PC PROBE

PC SOURCE PROBE

PC PROBE
PC SOURCE PROBE

20665 FOURTH STREET * SARATOGA, CA 95070 * (408) 741-5900

TABLE OF CONTENTS

CHAPTER 1. INSTALLING THE PC PROBE

CHAPTER 2. GETTING STARTED

CHAPTER 3. GENERATING AND USING SYMBOLS

CHAPTER 4. USING PROBE COMMANDS

CHAPTER 4. USING PROBE COMMANDS, continued

CHAPTER 5. DEBUGGING APPLICATIONS

CHAPTER 6. COMMAND REFERENCE

CHAPTER 6. COMMAND REFERENCE, continued

APPENDICES

FIGURES

TABLES

INTRODUCTION

Thank you for purchasing an Atron debugging tool. This manual describes the PC PROBE and its upgraded software option PC SOURCE PROBE. The standard PC PROBE software has symbolic debugging capabilities. The SOURCE PROBE option adds source level debugging features to the symbolic debugging capability of PC PROBE.

The PC PROBE is a hardware assisted software debugging tool. It is designed to provide flexible debugging capability for developing software or hardware add-on products for the IBM PC. PC PROBE can be used to debug applications level software running on the DOS operating system. PC PROBE can also be used by an experienced user to develop system level software such as device drivers, and new operating systems.

PC PROBE has capabilities which are not found in software only debugging tools. These capabilities are listed below:

1. PC PROBE can set breakpoints on reading, writing, or fetching memory. It can also breakpoint on input/output operations. These breakpoints can be on both address and data fields. The breakpoints can also happen on ranges of addresses.
2. PC PROBE can show you a real time trace of program execution. While your program is running, PC PROBE is saving the execution of the processor in real time memory on the PROBE card (1024 cycles are provided).
3. The software which operates PC PROBE is hidden and write protected on the PC PROBE card so that it cannot be corrupted by your undebugged program.
4. An on-board 128k memory provides room to store a hidden and write protected symbol table.
5. An RS232 port on the PC PROBE allows PC PROBE commands to be accepted from an external console without consuming the Com1 or Com2 ports.

ORGANIZATION OF THIS MANUAL

CHAPTER 1 contains the PROBE installation procedures.

CHAPTER 2 describes how to start the PROBE software. It also describes the PROBE menu driven human interface.

CHAPTER 3 describes the compiler and linker controls that you will need to use when you generate your code so that symbolic or source level debugging information can be passed to PROBE.

CHAPTER 4 presents an overview for the first time user on how to use the PROBE commands.

CHAPTER 5 contains a sample debugging session which also resides on the PROBE diskettes. This chapter also covers the more advanced debugging techniques such as debugging device drivers, interrupt routines, quit and stay resident routines, and boot loaders.

CHAPTER 6 is the Command Reference section which contains definitions and examples for each PROBE command.

APPENDICES contain additional information about the PC PROBE. Also included is a useful group of technical reports regarding common issues that arise during debugging.

INDEX indicates where to look for information based on key words or concepts.

CHAPTER 1
INSTALLING THE PC PROBE

INTRODUCTION

This chapter contains the set up and installation procedures for the PC PROBE. There are four major steps in the installation process.

1. Unpacking the PC PROBE
2. Setting the base address of the PC PROBE memory
3. Installing the PC PROBE
4. Running the PC PROBE diagnostics.

UNPACKING THE PC PROBE

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

1. PC PROBE printed circuit card and attached umbilical cable
2. PC PROBE floppy disks
3. PC PROBE STOP/RESET switch box
4. PC PROBE 8088 extraction tool
5. Plastic card guide
6. Spare socket.

SETTING THE BASE ADDRESS OF PC PROBE MEMORY

The PC PROBE contains 128k bytes of RAM memory which is used to store the PC PROBE software. This memory is write protected and will prevent the PROBE software from being modified during a debug session from erroneous action by your program. You can select the base address of the PC PROBE memory for any 64k byte boundary via the on-board jumpers. The on-board jumpers are shown in Figure 1-1, and match the physical layout of the board. The default jumper selections as shipped from the factory are also shown in Figure 1-1. Only change the base address if the default memory address conflicts with address space which is used by other boards plugged into your system. The default jumper setting is at address D0000 Hex. The base address of the PC PROBE is established by setting the jumper selections according to Table 1-1. The jumpers are set either left or right on the printed circuit card as shown.

Table 1-1. Memory Address Selection

A19	A18	A17	A16	FROM	TO
R	R	L	R	20000	3FFFF
R	R	L	L	30000	4FFFF
R	L	R	R	40000	5FFFF
R	L	R	L	50000	6FFFF
R	L	L	R	60000	7FFFF
R	L	L	L	70000	8FFFF
L	R	R	R	80000	9FFFF
L	R	R	L	90000	AFFFF
L	L	R	R	C0000	DFFFF
L	L	R	L	D0000	EFFFF*

* Factory default setting

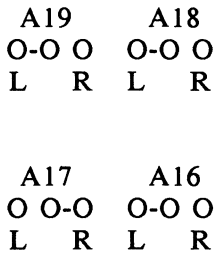


Figure 1-1. Default Jumper Settings

MEMORY CONFLICTS

When PROBE is installed in your system and the PROBE software is loaded, the memory map will look as follows:

Free memory or
other system resources

128k PROBE memory
PC PROBE base address

Free memory or
other system resources

DOS

PROBE VECTORS
(See appendix F)

If PC PROBE is jumpered for a base address of C0000, then it will conflict with the devices listed in Table 1-2. In general, high resolution graphics boards and network boards will conflict with PROBE if PROBE is jumpered at C0000.

Table 1-2. Hardware Conflicts at C0000

VENDOR	DEVICE
IBM	XT DISK CONTROLLER
IBM	EGA BOARD
IBM	NETWORK BOARD
TECMAR	TAPE BACKUP BOARD
TALLGRASS	TAPE BACKUP BOARD
CORVUS	HARD DISK CONTROLLER

If PC PROBE is jumpered for a base address of 80000, it will conflict with the system memory between 512k and 640k if this memory is present and selected.

There may be conflicts between PC PROBE and other software utilities executing in the PC. Some known examples of these conflicts are device drivers and quit and stay resident programs that use memory between 8000:0 and 9FFF:F. Other types of memory resident routines may want steal system vectors. Table 1-3 lists some of these conflicts and indicates what action should be taken.

