvantages to each of these options.

In order to load the symbol table before loading the program, it will be necessary to load the program one time, stop execution, and make note of the absolute address where the program loaded. Having this information, in the future it will be possible to load the symbol table at the absolute address retrieved as described above. The advantage of this method is that hardware breakpoints can be used to stop the program, rather than mandating the initial use of the stop button with a jump to self loop in order to halt execution at a specific point. The disadvantage to this method is that it requires that the program be loaded at the same absolute address every time it is to be debugged. Another minor disadvantage is the extra step initially required to determine where the program is being loaded.

Loading the symbol table after loading the program is somewhat less complicated. It does, however, require that a jump to self loop be used to stop the program at a point before the CS register is altered (as might be the case in a large model program). The STOP button is used to regain control, after the operating system loads it and begins execution. The symbol table can then be loaded using the Symbol adjustment *Absolute* option, with the load address being given as CS:0.

## DEBUGGING ROUTINES WHICH TAKE OVER THE KEYBOARD

Routines which take over the keyboard are tricky to debug because the BIOS keyboard routines are not reentrant. If a breakpoint is set inside this routine or inside your keyboard routine which takes over the keyboard interrupt, a lock of the system may occur because of the reentrancy problem. If this happens, switch to external console.

## DEBUGGING INTERRUPT DRIVEN SOFTWARE

Debugging programs which are running in an interrupt driven environment normally happens in one of the following ways: locking out all interrupts while in AT PROBE or using concurrent process debugging.

### Locking out all Interrupts While in AT PROBE

When first bringing up interrupt routines, it is most useful to lock out all interrupts while in the AT PROBE software. This is because an interrupt from a device which happens while in the AT PROBE software could potentially never return or crash the system if it is allowed to be serviced. To lock out interrupts while in the AT PROBE software use the Options Interrupt command as described in Chapter 5, COMMAND REFERENCE. This allows masking any or all of the interrupts from the system interrupt controller from being processed while in the AT PROBE software. This is especially necessary if a breakpoint has been set inside a non-reentrant interrupt routine in which periodic interrupts will continue to happen. If the keyboard interrupts must be locked out, then use the external console option with AT PROBE.

### Using Concurrent Process Debugging

In some systems it necessary to allow well-behaved interrupts to continue to process in the background even while the AT PROBE software has control of the system. For debugging this type of system, use the Options Interrupt command as described in Chapter 5, COMMAND REFERENCE to allow some interrupts to take control of the system whenever they happen.

## DEBUGGING ON A NON-DOS OPERATING SYSTEM

AT PROBE software is in a standard .exe file and was designed to run from DOS. It may be possible to use AT PROBE with other operating systems besides DOS by using the following procedure.

1. First boot DOS and load the AT PROBE software from drive A.

2. Insert the new operating system in drive A or simply open the floppy door if the new operating system is already on a hard disk.

3. Reboot the system using the DEBUGGING A BOOT LOAD SEQUENCE procedure which was described earlier.

4. If the new operating system does not destroy location 0:8 (the NMI vector), then control can be returned to AT PROBE with the STOP button. If the new operating system does destroy this vector, then you must use a utility in the new operating system to put this vector back before AT PROBE can get control.

5. You cannot use AT PROBE File,macro,windows commands that load or save to disk since they use DOS system calls. If the new operating system is a real time, time slicing version which uses keyboard BIOS routines, it would be best to switch AT PROBE to external console and execute the Options Int command to set all interrupts off before rebooting.

## DEBUGGING A SYSTEM WHICH CRASHES AND TAKES AT PROBE WITH IT

Some software applications which use interrupts and lots of DOS and BIOS interrupt calls can become quite complex to debug. In cases where the interrupts are not well behaved or there are many interacting reentrancy problems, the system may crash and not allow AT PROBE to work. These cases are best debugged on an external console. You should also lock out all interrupts while in the AT PROBE software with the Options Interrupt command. This ensures that interrupts in the system do not take control away from AT PROBE while you are in the AT PROBE software.

Another case which may cause AT PROBE to not respond is the modification of location 0:8 (NMI vector). For the AT PROBE to work properly, location 8 which stores the vector for the NMI must be left intact as initialized by the AT PROBE software. In the event that control of the AT PROBE is lost, it may be because this location has been modified. This event can be detected with the AT PROBE. Press the RESET button after control of the AT PROBE has been lost. Once reset, reload the AT PROBE software and display the trace memory. The trace memory is not cleared upon starting the AT PROBE software so that this previous event can be detected. The display of the trace information will show if location 0:8 has been modified if this event has not been cycled out of the trace memory. In the case where location 0:8 has been modified but the trace memory has cycled out the sequence of instructions that modified it, set a breakpoint on writing to location 0:8. Control of the AT PROBE will still be lost and the RESET button will be necessary, but this time the trace will contain the modifying code.

# CHAPTER 5 COMMAND REFERENCE

# CHAPTER 5
# COMMAND REFERENCE

This chapter provides the details for AT PROBE commands. Each command is followed by examples. The commands are listed alphabetically. The headers at the top of the page indicate the command/subcommand which is described on the page.

## FORMAT FOR DESCRIBING AT PROBE COMMANDS

In this chapter, the MENU BOX, DISPLAY WINDOW, and DIALOG BOXES produced by the commands are shown. Along the perimeter of the screens you will find numbers in circles. In the text you will find the corresponding numbers in circles along with a description of the screen information.

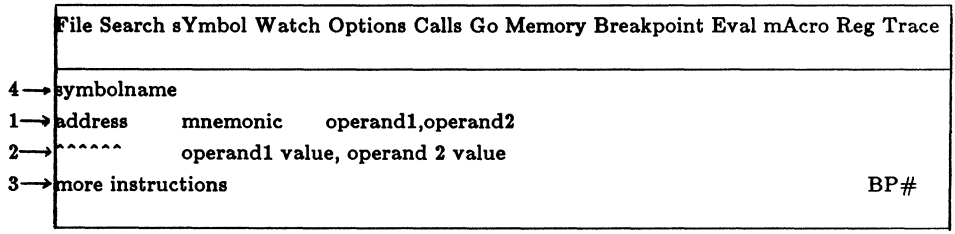
The prompts AT PROBE provides you in DIALOG BOXES are shown in italics in this chapter. In the text, a keystroke is specified as:

**<keyname>**

For example, <Enter> means type the key labeled "Enter" on your keyboard. Some <keynames> require two keys. For example <Alt-Key> means hold down Alt and type the keyname.

# CODE SCREEN

The default state of the DISPLAY WINDOW displays your program after it is loaded. It also displays the code at the current cs:ip after a breakpoint, Stop button or single step operation. For this reason, it is called the CODE SCREEN. The other AT PROBE commands overlay the DISPLAY WINDOW as needed. When commands have been specified and selected, the display or DIALOG BOX is replaced by the CODE SCREEN. The operation of the Function keys and cursor control keys on the CODE SCREEN is given in Chapter 2 and is not repeated here. This section describes in more detail the information on the CODE screen and operation during single step and breakpoints.




1 The instruction which matches the current program counter is indicated by the highlighted reverse video line or a different color on a color monitor. The address, mnemonic and operands are shown on this line. This instruction is not executed until a Function key is typed.

2 If the CODE SCREEN is displaying assembly language, the values of the operands for the current instruction are shown along with the effective address of the operand. AT PROBE calculates the address of the operand, if appropriate, and retrieves its contents in memory. Any symbol or near symbol which matches this calculated address is displayed, along with the contents of the memory at this calculated address. This shows you the operand values of the current instruction before it actually executes (i.e. before you type a function key). This saves you from having to bail out of the CODE SCREEN to see the values of the operands. The ^^^^^ on this line points to the instruction address for these operands ,i.e. the instruction just above the current line.

**********

Note that if the operand is pointing to a memory mapped IO device which changes content when it is read, AT PROBE's advance read of the operands may affect your program. You can disable AT PROBE from showing operand contents with the Options command.

**********

Also note that a highlight field spans the current instruction and its operands. This highlight field serves as a second cursor. The highlight field can be moved with the PgDn/PgUp and cursor keys. When moving the highlight field up the DISPLAY WINDOW past the current cs:ip, it will not move previous to the start of code which has executed from this screen unless there is a symbol within 100h bytes before that location and unassembly from that symbol to the start of code ends on an instruction boundary. When moving the highlight field down the CODE SCREEN below the current CS:IP, your program (as disassembled from memory or source code file) is shown. Typing Ctrl PgUp moves the highlight field to the start of code executed on this screen. Typing <HOME> moves the highlight to the next instruction to be executed.

3 If a sticky breakpoint has been set on executing an instruction, the sticky breakpoint is shown on the screen.

4 If the address of an instruction matches a label or line number, AT PROBE shows this. You can disable this with the Options command.

## SINGLE STEPPING in the CODE SCREEN

AT PROBE can single step your program from the CODE SCREEN with the following Function keys.

**F8** Executes a single step of the next instruction, steps into procedures and interrupts.

**F10** Executes a Program step. If the source line or instruction contains a function, procedure, interrupt, or call, the AT PROBE runs real time until it returns to the instruction after the currently highlighted instruction.

During single stepping in the CODE SCREEN, AT PROBE can do more than just take the next single step when F8 or F10 is typed.

If you are paging through your program on the CODE SCREEN and position the highlight field at an instruction, the following function keys can be used to set and execute to breakpoints at the highlight:

**F5** Start program execution and set a sticky breakpoint on executing instruction at highlight. Adds execute breakpoint to next available breakpoint in breakpoint screen and activates it. Shows breakpoint number in highlight to the right edge of the screen on the line of the instruction. When the instruction is displayed in the future, the breakpoint is displayed. All other active sticky breakpoints are still active. When breakpoint detected, control returns to the Command Menu.

**F7** Sets a temporary breakpoint on executing instruction at highlight and executes to that line then removes breakpoint. All other active sticky breakpoints are still active. When breakpoint detected, control returns to the Command Menu.

**F9** Sets or clears an execution breakpoint on line at highlight. Setting a breakpoint adds and activates a sticky breakpoint with the next available breakpoint number. Clearing deletes the breakpoint from breakpoint screen. Shows breakpoint number in highlight to the right edge of the screen on the line of the instruction. When the instruction is displayed in the future, the breakpoint is displayed.

If you type F5, F7, F8, or F10 and the highlight disappears and does not come back, then AT PROBE did not reach the target instruction and is still executing the program. To return control to the CODE SCREEN, press the Stop Button on the crash recovery switch box.

**Assembly language stepping**

During single step, AT PROBE implements Step Assembly language by setting the Trap flag except for the following conditions:

1. For interrupt instructions AT PROBE sets a software breakpoint at the target address of the software interrupt vector
2. For a segment load or a Repeat move string instruction, AT PROBE sets a software breakpoint at the instruction after the instruction to be stepped.

**Source stepping**

You can single step the program via source statement lines by putting the CODE screen in Source display with the F3 key. AT PROBE implements source level single stepping by setting a software breakpoint at each instruction which has a source line number, then executing a GO command. You may also limit the modules through which you single step while executing all code outside these modules in real time. This is done using the sYmbol Step-source-screen-modules selection command. This provides an automatic method of ignoring parts of you code which are already debugged and which you do not want to step into.

**Step count**

Single stepping can be by a single instruction or multiple instructions. This lets you go through your program in larger steps. You can change the Step count with the Options command.

**Step while condition**

Single stepping can be programmed to continue automatically while a a specified condition is true. You can set this condition by using the F8 and F10 keys in a conditional macro which has a while condition.

**Watch windows during single step**

While single stepping, you can pop up one or more Watch Windows. These windows are defined by the Window command and are popped up with a single Alt-Key. These windows are updated after each single step, therefore, you can keep an eye on anything in the target while you are single stepping.

# BREAKPOINT

The Breakpoint command lets you define, clear, activate and inactivate sticky breakpoints. If sticky breakpoints are active, they are automatically inserted when the Go command is executed or a function key starts program execution and sets a breakpoint. The Breakpoint command also allows you to set a pass counter on breakpoints and to monitor logic lines.

The Breakpoint command is invoked from the MENU BAR by typing:

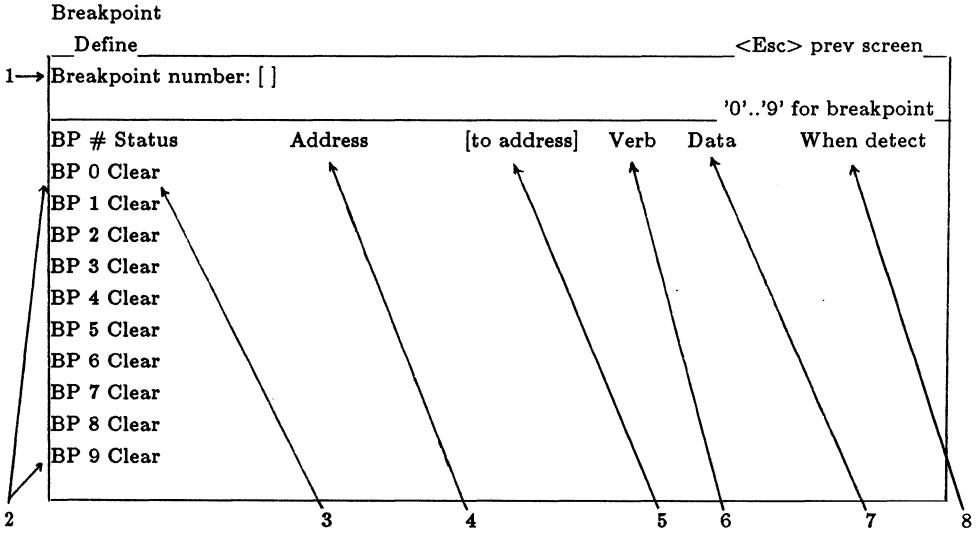
**B** (for Breakpoint)

The subcommands for Breakpoint are:

| Subcommand | Operation |
|---|---|
| **D**efine | Define/change breakpoint |
| **A**ctivate | Enable breakpoints during Go |
| **I**nactivate | Do not enable breakpoint during Go |
| **C**lear | Clear the breakpoint screen |
| **S**et pass counter | Set breakpoint pass counter |

## DEFINING AND EDITING BREAKPOINTS

When the Define subcommand is selected, the DISPLAY WINDOW shows a summary of the current conditions of sticky breakpoints. This is sometimes referred to in this manual as the Abbreviated Breakpoint Summary.




1 The DIALOG BOX first prompts you for the breakpoint number. AT PROBE lets you define up to 10 sticky breakpoints with breakpoint numbers from 0 to 9. To make it easy to remember which breakpoints are already defined, an abbreviated summary of the status of all 10 sticky breakpoints is shown in the DISPLAY WINDOW. A short explanation of these fields is given here. A more thorough explanation is given later.

2 This field is the Breakpoint number.

3 If a breakpoint has not previously been defined, its default status is shown as Clear.

4 This is the address of the currently defined sticky breakpoints.

5 This is the end address for a range breakpoint.

6 This is the verb of the currently defined sticky breakpoints. The default is Execute.

7 The Data field displays the size of the data if the data bus is included in the breakpoint.

8 This field displays the action to be taken when the breakpoint is detected.

Note that sticky breakpoints can be put into this breakpoint screen with the F5 and F9 keys while you are in the CODE SCREEN, File View, Memory Unassemble, Search, sYmbol sYmbol display, or Calls command. This is described in more detail in Chapter 2 as well as in each of these commands.

After you select the breakpoint number, the following will appear in the DISPLAY WINDOW.

```
    Breakpoint
      Define                                                <Esc> prev screen
4→  BP0 Status:<Inactive>{Clear|Active|Inactive}
1→  Verb:[Execute]          {Execute|Read|Write|Fetch|Input|Output|iNt ack|Any}

2→  Address:[]

3→  When BP detected: [Stop]      {Stop|Macro-window|Armbp|Reset trigger}
```

1 The first prompt sets the breakpoint verb. The breakpoint verbs are defined as follows:

| VERB | VERB DEFINITION |
|---|---|
| Execute | Instruction execution breakpoint via software interrupt instruction. May only be set in RAM memory. This type of breakpoint temporarily replaces the target code with the INT 3 instruction. The execute verb is the default. (1) |
| Read | Breakpoint on Read of memory address or range of memory |
| Write | Breakpoint on Write to memory address or range of memory |
| Fetch | Breakpoint on an opcode read from memory |
| Input | Breakpoint on read of IO port |
| Output | Breakpoint on write to IO port |
| Any | Breakpoint on Any access (i.e.Read, Write, Input, Output, or Fetch) of memory address or range of memory addresses. |

2 The second DIALOG BOX prompts you for the address for the breakpoint. You can use an expression or absolute number for the address.

3 This field lets you determine the action to take when AT PROBE has detected a breakpoint.

(1) Any time you use Execute in arming chains or use Pass counts, you will not get fully real time execution.

| Action | Description |
|---|---|
| Stop | Stop execution, display code screen |
| Macro-window | Stop execution, execute Macro or display a user defined Watch Window |
| Armbp | Arm detection of the next breakpoint |
| Reset trigger | Used in breakpoint sequences. Reset causes the first breakpoint in the sequence to be re-armed and all other points in the sequence to be disarmed. |

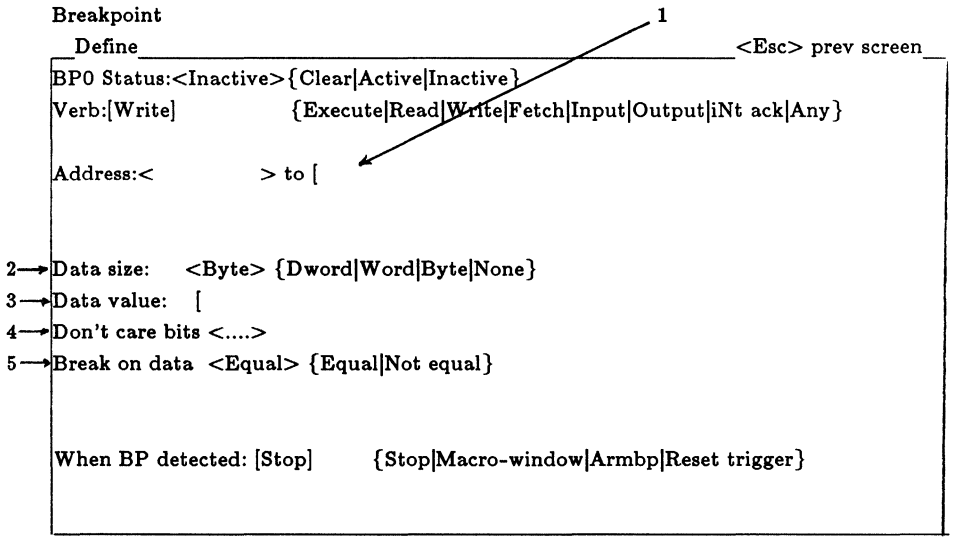
If you select STOP, AT PROBE simply stops program execution and re-displays the CODE SCREEN. Note, however, that if the Go command was invoked from inside a macro, the AT PROBE will continue to execute the remaining commands in the macro.

If you select the Macro/window option for the breakpoint, you are prompted for an Alt-Key (i.e. hold down Alt and type any key) to specify the Macro or Watch Window. When the breakpoint is detected, AT PROBE looks for the specified macro or Watch Window and executes it when the breakpoint has occurred.

The Armbp and Reset trigger options are used to define sequential breakpoints. See the Sequential Breakpoint section for more information.

4 This field lets you activate or inactivate a breakpoint or clear the breakpoint screen. AT PROBE looks for all active sticky breakpoints when the Go command or a function key starts execution.

The prompts just described are all that are necessary to set a simple Execute breakpoint. After activating an Execute breakpoint, you could start defining another breakpoint by typing <ESC>. For the other breakpoint verbs, the following DISPLAY WINDOW options would appear.




1 For any verb except execute, this field appears to let you enter a TO address to set a breakpoint on a range of addresses. You may enter any type of address expression or an address of the form:

**+ number**

In this case the TO address becomes breakpoint start address+number. If this field has previously been set to other values and you want to clear the field, type <space><enter>.

2 The breakpoint can be further qualified with the DATA FIELD. This is done by first selecting the size of the data. The size of the data field can be 1 or 2 bytes by choosing Byte or Word, except for odd addresses or for ranges, in which case only Byte or None are available. If you do not want to include the DATA FIELD in the breakpoint, then type <enter> to select the default which is <none> and the next two prompts will be ignored. Note that size of the data field can only be Byte or None if the breakpoint is on a range of memory (i.e. has a TO address).

3 If you select one of the types for the DATA FIELD, then you must specify the DATA VALUE which will cause the breakpoint with this prompt. The DATA VALUE can be an expression.

4 This field lets you mask out bits in the DATA FIELD of the breakpoint. This is useful if you are looking for a bit field such as a flag or an ASCII character in the breakpoint. You enter an X for each data bit position you want to mask and enter a period (.) for each data bit position you want to detect in the breakpoint. The initial default is all periods. The data you enter for this prompt is right justified.

5 The breakpoint can be selected to trap on the data being Equal or Not Equal to the DATA VALUE, except for odd addresses or for ranges, in which case only Byte or None are available. This is useful when you are looking for a change of state of a variable or a bit in memory. The breakpoint equal/not equal is only available on byte data, not word data.

See the Sequential Breakpoint Detection section for more information.

## NOTES ON BREAKPOINTS

### Hardware/software breakpoint implementation

Breakpoints defined with the verbs read, write, fetch, or any are detected with AT PROBE hardware. Four hardware break registers are available in AT PROBE. Software breakpoints defined with the verb execute are implemented by temporarily putting an INT 3 instruction at the breakpoint address. A maximum of four hardware breakpoints (2) and fifteen software breakpoints may be active when program execution begins including both STICKY and NON-STICKY breakpoints. AT PROBE will report an error if an attempt is made to activate more than four hardware breakpoints or fifteen breakpoints at once.

When a breakpoint is set for a read or write transaction, some additional instructions may be executed after the breakpoint. This is because the instruction is in the 80286 queue and cannot always be stopped from executing when the breakpoint occurs.

If no breakpoint is detected, you can regain control by pressing the STOP button on the external switch box.

### Parity errors

A parity error will cause a breakpoint. AT PROBE reports a breakpoint which has been caused by a parity error.

The ANY verb can be used to watch activity on either a single port or over a range of ports. To use the ANY verb on port I/O, prefix the address of the ports being watched with the pseudo segment "0:". If "0:" is not used, PROBE assumes the current value of DS: as the data segment value. Thus, to break on any port activity from ports in the range of 300 to 304 hex, use the ANY verb with a starting value of 0:300 and an ending value of 0:304.

(2) Each hardware breakpoint uses at least one hardware break register and a single hardware breakpoint may require all four break registers.

**Use of NMI**

The AT PROBE uses the NMI (non maskable interrupt) to cause a trap during all breakpoints which are not instruction execution breakpoints. The STOP button also uses the NMI.

**Multiple breakpoints consumed**

There are some conditions in the Breakpoint screen which use more than one of the available hardware break registers to implement a breakpoint. The most important if these conditions are listed below: the complete set of rules is in the "Breakpoint Rules" section.

1. Range breakpoints with data (only bytes are allowed) always take all four hardware break registers.
2. Range breakpoints with no data which cross 256 byte boundaries take 2 break registers.
3. Other ranges take one break register.

**Breakpoint restrictions**

Range Breakpoints have the following restrictions:

a. Verb can be anything but Execute.

b. Range breakpoints may only have a data size of Byte or None.

Since execution breakpoints are implemented via software interrupts, they may not be used for causing a break in prom memory. To do this, use a fetch breakpoint on the address in prom. This will cause a break when the instruction is fetched but not necessarily executed. This is because the 80286 CPU fetches its instructions into a queue before executing them. Instructions which are fetched into the queue are not necessarily executed since they may be preceded by a jump instruction which clears the queue. This is only a problem when trying to break on an instruction execution in prom memory.

**Sticky and non sticky breakpoints**

Breakpoints which are defined in the Go command or by typing F7 are activated for the duration of program execution and then are removed once control has been returned to the AT PROBE. These are commonly referred to as "NON-STICKY BREAKPOINTS." Breakpoints which are previously defined and activated by the Breakpoint command will be in effect when a Go command is issued, even if they are not specified in the new Go command. This is also true of breakpoints set by typing the F9 key. These are commonly called "STICKY BREAKPOINTS.".

Since the syntax of a sticky breakpoint is not interpreted until it is used, a syntax error in the breakpoint will not be found until the Go command is executed or a function key starts execution. Breakpoints set with the Go command are not sticky breakpoints.

## BREAKPOINT EXAMPLES

Define and activate breakpoint number 0 which detects a write to the range of addresses starting at 1000:0 and ending at 1000:4F6 when the value written is 1234. Execute macro <Alt-J> after the breakpoint. The key sequence to enter this breakpoint is shown followed by the Abbreviated Breakpoint Summary.

**B D 0 W 1000:0 <TAB> +4F6 <TAB> <TAB> B 1234 <TAB> <TAB> <TAB> M <Alt-J> <TAB> A <ESC>**

| BP # Status | Address | [to address] | Verb | Data | When detect |
|---|---|---|---|---|---|
| BP 0 Active | 1000:0 | +4F6 | Write | Byte | <Alt J> |

Define and activate breakpoint number 1 which traps executing an instruction at location PROMPT in module MAIN.

**B D 1 <TAB> MAIN.PROMPT <CursorUp> <CursorUp> A <ESC>**

| BP # Status | Address | [to address] | Verb | Data | When detect |
|---|---|---|---|---|---|
| BP 1 Active | MAIN.PROMPT | | Execute | | Stop |

Define and activate breakpoint numbers 2, 3, 4, and 5 for executing instructions at line numbers 45, 49, 78 and 118 in the current module.

**B D 2 <TAB> #45 <TAB> <TAB> A <ESC> 3 <TAB> #49 <TAB> <TAB> A <ESC> 4 <TAB> #78 <TAB> <TAB> A <ESC> 5 <TAB> #118 <TAB> <TAB> A <ESC>**

| BP # Status | Address | [to address] | Verb | Data | When detect |
|---|---|---|---|---|---|
| BP 2 Active | #45 | | Execute | | Stop |
| BP 3 Active | #49 | | Execute | | Stop |
| BP 4 Active | #78 | | Execute | | Stop |
| BP 5 Active | #118 | | Execute | | Stop |

Define and activate sticky breakpoint 6 to detect a FETCH from the lower 4k bytes of memory. Execute the macro assigned to the key <Alt-F> when this breakpoint occurs.

**B D 6 F 0:0 <TAB> +3FF <TAB> <TAB> <TAB> M <Alt-F> <TAB> A <ESC>**

| BP # Status | Address | [to address] | Verb | Data | When detect |
|---|---|---|---|---|---|
| BP 6 Active | 0:0 | +3FF | Fetch | None | <Alt F> |

Define and activate sticky breakpoint 7 to trap Any type of access between locations 8000:0 and 8000:FFF.

**B D 7 A 8000:0 <TAB> +FFF <TAB> <TAB> <TAB> <TAB> <TAB> A <ESC>**

| BP # Status | Address | [to address] | Verb | Data | When detect |
|---|---|---|---|---|---|
| BP 7 Active | 8000:0 | +FFF | Any | None | Stop |

For the next two examples, logic lines must be enabled through the Breakpoint Logic lines command.

Define and activate sticky breakpoint 8 to trap if the data pattern AAAA is written to location 1000:0 and bit 0 of the AT PROBE Logic lines is equal to 1.

**B D 8 W 1000:0 <TAB> <TAB> <TAB> W AAAA <TAB> <TAB> <TAB> XX1 <TAB> <TAB> <TAB> A <ESC>**

Define and activate a sticky breakpoint 9 to trap on read of IO port with the symbol PORT5.

**B D 9 I PORT5 <TAB> <TAB> <TAB> <TAB> <TAB> A <ESC>**

| BP # Status | Address | [to address] | Verb | Data | When detect |
|---|---|---|---|---|---|
| BP 9 Active | PORT5 | | Input | None | Stop |

Go back and redefine breakpoint 9 to make it trap on IO write to PORT5.

**B D 9 O <ESC>**

## SEQUENTIAL BREAKPOINTS

AT PROBE has the capability of detecting complex breakpoint sequences. Sequential breakpoints will not execute in real time. If you select Armbp from the field "When breakpoint detected" on the breakpoint screen, then following prompt appears in the Breakpoint screen:

*Arm bp:[ ]*

One breakpoint can arm or enable the detection of a following breakpoint. The breakpoint number which you insert in the current breakpoint definition tells AT PROBE to start looking for the next breakpoint when the current breakpoint occurs.

If you select Reset trigger from the field "When breakpoint detected" on the breakpoint screen, then the breakpoint is defined to clear the arming conditions set by all the other breakpoints. When a breakpoint with a Reset trigger field defined is detected, AT PROBE clears all Arm bp conditions which have been detected during program execution. After the Reset trigger has occurred, the process of arming breakpoints restarts again.

## SEQUENTIAL BREAKPOINT EXAMPLES

### THE BUG:

A stack variable is being randomly changed. The procedure which owns the variable, however, is OK and does not appear to be overwriting the variable. It is probably being overwritten by another procedure. The procedure which owns the variable writes to it all the time so you don't want to simply trap on a write to the variable. What you want is to find a write to the variable which is coming from anywhere other than the current procedure. AT PROBE can handle this with no problem. In words,

Breakpoint 0 is set for the address of the call which is suspected to be overwriting the local variable. The breakpoint verb is execute, and it arms Breakpoints 1 and 2.

Breakpoint 1 is set to Stop on a write to the local variable.

Breakpoint 2 is set for the address immediately after the address which causes breakpoint 0 to be triggered. The breakpoint verb is execute, and this breakpoint resets the chain.

The screens for these breakpoints look like this.

```
Breakpoint
  Define                                                  <Esc> prev screen
                      BP0 Status:<Active>{Clear|Active|Inactive}
Verb:<Execute>        {Execute|Read|Write|Fetch|Input|Output|Any}
Address:<call-start> to <>


When BP detected: [Armbp]    {Stop|Macro/watch|Arm bp|Reset trigger}
Armbp:<1 2>
```

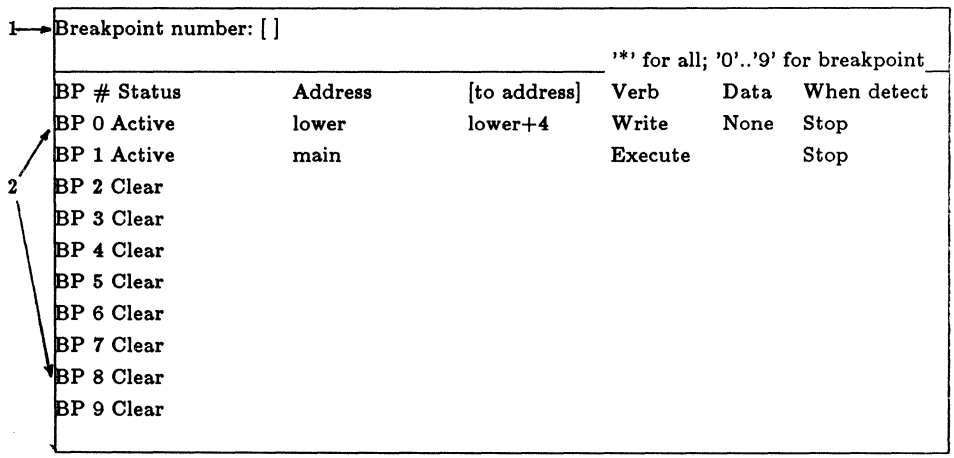
```
Breakpoint
  Define                                                    <Esc> prev screen
                        BP1 Status:<Active>{Clear|Active|Inactive}
Verb:<Execute>          {Execute|Read|Write|Fetch|Input|Output|Any}
Address:<stackvar-name> to <>
}
Data size:    <None> {Word|Byte|None}
When BP detected: <Stop>    {Stop|Macro/watch|Arm bp|Reset trigger}
```

```
Breakpoint
  Define                                                    <Esc> prev screen
                        BP2 Status:<Active>{Clear|Active|Inactive}
Verb:<Write>            {Execute|Read|Write|Fetch|Input|Output|Any}
Address:<call-end> to <>
}

When BP detected: <Reset trigger>      {Stop|Macro/watch|Arm bp|Reset trigger}
```

When program execution starts, the sequential breakpoint detection is triggered when the program executes the call instruction to the function suspected as causing the memory overwrite. If a write to the variable occurs, the AT PROBE stops. If, however, the program reenters the calling procedure, the sequential breakpoint detection is reset so that a write to the variable from this procedure can occur without stopping the AT PROBE.

## BREAKPOINT ACTIVATE, INACTIVATE, AND CLEAR

Three other subcommand choices for the Breakpoint command are Activate, Inactivate, and Clear. Inactivate lets you have breakpoints which are defined but are not enabled when the Go command or a Function key starts program execution. Clear lets you delete the Breakpoint definition for selected or all breakpoints. The following screen appears for these subcommands:




1 The DIALOG BOX prompts you for the sticky breakpoint number to activate, inactivate or clear:

*Breakpoint number: [ ]*

Select a breakpoint number or type '*' for all breakpoints.

2 An abbreviated breakpoint summary is shown to make it easy to see the current status of sticky breakpoints.

## EXAMPLE ACTIVATE/INACTIVATE BREAKPOINTS

Inactivate all sticky breakpoints.

**B I ***

Activate sticky breakpoints 1,2, and 5.

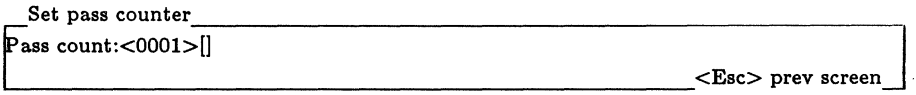
**B A 1 2 5 <Esc>**

Clear sticky breakpoints 3 and 4.

**B C 3 4 <Esc>**

## BREAKPOINT SET PASS COUNTER

By setting the pass counter, you can trap on the Nth breakpoint detected. All breakpoints that stop, execute a macro, display a watch window, or reset a chain will be counted. Breakpoints which arm other breakpoints are not counted.




If you type <enter> without a pass count, the pass count will not be changed. The pass count can have values from 1 to FFFF. The Breakpoint Set Pass Counter does not work in real time.

## BREAKPOINT RULES

For a discussion of the terms used in discussing breakpoints, see Appendix K.

1. For range breakpoints, the address that appears on the address bus must be within the range specified, or the breakpoint event will not be recognized.

   Example:

   Memory write breakpoint on range 101-107. A word write to address 100 will not trigger the breakpoint.

2. For range breakpoints with data, a false trigger may occur if the address that appears on the bus is within the range, but the data falls just above the range.

   Example:

   Memory write breakpoint on range 101-106 with data 12. A word write to address 106 with data 1234 will break, even though the 12 was written to address 107.

3. A range breakpoint can be programmed in only one hardware break register if it meets the following criteria:

   A. It does not cross a 256 byte boundary.
   B. It has no data.

4. A simple breakpoint with word data needs only 1 hardware break register. However, word data must start on a word boundary. If a breakpoint is to be set on a non-word boundary, it must be in the form of a range of byte data. This would require all four break registers.

5. If a pass count is used with breakpoints, the user's program will not execute at full speed. Each time a software breakpoint is hit, the AT PROBE checks to see if the pass counter has counted enough repetitions, and restarts the user's program if not.

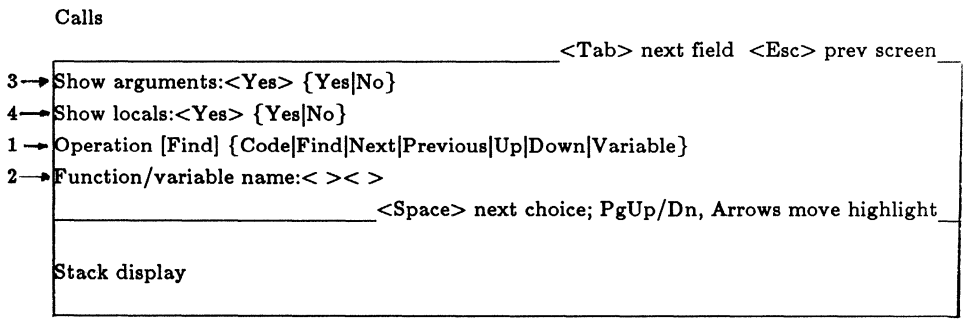
# CALLS

The Calls command lets you display current stack based information which includes local variables, calling routines, and return addresses. You can set sticky or non-sticky breakpoints on highlighted function names, view or execute functions. You can also search for specified strings within the stack display and change the value of variables.

The Calls command is invoked from the MENU BAR by typing:

**C** (for Calls)

The following screen appears:




1 The first prompt in the DIALOG BOX is for the Code subcommand.

If the CODE SCREEN is in "source" mode, typing C for Code will display the source code for the function which is highlighted. When the code is displayed, the Search command DIALOG BOX will be provided at the top of the screen to enable you to specify strings to search for in the code. See the Search Command for description of the subcommands that are now displayed and for an explanation of the breakpoints that can be set on the code screen using function keys.

If the CODE SCREEN is in "assembly" mode, the Code subcommand will unassemble and display the code for the highlighted function (See the Memory Unassemble Assemble command for a description of the AT PROBE standard line assembler.

Before typing "C", you can use the <F3> key to select source or assembler code to be displayed when the Code subcommand is invoked. The subcommand will display the code format that was last displayed on the CODE SCREEN unless you first press <F3> to select source or assembler.

The default is F for Find. The Find subcommand will search for occurrences of the string specified in the next prompt. The search will be conducted in the function or variable name portion of each line in the display. The search is not case sensitive.

Typing N or P will search for the next occurrence of the function or variable name. When no further matches can be found in the direction of search, the following message will be displayed:

*Function/variable name not found*

Typing U will move the highlight up the screen (down in the stack) to the called function name line. Typing D moves the highlight down the screen (up in the stack).

V for Variable can only be selected in the SOURCE PROBE. This command allows you to display variables according to their type and to change their value. Refer to the Memory Variable command for a general explanation of this subcommand. In addition to the basic capabilities of the Memory Variable command, Calls Variable provides the ability to examine and change a specific instance of a local variable when more than one instance of that variable was created through recursion. This is accomplished by highlighting the particular instance of that variable which is to be examined or changed, and then pressing V for Variable. Similarly, Calls Variable can be used to examine other stack based variables. If Calls Variable is used to inspect a variable on the stack which is other than the most recent instance

of that variable, the variables address will be entered into the "Variable name or address" DIALOG BOX. This is to distinguish it from other instances of the same variable.

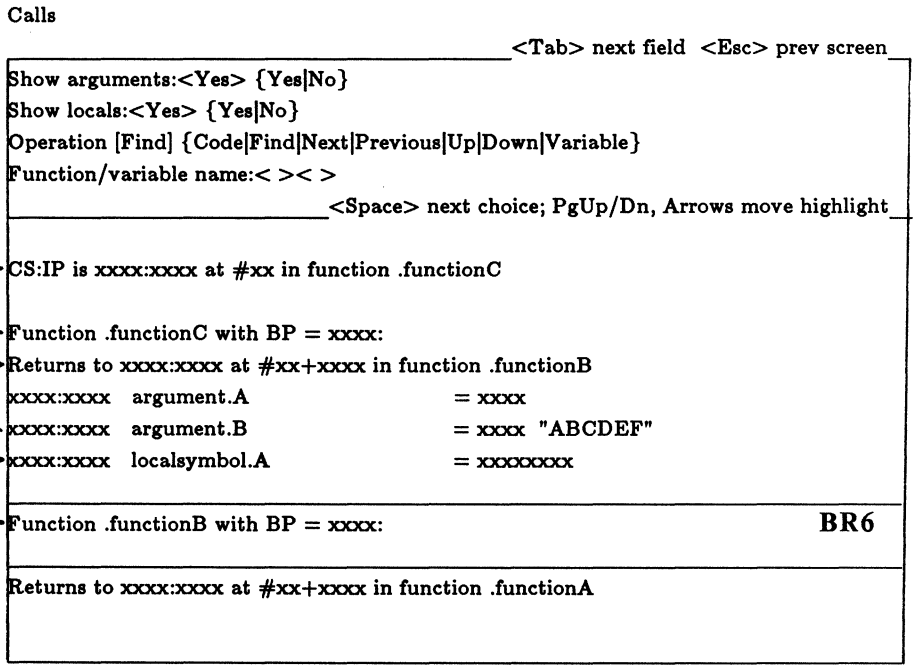
If a local register variable is on the stack - but is not in the current procedure - a "?" will be displayed for its value since the AT PROBE does not know its value.

2 The next prompt is for the Find subcommand. Type as many characters of a function or variable name as needed to correctly identify it.

3 Selecting 'Yes' for this prompt will display function arguments in the display of stack information. Selecting 'No' will clear the display of function arguments. This prompt is only displayed in the SOURCE PROBE.

4 Selecting 'Yes' for this prompt will display local symbols in the display of stack information. Selecting 'No' will clear the display of local symbols. This prompt is only displayed in the SOURCE PROBE.

The display of the stack information is as follows:




1 The address of the instruction pointer is displayed at the top of the screen together with the name of the currently executing function.

2 The current procedure name and the BP are displayed next.

3 The return address to the calling procedure.

4 Function arguments are displayed when "Yes" has been selected for "Show arguments". For character arrays, as much data as will fit on the right side of the display will be shown (for both arguments and local symbols).

5 Local symbols are displayed when "Yes" has been selected for "Show locals".

6 Breakpoints can be set on highlighted function name or return address lines and execution begun using the F5, F7 or F9 key.

## EXAMPLES OF USING THE CALLS COMMAND

After a breakpoint has occurred, display the calling sequence and show both arguments and local symbols.

**C**

Set a temporary breakpoint on returning to the function that called the currently executing function and begin execution.

**C <DnArrow to calling function> <F7>**

Display the current calling sequence and find any local symbol containing the character string "payrate".

**C <TAB> PAYRATE <enter>**

Find the next occurrence of a local symbol name with the same letters in it.

**N**

Look at the source code for one of the functions listed in the display of the calling sequence.

**C <DnArrow to desired function> C**

Move the highlight to next function "up the stack" (down screen).

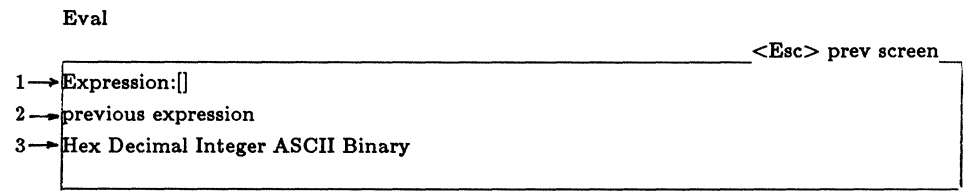
**C D**

# EVALUATING EXPRESSIONS

This command lets you evaluate expressions and display the results in several different bases. The command is invoked by typing:

**E** (for Evaluate)

The following DIALOG BOX appears:




1 You can type in any expression in the DIALOG BOX using the operators described in Chapter 3. After you type <enter>, this field is cleared and the expression and its results are displayed on the next two lines.

2 The expression just entered is displayed here.

3 The results of the expression just entered are displayed in the DIALOG BOX with the following formats:

HEX DECIMAL INTEGER ASCII BINARY

If the expression contains a segment and an offset it is evaluated to a 8 hex character number. This command serves to calculate real mode addresses in the same way as done by the 80286.

## EXAMPLES OF THE EVALUATE COMMAND

Evaluate expression: ((50*10)-1)/499

```
00000001H 1T +1T '.' 00000000,00000000,00000000,00000001Y
```

Evaluate expression: 'a'

```
00000061H 97T +97T 'a' 00000000,00000000,00000000,01100001Y
```

Evaluate expression: SS:SP+5 where SS=3000 and SP=0

```
00030005H 196613T +196613T '..' 00000000,00000011,00000000,00000101Y
```

Evaluate expression: ((AX-5)*2/10) where AX=1005H

```
00000200H 512T +512T '..' 00000000,00000000,00000010,00000000Y
```

Evaluate expression: 10T

```
0000000AH 10T +10T '.' 00000000,00000000,00000000,00001010Y
```

Evaluate expression: -1

```
FFFFFFFFH 4294967295T -1T '.' 11111111,11111111,11111111,11111111Y
```

Evaluate expression: AAAA:FFFF

```
000BAA9FH 764575T +764575T '...' 00000000,00001011,10101010,10011111Y
```

# FILE

The File commands let you load a program and symbol table, initialization conditions, Quit, or save the AT PROBE output from a debugging session in a log file.

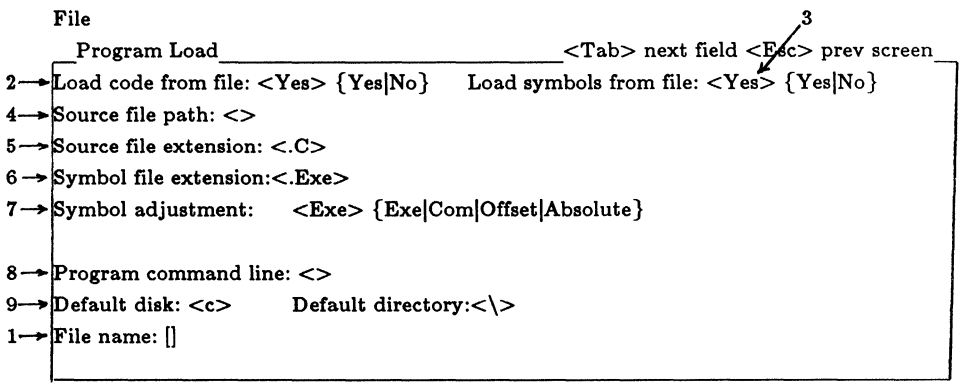
The File command has the following Subcommands:

| Subcommand | Operation |
|---|---|
| **Program load** | Load program, symbols with options |
| **View** | View files on disk (up to 10) |
| **Initializations** | Load AT PROBE initialization conditions |
| **Quit** | Return to DOS or run in the background |
| **Log file** | Log a debug session for later review |
| **Revision** | Returns the current version of the AT PROBE software. |

## FILE PROGRAM LOAD

This subcommand loads program, symbols, sets load options, and allows for entry of a program command line. Programs and symbols can also be loaded from the command line when invoking AT PROBE (See Chapter 2).

The following screen appears:




1 First you are prompted for the name of the file to load. If you cannot remember the name of the program, then use the wildcard capability of AT PROBE to display files. For example, if * is supplied for filename, the files in the current directory are shown in the DISPLAY WINDOW.

The program is loaded into system memory and the symbol table for the program is loaded into the AT PROBE symbol table. If the file contains only code and no symbols, the symbol table is not loaded.

2 This field lets you disable the loading of program code when the file is already loaded. For example, you would not want to load code if the program were already resident in memory and you simply wanted to attach a symbol table. Note, however, that since AT PROBE does not know where your program is located in memory when AT PROBE does not load the program, you will have to supply an adjustment for program symbols by setting the Symbol Adjustment field.

3 This field lets you disable the loading of program symbols when the program is loaded. Load time is reduced when symbols are not included. You would want to load without symbols to reload an already loaded program. The same symbol table will be used for the program in this case. Note that the sYmbol Load module command can be used to load only part of the symbols.

4 This field lets you enter the path for your source code files.

5 Field 5 is used when loading symbols from a .MAP file, but not for symbols from an .EXE file.

6 If the symbol table is in the .exe file, then the default .exe for the symbol file extension is used. This can be changed to .MAP to load from a MAP file.

7 This field tells AT PROBE how the symbol table file is related to the loaded program. The normal and default entry for this field is <exe>. When AT PROBE reads the symbol table from the program file, it automatically adjusts the symbol table addresses per the absolute locations of the loaded segments.

If <com> is selected, AT PROBE adjusts the location of the code segment down 100 hex locations to fit the .com format.

If either Offset or Absolute is selected the following prompt appears:

*Symbol adjust amount<0000:0000>*

The numbers before the colon are taken as the segment and the numbers after are taken as the offset. In the case of Offset, the address of each symbol is determined by the equation:

**program load address + offset + symbol address**

where:

program load address comes from the DOS loader
offset is the number you set in the [] field
symbol address comes from the symbol file

In the case of Absolute, the address of each symbol is determined by the equation:

**absolute + symbol address**

where:

absolute is the number you set in the [] field
symbol address comes from the symbol file

The absolute case is very useful for attaching symbols to loaded device drivers, quit and stay resident programs, or programs running on Non-DOS operating systems. See "Debugging a Device driver which installs itself" in the Advanced Debugging section for more details.

8 This field lets you pass a command line to the loaded program through the program segment prefix in the normal manner.

9 These fields let you change the default settings for the drive and path.

## EXAMPLES OF USING THE FILE PROGRAM LOAD COMMAND

Load the program VIEWORD.EXE with the default settings of the File Program Load subcommand.

**F P VIEWORD.EXE <enter>**

Load a the program INVEST.EXE with symbols loaded from the .EXE file. The executable file is in the default directory but the C source files are in PROJ\SOURCE. The program command line will pass the parameter "3" to the program.