Table 1-3. Software Conflicts

TYPE OF CONFLICT	ACTION
Software driver uses 80000 to 9FFFF	Put PROBE at another address.
Memory resident routines which take vectors	Load PROBE and do a Q R before loading memory resident routines which take vectors.

PROBE.CFG FILE

The jumper setting of the PROBE board should match the address contained in the file PROBE.CFG (configuration file). If you change the jumper setting from the factory default setting, then you should modify the address in the PROBE.CFG file with a text editor. Appendix C contains more information on the contents of this configuration file. If no configuration file is found by PC PROBE software, the default address of D0000 will be assumed.

INSTALLING THE PC PROBE

To install the PC PROBE follow these steps:

1. Disconnect power and remove the top cover of your PC.
2. Using the extraction tool provided, slip the tool under the 8088 processor and pry up gently to remove the processor from the socket (Figure 1-2). Be careful not to bend the 8088 pins.

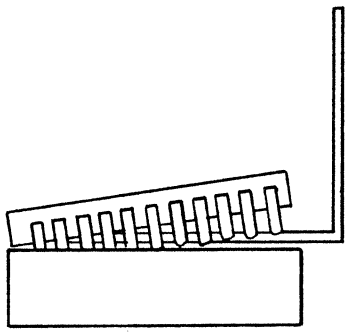


Figure 1-2. Removing the 8088

3. Insert the 8088 into the socket provided on the PROBE plug (Figure 1-3). Ensure that PIN 1 of the 8088 matches the red mark on the PROBE plug. If this is not done, damage will occur to the 8088 when power is applied.

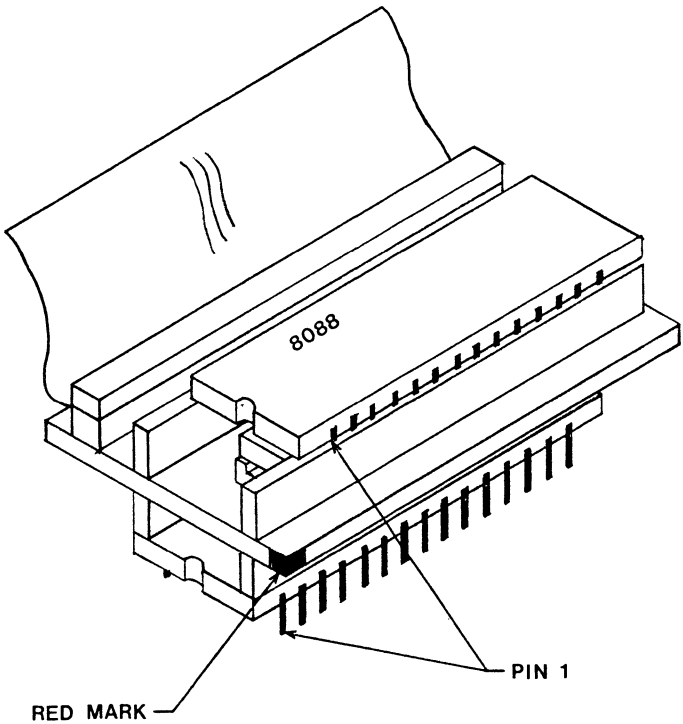


Figure 1-3. PROBE Plug

4. Insert the card guide supplied with the PC PROBE in the position you want to plug the PC PROBE into (if there not a card guide there already).
5. Insert the PROBE printed circuit card into the PC chassis. See Figure 1-4.

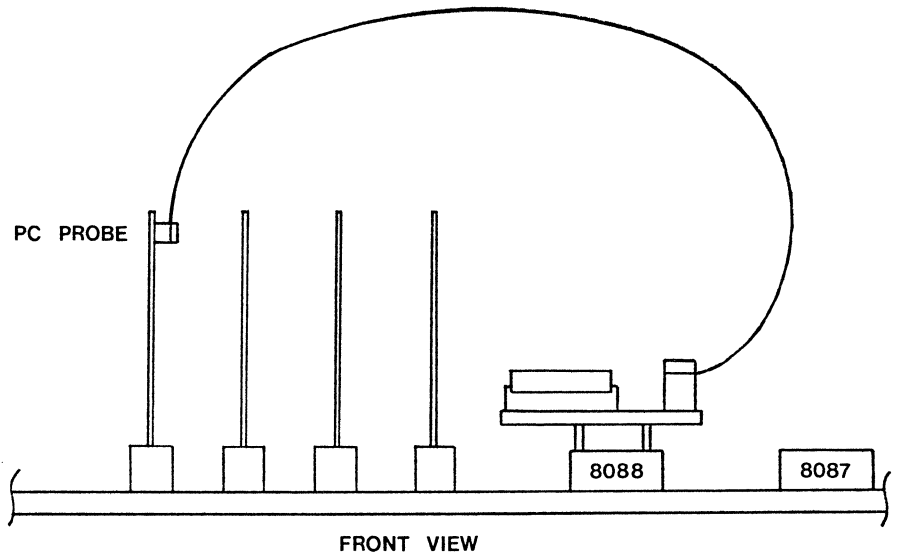


Figure 1-4. Installed PC PROBE Board

6. Insert the PROBE plug into the vacant 8088 socket. Ensure that the red mark on the PROBE plug is on the same end of the 8088 socket as pin 1 of the 8088. Pin 1 of the 8088 is normally indicated by a small niche in the 8088 socket. For an IBM PC this results in placing the red mark on the PROBE plug toward the back of the PC. In a COMPAQ, place the red mark towards the back of the COMPAQ. If the red mark of the PROBE plug is not on the same end as pin 1 of the 8088 socket, damage to the PROBE plug may result when power is applied.