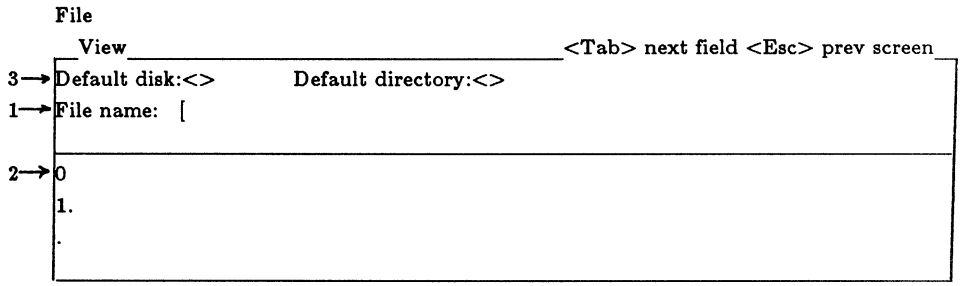
**F P INVEST.EXE <TAB> <TAB> <TAB> PROJ\SOURCE <TAB> <TAB> <TAB> <TAB> 3 <TAB> <TAB> <TAB> <enter>**

CMDRES.EXE has already been loaded into memory. Symbols loaded from the .EXE file are to be loaded at an absolute address of 2A4D:0.

**F P CMDRES.EXE <TAB> N <TAB> <TAB> <TAB> A 2A4D:0 <TAB> <TAB> <TAB> <enter>**

## FILE VIEW

Files can be viewed from AT PROBE while debugging with the File View command. This command displays the following window:

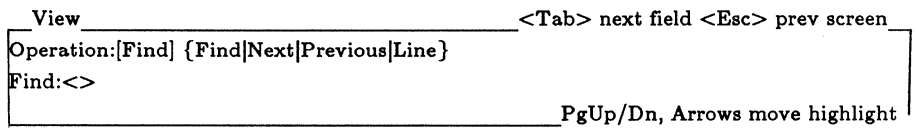



1 The DIALOG BOX first prompts you for the filename or file number. If you cannot remember the name of the file, then use the wild card (*) capability of AT PROBE to display files. In order for <F5>,<F7>, and <F9> to work correctly, the file name must exactly match the file name in the sYmbol Module-to-file-assignment command.

2 The file names you have previously viewed are shown in the DISPLAY WINDOW and they are assigned a number. Typing the number of the file and <enter> selects the file. If there are already 10 files assigned on the screen and you want to open another, you will be prompted for the number of a file (0 to 9) to close. When a file is opened and displayed in the DISPLAY WINDOW, it is automatically positioned to the point in the file where it was previously viewed. This lets you move quickly between 10 files without re-positioning the cursor in each file.

3 This is the default drive and pathname. Type <TAB> to get to these DIALOG BOXES to change the defaults.

Once the file name has been entered, the DIALOG BOX changes to the following:




The operation of this DIALOG BOX is the same as the DIALOG BOX for the Search command (see Search command in this chapter). The highlight field can be moved with the PgDn/PgUp and cursor keys. To move to the top or bottom of the file, use the Ctrl PgUp or PgDn keys. Like the Search command, breakpoints can be set at the highlighted line in the file and execution begun using the appropriate F5, F7 or F9 key.

## EXAMPLES OF USING THE FILE VIEW COMMAND

Display all files in the current pathname. The key sequence is:

**F V * <enter>**

Display all files in drive A: with a .HEX extension and override the default directory with "\".

**F V <TAB> A <TAB> \ <TAB> * .HEX <enter>**

View the file INIT.C in the default drive and directory.

**FVINIT.C <enter>**

View the file with the pathname A:\SRCFILES\MAIN.C:

**F V A:\SRCFILES\MAIN.C <enter>**

Change the default drive and directory to A:\TEMPFILES

**F V <TAB> A <TAB> \TEMPFILES**

View the file named FTOCIO.C. Display line #103 in this file.

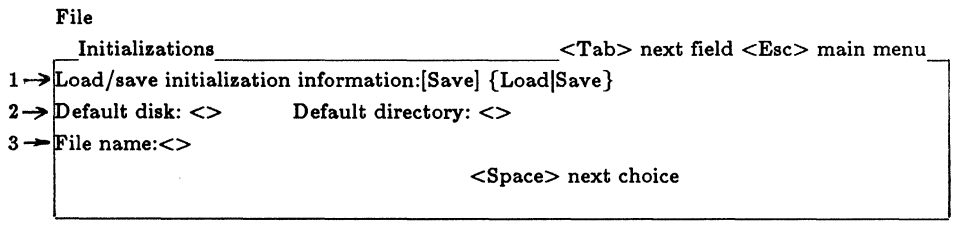
**F V FTOCIO.C <enter> L103 <enter>**

View a file which has already been assigned to number 4 in the DISPLAY WINDOW.

**F V 4 <enter>**

## FILE INITIALIZATIONS

Parameters which you set in AT PROBE can be saved to a disk file and recalled in future debugging sessions with the Init command. Initialization information stored in a previously saved file can be loaded to completely set up AT PROBE for a debugging session.




1. First you are prompted to Save or Load the initializations. Save is the default.

   After initializations have been saved, they can later be loaded using this DIALOG BOX or through the command line when AT PROBE is invoked (See Chapter 2).

2. The next prompts are for the default disk and directory.

3. The last prompt is for the initialization filename. The default drive and directory shown on this screen are used if you do not specify them. If you cannot remember the name of the file you want to use then type * and all files in the directory will be displayed.

The following items are saved with the initializations.

1. The Options are the settings found under the Options command.

2. The modulename settings found under the sYmbol command such as:

   a) Modulenames put into AT PROBE'S modulename table.
   b) If symbols will be loaded for the modulename when the Load command is invoked.
   c) If source level single stepping is to include or ignore a module when the Step Source command is invoked.
   d) File names assigned to the module names for use during Step Source commands.

3. The most recently loaded program.

4. Macros from a Macrofile. If AT PROBE has currently defined macronames which are the same as macronames found in the loaded Macrofile, then those macronames are left unchanged when loading initializations.

5. Window definitions from a window file are loaded. If AT PROBE has currently defined windownames which are the same as windownames found in the loaded file, then those windownames are left unchanged when loading initializations.

The information is stored as ASCII text in the initialization file. You may edit this text off-line with a text editor. The definition of the contents of this file is shown in Appendix G.

## EXAMPLES OF THE INITIALIZE COMMAND

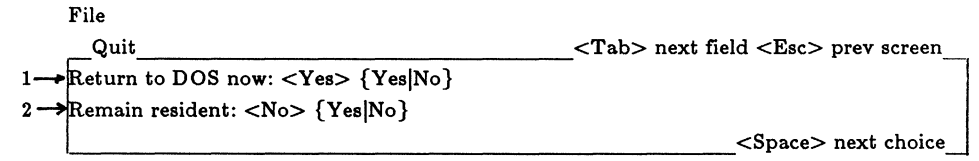
Display the files in directory \MAIN\DEMOFILES which have the extension .INI and save the current setup in an initialization file in this directory called DEMO.INI

**F I <TAB> <TAB> \MAIN\DEMOFILES <TAB> *.INI <enter>**
**<TAB> <UpArrow> DEMO.INI <enter>**

To load the initialize file named DEMO.INI from drive C, directory \DEMOS type this key sequence:

**F I L C <TAB> DEMOS <TAB> DEMO.INI <enter>**

## FILE QUIT




1 AT PROBE restores the user's screen and returns control to the operating system. This also removes the AT PROBE program prefix segment and restores all interrupt vectors to their original value unless the Remain resident option is selected.

2 You must type the <Tab> key to get to the prompt to select the Remain Resident option. In this case, AT PROBE returns control to the operating system but the program prefix segment and interrupt vectors 1, 2, and 3 remain in memory.

The Remain Resident command may only be issued once and the AT PROBE will exist in memory until the next RESET. In order to re-enter the AT PROBE you must press the STOP button. From then on, the Go command should be used to return to the DOS command level.

The remain resident option is useful whenever the File Program Load command cannot be used to load a program. For instance, it can be used when debugging installed device drivers, and user quit and stay resident programs.

The AT PROBE File Program Load command cannot be used to load a program after you have quit AT PROBE with the Remain resident option. AT PROBE can, however, still load a symbol table.

Sticky breakpoints are not active when you return to DOS through the Quit and remain resident option. They are only set with the Go command. Therefore, after doing a Quit and remain, press the STOP button to return to AT PROBE, define breakpoints and do a Go command to continue execution with breakpoints set.

## FILE LOG FILE

You can redirect AT PROBE output to a log file. This lets you save the history of a debugging session. A log file can be a disk file, printer, communications port, or file on a remote file server on a network. AT PROBE simply calls DOS with the filename handle. DOS then writes the output to the specified device. When this subcommand is selected, the following DIALOG BOX appears:

```
    File
     Log file                                  <Tab> next field <Esc> prev screen
3→ Default disk: <c>      Default directory: <\>
1→ File name:[]
2→ Enable log:<No> {Yes|No}
```

1 You are prompted for the name of the log file. Here are the devices you can specify for a log file:

| Filename | Description |
|---|---|
| filename | DOS filename either local or remote on a network |
| lpt1: | Lineprinter 1 attached to the 80286 system |
| lpt2: | Lineprinter 2 attached to the 80286 system |
| com1: | Com1 port on 80286 system |
| com2: | Com2 port on 80286 system |

Insure that the line printer is attached and turned on, however, otherwise AT PROBE will wait for the print device indefinitely.

2 After the log option is selected, you return to the MENU BAR. If a log file is currently open, you can close it and save it by selecting No for this field.

3 You must type the <Tab> key to get to the prompt to change the default drive and directory. You can also override these defaults by specifying the drive and pathname in the filename.

## FILE REVISION

To examine the current revision of the AT PROBE software.

File
Revision

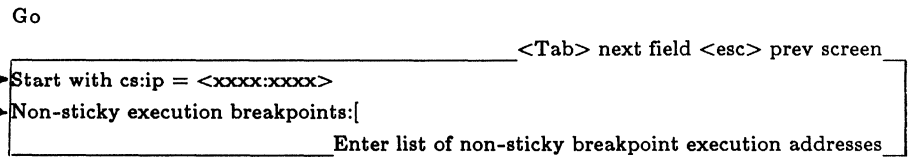
```
Atron's 80286 Source PROBE Version 04/06/88        (C)Copyright Atron 1987,1988
Configuration file is C:\PROBE\PROBE.CNF
```

# GO

The Go command executes the program being debugged. Execution starts at a specified address and stops when a defined breakpoint occurs. The Go command is invoked from the MENU BAR by typing:

**G** (for Go)

The DIALOG BOX for the Go command is as follows:




1 The Go command is optimized for quick start and/or quick set of simple execution breakpoints. Typing Go <enter> starts execution at current cs:ip. If you want to start execution from some other point, type <Tab> to get to the field which lets you change the start address.

You can optionally enter one or more addresses for non-sticky software execution breakpoints. You can enter as many as 15 breakpoints separated by commas.

All sticky breakpoints defined on the Abbreviated Breakpoint Summary screen are activated by the Go command.

For further discussion on breakpoints and an explanation of sticky and non-sticky breakpoints, see the Breakpoint command section in this chapter.

2 An address, symbolname or line number can be entered in this field as the starting address for program execution. The default is the current CS:IP.

If you want to execute a macro or display a window automatically after a Go command, then create a macro which imbeds the Go command into the macro. The macro can pause for you to set the breakpoints. When any breakpoint occurs, control returns to the macro and it can execute more commands or pop up a window.

## GO COMMAND EXAMPLES

Execute program starting at the current CS:IP displayed in the start field. Sticky breakpoints are already defined.

**G <enter>**

Execute program starting at 'init' in module chartio with breakpoints on lines 22 and 57 of the default module.

**G <UpArrow> ..chartio.init <TAB> #22,#57 <enter>**

Execute program starting at the current CS:IP and break on executing instruction at symbol 'tester' in module chartio.

**G ..chartio.tester <enter>**

Execute program starting at the current CS:IP and stops at CS:A333.

**G A333 <enter>**

# MACRO

Macros are a way to create your own commands and automate keystrokes which are repetitive. Macro commands are simply a group of keystrokes which are assigned to one keystroke. Each macro has a name which is the keystroke that invokes it. The commands to let you define, delete, edit, and display macros are invoked from the MENU BAR by typing:

A (for mAcro)

The Macro subcommands are shown here along with a short description of their operation.

| Sub command | Operation |
|---|---|
| Define | Define a new macro and assign it to an Alt-key |
| Edit | Change the definition of a current macro |
| Remove | Delete a currently defined macro. |
| Load | Load previously defined macros from file |
| Save | Save all currently defined macros to a disk file |

## MACRO DEFINE

To start the definition of a macro type subcommand D. The following screen appears:

```
    mAcro
     Define                                  <Tab> next field <Esc> prev screen
1 → Macro <Alt-Key>:     []
2 → Description:         <>
3 → Macro type:          <Normal>      {Normal|If|While|Count|Forever}

4 → Begin defining:<No> {Yes|No}
```

1 An Alt-Key means hold down the key Alt and then type any other key. The Alt-Key becomes the macroname. The Alt-key can be any key except <Alt-0>; thru <Alt-9>, <Alt-= (Alt equals)> and <Alt-- (Alt minus)>. These are saved for special use within the macro. <Alt-F10> is reserved for the Copy Paste process.

If the macroname already exists, you are given the following prompt:

*<Alt-Key> is a macro. Remove it? <Y> for yes, <any other> for no*

*or*

*<Alt-Key> is a watch window. Remove it? <Y> for yes, <any other> for no*

If you answer Yes, the current macro definition is deleted and you can proceed redefining the macro. If No, you are returned to the Macro <Alt-key> field. To change a macro use the Macro Edit command.

2 You are also prompted to add a Macro description to the Macro. This is an ASCII text string which describes your macro and can have up to 200 characters. The macro description is stored with the Macro. The Macro description can be viewed with the Macro Edit command. If you simply type <enter> in response to this prompt, no macro description is attached.

You cannot define a macro within a macro definition. AT PROBE commands are executed while the macro definition is in process.

3 You can define simple (normal) or conditional macros. For more details, see the next section "Conditional Macros". If you select Forever, the macro continues to execute repeatedly until the Stop button is pressed.

4 At this field you can start the macro definition process or type the <TAB> key to change other fields on this screen. Once macro definition begins, the MENU BAR reappears and the macro name appears in the lower right corner of the screen.

The macro is defined as all key strokes until the macroname (i.e. Alt-key) is typed again. The commands which define the macro also execute while the macro definition is in process. While the macro definition is in process, the macroname is displayed in the lower right area of the DISPLAY WINDOW to remind you that a macro is being defined.

Finish macro definition by typing the macro name again. (i.e. the Alt-Key)

## PASSING PARAMETERS TO THE MACRO

Parameters can be included in the macro definition. A parameter is defined as:

**<Alt-#> <enter>**

This means hold down the Alt key and type a number from 0 to 9. This is the reason that these Alt-Keys cannot be macronames. Up to 10 different parameters may be included in the macro definition, and each parameter may be used multiple times within the macro definition.

When the macro executes, it pauses the first time it encounters each different Alt-# and waits for you to input the parameter. The parameter is specified as all keystrokes you type until <enter> is typed. If there are 10 different #'s, then the macro will pause 10 times.

The same Alt-# may be used at several different places in the macro definition. When the .i.macro executes;, it will pause only the first time for this Alt-# parameter definition. The macro will use this definition each time it encounters this Alt-# during the current .i.macro execution;.

During definition of a macro which includes parameters (i.e.Alt-#'s), the parameter specification is passed on to the command in process. Commands execute while the macro is being defined. Note that the <enter> which specifies the end of the parameter during the definition or execution of the macro is not passed on to the command in process. Also note that you may define a macro which has parameters without actually having to use real parameters. This is done by simply typing:

**<ALT-#> <enter> <enter>**

This puts the ALT-# into the macro definition, and passes <enter> without a parameter to the command in process. This will typically produce an error message which you can simply ignore by typing any key.

A special Alt-Key is the <Alt--key (hold down the Alt key and type the minus sign). If <Alt--> is encountered, then the macro always pauses at the <Alt--> during execution to wait for input. This lets you enter a new parameter for this special Alt-Key each and every time it is encountered. This is especially useful in conditional macros which are described later.

While parameters are being entered into the definition of a macro, or the macro is pausing during execution to receive the parameter specification, the Alt-# for this parameter is displayed in the lower right of the screen. The definition of the parameters during macro execution is not maintained after the macro is finished executing. When this macro is invoked again, it will prompt you for the definition of each parameter. If you want the macro to always execute the same way, then do not use parameters.

## NESTING MACROS

A macro may be defined which invokes other macros during execution. This is done simply by typing the appropriate Alt-Key which invokes the nested macro into the definition of the macro. During execution, the nested macro may have parameters passed to it. You can think of Alt-#'s as global variable names which are recognized by both the outer level and nested macros.

## PASSING PARAMETERS TO NESTED MACROS

Parameters may be passed to the nested macro in one of two ways. The first time an Alt-# is encountered during macro execution, in either nested or outer level macros, it is specified. All further encounters of this parameter during execution of the outer level macro or the nested macro will use this specification for the Alt-# (i.e. parameter).

The second way of passing a parameter to a macro lets the parameter change each time the outer macro invokes the nested macro. This is done by using the following form to redefine a parameter during macro execution:

**<Alt-=> <Alt-#A> <Alt#-B>**

<Alt-=> means hold down the Alt-Key and type the = key. Alt-#B is the Alt-key to be redefined during macro execution. Alt-#A is the new Alt-key definition for Alt-#B during macro execution. When the Alt-= is encountered by the macro during execution, Alt-#B is replaced by Alt-#A. This lets you define macro modules which can be reused multiple times in a macro execution while having their parameters changed on the fly by other macros which call the macro module.

## DEFINING MACROS WHICH CONDITIONALLY EXECUTE

Macros can be defined to check for specified conditions before they will execute. The condition is tested each time the conditional macro is executed. This type of macro is defined by choosing one of the following Conditional types from the Macro Define screen.

| Type | Description |
|---|---|
| If | Start macro execution IF boolean expression is true |
| While | Execute macro again and again WHILE boolean expression is true |
| Count | Execute macro COUNT times |
| Forever | Execute macro continuously until the STOP button is pressed. |

### Count

If you select the Count conditional macro type, the following prompt appears:

*Loop count: [*

The macro will execute repetitively the number of times specified by the Count. Each time the macro reaches the end of its definition, it decrements the Count. When the Count reaches 0, this macro terminates execution. The Count can be an expression or simply a number. It can also be parameter passed from another macro. Since macros execute while they are being defined, the loop execution starts when the macro definition is ended (i.e. the Alt-Key is typed again.)

**While**

If you select the While conditional macro type, the following prompt appears:

*Condition : [*

The While condition is a boolean expression which is defined in Chapter 3 in the section called BOOLEAN EXPRESSIONS. The boolean expression is checked at the beginning of each execution of the macro. If it is true, the macro executes again. If it is false, the macro execution is terminated. Since you may be defining a conditional macro at a time when the condition is not true, the macro definition ignores the condition so that you can continue the definition.

**If**

Another type of condition for the Conditional macro definition is the IF condition. When this subcommand is selected, you are prompted with:

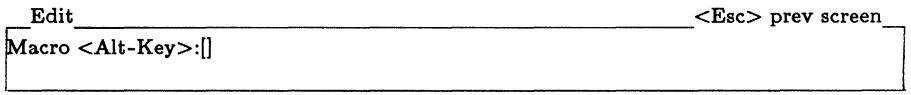
*Condition : [ ]*

The IF condition is a boolean expression which is defined at the start of this chapter in the section called BOOLEAN EXPRESSIONS. The boolean expression is checked at the start of execution of the macro. If it is false, the macro execution is terminated. Since you may be defining a conditional macro at a time when the condition is not true, the macro definition ignores the condition so that you can continue the definition.

In order for a macro parameter to be changed in a conditional macro that is in a loop, the Alt-- key must be used for the parameter. If Alt-- is encountered, then the macro always pauses at the Alt-- during execution to wait for input.

## MACRO EDIT

Once defined, macros can be edited by choosing the Edit subcommand from the MENU BOX. The names of the currently defined Macro and Watch window keys are listed in the DISPLAY WINDOW.




Type Alt-key to get macro definition or ? to see all currently defined macros. If the macro already exists, then the current definition of the macro is shown in the DISPLAY WINDOW as a sequence of keystrokes. A highlight field in the DISPLAY WINDOW can be moved with the cursor keys. The DIALOG BOX which lets you make changes to the macro definition looks like this:

*[ ]*

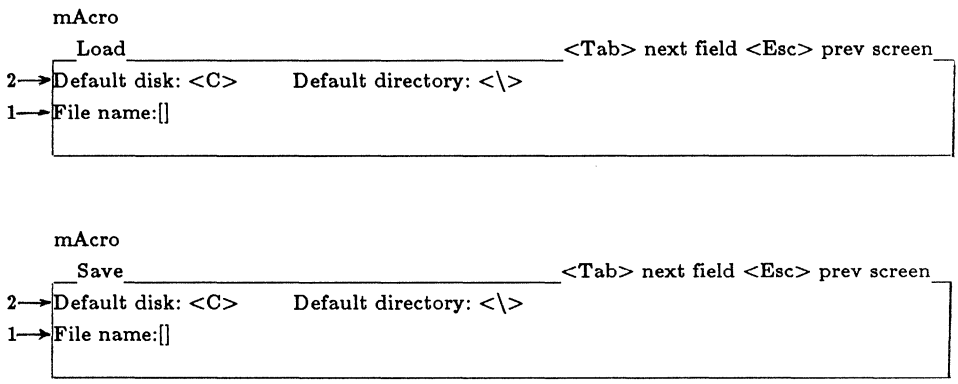
As the highlight field in the DISPLAY WINDOW is moved, the contents of the highlight filed is duplicated in the DIALOG BOX. You can use the AT PROBE edit keys to make changes, additions, or deletions to the keystrokes in the macro. You may put several keystrokes into the DIALOG BOX.

If you Edit a macro which does not exist, then the DISPLAY WINDOW only contains the name of the macro. The same DIALOG BOX as before is displayed. Key strokes are entered into the DIALOG BOX and transferred to the DISPLAY WINDOW when <enter> is typed. Using the Macro Edit command to define a new macro lets you assign pure ASCII text to a macro name. This is useful to simply save key strokes as with long complex symbolnames.

Since macros are saved in a file as simple ASCII text, they may be edited off line with your favorite text editor. The format of the macro text is described in Appendix F.

## MACRO LOAD AND SAVE

Macros can be loaded and saved from disk. The Load and Save subcommands prompt you for the filename:



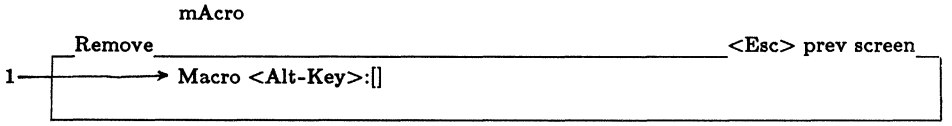

1 Type the name of the macro file to be loaded or saved.

2 You can change the default disk and directory by using <TAB> to get to these fields and enter a new pathname.

If the drive and directory are not specified, then the defaults shown on the screen are used. The wildcard character "*" can be used to display file names.

## MACRO REMOVE

A macro can be deleted by choosing the Remove subcommand from the MENU BOX. You are then prompted with:




1 Type A to remove all macros; ? to see all macros; or <Alt-Key> to select macroname.

The names of the currently defined Macro and Watch window keys are listed in the DISPLAY WINDOW.

By specifying the key which would normally activate this macro in response to this prompt, it is removed.

## MACRO EXECUTION

After a Macro has been defined, it can be executed by simply typing the Alt-Key which is the macroname. You will see the macro execute on the screen. If a macro pauses waiting for you enter to enter a parameter, the Alt-# for the parameter is in the lower right hand corner of the screen. The command in process which will accept the parameter is also shown on the screen.

If you want to bail out of macro execution, press the Stop Button.

## MACRO COMMAND EXAMPLES

Define a macro named Alt-A that will load a program and unassemble from the start of the program. The program name should be a parameter to the macro.

**A D <Alt-A> <enter> <enter> <enter> Y F P <ALT-0>**
**FTOC.EXE <enter> <enter> M U <enter> <Alt-A>**

This macro can now load the file FTOC.EXE with the following key sequence.

**<Alt-A> FTOC.EXE <enter>**

Define the same macro, but this time do not really load the file while the macro is being defined.