7. Set SWITCH #1 POSITION #2 on the PC system board to "On" if no 8087 is present in the system. The STOP button will not work if this is not done.
8. Plug the external STOP/RESET switch box cable into the PC PROBE STOP/RESET connector at the rear of the PC.
9. Replace the top cover and connect power to the PC.
10. If an external CRT is to be connected to PC PROBE, see Appendix C.
11. Power up the PC. If the PC does not power up, then check the following:

    a. Check for possible conflicts between PROBE memory and other system memory. This information can be found in this chapter in the section called "MEMORY CONFLICTS."
    b. Verify that the PROBE plug is not offset in the 8088 socket.
    c. Verify that the 8088 notch and the PROBE plug notch match and face the rear of the PC chassis (if this is an IBM PC).
    d. Check for broken off or misaligned pins on the 8088 or the PROBE plug socket. The PROBE comes equipped with an extra socket on the PROBE plug to protect it from damage. In the event a pin breaks off from this socket, remove and replace it with another socket (AUGAT 540-AG70D or equivalent)
    e. Verify that the PROBE is plugged into the 8088 socket which is next to the 8087 socket.
    f. Verify that the PROBE printed circuit card is firmly seated in the PC chassis.
    g. Verify that your system is compatible with the PC PROBE. See Appendix B.

    If the PC still does not power up, then contact ATRON technical support for assistance.

RUNNING PC PROBE DIAGNOSTICS

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

PROBEDIA

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

```
PC PROBE Diagnostics Version 1.0
(C)Copyright Atron Corp. xxxx

Enter board base address
{1,2,3,4,5,6,7,8,9,C,[D]}?
```

The choices are:

2 for 20000 - 3FFFF
3 for 30000 - 4FFFF
4 for 40000 - 5FFFF
5 for 50000 - 6FFFF
6 for 60000 - 7FFFF
7 for 70000 - 8FFFF
8 for 80000 - 9FFFF
9 for 90000 - AFFFF
C for C0000 - DFFFF
D for D0000 - EFFFF *factory default setting

The base address of the board must be jumpered to match the selected choice to perform the diagnostics. After you enter the address, the diagnostics start executing, and the following messages are displayed on the screen.

```
Board base address is xxxxx
Short RAM test started.
Short RAM test finished.
8741 Function test started.
  8741 Version is 03
8741 Function test finished.
Break logic test 1 started.
Break logic test 1 finished.
Trace RAM test started.
Trace RAM test finished.
Long RAM test started.
Long RAM test finished.
Hit the STOP button.
```

At the end of the diagnostics be sure to press the reset button to verify that it works.

If any of the tests fail, messages will be printed giving more detail on the nature of the failure. The PC PROBE is ready for operation only if it has passed all diagnostics. A failure in a diagnostic may mean a logic failure in the PC PROBE hardware. However, it may also mean there is a conflict or connection problem between the PC PROBE and the PC or other boards which are plugged into the PC. The following are some suggestions to try for different failures.

WHAT TO DO WHEN A PROBEDIA TEST FAILS

Bank Select Test Fails

Verify that the PC 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 PROBE, try removing all boards which are not part of a standard PC system. A standard PC system consists of the motherboard, monochrome board and/or color graphics adapter board, and disk controller board. If the PROBE is jumpered for 80000, temporarily remove or rename your CONFIG.SYS and AUTOEXEC.BAT files. Sometimes certain device drivers use memory in the 80000-9FFFF range. After you have the PC PROBE working satisfactorily, replace the boards one by one to see if there is a failure. Replace the lines of your CONFIG.SYS and AUTOEXEC.BAT files one by one.

STOP Button Fails

Note that if the STOP BUTTON diagnostic fails, it may be because SWITCH 1 POSITION #2 on the PC is not set to "ON". Also ensure that you press the button when requested.

If any other diagnostic fails or you cannot get the PC PROBE to work, contact ATRON technical support (408) 741-5900.

CHAPTER 2
GETTING STARTED

INTRODUCTION

This chapter tells you how to start the PC PROBE. It also describes the operation of the PC PROBE user interface including information on commands, error messages, choosing a console, and how to redirect list devices. This chapter also includes information about the files on your disks and a summary of the different versions of PC PROBE.

HOW TO START PROBE

To start the PC PROBE, enter:

PROBE [/path spec] [\]

To start the PC SOURCE PROBE, enter:

SOURCE [/path spec] [\]

[/path spec] is the path for the additional PROBE files required to run the selected PROBE product. If **[/path spec]** is not included in the command line and these files cannot be found in the current system directory, then PROBE queries you for the current path.

EXAMPLE: This command loads the PC PROBE software from directory ATRON on drive A even if this is not the current drive and directory.

A:\ATRON\PROBE /A:\ATRON

If you want to walk through an example, go to Chapter 5 at this point. Otherwise, keep reading for more background information.

MORE ON PROBE.CFG

The file PROBE.CFG contains configuration information which will be used by PC PROBE. The details on each configuration parameter are given in Appendix C. The default configuration parameters in this file are:

1. PROBE Memory base address: D0000
2. External console cursor motion: Set for VT100 terminal
3. Do not overflow the symbol table into system memory.
4. Do not use system memory for screen switching with a color monitor.

These may need to be changed if:

1. A different base address is selected.
2. An external console is to be used with PROBE.
3. Symbol table overflow requires allocating memory from the system memory space.
4. The screen swapping command will be used during debug, and a color monitor is attached to the PC.

INITIALIZATION MACRO

When the PROBE software is loaded, the [path spec] is searched for the INIT.MAC file. If no [path spec] was specified, the current default directory is searched for the INIT.MAC file. If found, this file is loaded as a macro automatically by PROBE. If the file contains a macro named INITIALIZE, that macro is executed automatically. It is useful to create this custom file and macro to set up PROBE completely for a debugging session. An example initialization macro is given below. The user can create the INIT.MAC file and put the text exactly as shown below into it.

```
MAC INITIALIZE =
LOA M PROBE.MAC
EM INIT FTOCNEW
END
```

This macro loads the additional macros from the PROBE.MAC file then executes the macro called INIT. The macro INIT has a parameter called FTOCNEW which is the sample file to be loaded and debugged. See Chapters 4 and 6 for more information on macros.

MENU WINDOWS

The user interface to all PROBE products is designed to minimize the need to refer to this manual. (Although reading it once is highly recommended.) A menu of commands is provided on the bottom two lines of the screen. The MORe command switches between the two alternate menus. Example menus are shown below:

MENU 1

```
Asign BP BYt COMpar CONsol DElete DIr DMa ECho EDit EMacro EVal FIl
FLag Go IF INIt INTrupt LIst LOAd LOGic LOOp MAcro MEnu MODule MORe
```

MENU 2

```
MOVe NEst NOBreak NOVerify POrt PRint PTr Quit Register SAve SCreen
SEarch SSource STep SYmbol Trace Unassemble WIndow WOrd MORe
```

When you type a command, the first unique characters (those shown in capital letters) are all that are needed to specify the command. When these first unique characters are recognized by PROBE as a unique command, the syntax for that command appears in the menu window. Since fields following the unique command name can be symbolic expressions, a space is needed between the command name and the first parameter to follow it. The menu window can be turned off and on with the MEnu command.