**A D <Alt-A> <enter> <enter> <enter> Y F P <Alt-0> <enter>**
**<enter> <ESC> M U <enter> <Alt-A>**

Note that the only difference is an <enter> where the file name was in the previous macro.

Edit the previous macro to delete the unassemble portion of the macro.

**A E <Alt-A> <enter> <DnArrow> <DnArrow> <DnArrow>**
**<DnArrow> <DelKey 9 times> <enter> <ESC>**

Use the <Alt-F10> key to capture a symbol named MAINMODULE.IOPROC.VARTEMP from the screen and store it as ALT-B (See Function Keys in Chapter 2)

**<Alt-F10> <cursor to beginning of symbolname> <Alt-F10>**
**<cursor to end of symbolname> <ALT-B>**

Define a macro named ALT-C which displays memory starting at an address passed to it from the previously defined macro ALT-B. Display 15 words at this address, pause to view memory and then press <ESC> and display the main registers.

**A D <ALT-C> <enter> <enter> <enter> Y M D W <ALT-B> <enter> +15 <enter> <ALT-0> <enter> <enter> <ESC> <ALT-C>**

Define a macro ALT-K that displays the file FTOCIO.C if the contents of memory location FILEPTR is "I". Enter the description of the macro as "Display file when flag is 'I'."

**A D <ALT-K> <enter> Display file when flag is "I" <enter> I [FILEPTR]B="I" <enter> Y F V FTOCIO.C <enter> <ALT-K>**

Define a macro which defines and activates a sticky breakpoint on writing to the variable FAHR. After each write, check to see if the contents of FAHR is greater than the contents of the variable FAHRMAX. If it is greater, stop the macro. If it is not, keep running the macro. Note that it would be better to define the breakpoint independently of this macro since it is defined through each loop.

**A D <ALT-X> <enter> <enter> W [FAHR]D<=[FAHRMAX]D <enter> Y B D 0 W FAHR <CTRL-CURSORLEFT> <enter> <UpArrow 3 times> A <ESC> <ESC> G <enter> <ALT-X>**

The following macros may be useful to define:

Define a set of macros which mimic the CodeView Ctrl key functions, i.e. make Alt-Function keys do the same thing as the CodeView Ctrl keys.

Define the <Alt-S> key as the single step key.

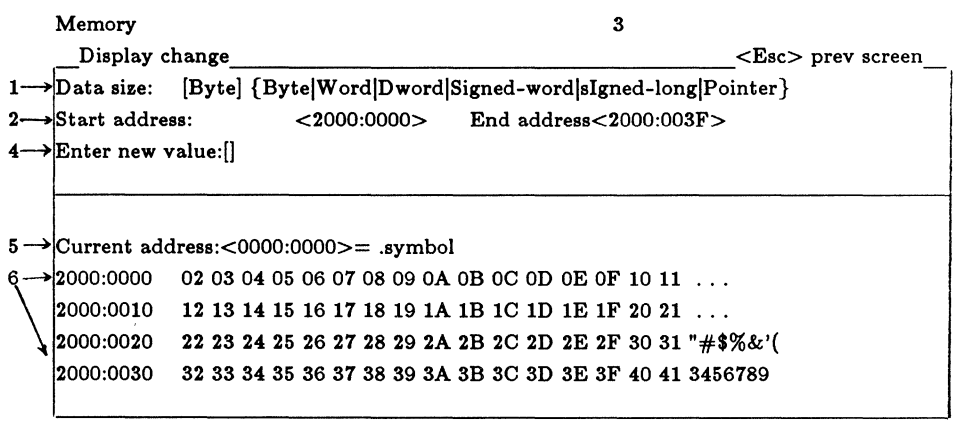
# MEMORY

The subcommands for the Memory command are:

| Subcommand | Operation |
|---|---|
| Display change | All non-float memory types |
| Io port | IO devices |
| Float display change | All floating point types |
| Unassemble assemble | Unassemble or assemble code into memory |
| Block operation | Operate on memory in blocks |
| Variable | Supports complex C data types |

## MEMORY DISPLAY CHANGE

You can display memory on screen in several different lengths and change it directly on the screen. Here is a sample memory display.




1 You select the type of the data to be displayed from this field.

2 This field prompts you for the start address of the memory to be displayed. You may enter any type of address expression. If you type <enter>, you will get the default start address shown on the screen.

3 This field prompts you for the end address. You may enter any type of address expression. The end address may also be of the form:

**+ number**

In this case the end address becomes start address+number. If you simply type <enter> without entering a new end address, you get the default end address shown on the screen. After the end address is selected, the memory is displayed in the DISPLAY WINDOW.

4 A location in memory is highlighted in the DISPLAY WINDOW. You are prompted to enter a new value into memory at this location. The value can be a symbolic expression, a number, or a string of "characters". If the value is a string, each character is written to the next unit (i.e. byte, word, or long). If the value is an expression, the standard AT PROBE editing keys can be used to make changes to the expression in the DIALOG BOX. After <enter> is typed, the value is deposited into memory and the highlight moves to the next address. The value deposited in memory is also displayed in the cell in the DISPLAY WINDOW unless you have the Read-after-write option set so that a read after write does not occur. You can move the highlight field in the DISPLAY WINDOW with the cursor and PgUp/PgDn keys. This lets you scroll through memory and make changes visually on the screen.

5 This is the current address of the memory location which is highlighted in the DISPLAY WINDOW. If the current address matches or is close to a symbol in the symbol table, then the symbol is also shown in this field along with the address.

6 A duplicate of the data is shown as ASCII in this area for the Display Byte command. The ASCII is not displayed for the Word.

Typing <TAB> while in the Memory Display command will move you to the previous fields. For example, you may want to display several small blocks of memory on the screen, one after the other. After displaying each block, type the <TAB> to select new start and end addresses. The previous blocks of memory are not erased from the screen but are scrolled up the screen.

## EXAMPLES FOR MEMORY DISPLAY CHANGE

This example demonstrates using the Memory Display command to show a line of words starting at location BUFFER in the current module. The key sequence is:

**M D W BUFFER <enter> <enter>**

To look at a screen-full of words at this point type:

**<PgDn>**

Change the value of 3 bytes of memory starting at TEMP to 0.

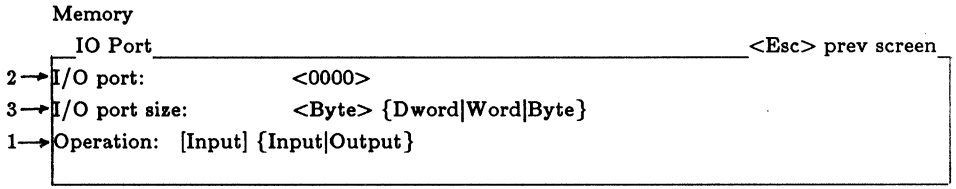
**M D B TEMP <enter> <enter> 0 <enter> 0 <enter> 0 <enter> <ESC>**

At 4000:0, change the first byte of memory to AA. At 5000:0, type in a pointer to 4000:0. Then display the byte at 4000:0 using the pointer.

**M D B 4000:0 <enter> <enter> A A <enter> <TAB> P 5000:0 <enter> <enter> 4000:0 <enter> <TAB> B [5000:0] P <TAB> +1 <enter>**

## MEMORY IO PORT

This command lets you display and change IO ports.




1 If Output is selected for the operation, you get the following prompt:

*Enter new value:[ ]*

Each time an output value is entered, the DISPLAY WINDOW shows:

Value output to port --> [IO port #]

You can continue to write the same value to this port by typing the <enter> key.

If the operation were Input, the DISPLAY WINDOW shows:

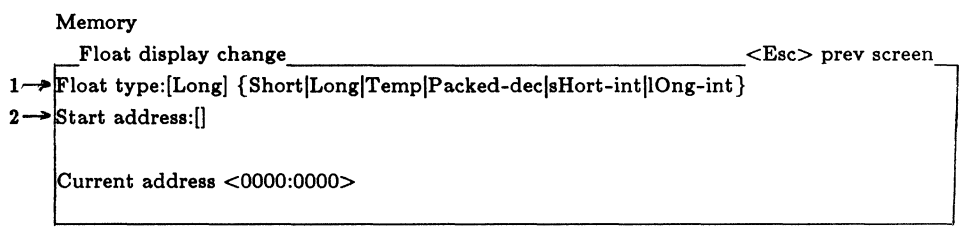
[IO port #] --> Value input from port

2 Tab to this field to select IO port #

3 Tab to this field to select port size.

## MEMORY FLOAT DISPLAY CHANGE

If you have a 80287 floating point co-processor in your system, you can display and change memory of a floating point format. When you select the Memory Float display command, the following DIALOG BOX appears:




1 These menu options let you select the floating point data type

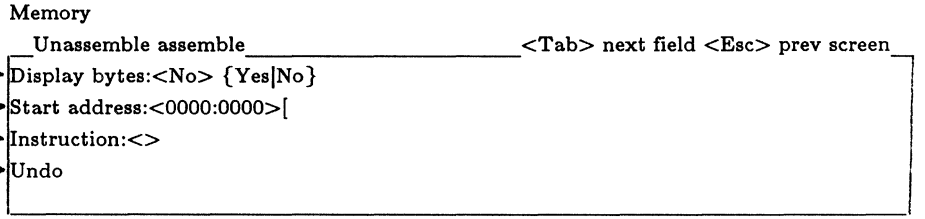
2 Once the start address is selected, the following prompt appears in place of the start address:

*Enter new value:[ ]*

If you do not have a 80287 in your target, the values displayed by this command are in hex. If you have the 80287, the conversions are done and the data is displayed in decimal in the standard floating point formats. Values can be entered in this format as well. In addition, the values of NAN (not a number) and INFINITY are also accepted. Once <enter> is typed, the next address is displayed. In this command, each line on the screen shows only one floating point value, since some values will fill nearly an entire line. The cursor keys, PgUp/Dn and highlighting work the same in this command as in all other DISPLAY WINDOWs.

## MEMORY UNASSEMBLE ASSEMBLE

Memory can be displayed as assembly language instructions with the Unassemble command. You can also enter new assembly code directly on this screen.




1 The first DIALOG BOX prompts you for the starting address of the memory to disassemble. If <enter> is typed with no start address then the default start address is assumed. After the start address is entered or the <default address> is taken, the DISPLAY WINDOW shows a page full of unassembled instructions with one line highlighted. You can move the highlight field with the cursor keys.

Note that symbols or line numbers which match the address fields are included in the unassembly to simplify the display. If you are running the SOURCE level version of AT PROBE, the source code which is associated with the unassembled instructions is also shown in the DISPLAY WINDOW.

2 This is the current default start address. It is the current program counter if a Go command or single step has been executed previous to this command. It is set to the last address of the previous Unassemble command display if no Go or single step has been recently executed.

3 Next, you are prompted to start assembling instructions. The instructions which are initially typed into the DIALOG BOX are transferred to memory and then unassembled into the DISPLAY WINDOW after each <enter>. The AT PROBE edit keys let you make corrections to the instructions in the DIALOG BOX. The new instruction overwrites the old instruction and unassembles the code on the screen directly under it. If the instruction is longer, the code below it will probably change. After typing <enter> you are prompted for a new instruction at the next address. To get to the other fields, type <TAB>.

The assembler will work for Co-processor instructions as well.

4 After an instruction is entered into memory, the following option appears in the DIALOG BOX:

*Undo last change: <No> {Yes|No}*

The Undo field gives you a chance to change the instruction you just changed back to the way it was.

5 This field lets you display the hex equivalent for instructions.

F5 and F7 set breakpoints on this screen, and start program execution. When a breakpoint is detected, control returns to the CODE SCREEN. F9 sets or clears sticky breakpoints.

## NOTES ON THE AT PROBE LINE ASSEMBLER

The following notes apply to the assembly language which is understood by the AT PROBE standard line assembler.

1. Standard assembly language mnemonics are used.
2. The assembler will automatically assemble short jumps and calls depending on the displacement of the destination address.
3. When a byte, word, or double-word size cannot be determined by the operand, the data type of the operand must be specified by byte ptr, word pt, or dword ptr.
4. Numbers are in hex. If a symbol is used as an address in a reference, only the offset will be used. If a symbol is used in a reference specified as FAR, then both the symbol's segment and offset values are used.
5. Prefix mnemonics are entered on a separate line.

   EXAMPLE:
   ```
   0642:0000 REP
   0642:0001 MOVSB
   0642:0002 LOCK
   0642:0003 XCHG BYTE PTR[.TEMP],AL
   ```
6. Segment override mnemonics are cs:, ds:, es:, and ss:

   EXAMPLE:
   ```
   0642:0000 MOV AX,CS:[.LOOPSTART]
   ```
7. The assembler will automatically assemble short or near jumps and calls depending on byte displacement to the destination address. The FAR prefix must be specified for inter-segment jumps or calls, otherwise, the current segment is used.

   EXAMPLE:
   ```
   0642:0000 JMP .LOOP          ;A 2 BYTE SHORT JUMP
   0642:0002 JMP FAR .START     ;A FAR JUMP
   ```

8. When a byte or word location cannot be determined by the operand, the data type of the operand must be specified with the prefix BYTE PTR or WORD PTR, DWORD PTR, QWORD PTR, etc.

   EXAMPLE:
   ```
   0642:0004 DEC WORD PTR [SI]
   ```

9. An immediate operand is distinguished from a memory location by enclosing the latter in square brackets.

   EXAMPLE:
   ```
   0642:0000 MOV CX,100              ;LOAD CX WITH 100H
   0642:0005 MOV CX,[100]            ;LOAD CX WITH THE
                                     ;CONTENTS OF MEMORY
                                     ;LOCATION DS:100
   0642:0007 MOV AL,CS:[.BUFFER]
   ```

10. All forms of register indirect commands are supported.

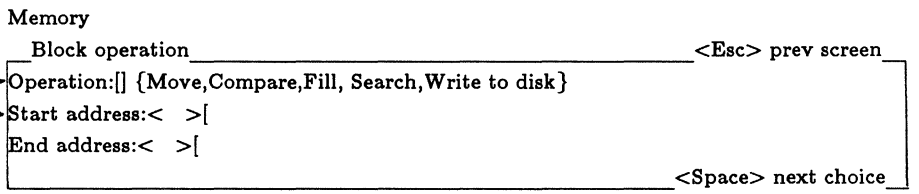
    EXAMPLE:
    ```
    ADD AX,[BP+SI+34]
    POP [BP+DI]
    PUSH [SI]
    ```

11. All opcode synonyms are supported.

    EXAMPLE:
    ```
    LOOPZ .LOOPONRDY
    LOOPE .LOOPONRDY
    JA .LOOPONRDY
    JNBE .LOOPONRDY
    ```

## MEMORY BLOCK OPERATIONS

The Memory Block operations lets you save, compare, find, move, and initialize blocks of memory.




1 First you select the type of block operation from these choices:

| Operation | Description |
|---|---|
| Move | Move block of memory to new destination |
| Compare | Compare two blocks of memory |
| Search | Find a string in memory |
| Fill | Fill memory with a string |
| Write to disk | Write a block to disk file |

2 Every operation prompts for both start and end address of the block to be moved, compared, searched, filled or written to disk.

The start and end address of the block are any type of address expression or simply type <enter> to accept the default start address shown on the screen. The end address may also be of the form:

**+ number**

In this case the end address becomes start address+number. If you simply type <enter> without entering a new end address, you get the default end address for the block.

Move and Compare both prompt for a destination address:

*Destination:[*

For the Move operation, the block defined by the start and end addresses will be moved to the destination address. Move can move an overlapping block in memory.

The Compare operation compares the block defined by start and end with a block of the same size starting at the destination address. Compare also has the following prompt:

*Report if bytes are: <Not equal> {Equal|Not Equal}*

The two blocks designated by the address prompts will be compared and a reporting of those bytes that match or don't match will be displayed in the DISPLAY WINDOW. If you chose for a report on those bytes which are not equal and they completely match, AT PROBE will report that they are identical. Otherwise, the reporting will have the following format:

| Offset | Src Dest Byte Byte |
|---|---|
| +0070 | 67-44 |
| +008F | 67-49 |

.

.

The Fill and Search operations prompt for the start and end address of the block and for specified bytes to fill the block or search for in the block. After the address prompts, the following prompt appears for these subcommands:

*Byte list:[*

For the Fill and Search operations, the string may be hex bytes, or an ASCII string in quotes.

The Fill operation fills the memory repeatedly with the list. The list written into memory at the destination address may only be partial since the block size may not be an even multiple of the list size. The list is treated as a sequence of bytes.

If the search command is used, it will further prompt for whether the search should be for a value equal to the search string, or a value not equal to the search string. If Search finds a match, it reports:

*Search string found at offset +xxxx*

The above message is displayed for every match found. If no match is found, AT PROBE reports:

*Search data not found in block*

The Write operation saves a block of memory to a disk file as a string of bytes. The prompts for this operation are start and end address of the block to be saved and the name of the file in which it will be saved. There is also a default disk and directory prompt which you can override by typing <TAB> to these fields and typing new path information.

## EXAMPLES OF MEMORY BLOCK OPERATION COMMANDS

Save a block of memory to disk from ARRAY-START to ARRAY-END in a file called SAVEBLOCK. Use drive D with directory \TEMPSAVE.

**M B W ARRAY-START <TAB> ARRAY-END <TAB> D <TAB> \TEMPSAVE <TAB> SAVEBLOCK <enter>**

Compare the block of memory with the defaults from the previous example with a destination block at 5000:0

**C <enter> <enter> 5000:0< enter>**

Find the string "Now is the time" in the block of memory from 3000:0 to 3000:FF. Report if bytes are equal.

**M B S 3000:0 <enter> +100 <enter> "Now is the time" <enter> E**

Move the block of memory from ARRAY-START to ARRAY-START +3FF into NEWARRAY.

**M B M ARRAY-START <enter> +3FF <enter> NEWARRAY <enter>**

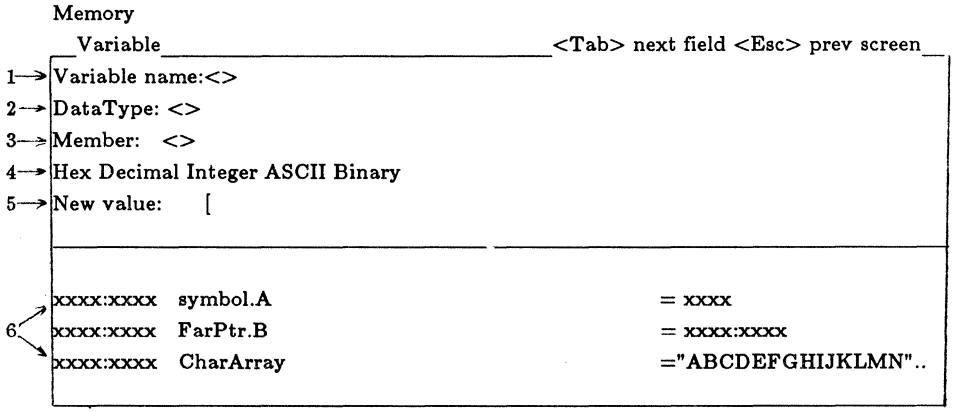
Fill the block of memory starting at ARRAY-START and ending at ARRAY-END with the string "Now is the time".

**M B F ARRAY-START <enter> ARRAY-END <enter> "Now is the time" <enter>**

## MEMORY VARIABLE

Variables can be displayed or changed with the Memory Variable command. The variables will automatically be displayed in the data type assigned to the variable.

This command applies only to the SOURCE PROBE.




1 The first field prompts you for the name of the variable to be displayed or changed.

Structure field names and array indices are allowed.

Expressions such as "[Ptr]" may be given to display memory pointed at by a pointer variable. In this case, if "Ptr" points to a structure, then fields within a structure can be specified just as if the original structure name were specified, as in "[Ptr].fieldname", etc.

2 The second field displays the data type of the variable entered in VARIABLE NAME. Optionally, the data type of the variable being examined can be altered using the DATATYPE DIALOG BOX. Note that if a complex data structure which was defined without a tag is used as a type cast, VARIABLE NAME will simply be designated as "struct".

When a variable name is first input, the type in the DATATYPE field is set to the default type for that variable and the cursor jumps to the NEW VALUE field. To alter the default type cast, press <TAB> twice. The DATATYPE prompt will change to <Same|Change|None>. If **C** is entered for Change, you will be prompted to input the name of a variable with a type the same type as the type you wish to cast the current variable to (3). Complex expressions can be used to specify type casts. If N is entered to the DATATYPE prompt, the variable in question will be uncast, that is represented as a 16 bit word. Entering S to the DATATYPE prompt will cause the cursor to move to the next field.

Examples:

If **P** is a pointer to an integer, specifying **[P]** will cause the variable to be cast to the type of dereferenced **P**, that being to type integer. If **C** is a complex structure with **S** being the address of the first element of an array of chars (a string) contained as a member within **C**, entering **C.S** will cast the variable being referenced to the type of an array (or string) of chars.

3 The third field optionally displays the name of the current member of the data structure being examined. If a simple data type is being examined, this line will remain blank.

4 After the variable name is entered, its value is displayed according to its type. Character strings are displayed in quotes, far pointers are displayed in segment:offset format, enums in ASCII and other types are shown in hex, decimal, integer, ASCII and binary formats.

5 A new value for the variable can be entered here. The value entered should be according to the variable type. For instance, a character array should be entered in quotes.

(3) It is necessary to extract type casting information from existing variables, due to the way Microsoft (R) symbolic information is stored. This function does not work with compilers which do not provide Microsoft symbolic information.

6 The DISPLAY WINDOW shows the address, name and value of the variable. This display is updated when a new value is entered for a variable. Structures and arrays are displayed in a multi-line format. The highlighted cursor can be used to select an array element to be changed. Also the PgUp/PgDn and Ctrl-PgUp/Ctrl-PgDn keys can be used to move the cursor when there is more than one display line.

The Memory Variable command supports all C data types.

Character arrays are printed in a one-line format. The first 14 characters are displayed on the right side of the screen with continuation indicated.

The Variable subcommand displays all values except character arrays and enums in hex. Values are displayed as 1, 2, 4 or 8 bytes, depending on the type of the variable. Enum variable value are displayed as an ASCII character string if the numeric value has an associated string. Otherwise, the value is displayed as a 2-byte number in hex, unsigned decimal, decimal ASCII and binary.

Enum values may be assigned as either ASCII strings (enum element names) or as hex values.

Note: If you set a breakpoint at .procedurename, execute to that point, and then try to display a local variable, the value displayed will be incorrect because the BP register does not get initialized until several instructions into the procedure. Source stepping one statement will set up the BP register.

# OPTIONS

The Options command lets you set a number of parameters which affect how AT PROBE works while in the code screen. The options affect the interactive aspects of displaying code and single stepping program execution.

All the options that are selected from the Options command can be saved in an initialization file (See the File Initializations command in this chapter). The options will be set automatically when the initialization file is loaded.

This command is invoked from the MENU BAR by typing:

**O** (for Options)

The Options command has the following subcommands:

| Subcommands | Operation |
|---|---|
| Screen | Selects screen switch and remote consoles |
| View operands | Display operand contents during step |
| Mix source | Show source code with assy during step |
| sYmbols displayed | Show symbols with code during step |
| Case sensitivity | Include case in symbol interpretation |
| sTep count | Number of steps to take during step |
| Interrupts | Controls interrupt system while in AT PROBE |
| Read after write | Verify changes made by the AT PROBE memory command |
| Function call | Selects linkage style |

## OPTIONS SCREEN

When this subcommand is selected, the following DIALOG BOX appears:

```
    Options
     Screen_____________________________<Tab> next field <Esc> main menu_
1-> Switch screens:<Flip> {Flip|Switch|No-switch}
2-> Activate console:[Local] {Local|Other|Remote|1-Com1|2-Com2}
3-> Baud rate: <38400>{2400|9600|19200|38400}
```

1 AT PROBE can isolate its screen from the application by creating two virtual screens. If the AT PROBE console is selected to be Local, this field selects the type of screen multiplexing between the AT PROBE screen and the applications screen. If screen switching or flipping is enabled, the application screen is displayed whenever your program is executing from a Go command or a single step. In this manner, output from the program is not confused with output from AT PROBE. When AT PROBE has gained control from the application program after a breakpoint or after each single step, the application screen is saved and the AT PROBE screen is displayed. There are three choices:

Screen Flip - the user screen is put on video pages 0, 1, or 2, (depending on the page selected using the BIOS) and the AT PROBE screen uses video page 3. This is slightly faster than screen switching, and requires that a video card supporting multiple display pages be used. In order to use screen flipping, the application being debugged must remain in the same video mode as used by AT PROBE (mode 02 on monochrome, 03 on color) and must be using the 25 line display. If screen flipping is not available in the current display mode, the **Flip** option will not be displayed in the dialog box.

Screen Switch - stores the contents of the application screen in AT PROBE memory and puts up the AT PROBE screen. Screen switching is the default.

No switch - Both the AT PROBE and the application share the same screen in this mode.

Screen switching may only be used from the local console. They have no meaning from the remote console or other console since the application screen is always displayed on the system console.

You can switch between a display of the application screen and the AT PROBE screen with the F4 function key when screen switching is selected.

2 This field lets you select the active console interface for AT PROBE. AT PROBE starts with the console set to the standard keyboard and monitor. You can change this by selecting one of the options from this field.

L for Local restores the communication to the AT PROBE from the standard keyboard and monitor. This is the default when AT PROBE first signs on. If you have switched the console to one of the other choices shown below, you can switch back to the local console with this option. Local console is the default.

R for Remote console uses the serial RS232 port on the AT PROBE board for commands. This frees up the keyboard and the monitor for applications programs. It also eliminates the use of all DOS and ROM BIOS calls (except for disk use) since the software driver for this port is contained in the AT PROBE software. This is very useful for debugging routines which steal DOS interrupts.

Typing 1 or 2 specifies using the COM1 or COM2 port. The function is the same as the Remote console case.

The connection to the COM1 or COM2 RS232 ports is described in Appendix E. The external CRT configuration parameters are also described in Appendix E.

O for Other console allows the screen display for AT PROBE commands to appear on an alternate video monitor driven from an alternate video display controller board plugged into the system. A system which uses a monochrome monitor may have a graphics monitor installed also. In this case, the monochrome monitor would be used for your program output and the graphics monitor would be used for AT PROBE output. If the system normally uses the graphics monitor and also has a monochrome monitor, then your program output is put on the graphics monitor and AT PROBE output is put on the monochrome monitor.

*Press the <Enter> key on the remote keyboard*

This prompt allows the AT PROBE to check that the remote console is connected and operating at the correct baud rate. If the character received by the AT PROBE is not <enter>, the AT PROBE will remain at the local keyboard. Typing <enter> at the local keyboard will also cause AT PROBE to remain there.

3 This field lets you set the baud rate for the Remote, Com1 or Com2 serial ports. You only need to enter the first character to select the baud rate. 9600 baud is the default. The baud rate may be specified in the PROBE.CNF file.

## OPTIONS VIEW OPERANDS

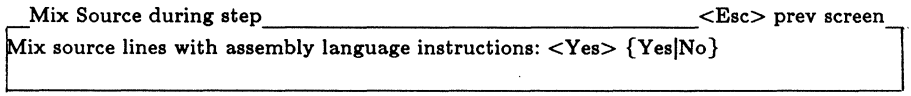
During single step operations with the F8 or F10 function key, the code screen can display the contents of operands if the screen is in assembly language mode. If this is not desirable, enter No to Options View operands.

```
Options
 View operands during step                          <Esc> prev screen
Read and display operands on code screen: <Yes> {Yes|No}
```

The default is Yes.

## OPTIONS MIX SOURCE DURING STEP

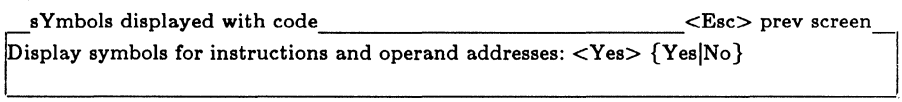
If the code screen is in assembly language mode, you can also include source code mixed in with the display by selecting the option in the screen below.




The default is Yes.

## OPTIONS SYMBOLS DISPLAYED WITH CODE

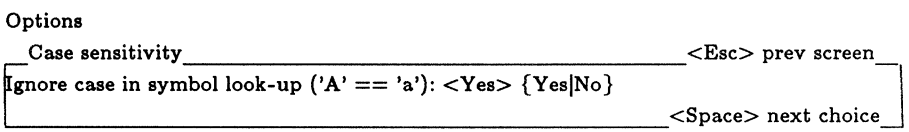
When AT PROBE displays assembly language in the CODE SCREEN, you can also include the display of linenumbers and symbols for address and operand fields. The screen below lets you make the choice.




The default is Yes.

## OPTIONS CASE SENSITIVITY

When symbols are referenced, case can be ignored or used in correctly identifying the symbol. If your code has symbols that are only differentiated by case, you should select No for this option.




The default is Yes (ignore case).

## OPTIONS INTERRUPTS

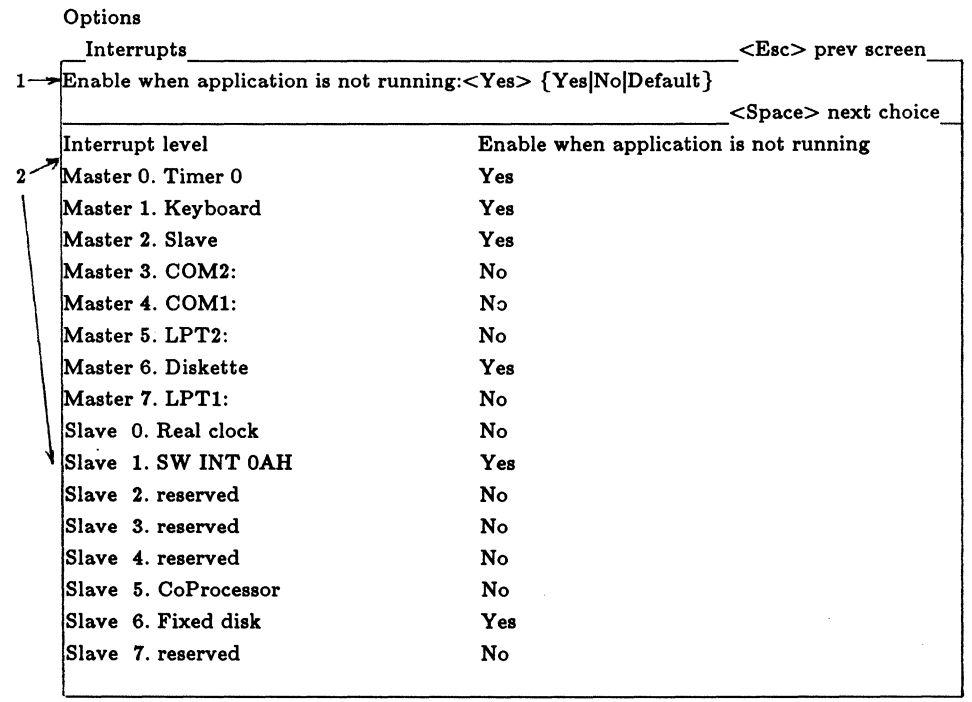
You can enable or disable interrupt masks in the two 8259 interrupt controllers while in AT PROBE software. The background interrupts such as the real time clock, disk controller operations, and keyboard servicing will continue to request servicing while the AT PROBE software is executing. However, if the routines which service these requests are not working, they could prevent AT PROBE from operating. This could happen if they never return to the AT PROBE software. This could also happen if AT PROBE took control via a breakpoint you set within a non-reentrant interrupt routine, and then a new interrupt tried to take control away from AT PROBE. To prevent this from occurring, use the Options Interrupt command to mask off the selected interrupts after entry into AT PROBE software. You would normally execute this command after you had just loaded your program and before executing it for the first time. The Options Interrupt command sets the state of the 8259 interrupt mask register while in the AT PROBE software. The mask change occurs as soon as the Options Interrupt command is executed. This does not affect your program since the mask is always restored to the value it had when AT PROBE was entered from a breakpoint. Also, references to I/O port 21h and A1h will reflect the mask used in your program: they will not affect the mask used in AT PROBE. There are two cases when this mask is then changed again.

1. ENTERING THE APPLICATION CODE. When the applications program is started with the Go command, this mask register is set to the state it was in when the last breakpoint occurred. For the first Go command the mask register is set to the state it was in when AT PROBE was started.

2. ENTERING AT PROBE SOFTWARE. When AT PROBE software is entered from a breakpoint in the users program, it sets the mask to the state which was specified by the Options Interrupt command. If no Options Interrupt command has ever been issued, it sets the mask to the state it was in when the AT PROBE software was loaded.

The keyboard interrupt can only be masked from a remote CRT. If the disk controller interrupts are masked, then commands which use the disk such as Load, Save and Edit subcommands cannot be executed.

When the Options Interrupt command is selected, the following DIALOG BOX appears:




1 This prompt allows you to enable or disable specified interrupts. Select Yes to enable, No to disable or D to take the default for the interrupt at the highlight. The default is the state saved when AT PROBE was loaded.

2 The DISPLAY WINDOW above contains a sample display of the interrupt states. It will generally be the default. However, a program you have run may have changed some of the values.

## OPTIONS STEP COUNT

You can select the number of single steps to take each time the F8 or F10 function keys are typed while in the Code screen.

```
Options
  sTep count                                          <Esc> prev screen
Steps to take for each <F8> or <F10> key:<0001>[]
```

The default is 1 step. You can select up to FFFF steps to be taken for each press of the F8 and F10 keys.

## OPTIONS READ AFTER WRITE VERIFICATION

When you change the contents of memory with any of the MEMORY subcommands, AT PROBE automatically reads back the change to insure that it occurred. You can disable this verification with the screen below.

```
Options
  Read after write verification                       <Esc> prev screen
Perform memory read-after-write verification: <Yes> {Yes|No}
                                                      <Space> next choice
```

## OPTIONS FUNCTION CALL LINKAGE STYLE

In order for the Calls command to correctly display the calling stack and for local variables to be evaluated in expressions, the style of function call linkage must be specified.

When this subcommand is selected, the following DIALOG BOX appears:

```
Options
 Function call linkage style                          <Esc> prev screen
Call linkage style:[Microsoft (r)] {Microsoft (r)|Lattice (r)}
```

Type M for Microsoft or L for Lattice. The default linkage style is the one used by Microsoft. It assumes the following order of instructions:

```
PUSH BP
MOV BP, SP
SUB SP, XX
```

The linkage for Lattice is as follows:

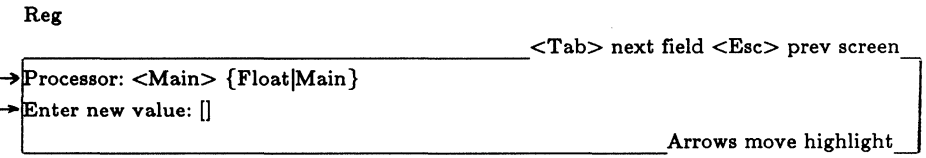
```
PUSH BP
SUB SP, XX
MOV BP, SP
```

# REGISTER COMMAND

The registers and flags in the 80286 can be displayed and changed with the register command. You invoke this command from the MENU BAR by typing:

**R** (for Register)

The following screen appears:




1 The DIALOG BOX prompts lets you change the value of one of the registers or flags. The register to be changed is indicated by the highlighted field. You can position the highlight on the desired register and enter an expression. Optionally, to avoid having to move the highlight field to a register, you can enter the following value:

**registername = value**

The value is deposited directly into registername. Registername is a standard register definition.

2 This field shows you which processor the registers in the DISPLAY WINDOW is displaying. The Main processor displays the 80286 registers. Float displays the 80287 numeric coprocessor registers. The display formats are shown next.

### 80286 registers and flags

```
AX=0007  CS=12FD  SS=14AD  DS=14AD  ES=14AD  GDTR=000000,FFFF
BX=1314  IP=0136  SP=1310  SI=0082  DI=132D  IDTR=000000,FFFF TR=0000
CX=0019          BP=1316                MSW=FFF0=TS0 EM0 MP0 PE0
DX=000E          FL=0206=O0 D0 I1 T0 S0 Z0 A0 P1 C0
```

### 80287 registers and flags

```
ST(0)= + 1.6148186360801886 E + 4335   Tag=Empty
ST(1)= + 1.6148186360801886 E + 4335   Tag=Empty
ST(2)= + 1.6148186360801886 E + 4335   Tag=Empty
ST(3)= + 1.6148186360801886 E + 4335   Tag=Empty
ST(4)= + 1.6148186360801886 E + 4335   Tag=Empty
ST(5)= + 1.6148186360801886 E + 4335   Tag=Empty
ST(6)= + 1.6148186360801886 E + 4335   Tag=Empty
ST(7)= + 1.6148186360801886 E + 4335   Tag=Empty
FPCW=037F=   PM1 UM1 OM1 ZM1 DM1 IM1
                          I=Proj  R=Near  P=64-bit
FPSW=4100=ES0 PE0 UE0 OE0 ZE0 DE0 IE0 ST0 C31 C20 C10 C01 B0
```

These register names are recognized by AT PROBE and can be used in expressions exactly as they are shown above. If you want to specify a hex number to AT PROBE which coincides with a register name, you must precede the hex number with 0.

## EXAMPLES OF USING THE REGISTER COMMAND

Change the value of register DS to AAAA.

**R <RtArrow> <RtArrow> <RtArrow> AAAA <enter>**

Change the value of the ES to EFF

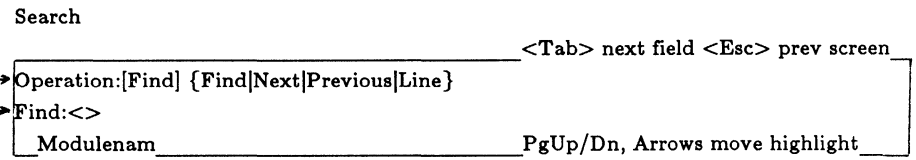
**R ES=EFF <enter>**

# SEARCH

The Search command operates in the CODE SCREEN. It lets you find all occurrences of a specified string in the source code for the module that is currently executing. It is invoked from the MENU BAR by typing:

S (for Search)

The following DIALOG BOX appears:




1 The first prompt is for the action to be taken in the Search command.

The default is F for Find. Choose this option to begin searching for an occurrence of the string specified in the next prompt.

When and if the specified string is found, you can type N for the Next occurrence of the string in the current module. When no further occurrences of the string are found in the module, the following message will be displayed:

*Did not find string from here to end of file*

Type P for Previous to return the highlight to a previous occurrence of the string. If there are no other occurrences of the string between the current position in the file and the start of the file, the following message will be displayed:

*Did not find string from here to start of file*

Type L to display a source code line number.

2 For the Find subcommand, the second prompt is:

*Find:[*

For this prompt, type the string to be searched. The search looks for any line which has an occurrence of the string. The search is not case sensitive. When a match is found, the line containing the match is highlighted. For the Line subcommand, the second prompt is:

*Line:[*

Type the number (in decimal) of the source code line you want to display. The name of the module that is currently executing is displayed on the bottom left of the DIALOG BOX.

The highlight field can be moved with the PgDn/PgUp and cursor keys. To move to the top or bottom of the file, use the Ctrl PgUp or PgDn keys.

While the file is being displayed in the DISPLAY WINDOW, F5 and F7 set their breakpoints then transfer control to the program. Upon breakpoint or Stop button, control returns to the CODE SCREEN. F9 sets or clears a sticky breakpoint on executing a line number or label. The breakpoint number is shown on the screen at the right edge.

## EXAMPLES OF USING THE SEARCH COMMAND

Find any occurrence of the string "BuildArray" in the module that is currently executing.

**S <TAB> BUILDARRAY <enter>**

Type N or P for the next or previous occurrences of this string.

Search for line 22 of the source code in the module that is currently executing.

**S L 22 <enter>**

From the previous example, the highlight is now on line 22. Set a temporary breakpoint and execute to that line.

**<F7>**

# SYMBOL

The Symbol command lets you view the AT PROBE symbol table along with several attributes for symbols and modules.

The Symbol command has the following subcommands:

| Subcommand | Operation |
|---|---|
| sYmbol display change | Display, change or delete symbols |
| Default modulename | Sets default modulename for symbols |
| Load module selections | Load symbols from only from selected modules |
| Step source screen modules | Source step only in selected modules |
| Module to file assignment | Display/change file names assigned to modulenames |

## SYMBOL DISPLAY CHANGE

The sYmbol display change command lets you display, change, add, and delete symbols from the AT PROBE symbol table. The following MENU BOX is displayed:

```
    sYmbol
    __sYmbol display change_____________________<Tab> next field <Esc> prev screen_
2 → Display type:<Externals> {Externals|Internals in module|Lines}

3 → Operation:<Find> {Find|Add|Remove|Move|Clear Table|Variable}
1 → Symbol name:<      >[
                                                        Up/Dn, Arrows move highlight
    Address                 Name                    Type
4 → xxxx:xxxx               xxxxxxxxxxxxx           xxxxxxxxxx
    xxxx:xxxx               xxxxxxx                 xxxxxxxxxxxxxxxxx
```

1 The first prompt is for the name of the symbol to find, add, remove or move. The symbol at the highlighted line is displayed to the left of the prompt and is the default. To select a symbol, you can type the name of the symbol at the prompt or move the highlight to the line containing the symbol.

2 The next prompt is for the display type.

Type E for Externals. Externals is the default. The display shows all external symbols.

Type I for Internals in module. When you select Internals or Lines, you are prompted for the name of the module.

*Module name:<  >*

The names of the modules are displayed in the DISPLAY WINDOW. To select a module, you can type the name of the module at the prompt or move the highlight in the DISPLAY WINDOW to the module name.

Type L for Lines. When you select Lines, you are prompted for the module name (see Internals option above).

3 The Operation prompt has the following choices:

Type F for Find. Find is the default. Find searches for a match on symbols or line numbers. The highlight is positioned on the line where the match is found. If there is no match, the following message is displayed:

*Could not find symbol with name*
*starting with input string*

Typing A for Add allows you to add a symbol. For both the Add and Move subcommands, the following prompt will be displayed:

*Symbol address:<  >[*

Enter the name (1) and the address of the symbol to be added.

If you type R for Remove, you can select symbols or line numbers to be removed from the list.

Type M to Move symbols to a new address. You will be prompted for the address where the symbol will be moved (See Add option above).

Type C for Clear Table if you wish to remove all symbols from the AT PROBE's memory. If you select this option, the following message will be displayed to verify whether or not you want to clear all symbols in the table:

*Remove all externals, internals, and lines?*
*(Y) for yes; <other> for no*

V for Variable can only be selected in SOURCE PROBE. This command allows you to display variables according to their type and to change their value. Refer to the Memory Variable command for a general explanation of this subcommand. Variables can be selected to be displayed or changed by moving the highlighted cursor to the symbol or by typing the symbol name at the prompt.

4 The DISPLAY WINDOW shows symbols, line numbers or module choices. In the example DISPLAY WINDOW, symbols are displayed with their address, name and type.

In addition to the cursor keys, the PgUp and PgDn keys can be used in the DISPLAY WINDOW to move the highlight to symbols, modules or line numbers.

F5 and F7 operate if highlight is on a line number or procedurename in the symbol display. Control then transfers to the program. Upon breakpoint or Stop button, control returns to the CODE SCREEN. F9 sets or clears a sticky breakpoint on executing a line number or label. The breakpoint number is shown on the screen with the symbol.

## EXAMPLES OF USING SYMBOL DISPLAY CHANGE COMMAND

Display the symbol Testvar.

**Y Y TESTVAR <enter>**

Display all symbols for the module \FTOCIO

**Y Y <TAB> I FTOCIO <enter>**

The module could also have been selected with the cursor keys.

Starting from the where the previous example leaves you on the screen, display the symbol Getval and change its address to Getval+1.

**<TAB> GETVAL <enter> <TAB> <TAB> <TAB> M <TAB> GETVAL+1 <enter>**

Delete all symbols in all modules.

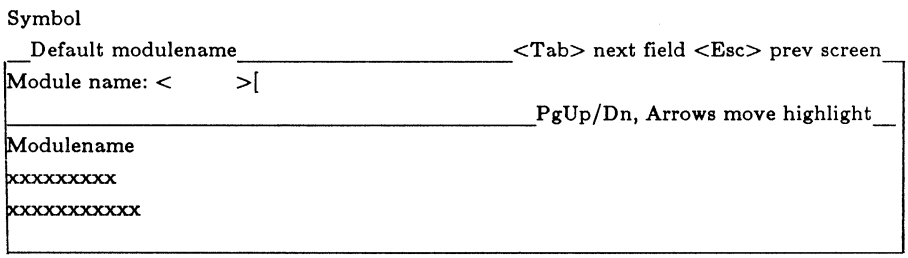
**Y Y <TAB> <TAB> C Y**

Add the symbol Zcount at address 3000:0

**Y Y <TAB> <TAB> A Zcount <TAB> 3000:0 <enter>**

## SYMBOL DEFAULT MODULENAME

It is tedious to type in the complete modulename for every symbol if you are always working within the same module. AT PROBE lets you define a default modulename which will automatically be included in front of symbols. This is done with the Default-modulename subcommand which displays the following screen:




All currently loaded modules are displayed on the screen and one of the modulenames on the screen is highlighted. The cursor keys move the highlighted field. You can simply type <enter> to select the highlighted modulename or type in a new modulename.

If no modulename is specified when you specify symbol or line number, then the default module name is used.

## SYMBOL LOAD MODULE SELECTIONS

The loading of symbols into the symbol table can be limited to specified modulenames. This is done with the Load-Module-selections subcommand. When invoked the following screen appears:

```
    Symbol
    Load module selections                    <Tab> next field <Esc> prev screen
2 → Module name <       >[
1 → Load symbols for module: [Yes] {Yes|No}
                                              PgUp/Dn, Arrows move highlight
    Module name          Load symbols for module?
3 → xxxxxxxx             Yes
    xxxxxxx              No
```

1 The first prompt lets you choose whether or not to load symbols for the selected module.

2 The module name can be entered here or selected in the DISPLAY WINDOW with the cursor keys. The module displayed to the left of the prompt and at the highlight is the default.

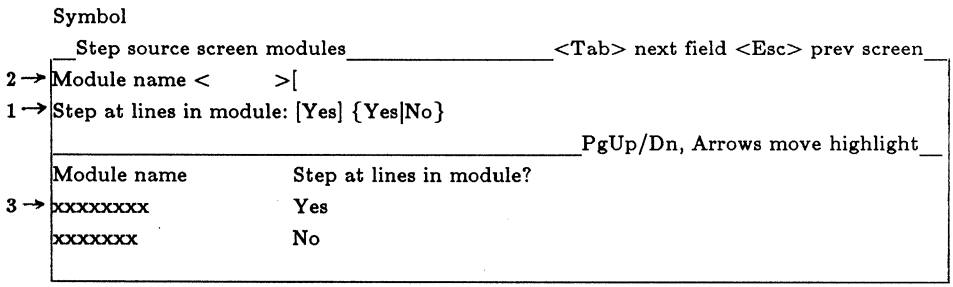
3 The DISPLAY WINDOW shows the currently loaded modules and the choice for loading symbols. The module can be selected by moving the highlight with the cursor keys. To select modules that are not currently displayed, use the PgUp/Dn keys.

Symbols in these modules may have previously been loaded from the AT PROBE initialization file. If the File Program Load command is invoked before this command is used, then it assumes that all modules and all symbols will be loaded into the AT PROBE symbol table and all modulenames are loaded into this selection list table. If some modules have been specified, symbols will be loaded or not depending on the selection here. Any module not listed will have symbols loaded. The init file can be used to perform these selections.

## SYMBOL STEP SOURCE SCREEN MODULES

AT PROBE can single step a high level language program by statements with the Step Source command. It may be desirable to limit the single stepping of source code to only specified modules. When the program goes outside of the specified modules, it runs real time until it gets back into the selected modules. This can be done with the Step-source-screen-modules subcommand. When invoked the following screen appears:

This command applies only to the SOURCE PROBE.



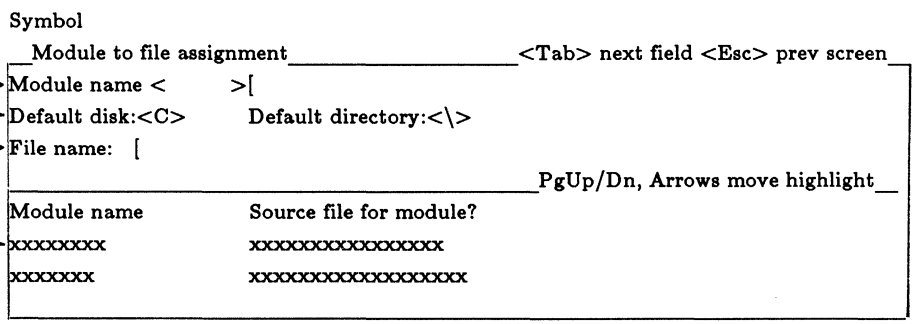

1. The first prompt lets you choose whether or not to source step in the selected module.
2. The module name can be entered here or selected in the DISPLAY WINDOW with the cursor keys. The module displayed to the left of the prompt and at the highlight is the default.
3. The DISPLAY WINDOW shows the currently loaded modules and the choice for source step. The module can be selected by moving the highlight with the cursor keys. To select modules that are not currently displayed, use the PgUp/Dn keys.