ENTERING COMMANDS

In order to simplify command line editing and avoid learning a new editor, the DOS command line editing has been implemented in PROBE. A description of the DOS editing keys is described as follows:

F1 or -->	Copy and display one character from retained line.
F2	Copy all characters up to specified character.
F3	Copy all remaining characters from retained line.
F4	Skip all characters to specified character of retained line.
F5	Accept edited line for more editing as retained line.
Ins	Toggle - insert characters.
Del	Skip one character in retained line.
Esc	Cancel current line.

Commands which dump data to the screen can be terminated with the Ctrl Break (Ctrl C) key or the STOP button on the external switch box. The screen dump will pause if Ctrl S is typed.

ERROR MESSAGES

When a command or command parameter is not recognized by PROBE, an error message is printed to indicate the problem. Appendix A contains a summary of these error messages.

CONSOLES

When PROBE is first started, command entry is done through the PC keyboard. However, you have several choices for console IO. The simplest choice creates two virtual screens and isolates the PROBE screen from the application screen. This is done with the SCreen switching command.

FORMAT: **SC S**

If you have two video controller boards in your PC, you can move the PROBE display to the screen other than the one which is the current default. This is done with the CONsole command.

FORMAT: **CON Other**

A third choice is to switch to an entirely separate CRT which you connect to the RS232 port of the PROBE. This is also done with the CONsole command.

FORMAT: **CON Remote**

The description on how to connect a remote console and set up its configuration parameters is shown in Appendix C. The keys for command line editing are different for a remote console keyboard, as it may not be a PC. More information on the editing keys is also given in Appendix C.

For more details on the other options for these commands and connections, refer to the specific command in Chapter 6, COMMAND REFERENCE.

LIST DEVICES

You may want to save the results of a debugging session. The LIst command directs a copy of the information sent to the console to a list device such as LPT1 or a filename.

FORMAT: **LI listdevice**

For more details on the other options for this command, refer to the LIst command in Chapter 6, COMMAND REFERENCE.

FILES ON YOUR PC 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 E for each version of PC PROBE software. Only those used for "RUNNING" are required for the actual execution of PROBE software.

VERSIONS OF PC PROBE SOFTWARE

This manual describes the standard versions of PC PROBE and PC SOURCE PROBE software. Other versions of these software products are available which are optimized for specific applications. These other versions are summarized as follows:

/PL VERSIONS

These versions support the overlay manager of the PLINK86 linker. The available products are:

PC PROBE/PL
PC SOURCE PROBE/PL

The commands which are affected by the /PL versions are discussed in Appendix J.

/87 VERSIONS

These versions add full support for the 8087 numeric data processor to PROBE. Memory and 8087 registers can be displayed and changed via all floating point data types.

PC PROBE/87
PC SOURCE PROBE/87

The commands which are affected by these versions are the Numeric REGisters, Numeric Flag, FLOat, and Trace commands. See each command for more information.

CHAPTER 3
GENERATING AND USING SYMBOLS

INTRODUCTION

Before debugging your program, the steps of assembling, compiling, and linking must have already been completed. This chapter discusses important points regarding the use of assemblers, compilers, and linkers which are compatible with PROBE. It describes how to generate symbolic debugging information and pass this information to the PROBE. Also included is a summary of the commands which let you load, use, redefine, or delete these symbols.

GENERATING SYMBOLIC DEBUGGING INFORMATION

PC 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 PROBE from the compiler using controls which are discussed next in this chapter. This symbolic information consists of public variables, public procedures, functions, subroutines, modulenames, and high level language line numbers. Using symbols greatly simplifies the debugging process. In addition, if the PC SOURCE PROBE version of the software is running on PC PROBE, then source level debugging can be achieved.

To maximize the availability of symbols to be used during debugging, it is suggested that the following approach be used during program development. When debugging individual program modules, maximize the number of variables declared PUBLIC or statically allocated since this will pass the maximum number of symbols to the symbol table. Then, when the module has been debugged, remove extraneous PUBLICS.

The sections which follow describe the things that need to be considered when using PROBE with C, Pascal, and Assembly language. You may want to go directly to the section which applies to you.

USING C WITH PROBE

Symbolic debugging records can be passed from the C compiler through the linker to the .MAP file if the appropriate compiler and linker controls are specified. The available symbolic debugging information consists of linenumbers and global symbols. Currently, no local records (i.e. stack based variables) are passed from these compilers.

C SOURCE DEBUGGING

In order to do source level debugging with the PC SOURCE PROBE, a modulename must be associated with the line numbers for that module. A module is a single unit of compilation and has a name called the modulename. The modulename is matched to the appropriate source file using the ASIgn command in the SOURCE PROBE. The following examples show how the modules are named for various versions of the compilers and linkers.

I) Lattice C Version 2.14
DOS Link Version 2.20 (or 2.00)

A) lc1 ... -d small code; small data
lc2 ...
link ... /m /l

The modules are all named PROG as a default by the compiler. During the loading of the symbol table, PROBE finds that all of the modulenames are the same. PROBE then looks for a high line number to low line number transition. When PROBE finds this, it automatically changes the modulenames which are in the PROBE symbol table to PROG, PROG1, PROG2... If the SOURCE PROBE is going to be used, the appropriate source files must be assigned to the modulenames in the order they were linked as shown below. The ASIgn command is executed while in SOURCE PROBE.

```
ASI PROG firstsourcefile
ASI PROG1 secondsourcefile
ASI PROG2 thirdsourcefile
```

B) lc1 ... -d -mD small code; large data
lc2 ...
link ... /m /l

The modules are all named CODE as a default by the compiler. During the loading of the symbol table, PROBE finds all of the modulenames are the same. PROBE then looks for a high line number to low line number transition. When PROBE finds this, it automatically changes the modulenames which are in the PROBE symbol table to CODE, CODE1, CODE2... If the SOURCE PROBE is going to be used, then the appropriate source files must be assigned to the modulenames in the order they were linked as shown below. The ASIgn command is executed while in SOURCE PROBE.

```
ASI CODE firstsourcefile
ASI CODE1 secondsourcefile
ASI CODE2 thirdsourcefile
```

C) lc1 ... -d -mP large code; small data
lc2 ... -smodulename
link ... /m /l

The modules are named with the -smodulename control in the compiler. If the SOURCE PROBE is going to be used the appropriate source files must be assigned to the modulenames in the order they were linked as shown below. The ASIgn command is executed while in SOURCE PROBE.

```
ASI firstmodulename firstsourcefile
ASI secondmodulename secondsourcefile
ASI thirdmodulename thirdsourcefile
```

D) lc1 ... -d -mL large code; large data
lc2 ... -smodulename
link ... /m /l
Modulenames are the same as C above.

II) Lattice C Version 2.14
DOS Link Version 2.30 (DOS 3.1)

A) lc1 ... -d small code; small data
lc2 ...
link ... /m /l
Modules have the same name as their respective object files.

B) lc1 ... -d -mD small code; large data
lc2 ...
link ... /m /l
Modules have the same name as their respective object files.

C) lc1 ... -d -mP large code; small data
lc2 ...
link ... /m /l
Modules have the same name as their respective object files.

D) lc1 ... -d -mL large code; large data
lc2 ...
link ... /m /l
Modules have the same name as their respective object files.

III) MicroSoft C compiler Version 3.00
MicroSoft Linker Version 3.01 (shipped with compiler)
(Note: /map and /linenumbers must be spelled out below)

A) msc ... /Zd /Od /AS small code; small data
link ... /map /linenumbers
Modules have the same name as their respective object files.

B) msc ... /Zd /Od /AM large code; small data
link ... /map /linenumbers
Modules have the same name as their respective object files.

C) msc ... /Zd /Od /AL large code; large data
link ... /map /linenumbers
Modules have the same name as their respective object files.

IV) Computer Innovations C compiler Version 2.30
Dos Linker Version 3.01

A) CC filename.c -x2 ...
link ... /map /linenumbers
Modules have the same name as their respective object files.

SINGLE STEPPING PROGRAMS BY C SOURCE STATEMENTS

This section only applies to the PC SOURCE PROBE when it is used with the Lattice C compiler.

1. In order to get line numbers that correspond to "}" statements you must put a ';' after the '}'. If this is not done, no line number record is output by the compiler for this source line. In other words, you must have:

```
while (expression) {
    statements;
    if (expression) {
        statements;
    };
};
```

2. Also, functions that return by just "falling out the bottom" should include a return statement before the final '}' for the same reason.

```
return;}
```

3. For the last case in a switch statement, do not include the final "break;"

```
switch (    ) {
case 0:  statements; break;
      case 1:  statements; break;
      .
      .
      case n:  statements; (no break here)
      };
```

Note that none of these recommendations produce additional code in the .exe file. They each just produce more line number records from the compiler.

GENERATING A C LIST FILE WITH LINE NUMBERS

This utility is provided as a convenience. It is not necessary for use by PROBE software. The CLIST program found on the PROBE diskette accepts an input "C" source file and produces a listing file. It also expands tabs into spaces from the source file so that the file may be printed on any line printer.

FORMAT: **CLIST [sourcefile [,destinationfile [,spaces per tab]]] [options]**

If the files are not listed, then the user is prompted for the file names. Source lines are transferred to the **destinationfile** in the format specified later. All tabs are expanded to spaces with tab stops every specified number of columns. The two **options** which can be specified are i and c:

-i<include drive>

I specifies the drive to be searched for all include files. example-ia (no intervening spaces)

-c

C specifies that comments do not nest. An */ ends all comments currently in effect, no matter how many /* have occurred.

This is the LISTING FORMAT which is produced:

C ***** 00000. xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

C — Is a comment indicator. The 'C' is placed in this column if the first character in the source line is considered to be inside a comment.

***** — Is the include nesting level from the include files which are currently being used. Each include file which includes another file will add one more * to this field. There are a maximum of 5 *.

00000 — Is the line number of the line in the current file. Each <CR> in the file increments the line count. The line count starts at 1 in each file.

xxxxx — Is the line of source code.

EXAMPLES: Produce a listing in ftocm.lst with tabs set every 8 spaces. Include files are on the default drive and comments nest.
clist ftocm.c, ftocm.lst, 8

Produce a listing directly to the line printer with tabs set every 4 spaces. Include files are on the default drive and comments nest.
clist ftocm.c, lpt1:, 4

Produce a listing directly to the line printer with tabs set every 4 spaces. In addition, all include files exist on drive C, and comments do not nest.
clist ftocm.c, lpt1:, 4 -ic -c

You are prompted for the destination file and for the number of spaces per tab. All include files exist on drive A, and comments nest.
clist ftocm.c -ia

USING PASCAL WITH PROBE

The IBM or Microsoft Pascal compiler will always generate symbolic information into the object files and no special compiler controls are necessary. The modulenames are derived from one of the following in the Pascal source code:

Program statement
Module statement
Implementation statement

The symbols from the Pascal runtime library generally are of no use and consume too much symbol table space. Therefore, it is recommended that the STRIP utility be used to eliminate these symbols. The /l/m control should be used in the DOS linker to include symbols and linenumbers in the MAP file. The Pascal compiler run time libraries do a special move of the data for some segments at the beginning of the program. See Appendix G for more information.

USING ASSEMBLY LANGUAGE WITH PROBE

In the Macro Assembler, public symbols are available as symbols. The /m control in the linker must be used to pass these symbols to the Map file.

USING THE DOS LINKER WITH PROBE

The DOS linker will pass symbolic information generated by the compiler or assembler from the .obj file to a .MAP file if the /l/m linker controls are applied. See the DOS LINK command for more information.

USING PLINK86 WITH PROBE

If you are using the PLINK86 linker, refer to Appendix J. Special versions of PROBE software are available to support this linker. The standard versions of PROBE software can also be used by using the STRIPPE utility to convert the symbol table MAP file. See Appendix J for details.