The current modules are shown in the DISPLAY WINDOW. These may have been previously loaded from the AT PROBE initialization file. If the File Program Load command is invoked before this command is used, then it is assumed that all modules will be single stepped with the Step Source command. The modules to be stepped could have been specified by loading an initializations file.

## SYMBOL MODULE TO FILENAME ASSIGNMENTS

For source level single stepping or for including source code in Trace displays, source files must be assigned to modulenames. To do this select the Module assignments subcommand and the following screen is displayed:

This command applies only to the SOURCE PROBE.




1 Type the name of the source file for the selected module or accept the default. If the correct source file name is displayed, but without the correct pathname, you can accept the default disk and directory as the source file prefix by simply pressing <enter>.

2 The module name can be entered here or selected in the DISPLAY WINDOW with the cursor keys. The module displayed to the left of the prompt and at the highlight is the default.

3 This is the default drive and directory. Type <Tab> to get to the DIALOG BOXES which let you change these defaults.

4 The DISPLAY WINDOW lists the currently loaded modules and the names of the corresponding source code files. The module can be selected by moving the highlight with the cursor keys. To select modules that are not currently displayed, use the PgUp/Dn keys.

The current modules are shown in the DISPLAY WINDOW. These may have been previously loaded with the File Initializations Load command. If the File Program Load command is invoked before this command is used, and the object module format includes the assignments, this table will receive default initializations.

You remain in this command to change other modules until you type <Esc>.

Default assignments are made from the load module if the Microsoft C compiler version 4.0 or greater is used. If no assignments are available from the load module, use the default drive and modulename on this screen as the pathname and fill in the table with this and the filename. The user can edit these if they are not correct.

Default source file pathnames and file extensions can also be picked up from the File Program Load subcommand which prompts for these defaults.

# TRACE

While your program is executing, AT PROBE is continually saving a history of program execution into high speed AT PROBE memory. This AT PROBE command displays the real time program execution before (and optionally after) the breakpoint was detected. The trace command is invoked from the MENU BAR by typing:trace

T (for Trace)

The subcommands for the Trace command are:

| Subcommand | Operation |
|---|---|
| Instruction | Display trace data with prefetch filtered |
| Unprocessed ins | Display trace with prefetch not filtered |
| Save-to-disk | Save trace data to disk |
| Raw data | Display trace data in hex format |
| DMA cycles | Toggle DMA cycles on/off in trace display |

## TRACE INSTRUCTIONS

The Trace Instructions command provides the most useful form of the trace display. AT PROBE analyzes the trace data which was collected in real time and processes it to produce an easy to understand trace display. The processing does the following:

1. The 80286 pipeline has been modeled in AT PROBE software. AT PROBE analyzes the trace data and tosses out prefetched but unexecuted instructions so you don't have to guess which instructions executed and which did not.
2. AT PROBE analyzes the trace data and displays the memory reference cycles directly under the instructions which executed them. If AT PROBE did not do this, you would have to do this yourself mentally. This is because the 80286 pipeline fetches opcodes many bus cycles earlier than the memory reference cycles which go with those instructions.

The trace shows assembled instructions and operands, data transferred during execution cycles, stack operations, and interrupt cycles. Program symbols are included in the trace data to make the identification of program operation easy to understand. Trace data is viewed after the detection of a breakpoint or execution has been terminated with the Stop Button. A screen similar to the following appears:

```
    Trace
    Instructions                          <TAB> next field <ESC> prev screen
6 → Search addr:          <Any>[xxxx:xxxxxxxx]          to<>[xxxx:xxxxxxxx]
    Verb:       <Any>{Read|Write|Input|Output|Any}
    Data size:  <Any> {Byte|Word|Dword|Any}          Data:<>
    Begin search of trace:[No]      {Yes|No}

     00DA20  MOV     BP,SP
     00DA22  MOV     AX,0004
     00DA25  CALL    $+00C8
     01094C  WRITE   -   00C8  .__end+07CC
5 ------------> .____chkstk:
1 →  00DAED POP     CX
2 →  01094C  READ   -   00C8  .__end+07CC
     00DAEE  MOV     BX,SP
     00DAF0  SUB     BX,AX
     00DAF2  JB      $+000C
     00DAF4  CMP     BX,WORD PTR [0082]
     00F8E2   READ   -   0A20  .STKHQQ
     00DAF8  JB      $+0006
     00DAFA MOV      SP,BX
     00DAFC  JMP     CX
3 → B 00DA28  PUSH    SI
     010948  WRITE   -   005D  .__end+07C8
```

1 This line shows the instruction address, instructions and operands.

2 Any memory reference cycles used by the instruction are shown on this line with their address, type of cycle, and data on the bus during the cycle. If the address of the memory reference cycle matches or is near a program symbol, the symbolname is shown to the right.

3 A "B" in this column indicates that this is the cycle which caused the breakpoint.

4 When you use the Trace search commands (described later) the line of "Found" code is marked with a leading "S".

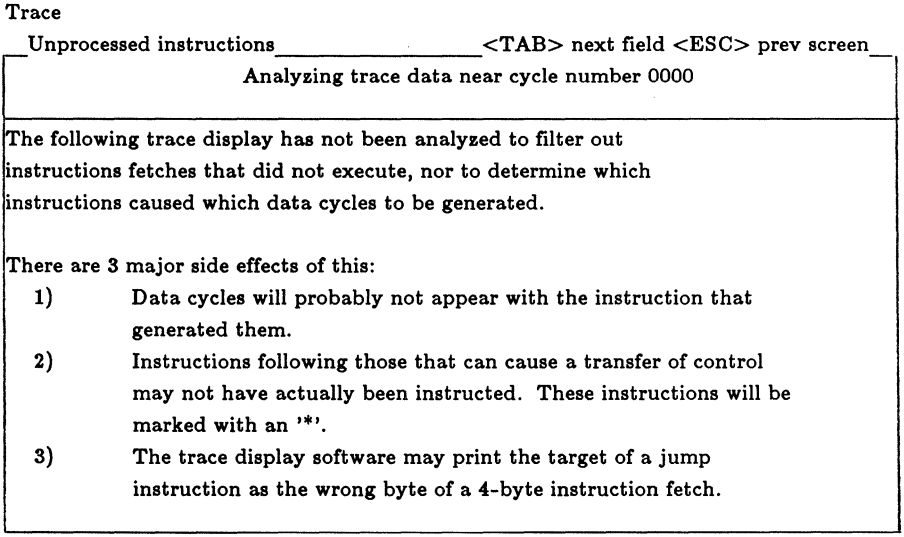
5 Procedure names or high level language line numbers which match the address field of an instruction are shown before the instruction. If you are using the SOURCE rather than the AT PROBE version of the software, then the actual program line precedes the assembly language.

6 You can scroll and search for data in the trace display. This is described later in this command.

If trace regions were active during the collection of the trace data, the trace regions are separated in the DISPLAY WINDOW by a row of * characters.

## TRACE UNPROCESSED INSTRUCTIONS

The Trace Unprocessed instructions command shows you the trace display in a form similar to the Trace Instructions. The difference is that AT PROBE does not filter out unexecuted prefetched instructions or place memory reference cycles directly under the instructions which generated them. It simply displays the opcodes and memory reference cycles on the bus in the order in which they occurred. When this command is invoked the following message appears on the screen to remind you of these facts.




Type any key to start the trace display and you will get a screen similar to Trace Instructions. AT PROBE will sometimes provide you with the above message and display the trace as Unprocessed instructions even when you use the Trace Instructions command. This might happen if AT PROBE could not accurately analyze the trace data to filter out prefetch and tie memory reference cycles to instructions.

### Waiting for the Trace Command to Work

The trace command does an exhaustive search of the collected trace data to determine the correct trace. In some cases, the process may take a long time due to:

1. A non-existent instruction occurring in the data.
2. A long string move operation, which has cycled the instruction fetch portion of the instruction out of the trace data.

If AT PROBE cannot display accurate instruction execution, the following is printed. You may use the Trace Raw command to try to see what happened.

The following trace display has not been analyzed to filter out instructions fetches that did not execute, nor to determine which instructions caused which data cycles to be generated.

The following trace display has not been analyzed to filter out instructions fetches that did not execute, nor to determine which instructions caused which data cycles to be generated.

There are 3 major side effects of this:

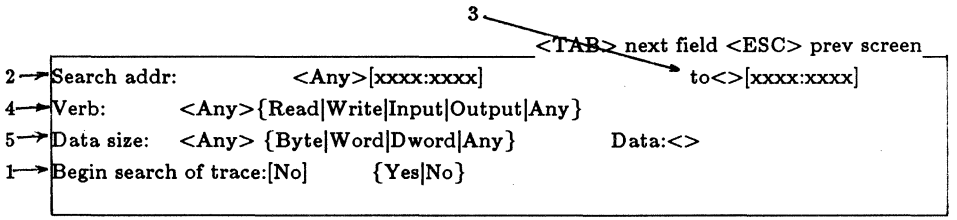
1) Data cycles will probably not appear with the instruction that generated them.
2) Instructions following those that can cause a transfer of control may not have actually been instructed. These instructions will be marked with an '*'.
3) The trace display software may print the target of a jump instruction as the wrong byte of a 4-byte instruction fetch.

## SEARCHING TRACE DATA

While you are in any of the Trace commands which display data in the DISPLAY WINDOW, the following keys let you scroll through the trace display:

PgUp/PgDn and cursor keys move you through the trace data.
Ctrl PgUp moves you to the start of the trace data.
Ctrl PgDn moves you to the end of the trace data.

AT PROBE also has a built in editor which lets you search trace data fields for specific events. The DIALOG BOX prompts look like this:




1 If you type Y, AT PROBE starts searching from the current location in the DISPLAY WINDOW until the end of the Trace data for matches between the trace data and the other fields in the DIALOG BOX. If you type N, then AT PROBE lets you set other search conditions in the DIALOG BOX.

2 This field lets you enter an address expression for the search address.

3 If you want to search for any address within a range, then set the TO address. You may enter any type of address expression or an address of the form:

**+ number**

In this case the TO address becomes search address+number. If this field has previously been set to other values and you want to clear the field, type <space><enter>.

4 Set the verb field for the search.

5 Set the Data size and Data value fields for the search.

Typing <enter> in response to any of these fields takes the default in the < >. If a search is completed successfully, the trace data is positioned in the DISPLAY WINDOW with the first line which matches the search marked with an 'S'. Any additional lines in the trace data which match the search condition are also marked with an 'S'. You can continue the search from here by simply typing Y again in response to the prompt. If the search is not successful, the following message is displayed:

*Specified values not found from current locations to the end of trace.*

## TRACE SAVE TO DISK

This command lets you save the trace to a disk file. You could also use the File Log command to open a log file to disk then display the trace data, but this is much faster. The following screen appears.

```
Trace
  Save to disk_____________________________<TAB> next field <ESC> prev screen_
1→Data Format: [ASCII] {Raw|ASCII}

2→Default disk: <c>      Default directory: <\>
3→File name:  [
```

1 Type A for ASCII if you want to save the trace in a form that you can read. This option is valuable if you wish to store the trace data and examine it at another time or if you wish to look at the data on paper. The ASCII form of the trace data allows you to use your favorite text editor to examine the data.

Type R for Raw to save a machine readable form of the trace data that can be sent to Atron to help resolve trace problems.

Atron has modeled the 80286 pipeline in order to filter unexecuted instructions and tie memory reference cycles to instructions. If this model has bugs, you can help us find them by saving trace data when the Trace Instructions command fails (i.e. gives you Trace Unprocessed instructions display instead). With this Raw data, we can exactly recreate the state of the AT PROBE when it failed in your target. There are other situations when the AT PROBE appears to be effecting the operation of your system or program execution that we can often identify at Atron with this Trace data. In such situations, do a Raw save and send the resulting file to Atron to the attention of the Technical Support group. When we receive the information, we will report our results, work-arounds, or corrections to you as soon as we analyze the Trace data.

2 You are prompted for the pathname of the file in which the trace data will be saved.

3 You are prompted for the filename to save the data. You can also save it by directing the trace data to a printer. Use the standard name for your printer port.

When the filename has been entered, AT PROBE first analyzes the trace data before saving it. In the DIALOG BOX, the following message will be displayed:

*Analyzing trace data near cycle number xxxx*

After the trace data has been analyzed, the following message will be displayed in the DIALOG BOX:

*Writing trace data*

## TRACE RAW

This command is meant to be used be used by Atron Technical Support to diagnose any trace synchronization problems that might arise. It is not intended for use in the process of debugging application programs.

The Trace Raw command is a hex display of the real time trace data. The display of the data appears under these columns:

```
Trace
  Raw                                    <TAB> next field <ESC> main menu
Search addr:          <Any>[xxxx:xxxx]                  to<>[xxxx:xxxx]
Verb:        <Any>{Read|Write|Input|Output|Any}
Data size:   <Any> {Byte|Word|Dword|Any}          Data:<>
Begin search of trace:[No]       {Yes|No}

Cycle Addr Data Cod#Int MRW IORW RUN BP DMA S10 BHE Mstr# PeAck IOCHK#
                        Raw trace data
```

| | |
|---|---|
| Cycle | number of the cycle (0-7ff) in the trace data. |
| Address | 24 bits of 80286 addresses. |
| Data | 16 bits of 80286 data. For any given cycle the data here may not be valid. The Strobe field indicates which portion of the data bus contains valid data. |

*Mstr# and IOCHK# are active low.*
*See Intel 80286 CPU documentation for an in-depth description.*

| | |
|---|---|
| Cod#Int | 80286 Cod/Int/ line, inverted. |
| MRW | Memory read, Memory write. |
| IORW | I/O read, I/O write. |
| Run | AT PROBE run bit. High while the board is run. |
| BP | Hardware breakpoint, misc breakpoint. |
| Ref | Refresh |
| DMA | Direct memory access. |
| S10 | 80286 S1/ and S0/ lines, inverted. |
| Mstr# | Bus master/. |
| PeAck | Peripheral Acknowledge. |
| IOCHK# | I/O channel check. |

The cycle listed in the bottom line of the DIALOG BOX will reflect the cycle number of the first trace cycle where Run == 0. For hardware breakpoints, this cycle usually will be very close to the cycle where the breakpoint was triggered (usually the next cycle). After a software breakpoint is hit, the Run bit will remain high until the AT PROBE software can manually stop the board, so the pseudo trace counter will be further off. Pressing HOME will cause the display to center around the cycle indicated by the trace counter. If the trace counter is shown as 0000, this probably means that the Run bit went low just as trace memory wrapped around; hitting <Ctrl-PgDn> will move to the end of the display.

# WATCH

You can create your own custom data displays called Watch windows which can be popped up over the DISPLAY WINDOW. These Watch windows of data are defined by the Watch command. The Watch command assigns the Watch window to any Alt-Key (i.e. hold down ALT while you type any other key). The window can be popped up at any time by typing the Alt-Key. It can be popped down by typing the same Alt-Key again. If more than one window is popped up at a time, they "stack" one under the other. If a command is in process under the pop up window, it is temporarily suspended until all the Watch windows are popped down.

If a window is being popped up during the definition of a macro, the Alt-Key which pops the window down is not passed on to the macro. This has the effect of pausing the macro when the window is displayed. You then pop down the window with the appropriate Alt-Key and the macro continues execution. This feature enables you to view Watch windows while building macros and other windows.

The Alt-Key which popped up the Watch window is shown in the window. When multiple windows are popped up, the must be popped down starting with the one on the bottom of the screen.

The Watch command is invoked from the MENU BAR by typing:

W (for Watch)

The subcommands for the Watch command are:

| Subcommand | Operation |
|---|---|
| Define-edit | Define or edit a pop up Watch window |
| Remove | Delete currently defined window |
| Load | Load a file of predefined windows |
| Save | Save current window definitions to disk |

## DEFINING AND EDITING WATCH WINDOWS

When the Define subcommand is selected, the following prompt appears:

*Enter <Alt-Key> that will activate this window:[ ]*

An Alt-Key means hold down Alt and type any other key. This Alt-Key becomes the key which will pop up the window. This Alt-Key may have already been assigned to a window. In this case, the subcommands let you edit the window definition. If you enter the wild card key *, the names of all windows are displayed. Once entered, the Define Edit subcommand displays another the following DIALOG BOX with the Watch window subcommands.

```
Watch
  Define edit                          <Tab> next field <Esc> prev screen
┌──────────────────────────────────────────────────────────────────────────┐
│Operation: [Add field] {Add field|Move field|Change field|Remove field}    │
│                                                    <Space> next choice   │
└──────────────────────────────────────────────────────────────────────────┘
```

## ADDING A FIELD

A field is an area of the window which displays the data. When the Add-Field subcommand is selected, another line is displayed in the DIALOG BOX.

```
Watch
  Define edit                          <Tab> next field <Esc> prev screen
┌──────────────────────────────────────────────────────────────────────────┐
│Operation: [Add field] {Add field|Move field|Change field|Remove field}    │
│Field type:[Label] {Label|Expression|Zero String0|String len|Range mem}   │
│                                                    <Space> next choice   │
└──────────────────────────────────────────────────────────────────────────┘
```

To add a field to the Watch window:

1. Select the Add field subcommand.
2. Select the Field type.
3. For the Expression and Range-of-memory Field types, select the appropriate Data Type.
4. Each Field type has DIALOG BOXES that prompt for expressions, start addresses, etc.

A summary of the Field types and their corresponding Data types and DIALOG BOX prompts are summarized below.:

| FIELD TYPE SUBCOMMAND | DATA TYPE | ADDITIONAL DIALOG BOXES |
|---|---|---|
| Label | String | Label:[ |
| Expression | Byte | Expression:[ |
| | Word | " |
| | Dword | " |
| | Unsigned | " |
| | Integer | " |
| | ASCII | " |
| | Symbol | " |
| Zero string0 | Zero-terminated | String address:[ |
| String len | Length-defined | String address:[<br>String length:[ |
| Range-of-memory | Byte | Range start address:[<br>Range end address:[ |
| | Word | " |
| | Dword | " |
| | fShort | |
| | fLong | |
| | fTemp | |
| | fPacked | |

Each field in a Watch Window has a DATA TYPE. These are shown in the previous table and they are explained below. A field which contains an Expression will first evaluate the expression then put the data into the field in the DATA TYPE you select for the expression.

| DATA TYPE | DESCRIPTION |
|---|---|
| String | Labels can be any string |
| Byte | Byte value in hex |
| Word | Word (two bytes) value in hex |
| Dword | Long (four bytes) value in hex |
| Unsigned | 32 bit unsigned integer |
| Integer | 32 bit signed integer |
| ASCII | Byte value in ASCII |
| Symbol | Symbol name which matches expression. |
| Zero-terminated | ASCII string terminated with 0 |
| Length defined | ASCII string defined by length |
| fShort | Short 32 bit real |
| fLong | Real 64 bits |
| fTemp | Temp Real 80 bits |
| fPacked | 80 bit Packed Decimal |

At the end of choosing the DATA TYPE for the field and filling in the answers to the AT PROBE prompts, the final DIALOG BOX for the following instruction is displayed:

*Place highlight in location for field and press <enter>*

The size of the field in the window is automatically adjusted to fit the data put into the field.

Use the cursor keys to move the solid block cursor shown in the DISPLAY WINDOW to the position within the window where the field is to be displayed. If other fields are already defined for this window, they are shown so that the new field will not conflict with the current fields. You should insure that a field is not positioned such that it writes over another currently defined field. If you do, the data from one field will overwrite the data of other fields.

## MOVE A FIELD

If the Move Field subcommand is selected, the following prompt appears:

*Place highlight in field to be moved and press <Enter>*

The solid block cursor can be moved with the cursor motion keys to the field you want to move within this window. Typing <enter> then gives you this prompt:

*Place highlight in location for field and press <Enter>*

Move the solid block cursor where you want the field to move then type <enter>.

## CHANGE A FIELD

If the Change-a-Field subcommand is selected, the following prompt appears:

*Place highlight in field to be changed and press <Enter>*

The solid block cursor can be moved with the cursor motion keys to the field you want to change in this window. The solid block moves only to the starting position of each field with the cursor keys. Typing <enter> brings up the prompts from the Add-a-field subcommand so that you can edit and make changes to the field. Only the Data Type can be changed for the field.

## REMOVING A FIELD

If the Remove-a-field subcommand to the previous screen is selected, the following prompt appears:

*Place highlight in field to be removed and press <Enter>*

The solid block cursor can be moved with the cursor motion keys to the start of the field you want to delete for this window. Deleting this field leaves the remaining fields in the window in place.

## WATCH REMOVE WINDOW

A window can be deleted by selecting the Remove-a-window subcommand. The following prompt appears:

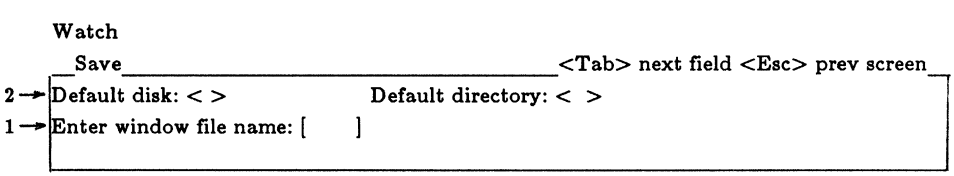
*Enter <Alt-Key> that will activate this window: [ ]*

Entering A for this prompt will delete all currently active windows. The following message will be displayed if you select this choice:

*Remove all watch-windows? <Y> for yes, <any other> for no.*

## WATCH LOAD AND SAVE

Currently defined windows can be saved to a disk file with the Save subcommand. These windows can be loaded again with the Load subcommand. For either subcommand, the following screen appears:




1 The DIALOG BOX prompts you for the filename to load or save the windows.

*File name:[*

If you do not specify the drive and pathname for the file, the defaults will be assumed.

2 This is the default disk drive and directory. By typing <TAB>, you can invoke prompts to change these defaults.

Note that Watch windows that have been saved in a file that is currently loaded can also be saved with the File Initializations command and loaded with the File Initializations Load command. If Windows are currently already defined or loaded in AT PROBE, then Windows from new loads will not redefine the currently defined windows. If you want to replace the current definitions with new, remove the specific Watch window names.

## OTHER NOTES ON WATCH WINDOWS

If you try to define a window which is already assigned to a Macro, the following prompt appears:

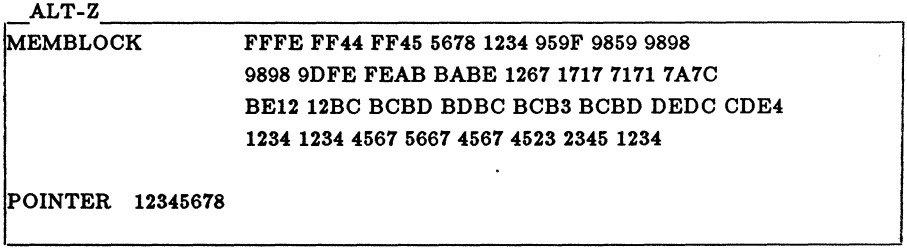
*<Alt-Key> is a macro. Remove it: <Y> for yes, <any other> for no.*

You can then elect to delete the macro in favor of the window. You can also edit Watch windows off line with your favorite text editor. See Appendix F for doing this.

## EXAMPLES OF USING THE WATCH WINDOW COMMANDS

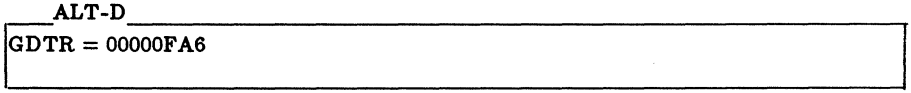
Define a window named Alt-Z which display the block of memory 3F words long which starts at location ARRAY. Put a label on this array called MEMBLOCK. Then print the Double-word which is pointed to by the variable at VARPOINTER. Put a label on this called POINTER. A sample of this window is shown below.

**W D <Alt-Z> <enter> A L MEMBLOCK <enter> <enter> R W ARRAY <enter> +3F <enter> <RtArrow 9 times> <enter> L POINTER <enter> <DnArrow 6 times> <LtArrow 9 times> <enter> R D VARPOINTER <enter> +1 <enter> <RtArrow 9 times> <enter> <Esc>**




Define a window named Alt-D which displays the contents of the Global Descriptor Table Register (GDTR) as a Double-word. Put a label GDTR in front of the data.

**W D <Alt-D> <enter> A L GDTR = <enter> <enter> E D GDTR <enter> <RtArrow 7 times> <enter> <Esc>**



Define a window named Alt-A which displays the value of a pointer represented by the symbol POINTER and label it POINTER =. Then display the 0 terminated string pointed to by this pointer.