SYMBOLIC DEBUGGING

LOADING SYMBOL TABLE FILES

Once you have generated the symbol information and you are under control of the PROBE software, you can load the symbol table into the write protected 1 megabyte address space of the PROBE. This is done with the following command:

LOA S symboltablefile {options}

EXAMPLE: To load the MAP file for the program FTOCNEW.EXE:
LOA S FTOCNEW.MAP

This is the simplest form of the load symbol table command. See Chapter 6, COMMAND REFERENCE, for the many options which can occur for the LOAd symbols command.

Another way to load symbols into the PROBE symbol table is to specifically define them. This can be done with the SYmbol command:

SY .symbolname = symbolvalue

EXAMPLE: To define a symbol equal to the current CS:IP:
SY .START = CS:IP

The symbols in the symbol table command can be displayed simply by typing SY <enter>.

SYMBOL TABLE OVERFLOW

When the symbol table for the program is too large to be loaded into the allocated PROBE memory, there are two choices:

1. Use the SELect command to load only line numbers from specified modules when loading the symbol table. See the SELECT command in Chapter 6.
2. Use the symbol stripping utility to strip out symbols from the MAP file before loading the program.

SELECTIVELY LOADING SYMBOLS

If you know the symbol table is too large to be loaded into PROBE, then you can eliminate linenumber records for all modules except for those you specifically select. This is done with the SELect command:

SEL [..modulename1, ..modulename2,...]

EXAMPLE: To load all line number records for the two modules shown here:
SEL FTOCM_CODE, FTOCIO_CODE

If you had a symbol table overflow, and you want to try the load again with selected modules, be sure to delete all of the symbols currently in the PROBE symbol table. This is done with the DElete Symbols ALL command:

DE S ALL

Or you can simply delete a single symbol using the following format:

DE S .symbolname

STRIPPING SYMBOLS FROM THE MAP FILE

The second alternative to reducing the number of symbols loaded is to prestrip them from the MAP file. The STRIP utility on the PROBE disk will strip symbols from the .MAP file generated by the linker.

FORMATS: **strip inputfile.map outputfile.mp1, datafile**
or
STRIP

The strip utility is executed after the linker generates a link map for the user's program. **Inputfile.map** is your MAP file generated by the linker. **Outputfile.mp1** is the new MAP file named by you with the requested symbols and line numbers stripped out. **Datafile** is a file containing the list of symbols and modules to be stripped from inputfile.map. Once created, datafiles are not normally changed until the program symbols overflow the symbol table space. You may also want to change the datafile if you change the version of your

compiler. To create a datafile start with a .MAP file, and then use your favorite text editor to delete the following items.

1. Remove the symbols from this .MAP file which are to be included in the final symbol table (i.e. outputfile.mpl). The remaining symbols are the ones to be stripped from new .MAP files created with the linker.
2. Delete all linenumbers.
3. Delete segment map.
4. Delete group map.
5. Delete "Address Publics by Line".
6. Delete the modulename statements which are in the .MAP file as: "Linenumbers for module modulename" for all linenumbers which are to be included in the final symbol table.

What remains is a file of symbols and "Linenumbers for module modulename" statements which will be stripped from the inputfile.map to produce the outputfile.mpl.

Sample datafiles are included on the PC PROBE disk for stripping symbols and module line numbers for several standard high level language libraries. The symbols found in these libraries are of little use during debugging. If the STRIP utility is invoked without specifying additional files, the utility will prompt for the file names before execution.

EXAMPLE: strip userprog.map, b:userprog.mpl, strip.pas

The format for symbols in the MAP file is given in Appendix G, Symbol Table Map Format.

USING SYMBOLS IN COMMANDS

A symbol can be used any place an expression or value is expected in PROBE commands. A symbol can be in either of two forms:

.symbolname
or
..modulename#linenumber

Procedure names and function names are treated the same as symbol names. A single dot in front of a symbolname distinguishes it from a hex character. A double dot in front of a module name distinguishes it from a symbolname. If no ..modulename is specified when you use a linenumber as a symbol, then the current default modulename is used. After the symbol table is first loaded, the default modulename is simply the first loaded module. You can specify the default modulename with the following MODule command:

MOD modulename

EXAMPLES: This is an example Go command which shows both symbolnames and modulenames used. START is a symbol which represents the starting address in the Go command. STOP1, STOP2, and MAIN#38 are symbols which represent instruction execution breakpoints.
G =.START, .STOP1, .STOP2, ..MAIN#38

This example displays the bytes between TEMP and TEMP+10:
BYTE .TEMP .TEMP+10

SYMBOL TABLE MAP FORMATS

A description of the symbol table formats which are found in the MAP file are described in Appendix G. If the executable code is linked by a non-DOS linker, a utility must be written by the user to convert the information produced by the user's linker to the standard DOS formats. The information in Appendix G will aid you in this process.

CHAPTER 4
USING PROBE COMMANDS

INTRODUCTION

This chapter presents an overview on using PROBE commands for the first time user. If you are familiar with PROBE, you may wish to skip this chapter and go directly to Chapter 6, COMMAND REFERENCE. This chapter describes the PROBE commands in the following functional groups:

DISPLAY AND CHANGE MEMORY
DISPLAY AND CHANGE IO
DISPLAY AND CHANGE REGISTERS AND FLAGS
BLOCK OPERATIONS ON MEMORY
ASSEMBLE AND UNASSEMBLE MEMORY
LOADING PROGRAMS AND SYMBOL TABLE
STARTING PROGRAM EXECUTION AND SETTING BREAKPOINTS
SINGLE STEP PROGRAM EXECUTION
REAL TIME TRACE
MACRO COMMANDS
EVALUATE EXPRESSIONS

The commands associated with each of these functional groups are briefly described in this chapter. The interaction of some commands in setting parameters for other commands is also described. In the discussion which follows, several common terms and definitions are used. If they are not self-obvious, definitions can be found in the first few pages of Chapter 6, COMMAND REFERENCE.

DISPLAY AND CHANGE MEMORY

BYTES, WORDS, POINTERS, and FLOATING POINT

You can display a block of memory in byte, word, pointer data, or floating point types with the following commands:

BY [range]
WO [range]
PT [range]
FLO [type] [range]*

Range specifies the area of memory to be displayed. There are two ways to express **range**:

startaddress endaddress
or
startaddress L length

If **range** is not specified in these commands, then a default block length of memory is displayed. The PgDn key will also display the next block of memory after one of these commands has been executed.

*NOTE: Only applies to /87 versions. If you have the /87 version of the PROBE software then you can display floating point numbers in memory in all of the data types shown in Table 4-1. An 8087 must be present in the system before any of the floating point commands will operate. The default for **[type]** is R in this command.

Table 4-1. 8087 Data Types

TYPE	DESCRIPTION	#BYTES
I	32 BIT INTEGER	4
L	LONG 64 BIT INTEGER	8
P	PACKED DECIMAL	10
S	SHORT 32 BIT REAL	4
R	REAL 64 BITS	8
T	TEMP REAL 80 BITS	10

There are two methods that can be used to change memory. The first method lets you specify a string of data which will be deposited in memory when you enter it. The following commands are used for this method:

BY start address = value [,] value [,] value...
WO start address = value [,] value [,] value...
PT start address = value [,] value [,] value...
FLO [type] start address = value [,] value [,] ...

In the second method, you get to look at each memory location before changing it. After the new data is entered, the next location is displayed and can be changed. Typing <enter> alone on a line ends the changes. The following commands are used for this method:

BY start address = <enter>
WO start address = <enter>
PT start address = <enter>
FLO [type] start address = <enter>

The following examples use these display and change memory commands.

EXAMPLES: This BYte command stores a list of bytes.
BY .START = 50,51,52

The next 3 bytes can be displayed can be displayed with the following command. Since no length was specified, PROBE uses the previously specified length of 3.
BY <enter>
3000:0003 40 41 42 *@AB*

This WOrd command stores a list of words but lets you look at each one before changing it.
WO .START =
3000:0000 0000 - 0001 <enter>
3000:0002 0001 - 0002 <enter>
3000:0004 0002 - 0003 <enter>
3000:0006 0004 - <enter>

EXAMPLES, continued

Registers can also provide the start address.
WO SS:SP = 0, 0, 0, 0
WO SS:SP+BP L 3

ASCII strings can be stored.
BY .START = 'THIS IS A LINE'

This Pointer command puts pointer values on the stack.
PT SS:SP = 3000:AEF0

MEMORY NOVERIFY CONDITION

When PROBE changes memory with commands like BYte, WOrd, and PTr, it does an automatic read after write verification to ensure that the change really occurred. Some peripheral devices, which are addressed as memory, will malfunction if a read after write occurs. The NOVerify command given below lets you suppress all read after write activity.

NOV Noread

DISPLAY AND CHANGE IO PORTS

The PC IO ports can be displayed and changed with the following POrt command.

PO [Word] portnumber

By including **Word** in this command you can read 16 bit rather than 8 bit ports. You can also write a byte or word value to a port with the command below.

PO [Word] portnumber = value

EXAMPLES: To display the port addressed by the symbol PORT5:

```
PO .PORT5
AE
```

To change the port:

```
PO .PORT5 = AA ;in hex
PO .PORT5 = 'A' ;in ASCII
PO .PORT5 = 10T ;in decimal
```

DISPLAY AND CHANGE REGISTERS AND FLAGS

You can display the registers and flags of the 8088 and 8087 with the following register and flag commands:

Reg [8088 registername]
FLA
NR [{S | T} (#)]
NF [C|S]

The first two commands display 8088 registers or flags. The second two commands display 8087 stack and tag registers or control and status flags. If no register is specified in the commands, then all registers are displayed. The register and flag names are shown in the tables below.

8088 REGISTERS

AX	CS	SS	DS	ES	AH	AL
BX	IP	SP	SI	DI	BH	BL
CX	BP			CH	CL	
DX	FL			DH	DL	

8087 REGISTERS

ST (0)	Tag 0
ST (1)	Tag 1
ST (2)	Tag 2
ST (3)	Tag 3
ST (4)	Tag 4
ST (5)	Tag 5
ST (6)	Tag 6
ST (7)	Tag 7

8088 FLAGS

FLAG NAME	SET	CLEAR
Overflow	O1	O0
Direction	D1	D0
Interrupt	I1	I0
Trap	T1	T0
Sign	S1	S0
Zero	Z1	Z0
Aux carry	A1	A0
Parity	P1	P0
Carry	C1	C0

8087 FLAGS

Control
Status
Instruction
Operand

The registers and flags can be changed to a specified value using the commands below. If a value is not specified, then the register or flag is displayed along with a prompt to let you change it.

R registername = [value]
FLA flagname = flagvalue
NR {S|T} (#) = [value]
NF {C|S} = [value]

EXAMPLES: To display all 8088 registers type:
R

```
AX= 1234    CS= 0000    SS= 1000    DS= 0011    ES= 0100
BX= 0104    IP= 1000    SP= 5000    SI= 0000    DI= 0000
CX= 0002                BP= 4000
DX= ABAB                FL= O0 D1 E1 S0 Z1 A1 P0 C0
```

To change the 8088 IP register to 2000:
R IP =
IP = 1000 - 2000 (the 2000 was entered by the user)

To put an ASCII value into the 8088 AL register:
R AL = 'Q'

To set 8087 tag register 4 to a 2:
NR t (4) =2

To display 8087 stack register 0:
NR s 0
ST (0)=-0.1859660090 E 16363

To display 8088 flags:
FLA
O0 D1 I0 T0 S0 Z1 A1 P1 C1

To set the 8088 interrupt flag:
FLA I = 1

To change the 8087 control word to 0FFF:
NF C = 0FFF

INITIALIZING REGISTERS AND FLAGS

When a .EXE file is loaded, the SS:SP, CS:IP, DS, and ES registers are set by the loader and operating system to initial conditions. It may be desirable to reinitialize the registers to these conditions when restarting program execution if a program or procedure has left the stack and registers in an indeterminate state. This can be done with the INItialize command with the format shown below. See Chapter 6, COMMAND REFERENCE, for a discussion on the restrictions for this command.

INI

BLOCK OPERATIONS ON MEMORY

PROBE provides you with a flexible set of commands to move and compare blocks of memory. In the MOVe command, the block of memory specified by the **range** is moved to a location starting at **destinationaddress.** The memory is moved on a byte by byte basis starting with the first address of **range.** With the PROBE COMpare command, the contents of the block of memory specified by **range** is compared to the same size block starting at the location specified by **destinationaddress.** The entire block is compared, and the mismatches are displayed.