**W D <ALT-A> <enter> A L POINTER= <enter> <enter> Z [POINTER] P <Rt Arrow 9 times> <enter>**

Load the file of windows from the file REG.WIN.

**W L REG.WIN <enter>**

Save all currently defined windows in a file called WINSAVE.WIN. First change the current directory to \WINDOWS and the current drive to B.

**W S <TAB> B <TAB> \WINDOWS <TAB> WINSAVE.WIN**

Remove the window assigned to AltD.

**W R <Alt-D> <enter>**

Remove all currently defined Watch windows.

**W R A Y**

# APPENDICES

## APPENDIX A
## AT PROBE ERROR MESSAGES

These are the error messages which AT PROBE will display. To clear the error message and resume keyboard input to AT PROBE, strike any key. The error messages in this Appendix are arranged in alphabetical order for easy reference.

*"80287 numeric co-processor not present."*
The 80287 co-processor is not present in the system so its registers cannot be displayed.

*"<Alt > is already defined. File definition ignored."*
The key specified is defined and, therefore, the definition read from the file was ignored.

*"<Alt > is not a macro key."*
The key typed is not a macro key and cannot be removed.

*"<Alt > is a macro. Remove it? <Y> for yes, any other for no"*
The key typed is already a macro key and must be removed before the macro or watch window can be defined.

*"<Alt > is a watch-window. Remove it? <Y> for yes any other for no"*
The key typed is already a watch window and must be removed before the macro or watch window can be defined.

*"<Alt > is not a watch-window key."*
The key typed is not a watch-window key and cannot be removed.

*"Absolute addresses have no SEG and OFFSET"*
The "seg" and "offset" operators cannot be used on an absolute address in /PRO versions.

*"Access denied to file."*
DOS denied access to the specified file.

*"Address is not in a function compiled and loaded with debug information"*

The end() and sizeof() operators need debug information.

*"All BPs are defined. Use Breakpoint Clear"*

All breakpoints are currently defined and one cannot be allocated for the <F5> or <F9> key. Use Breakpoint Clear to clear one (or more) BPs.

*"Arm list must contain at least one BP to arm."*

If a breakpoint is of type "Arm-bp", it must arm some BP (Breakpoint).

*"Array index is out of bounds."*

The input array index is not within the defined range for the array.

*"Array index or operator expected but not found."*

An array index or operator is expected after an array name.

*"Assembler internal error."*

An internal error occurred while assembling the instruction (Memory Unassemble).

*"Attempt to read past end-of-file."*

An attempt to read past the end-of-file was detected.

*"Attempted division by 0."*

The expression would result in a division by 0.

*"Bad drive request: Abort, Retry, Ignore?"*

A request was made to a bad drive in the DOS file system.

*"Base is 24 bit value. High bits ignored"*

Memory selector command only allows 24 bits for base value on an 80286 using AT PROBE /PRO.

*"BPn cannot be programmed together with xxxxxx"*
While arming new breakpoints, the AT PROBE's hardware break registers or software breakpoints were all used, but BPn has not yet been programmed.

*"BPn Data value: xxxxxxxxxxxxxxxxxx"*
The data value expression is illegal (Breakpoint).

*"BPn End address: xxxxxxxxxxxxxxxxx"*
The end address expression is illegal (Breakpoint).

*"BPn Start address: xxxxxxxxxxxxxxx"*
The start address expression is illegal (Breakpoint).

*"Breakpoint may not arm itself, valid ArmBP is 0 to 9"*
A breakpoint is not allowed to be armed by itself. A breakpoint is only allowed to arm breakpoints number from 0 to 9 (Breakpoint).

*"Cannot find file assignment corresponding to current CS:IP value."*
The current CS:IP does not exist in a module for which a Module-to-file assignment is known.

*"Cannot find file assignment for that line."*
The Calls command cannot associate the highlighted line number with a file.

*"Cannot open source level screen file xxxxxxx"*
The file specified in the Module-to-file assignment list cannot be opened.

*"Cannot set software breakpoint - address is not in RAM."*
The software break instruction can not be written to memory.

*"Code generator internal error."*
Internal error in code generator (Memory Unassemble).

*"Could not find symbol with name starting with input string"*
There are no symbols which start with the string typed.

*"Could not open configuration file xxxxxxxx"*
The configuration file specified could not be opened. Thus, all configuration options are set to default.

*"Count is 5 bit value. Higher bits ignored."*
The word (or dword) count for a call gate in the memory selector command (AT PROBE /PRO) is a 5 bit value.

*"CRC error: Abort, Retry, Ignore?"*
A CRC error was detected on the disk access.

*"Default address type changed to xxxx"*
After a breakpoint, the protected mode bit in msw (cr0) was set differently than the current address type so the default address type was changed. If changed to "Protected" then all addresses now have the form "selector:offset". If changed to real, then all addresses now have the form "segment:offset".

*"Did not find string from here to end of file."*
*"Did not find string from here to start of file."*
The specified search string was not found in the file from the current location to the end/start of the file (depending on if search was for next/previous occurrence).

*"Displacement too large."*
The displacement for the memory reference is too large for this type of instruction (Memory Unassemble).\

*"DOS busy: cannot open file - Run until line encountered? <Y> = yes, <other> = no"*
If the STOP button is pressed while a DOS request is in process, (the DOS busy byte is non-zero) this message appears. Entering 'Y' will run until the next line number in the program is encountered: 'N' will display the assembly screen at the current CS:IP and will not allow files to be opened. See busy option in PROBE.CNF.

*"DOS busy flag set: Opening file now would illegally re-enter DOS"*
This means the DOS busy flag is non-zero and the file cannot be opened without destroying the current state of DOS. See busy option in PROBE.CNF.

*"DOS critical error: Abort, Retry, Ignore?"*
A DOS critical error occurred during file access.

*"Drive not ready: Abort, Retry, Ignore?"*
The specified drive is not ready.

*"Error: Structure or enumeration type too big. File offset = xxxx"*
The symbolic information read from the .EXE file is invalid.

*"Error: Unknown .EXE debug record. File offset = xxxx"*
The symbolic information begin read from the .EXE file is invalid.

*"Extra characters at end of line."*
There were more characters typed for instruction than expected (Memory Unassemble).

*"File error reading .EXE debug information. File offset = xxxx"*
The symbolic information begin read from the .EXE file is invalid.

*"File not found."*
The specified file was not found.

*"File system error."*
An unknown type of file system error occurred.

*"Function/variable name not found."*
The name typed is not a recognized symbol name (Calls).

*"General error reading .EXE debug information. File offset = xxxx"*
An error occurred while reading symbolic information from the .EXE file.

*"Granularity = 4k. Limit expanded to 4k boundary."*
The limit for the selector in Memory Selector (AT PROBE /PRO) must end with FFFF if granularity is set to 4k.

*"Granularity = Byte. Limit is 20 bit value. Higher bits ignored."*
Only the low 20 bits can be used for limit if Granularity = Byte in the Memory Selector command (AT PROBE /PRO).

*"Illegal baud rate specified: xxxx"*
The baud rate specified in the configuration file is illegal. It must be {2400 | 4800 | 9600 | 19200 | 38400}.

*"Illegal destination operand."*
The first (destination) operand specified is illegal (Memory Unassemble).

*"Illegal floating point format."*
The floating point number is not in a legal format.

*"Illegal indirection: A register variable in brackets must be by itself."*
Register based variables must be indirected immediately.

*"Illegal interrupt."*
Interrupt numbers are: 0 <= Int <= FF (Memory Unassemble).

*"Illegal line number specification"*
The line number specification is not a valid decimal number.

*"Illegal operation."*
The operation specified is illegal (Memory Unassemble).

*"Illegal segment override."*
The segment register used as override is not allowed (Memory Unassemble).

*"Illegal shift operation."*
This shift factor is illegal (Memory Unassemble).

*"Illegal source operand."*
The second (source) operand specified is illegal (Memory Unassemble).

*"Insufficient memory available to load program."*
DOS reported that there is not enough memory for the program to be loaded.

*"Internal error: "*
An internal error occurred while parsing breakpoints.

*"Internal error: Unable to classify breakpoint."*
*"Internal error: Unknown single breakpoint."*
*"Internal error: No Possible break modes."*
*"Internal error: Unknown count breakpoint."*
*"Internal error: Invalid break reason."*
These are all conditions that should never happen, and should be reported to Atron if they do.

*"Internal information in EXE file is corrupt."*
The EXE file to be loaded contains corrupted data.

*"Interrupt 1 while not single stepping."*
An interrupt 1 was detected but the AT PROBE was not attempting to step in the user's program. This probably means a bad value for the FL register was popped from the stack.

*"Interrupt number is outside table limit"*
The interrupt number in the Memory Selector command for AT PROBE /PRO is larger than the table limit in the IDTR.

*"Invalid address."*
The address typed for the instruction is illegal (Memory Unassemble).

*"Invalid expression."*
The expression typed in the instruction is illegal (Memory Unassemble).

*"Invalid file access."*
An invalid access was made to the file.

*"Invalid instruction mnemonic."*
The mnemonic is not an 80286 instruction (Memory Unassemble).

*"Keyboard interrupt must be enabled. Enable now? <Y>=Yes, <other>=no"*
The keyboard interrupt must be enabled (Option Interrupts) before control can be returned to the local console (Option Screen).

*"Limit is 16 bit value. Higher bits ignored"*
The limit of a descriptor in the Memory Selector command (AT PROBE /PRO) of an 80286 is a 16 bit value.

*"Local symbol is not active on the stack at this time."*
The local symbol name entered is not yet active.

*"Local variable address must be "SI", "DI", or "SS:BP+ expression ""*
When specifying the address of a local symbol, it must be "SI" or "DI" if it exists in a register, or "SS:BP+..." if it is stack based.

*"Macro name may not be changed while editing macro."*
All information in a macro may be changed while editing except the name.

*"Macro nesting > 5. Macro will not execute."*
Macro are allowed to nest only to a level of 5.

*"Missing comma."*
Missing ',' between operators in new instruction (Memory Unassemble).

*"Missing operand"*
A required operand is missing from new instruction (Memory Unassemble).

*"Must be a memory operand."*
The noted operands must exist in memory (Memory Unassemble).

*"Must be a stack operand."*
The noted operand must exist on the stack (Memory Unassemble).

*"Must be: 1 <= Step count <= FFFF"*
The step count for code screens must be between 1 and FFFF.

*"Must have Display type == Line to define a line number"*
May only define new line numbers if currently displaying line numbers.

*"No debug information in .EXE file."*
There is no debug information in the .EXE file.

*"No line numbers in symbol table. Cannot display source level screen."*
There must be line numbers in the symbol table before the source screen can correlate addresses to line numbers.

*"No line near current CS:IP. Run until line encountered? <Y>=yes, <other>=no"*
There is no line number near CS:IP for source screen display. Type 'Y' to run to a line, 'N' to display the assembly screen.

*"No module is assigned to file "*
The filename for the file being edited does not EXACTLY match any filename in the Module-to-file assignment list.

*"No name to find yet; do a Find first."*
Cannot find Next/Previous occurrence of a Find if no Find has been done (Calls).

*"No qualified cycles were fount in trace."*
The trace qualification never put any cycles into trace memory.

*"No user executed instructions were found in trace memory."*
A GO or Step (<F8> or <F10) must have been performed before trace data is valid.

*"Note: Breakpoints will be emulated or counted in software (not real-time)."*
This message appears during a Go if breakpoints must be sequenced by the AT PROBE software rather than by the AT PROBE hardware. It can occur because of arming chains that do not fit any available complex break mode, or because of a pass count used with software breakpoints. See "Breakpoint Rules" under "Breakpoints" in chapter 5.

*"Only a quoted string is allowed for a character array"*
Only quoted strings are allowed as new values for character arrays in Memory Variable command. Either single or double quotes are allowed.

*"Operand error."*
The noted operand is invalid. Possible mis-matched operand sizes or this type of operand is not allowed (Memory Unassemble).

*"Offset is 16 bit value. Higher bits ignored"*
The offset of a gate is a 16 bit value on a 80286 in the Memory Selector command (AT PROBE /PRO).

*"Operator expected but not found in expression."*
An operator was expected in the typed expression but was not found.

*"Overlay for symbol is not loaded at this time."*
The overlay for the symbol name is not in memory at this time.

*"Pass count must be in range 1-FFFF."*
The pass count is a 16 bit counter (Breakpoint).

*"Path not found."*
An element of the specified path did not exist.

*"Possible bit values are {0 | 1 | X} optionally separated by <Spaces>"*
Bit values (e.g. for logic lines) are either '0' for a logic 0, '1' for a logic 1, or 'X' for don't care (Breakpoint).

*"Possible don't care values are '.' or 'X' optionally separated by <Spaces>"*
When entering don't care values, they must be either '.' for a care bit or 'X' for a don't care bit (Breakpoint).

*"Printer out of paper: Abort, Retry, Ignore?"*
The printer is out of paper and cannot be written to.

*"Program environment area has been corrupted."*
The program environment area has been corrupted and DOS can no longer load programs.

*"Program terminated, termination code = xx"*
The program terminated (AH=4c, Int 21) with the termination code (al=xx).

*"Quit Remain has been performed. You may not Quit or Quit Remain again."*
One a Quit and Remain Resident has be selected, you may not quit. You must have a program loaded and GO, or you must re-boot.

*"Read fault: Abort, Retry, Ignore?"*
A read fault occurred reading from the device.

*"Register variable not allowed in address expression (except in brackets)."*
A register based symbol name is not allowed in an address expression unless it is dereferenced.

*"Register variables not int the current function are not allowed"*
Register variables are only allowed if CS:IP is currently in the function containing the variable for Memory Variable.

*"Relative jump out of range."*
The input relative jump is more than 128 bytes away (Memory Unassemble).

*"Remove all externals, internals, and lines? <Y> for yes; <any other> for no"*
Type 'Y' to clear the symbol table of all symbols.

*"Remove all macros? <Y>=yes; <any other> for no"*
Type 'Y' to remove all macro definitions (Macro Remove).

*"Remove all watch-windows? <Y> for yes; <any other> for no"*

*"Sector not found: Abort, Retry, Ignore?"*
The sector was not found by the DOS file system.

*"Seek error: Abort, Retry, Ignore?"*
A seek error was detected by the file access.

*"Selector value is outside table limit"*
The selector in the Memory Selector command for AT PROBE /PRO is larger than the table limit in the GDTR or LDT.

*"Specified size is too big for indirection through a register variable."*
The size of indirection specified is too large.

*"Structure field name or operator expected but not found."*
A field name or operator is expected after structure/union/bit field name.

*"Symbol/Macro/Watch-window allocation table is full."*
The AT PROBE memory allocation table is full and no new symbols, macros, or watch-windows may be added until some are deleted.

*"There is no address corresponding to this line in the symbol table."*
The highlighted line in the file does not correspond in a known way to an address in memory.

*"There were more '(' than ')'."*
Parenthesis are mismatched.

*"There were more '[' than ']'."*
Brackets are mismatched. This error can occur in expressions like "[[pointer].structmember]" if "pointer" is not of type "struct *". In this case expression parsing stops at the '.', hence the message.

*"Too many '('s to be parsed"*
The expression contained too many '(' and cannot be parsed. Use less open parens or close some out.

*"Too many hardware break registers needed."*
Too many hardware breakpoints are needed to implement the currently active breakpoints.

*"Too many open files."*
DOS reported too many open files. No more may be opened until one is closed.

*"Too many operands."*
Too many operands were listed for this instruction (Memory Unassemble).

*"Too many software breakpoints needed."*
Too many software breakpoints are needed to implement the currently active breakpoints.

*"Too many trace on/off conditions."*
Too many trace on/off conditions were specified to implement the trace qualification.

*"Too many trace qualification regions."*
Too many trace qualification regions were specified to implement the trace qualification.

*"Trace on/off address: "*
The on/off address for qualification was illegal.

*"Trace qual data value: "*
The data value for qualification was illegal.

*"Trace qual end address: "*
The end address for qualification was illegal.

*"Trace qual start address: "*
The start address for qualification was illegal.

*"Type file number of file to close before opening xxxxx"*
The file list for File View is full and one file must be "closed" before the new one can be opened.

*"Type mismatch."*
The operands input are of a different type (Memory Unassemble).

*"Unable to match data cycles with prefetched instructions in trace."*
This may be caused by uninitialized trace memory (no Go or step command has been entered), or a long string instruction was recently executed. As a result, data cycles listed below may not appear with the instruction that generated them. Also, instructions with an "*" may or may not have executed. Press any key to continue, <Esc> to abort. The trace data resident on the AT PROBE board cannot be decoded.

*"Unknown command: Abort, Retry, Ignore?"*
An unknown command was issued to the DOS file system.

*"Unknown DOS error while loading program."*
DOS reported an unknown error while loading the program.

*"Unknown media: Abort, Retry, Ignore?"*
The media type is not known by DOS file system.

*"Unknown operand size."*
The operands size for the instruction is not known and must be specified (e.g. INC [xxxx]) (Memory Unassemble).

*"Unknown symbol file (Could not find "Publics by" in map file)"*
The file listed for loading of symbols was of an unknown type. That is, it was not a ".EXE" file and it was not a ".MAP" file which contained the "Publics by" string.

*"Unknown unit: Abort, Retry, Ignore?"*
An unknown unit was specified.

*"Unknown version of debug information in .EXE file."*
There is debug information in the .EXE file but it cannot be loaded.

*"Value expected or symbol name not found."*
A value was expected at the noted location but one was not found. This may be because the symbol name typed was not found.

*"Value written to memory is different from value read back."*
The value written to memory is not the same as the value that was read back from the same address. This probably means that the write occurred to an area of memory that does not contain functional RAM.

*"Values not found from current location to the end of trace"*
The search values were not found from the cursor location to the end of trace.

*"Write fault or disk full."*
A write fault occurred while writing to the disk. The disk may be full.

*"Write fault: Abort, Retry, Ignore?"*
A write fault occurred writing to the device.

*"Write-protect error: Abort, Retry, Ignore?"*
The disk selected to write file to is write protected.

## APPENDIX B
## FILES ON YOUR AT PROBE DISKETTES

There are several files on your AT PROBE diskettes which may or may not be needed depending upon what you are doing. Only those used for executing AT PROBE are required. A list of these files and their purpose is given below:

| AT PROBE FILES | VERSION | DESCRIPTION |
|---|---|---|
| atprobe.exe | | for executing PROBE on IBM (R) BIOS systems |
| atprbhp.exe | | for executing PROBE on Phoenix (R) BIOS systems |
| probe.sys | | for debugging device drivers |
| p2.exe | | for executing AT PROBE |
| atpdia.exe | | diagnostic confidence test on IBM (R) BIOS systems |
| atpdiahp.exe | | diagnostic confidence test on Phoenix (R) BIOS systems |
| probe.cnf | | sets AT PROBE configuration parameters |
| demo.c | | c source file for demo program module 1 |
| demo2.c | | c source file for demo program module 2 |
| demo3.asm | | assembly language source file for demo program module 3 |
| demo4.c | | c source file for demo program module 4 |

| | |
|---|---|
| demo.mac | macro file for demo program |
| demo.wch | window file for demo program |
| demo.exe | executable file for demo program |
| pc.cnf | PROBE.cnf type file configured for pc as external console. Copy it to PROBE.cnf |
| hp.cnf | PROBE.cnf file configured for HP terminal as external console |
| tv970.cnf | PROBE.cnf type file configured for Televideo 970 as external console. |
| wyse.cnf | PROBE.cnf type file configured for Wyse terminals external console |
| termcom.exe | makes a pc a terminal emulator for use with AT PROBE as an external console. AT PROBE uses pc.cnf |
| termmode.exe | can be used to make a PC run at speeds above 9600 baud. |
| termmod2.exe | similar to termode, but configures com2: |
| readme.doc | (optional) describes any changes to the PROBE subsequent to the release of the current PROBE manual |
| drinit.asm | source code for probe.sys driver "Init" function, which does most of the work |

| | |
|---|---|
| driver.equ | equates used by the probe.sys .asm files |
| drivers | "make" description file for probe.sys |
| drmain.asm | source code for probe.sys DOS driver interface |
| drprot2.asm | source code for probe.sys real/protected mode switching on a 286 with an IBM (R) BIOS |
| drprot3.asm | source code for probe.sys real/protected mode switching on a 386 |
| drprothp.asm | source code for probe.sys real/protected mode switching on a 286 with a Phoenix (R) BIOS |
| *.hlp | AT PROBE help files |

| SOURCE PROBE files | VERSION | DESCRIPTION |
|---|---|---|
| atsource.exe | | file for executing AT Source PROBE on an IBM (R) BIOS system |
| atsrchp.exe | | file for executing AT Source PROBE on a Phoenix (R) BIOS system |
| s2.exe | | for executing AT Source PROBE |

All other files are duplicates of those on the PROBE diskette.

## APPENDIX C
## GENERATING SYMBOLS

AT PROBE allows you to use the symbolic information from your program during debugging instead of absolute numbers. The symbolic debugging information is passed to the AT PROBE from the compiler using controls which are discussed here. This symbolic information may consist of public variables, public procedures, functions, subroutines, modulenames, and high level language line numbers. Some compilers will also produce symbols for local variables and procedures.

### Generating Symbols using MICROSOFT C Ver. 4.0 or 5.0

To make all symbolic information available to AT PROBE using Microsoft C Compiler version 4.00 or later, compile with the /Zi option. To generate only line numbers use the /Zd option. Use the /Od option to disable compiler optimization (this will make debugging much simpler). Next, using the Microsoft linker version 3.51 or later with the /co option. The symbol table information is now in the .exe file and is loaded with the File Initializations or File Program Load command. Note that /Zi does not increase the size of the program image that is loaded; it puts debug information after the load image in the .exe file.

### Generating Symbols for other compilers

Compilers other than the Microsoft C compiler version 4.0 and later must set compiler flags such that a Microsoft compatible MAP file is created. For example, if the Computer Innovations compiler version 3.01 is used with DOS link version 2.30, the command line would be:

```
CC filename.c -x2 ...
link ... /map /linenumbers
```

Modules will have the same names as their respective object files.

## APPENDIX D
## CONFIGURATION FILE

The configuration file is an ASCII file named PROBE.CNF, and the file provides AT PROBE system information to be used during some commands. You can use a text editor to change this file. Your text editor should store PROBE.CNF as ASCII text and should not include other hidden text editor control information in PROBE.CNF. PROBE.CNF specifies the following:

### REMOTE CONSOLE ESCAPE SEQUENCES AND SPECIAL CHARACTERS

Here are some definitions for special characters specified in the remote console escape sequences which will be used in describing the configuration file. No distinction is made between upper-case and lower-case characters.

| | |
|---|---|
| X | an ASCII 'X'. AT PROBE will add the column number to the input offset to create an ASCII representation of the column number and transmit the resulting two character sequence ("00"-"84") to the remote terminal. |
| Y | an ASCII 'Y'. AT PROBE will add the row number to the input offset to create an ASCII representation of the row number and transmit the resulting two character sequence ("00"-"84") to the remote terminal. |
| V | an ASCII 'V'. AT PROBE will add the column number to the input offset and send the result in binary format (0-ffh) to the remote terminal. |
| W | an ASCII 'W'. AT PROBE will add the row number to the input offset and send the resulting result in binary format (0-ffh) to the remote terminal. |
| S | an ASCII 'S'. AT PROBE will sent the character at this point in the escape sequence. |
| N | an ASCII 'N'. AT PROBE will send a NULL character (00h) at this point in the escape sequence. |

Here are the parameters in the PROBE.CNF file:

**ADDR** = base address [, software execution address]

Base address is the start of the 1 meg of memory on the AT PROBE board. It can be set to either 100000 or D00000 (default). Software execution address is the start on an extra megabyte of memory that, if specified, will be used in addition to the memory on the AT PROBE board to provide extra symbols table space.

**BUSY** = xxxx:yyyy

Set to the address of the DOS "busy" byte. This byte is non-zero if a DOS request is in progress and is examined when the STOP button is pressed to determine if control can safely returned to AT PROBE without destroying the current state of DOS. If omitted or specified as zero ("0"), the segment value used is the segment address for the INT 20 vector (located at 0000:0082). If omitted or specified as zero the offset value is set according to the following:

DOS 2.xx -- 012Dh
DOS 3.00 -- 0311h
others -- 02CFh (1)

This value can be set using a standard PROBE expression. For example, to use the segment form the INT 21 vector with offset 02cfh, the string [0:(21*4)+2]:02cf would be entered. To disable the DOS busy check, set BUSY = ffff:ffff.

(1) These values have been tested and found to work on DOS 2.10, 3.00, 3.10, 3.20, and 3.30

**FIELD = {L|C}**

The default, 'L' specifies that on a remote terminal the all of the current line of code will be displayed in reverse video. If 'C' is specified, then only the first character of the current line is highlighted. This has no effect on fields of less than one full line of length.

**FLOAT = {y|n}**

If a 80287 floating point co-processor is present in the computer FLOAT should be set to "Yes", otherwise FLOAT should be set to "No". In some cases it is not possible to accurately test for the co-processor, and so AT PROBE may cause the system to hang when it attempts to perform a floating point operation.

**COLOR = FC [,BC]**

Set the Foreground Color and Background Color of the monitor. This configuration switch is ignored for monochrome monitors. Colors are:

'R' -- Red 'G' -- Green 'B' -- Blue
'C' -- Cyan 'Y' -- Yellow 'M' -- Magenta
'W' -- White 'K' -- blacK

If omitted, the color is set to "W, K" (white on black). If the background color is omitted, it is set to black.

**Examples:**

| | |
|---|---|
| COLOR = Y,K | Yellow on black |
| COLOR = Blue,White | Blue letters on white background |
| COLOR = green | Green letters on black background |

**MONO = {y|n}**

If no, then AT PROBE waits for retrace to write to screen. If yes, AT PROBE does not use any color attributes and does not wait for retrace to write to the screen.

**LOGIC = {y|n}**

Tell AT PROBE if logic PROBEs are installed. You can override this when you are executing PROBE with the Options Logic PROBEs command. The default value is 'N'.

**BAUD = xxxx**

xxxx can be 1200, 2400,4800,9600 19200, 38400. Sets the baud rate for external console using the AT PROBE serial port or the COM1 or COM2 ports. If omitted, or if any other value is listed, 9600 baud will be used. No parity, 8 data bits, and 2 stop bits are always used and cannot be changed.

**ROWS = xx**

This is the number of rows on the external console. May be up to 32 hex (50 decimal). This value must be specified in hexadecimal. The default is 19.

**LINES = xx**

This is the number of lines on the local console. This defaults to 19 (25 decimal) but can be set to 2B (43 decimal) or 32 (50 decimal). This tells the software how many lines are currently configured on the monitor at the time the AT PROBE software is loaded.

**CLEAR = esc sequence in hex**

This is the esc sequence used to clear the external console's display from the cursor position to the end of the current line. The default value is 1B 32.

**INVERSE = esc sequence in hex**

This is the esc sequence used by the terminal to put a character into inverse video. The default value is 1B 35.

**NORMAL = esc sequence in hex**

This is the esc sequence used by the terminal to put an inverse video character back into normal mode. The default value is 1B 36.

**WAIT = xxxx**

This is the time to wait after moving the cursor or providing a line feed character to the terminal. It is a 32 bit hex number which is determined by the speed of the system. The default value is 80h.

**MOVE = esc sequence in hex**

Move cursor to row Y, column X (see X, Y definition).
or
Move cursor to row W, column V (see W, V definition).

The default value is 1B 33 W V.

**OFFSET = xx [,xx]**

Set offset to be added to V, W or X, Y. This is used in circumstances where the upper left corner of the display is an offset of some value from 0,0. For example, if the terminal being used expects the upper left corner of the display to be accessed as 1,1, OFFSET should be set to 1,1. If the terminal expects the value of 20 hex to be added to row and column values, then OFFSET should be set to 20,20. The row offset is listed first. If omitted, the column offset is set to the row offset. The default value is 0,0.

## DEFAULT CONFIGURATION FILE

The default PROBE.CNF file as supplied on your distribution diskette does not contain any settings, therefore, AT PROBE assumes the defaults described earlier. If AT PROBE does not find the file PROBE.CNF in the directory from which the executable files were loaded, then these default parameters are used..

## APPENDIX E
## EXTERNAL CONSOLE CONNECTION

### CONNECTING AN EXTERNAL CRT TO AT PROBE'S SERIAL PORT

AT PROBE assumes the following transmission parameters for transmitting and receiving data via the on-board serial IO channel.

9600 baud (this is the default)
8 data bits
2 stop bits
no parity

This can be reconfigured using either the PROBE.CNF file or from the Options Screen menu.

The interface for the AT PROBE on-board serial IO channel is standard RS232 for transmit and receive with the signals provided on the pins shown in Figure C-1. This is the same as the standard COM1 PORT configuration. Only TXD, RXD, and GND are supported. Note that your terminal may require you to connect pins 4 to 5 (RTS - CTS) and 6 to 20 (DSR - DTR) at the terminal.

### USING THE PC AS A TERMINAL EMULATOR

Another PC can be used as an intelligent terminal by using the Termcom program provided on the AT PROBE diskette. In this case connect pins 5-6-20 on the PC. Termcom is a terminal emulator program which makes the COM1 port of a second PC look like an intelligent terminal to AT PROBE.

1. On the PC which is acting like a terminal, use the mode command to set the baud rate.

**mode com1: 9600,n,8,2**

For baud rates above 9600 use the termmode (for Com1:) or termmod2 (for Com2:) program with the following parameters:

| | |
|---|---|
| /Bnnnn | nnnn = 24, 48, 96, 19.2, 38.4, 57.6, or 115.2 baud |
| /Dn | n = 5, 6, 7, or 8 data bits |
| /Sn | n = 1 or 2 stop bits |
| /Pc | c = "N", "O", "E", "0", or "1" to indicate parity |

2. Connect an RS232 cable between the AT PROBE com port and the COM1 port of the PC terminal emulator.

3. On the PC terminal emulator type:

**termcom [displaylines]** (2)

where the displaylines option is 2B or 43 for EGA/VGA 43 line mode and 32 or 50 for VGA 50 line mode. If termcom is used in 43 or 50 line mode, PROBE.CNF needs "ROWS = 2B" or "ROWS = 32" to tell the AT PROBE software that the terminal has that many lines.

4. Once AT PROBE has been brought up, switch to the external console.

Commands which send data to the screen can be terminated with the Ctrl C key or the STOP button on the external switch box.

(2) added Version 2.03

## USE OF NON-ASCII KEYS FROM A REMOTE CONSOLE

The non-ASCII keys on the PC keyboard can be duplicated on the remote console by use of the '^' character. This character (5EH, <Shift-6> on the IBM keyboard) is used to signal that a non-ASCII specifier is coming next in the input stream. A double '^' sequence signifies that an <Alt> key is coming next. These sequences are generated automatically by the TERMCOM.EXE program, version 2.01.

| PC keyboard key | Remote sequence |
|---|---|
| <CursorUp> | ^P |
| <CursorDown> | ^L |
| <CursorLeft> | ^S |
| <CursorRight> | ^R |
| <Ctrl-CursorRight> | ^F (3) |
| <Ctrl-CursorLeft> | ^G (4) |
| <PgUp> | ^U |
| <PgDn> | ^D |
| <Ctrl-PgUp> | ^T |
| <Ctrl-PgDn> | ^B |
| <Home> | ^H |
| <End> | ^E |
| <Ctrl-Home> | ^Y |
| <Ctrl-End> | ^Z |
| <Ins> | ^I |
| <Del> | ^K |
| <Alt--> (minus) | ^^- |
| <Alt-=> | ^^= |

(3) added Version 2.02
(4) added Version 2.02

| | | |
|---|---|---|
| <F1> | | ^1 |
| | to | |
| <F10> | | ^0 |
| <Alt-A> | | ^^A |
| | to | |
| <Alt-Z> | | ^^Z |
| <Alt-F1> | | ^^1 |
| | to | |
| <Alt-F10> | | ^^0 |
| <Alt-1> | | ^^! (5) |
| | to | |
| <Alt-0> | | ^^) (6) |

(5) Note: '!' is <Shift-1>
(6) Note: ')' is <Shift-0>

## APPENDIX F
## TEXT FORMATS FOR MACROS, WATCH WINDOWS, AND INITIALIZATION FILES

Macros, Watch Windows, and Initialization files are stored as text files. Macro files are created with the Macro Save command. Watch Windows are created with the Watch Window Save command. Initialization files are created with the Initialize Save command. Macros and windows can be edited on-line with the Macro Edit and Watch Window Edit commands. Macro, Watch Window, and Initialization files can also be edited off-line with a standard text editor and stored as text files. Be sure that your editor only stores the file as pure ASCII text and does not include additional control codes. The formats for these files is described below.

## MACRO FILE FORMATS

In macro editing, both on- and off-line, the special keyboard keys are specified exactly as described below:

| SPECIFICATION | DESCRIPTION |
|---|---|
| <Enter> | Enter key. |
| <Esc> | Esc key. |
| <Tab> | Tab key. |
| <BS> | <- backspace key. |
| <Home> | Home key. |
| <End> | End key. |
| <PgUp> | PgUp key. |
| <PgDn> | PgDn key. |
| <CtrlHome> | Home key with the Ctrl key held down. |
| <CtrlEnd> | End key with the Ctrl key held down. |
| <CtrlPgUp> | PgUp key with the Ctrl key held down. |
| <CtrlPgDn> | PgDn key with the Ctrl key held down. |
| <CursorUp> | Up arrow key. |
| <CursorDown> | Down arrow key. |
| <CursorLeft> | Left arrow key. |
| <CursorRight> | Right arrow key. |
| <CtrlCursorLeft> | Left arrow key with the Ctrl key held down. |
| <CtrlCursorRight> | Right arrow key with the Ctrl key held down. |
| <Ins> | Ins key. |
| <Del> | Del key. |
| <F1> to <F10> | function keys. |
| <Alt?> | Alt key. |

**Where:**

Possible values for '?' are: any letter A..Z, Space (i.e.<Alt >) any number 0..9 - or = F1 to F10

The format for macros in a macrofile are:

MACRONAME [':' MACROTYPE]['.' MACRODESCRIPTION]
MACRODEFINITION
BLANKLINE

**Where:**

MACRONAME = an AltKey
MACROTYPE = I | LF | LC | LW

'I' for a conditional macro
'LF' for a loop forever macro
'LC' COUNT for a loop count macro.
COUNT is an expression
'LW' CONDITION for a loop while macro.
CONDITION is a boolean expression which is either TRUE (not 0) or FALSE (0)

MACRODEFINITION = keystrokes for macro

BLANKLINE a line containing only a Cr (or Cr,Lf) or the End-of-file to signal end of macro.

**Examples:**

This macro file contains the macro Alt-R. Macro <AltR> is a macro which will cause AT PROBE to reload the file currently being debugged.

<AltR>.reload executable file
fp<Tab>
<Enter>
n<Enter>
<Enter>
<Enter>
<Enter>

## WINDOW TEXT FILE FORMATS

The format for Windows in a window file are:

```
WINDOW_NAME
FIELD_SPEC
FIELD_SPEC
FIELD_SPEC
...
FIELD_SPEC
NULL_FIELD
<Blank line>
```

**Where:**

WINDOWNAME = an AltKey

FIELDSPEC = ROW ',' COL <enter>
{EXPR | ZEROSTR | LENSTR | RANGE | LABEL}

NULL_FIELD = "FF,FF" <Enter>

ROW = hex number.
Row relative to start of watch-window for field.

COL = hex number.
Column for field (01..4E).

EXPR = 'E' EXPRTYPE <Enter>
expression <Enter>

EXPRTYPE = {'B' | 'W' | 'D' | 'U' | 'I' | 'A' | 'S'}
(for Byte,Word,Dword,Unsigned,Integer,ASCII,Symbol)

ZEROSTR = 'S' 'Z' <Enter>
address <Enter>

LENSTR = 'S' 'L' <Enter>
address <Enter>
length <Enter>

RANGE = 'R' RANGETYPE <Enter>
start address <Enter>
end address <Enter>

RANGETYPE = {'B' | 'W' | 'D' | 'S' | 'L' | 'T' | 'P'}
(for Byte,Word,Dword,fShort,fLong,fTemp,fPacked)

LABEL = 'L' <Enter>
label <Enter>

EXPRESSION = standard AT PROBE expression.

ADDRESS = standard AT PROBE address expression.

LENGTH = length of string.

**Example:**

To create a watch-window that will be opened by typing <AltW> that will display register AX on the first line of the window, followed by the zero-terminated string at DS:SI. The next line of the window will contain the first 10H bytes of the string in byte format.

| File definition | Explanation |
|---|---|
| <AltW> | Watch-window name |
| 00,01 | Row 0, Column 1 |
| L | Label field |
| AX= | "AX=" is the label |
| 00,04 | Row 0, Column 4 |
| EW | Expression,Word field |
| AX | Value for expression is in register AX |
| 00,0C | Row 0, Column C |
| L | Label field |
| DS:SI--> | "DS:SI-->" is the label |
| 00,14 | Row 0, Column 14 |
| SZ | String,Zero-terminated field |
| DS:SI | Address for string is DS:SI |
| 01,0C | Row 1, Column C |
| L | Label field |
| DS:SI= | "DS:SI=" is the label |
| 01,12 | Row 1, Column 12 |
| RB | Range,Byte field |
| DS:SI | Start address for range is DS:SI |
| DS:SI+10 | End address for range is DS:SI+10 |
| FF,FF | Null field at end of definition. |

## INITIALIZE FILE FORMATS

The initialization file consists of blocks of information for each set of data stored in the file. The blocks of information may be in any order in the file. (The INIT block is the last block loaded from the file. It is never saved). Each block must be separated from the next block by a blank line. The blocks are always saved (and the file is searched so that they are always loaded) in the following order:

OPTION
MODULE
PROGRAM
MACRO
WINDOW
INIT

**OPTION block:**

The OPTION block contains the settings for the sub-commands in the Option command.

```
OPTION =
Screen, View-ops, Mix-source, sYmbols, Case, sTep-count,
Read-verify, Functions
<Blank line>
```

SCREEN= {'S' | 'F' | 'N'}
(for Switch,Flip,None) Screen switching type.

VIEW-OPS= {'Y' | 'N'}
(for Yes or No) View operands during step.

MIX-SOURCE= {'Y' | 'N'}
(for Yes or No) Mix source lines with assembly.

SYMBOLS= {'Y' | 'N'}
(for Yes or No) Display symbols with assembly.

CASE= {'Y' | 'N'}
(for Yes or No) Symbol case sensitivity.

STEP-COUNT= {xxxx}
Steps to take for each <Enter>.

READ-VERIFY= {'Y' | 'N'}
(for Yes or No) Verify memory writes.

FUNCTIONS= {'M' | 'L'}
Function linkage style.

**Example:**

```
OPTION=
None,Yes,Yes,Yes,No,0001,Yes,M
```

**MODULE block:**

The module block contains the information specific to each module to be loaded into the symbol table. The options can be used to limit the number of symbols loaded into the symbol table.

```
MODULE=
modulename,Load?,Step?,filename
modulename,Load?,Step?,filename
...
modulename,Load?,Step?,filename
<Blank line>
```

Modulename = The name of the module as defined in the .EXE file or .MAP file

Load? = {'Y' | 'N'}
(for Yes or No) Load symbols/lines for this module

Step? = {'Y' | 'N'}
(for Yes or No) Source level step in this module

filename = ????
The name for the source file that generated this module.

**Example:**

```
MODULE=
ftocm,Yes,Yes,c:\source\ftocm.c
ftocio,No
calc,Yes,Yes,c:\util\calc.c
```

**PROGRAM block:**

The program block consists of the program name (and symbol file name) to be loaded, along with any command line arguments.

```
PROGRAM=
programname arguments
<Blank line>
```

The syntax for this block is the same as the syntax for loading a program from the command line when starting AT PROBE. That is:

```
filename.exe[.map] <arguments>
```

**Example:**

```
PROGRAM=
ftoc.exe.map 10, 20, 1
```

**MACRO block:**

The macro block consists of a list of filenames, each of which contains macros to be loaded. Only the last mAcro Save (or mAcro Load) command file name is saved to the init file.

```
MACRO=
filename
filename
...
filename
<Blank line>
```

**Example:**

```
MACRO=
c:\Atron\debug.mac
c:\Util\util.mac
c:\source\ftoc.mac
```

**WATCH-WINDOW block:**

The watch-window block consists of a list of filenames, each of which contains watch-windows to be loaded. Only the last Watch Save (or Watch Load) command file name is saved to the init file.

```
WATCH=
filename
filename
...
filename
<Blank line>
```

**Example:**

```
WATCH=
c:\Util\memory.win
c:\source\ftoc.win
```

**INIT block:**

The init block contains a macro name. This macro will begin execution as soon as all of the other blocks have been loaded from the init file.

The INIT block is never saved.

```
INIT=
<Alt Keyname>
<Blank line>
```

**Example:**

```
INIT=
<AltI>
```

## APPENDIX G
## PROBE/MS DOS INTERFACE DESCRIPTION

**Start-up:**

**INT 11**

Get video equipment type

**INT 21**

| | |
|---|---|
| ah=1; | Read character for ".EXE" path. |
| ah=2; | Print character for start-up error message. |
| ah=4C; al=1 | Abort because of error |
| ah=3D; al=0 | Open file "PROBE.CNF", "S2.EXE", "P2.EXE" |
| ah=3E | Close file |
| ah=3F | Read from file |
| ah=42 | Seek in file |
| ah=19 | Get current disk |
| ah=47 | Get current directory |
| ah=4A | Set block size to size of PPS and real/protected mode interface code |

**Execution:**

**INT 10 (Local/Other console only)**

| | |
|---|---|
| ah=0 | Set video mode |
| ah=2 | Move cursor |
| ah=3 | Get cursor position |
| ah=5 | Set video page |
| ah=11 | Set CRT palette |
| ah=15 | Get video mode/page |

**INT 16 (Local/Other console only)**

| | |
|---|---|
| ah=0 | Get character from keyboard |
| ah=1 | Character ready at keyboard? |
| ah=2 | Get shift status |

INT 21

| | |
|---|---|
| ah=0D | Flush disk buffers after closing any file |
| ah=1A | Set DTA when directory printed/Program-load |
| ah=29 | Parse filename into DTA for Program-load |
| ah=30 | Get DOS version for Program-load |
| ah=31; al=0 | Quit and remain resident command |
| ah=3C | Create file handle<br>Init/Macro/Window/Trace<br>save Log-file |
| ah=3D | Open file handle<br>Init/Macro/Window<br>Load View file<br>Program-load<br>Source code display<br>Log-file reopened |
| ah=3E | Close file handle |
| ah=3F | Read from file |
| ah=40 | Write to file |
| ah=42 | Seek in file<br>Program-load<br>View file<br>Reading config file<br>Reading init file<br>Reading symbol file |
| ah=48 | Allocate memory before Program-load |
| ah=49 | Release allocated memory before Prog-load |
| ah=4B; al=01 | Program-load; do not begin execution |
| ah=4C; al=FF | Terminate loaded program before new load |
| ah=4C; al=00 | Quit |
| ah=4D | Get termination code for process after execution of program |
| ah=4E | Get first matching file when directory printed |
| ah=4F | Get next matching file when directory printed |

## OTHER RESOURCES USED BY AT PROBE

AT PROBE uses 43k bytes of memory to initially load itself onto the PROBE board. After that, it releases all but 8k bytes. These 8k bytes remain allocated until PROBE exits.

## APPENDIX H
## LANGUAGE COMPATIBILITY

| Software | Version | Manufacturer | |
|---|---|---|---|
| C | | Microsoft | |
| | | IBM | (no local symbol support) |
| | | Lattice | (no local symbol support) |
| | | Metaware | (no local symbol support) |
| | | Borland | (no local symbol support) |
| | | Computer Innovations | (no local symbol support) |
| Assembler | | IBM/Microsoft (7) | |

(7) In order to debug with line numbers from MASM 5.0 or greater, the one segment per module convention of Microsoft's C compiler must be observed.

## APPENDIX I
## GLOSSARY OF BREAKPOINT TERMS

**Breakpoint:**

An event which normally causes an interruption of user program execution and a return to the Atron PROBE code.

**Hardware Breakpoint:**

Any of the following types of events:

Memory Read
Memory Write
I/O Read
I/O Write
Instruction Fetch

**Hardware break register:**

The hardware on the AT PROBE board that detects bus cycles that match specified conditions. There are four break registers on the AT PROBE board, and more than one at a time may be needed to implement a single hardware breakpoint as specified by the Breakpoint Define command.

**Software Breakpoint:**

An instruction execution breakpoint that is achieved by inserting an "Int 3" instruction in the user code at the desired address.

**Simple Breakpoint:**

Any hardware breakpoint that can be achieved by using one of the four hardware break registers.

Examples are:
Memory write to a given address.
Instruction fetch at a given address.

**Range Breakpoint:**

Any hardware breakpoint that is defined over a range of addresses. Range breakpoints normally require two hardware break registers.

An example is:

Memory write anywhere in the address range 7000:0 to 8000:0.

Under some circumstances only one hardware break register is needed for a range of addresses. This kind of breakpoint is referred to as a "Simple Range Breakpoint". The "Breakpoint Rules" section under the "Breakpoint" command description describes the circumstances under which range breakpoints are treated as simple ranges.

**Simple Range Breakpoint:**

A range breakpoint that only requires one hardware break register. See "Range Breakpoint" above.

The term "Simple Range Breakpoint" refers to the internal form of certain breakpoints that are specified as "Range Breakpoints" by the user.

**Data Breakpoint:**

Any hardware breakpoint that is defined as happening only when specified data values are read or written.

Examples are:

Value 1234 written to a given address.
Value 12 written to any of a range of addresses.

**Pass Count:**

A repetition factor to be applied to one or more breakpoints. If a pass count is used, it works this way: Any time an event happens which would otherwise cause the AT PROBE to stop user program execution and return to the debugger, this event increments the pass counter. If the specified number of repetitions has not occurred yet, then user program execution continues, with breakpoints set at their initial conditions. (More on initial conditions under "Arming".)

An example is:

Pass count = 5.
Memory write to a given address.

In this example, the user program will execute until the fifth time the specified memory write occurs.

Another example:

Pass count = 5.
Memory write to a given address.
Memory read from a different address.

This example will stop when the total number of occurrences of the two breakpoint conditions equals five. E.g. four of the memory writes followed by the memory read will stop execution.

**Arming:**

Rather than stopping execution, a breakpoint event can arm another breakpoint condition. When breakpoint "A" arms breakpoint "B", only breakpoint "A" is armed initially. (If event "B" happens before event "A", the user program won't stop.) Once event "A" happens, then breakpoint "A" is disarmed and breakpoint "B" is armed. Event "A" followed by event "B" will stop the user program and return to the AT PROBE.

**Resets:**

A breakpoint which arms another breakpoint can arm another breakpoint as well, which will "reset" the breakpoints back to the initial state.

An example:

A arms B.
A arms C.
C resets.

In this example, event "A" followed by event "C" will act as though event "A" had never happened; the AT PROBE will be back looking for event "B". For instance:

A then B will stop execution.
A then C then A then B will stop execution.
A then C then B will not stop execution.

# APPENDIX J
# TECHNICAL REPORTS

## USER PROCESSED NMI

The AT PROBE uses the non-maskable (NMI) interrupt to generate the breakpoint.

## INTERRUPT 3

Software interrupt 3 is used by the AT PROBE for the generation of software breakpoints and should not be used by the user's program.

## STACK USAGE DURING BREAKPOINT

After a breakpoint has been detected, 28 bytes of information are pushed onto the stack. However, after control is received by AT PROBE, the stack pointer is adjusted to remove this data. When starting program execution again, the stack is also restored correctly.

## GETTING TO DOS COMMANDS FROM AT PROBE

To execute DOS commands under the watchful eye of AT PROBE, do the following:

1. Load AT PROBE.

2. Now do a quit and stay resident command from AT PROBE - **FQ <TAB> Y Y.**

3. AT PROBE can now be re-entered by using the STOP button as long as its vectors have not been modified.

## MODIFYING THE AT PROBE SOFTWARE FROM THE APPLICATIONS PROGRAM

The memory on the hardware versions of AT PROBE is write protected. This means that if the applications program tries to write to the memory area of the AT PROBE it will not change the memory.

## INTERRUPTING CRITICAL CODE SECTIONS IN DOS

If the STOP button on the external switch box is pressed or a breakpoint occurs while the executing user's program is inside a non re-entrant BIOS call such as the keyboard int 16 or monitor service routines (INT 10), then the absolute locations which these routines address will be modified and the results will be indeterminate (the system may appear to lock up). This is because the AT PROBE software uses BIOS calls to do some console IO. To eliminate this from happening, switch to the external console where no BIOS calls are made by AT PROBE software since the external console routines are in the AT PROBE software.

# INDEX



## A



## B

## C

## D



## E

## G



## H



## I

## J



## K



## L



## M

## N



## O



## P



## Q



## R



## S

## T



## V

## W