MOV range destinationaddress
COM range destinationaddress

EXAMPLES: To compare the 100 locations starting at DS:200 to the locations at 1000:100:
COM 200 L 100T 1000:100

To compare the 257 locations starting at TEMP to NEWTEMP:
COM .TEMP .TEMP+100 .NEWTEMP

To compare the 32 locations starting at LOOP to 100H in the CS segment:
COM .LOOP L 20 CS:100

To move the 32 locations starting at LOOP to 100H in the CS segment:
MOV .LOOP L 20 CS:100

Memory can be filled with the FIll command. If the number of items in **list** is fewer than the number of bytes in **range**, then **list** is repeated until the end of the range is reached. If **list** contains more items than there are bytes in **range**, then the excess **list** items are ignored. With the SEArch command, the block of memory indicated by **range** is searched for the string indicated by **list**. All locations within **range** which match **list** are displayed.

FI range list
SEA range list

EXAMPLES: To fill the 33 locations starting at CS:FF00 with 100:
FI CS:FF00 FF20 100T

To fill the 257 locations starting at TEMP with the string "ZERO":
FI .TEMP .TEMP+100 "ZERO"

To search the stack for the string 'STACK':
SEA SS:SP-FF L FF 'STACK'

ASSEMBLE AND UNASSEMBLE MEMORY

The PROBE provides an on-line symbolic assembler and disassembler. You can start assembling instructions into memory using standard 8088 and 8087 mnemonics. Symbols from the PROBE symbol table can be used in place of absolute numbers. The Assemble command format is:

ASM [start address] <enter> <assy language stmt>
<enter> only to stop assembly

If **[start address]** is not specified, then the assembly starts where the most recent Assemble command ended. The description of your assembly language options is beyond the scope of this chapter and you should go to Chapter 6, COMMAND REFERENCE, for this information.

The Unassemble command displays memory as 8088 and 8087 assembly language instructions (or near assembly language when more clarity is needed in the disassembly). The format for the unassemble command is:

U range

The specified **range** of memory is unassembled to the next highest whole instruction. The PgDn key will unassemble the next screen full of instructions. The start of **range** must correspond to the first byte of an instruction or all of the unassembled instructions which follow may be incorrect. All indirect references to memory are specified as word or byte in the unassembly to simplify the understanding of the data type which is being addressed. All relative jumps and calls show the absolute offset rather than the relative offset for ease of determining the target address. The target addresses are also shown with a symbol comment at the end of the instruction. If a matching symbol does not exist, then the nearest previous symbol plus an offset is used. Data operands also show symbols as a comment if base or index registers are not involved.

LOADING PROGRAMS AND SYMBOL TABLE

Normally, the first thing you do after starting PROBE is load the program to be debugged. The command format below loads the applications program named by the **filespec** into PC memory. **[Optional parameters]** in the LOAd command are passed to the loaded program in its program prefix segment in the normal manner. *The LOAd command should be used to load the program before the LOAd S command is used to load the symbol table.*

LOA filespec [optional parameters]

After loading the program, the symbol table can be loaded to give you symbolic and source level debugging capability. If PROBE loads the program, then it automatically adjusts the symbol table to correspond to the loaded program. If PROBE did not load the program to be debugged as in the case of an installed device driver or boot loader, refer to the LOAD command in Chapter 6, and the sections in Chapter 5 called "Debugging a Boot Load Sequence" and "Debugging a Device Driver Which Installs Itself" for more information.

LOA Symbols filespec

If you cannot remember the name of the files you want to load, use the DIrectory command given below. Note that all of the standard DOS pathname and wildcard options are supported with the DIrectory command.

DI [filespec]

EXAMPLES: These commands load a standard .EXE file and pass a parameter called LPT1 to the program. Then a symbol table file is loaded.

```
LOA DUMP.EXE LPT1:
LOA S DUMP.MAP
```

To display all macro files in the current directory of drive B with a .MAC extension:

```
DI B:*.MAC
INIT.MAC
SOURCE.MAC
```

STARTING PROGRAM EXECUTION AND SETTING BREAKPOINTS

As with most debuggers, program execution with PROBE can start with the Go command. You have the option of specifying a start address in this command. If you do not, then the current CS:IP is used. In addition, you can set **breakpoints** in the command which starts program execution. The format of the Go command is:

G [=address,] breakpoint, breakpoint...

Breakpoints in the Go command are non-sticky. This means you have to specify them again when you issue another Go command. Another way of setting breakpoints is with the BP command. Breakpoints set by the BP command are sticky, that is, they are automatically inserted for each Go command. Sticky breakpoints are deactivated with the DElete B command. The format of the BreakPoint command is:

BP breaknumber = breakpoint

Breaknumber is from 1 to 8 and **breakpoint** matches the definitions shown next. Symbols, registers, etc. are interpreted as absolute values and are substituted into the breakpoint definition when program execution starts. In other words, BPs are evaluated when the Go command is executed. A total of 19 breakpoints may exist between sticky and non-sticky breakpoints. The format for breakpoint is:

[IO] address [verb] [DATA datavalue]

Address is a memory address for the breakpoint unless preceded by the key word **IO** which indicates that the address is for IO. If no segment register is specified for **address**, then the CS register value is assumed. **Address** may be a range of addresses by replacing a hex character specified in the address with the character x. This indicates a "don't care" condition for this hex character in the breakpoint. Range breakpoints cannot be used for execution breakpoints. **[verb]** is one of the following:

R for read
W for write
A for all (i.e read, write, or execute)
D1 for DMA transaction on channel #1
D2 for DMA transaction on channel #2
D3 for DMA transaction on channel #3

If **[verb]** is not included in the breakpoint, then break on instruction execution is assumed. The 8088 distinguishes execute from read. Read is used to read a data value. Execute is used to execute an instruction.

The key word **DATA** indicates that the data portion of the operation must also match **[datavalue]** to cause the break at address. The **data** is byte data. If byte data is specified in the breakpoint, and a word read operation occurs by the cpu, the other half of the word is ignored. If **DATA** is not included, then **datavalue** is ignored in the breakpoint. **Datavalue** may only be a byte quantity. Also, **datavalue** is not allowed for execution breakpoints.

Once a breakpoint has been defined, you can display its definition with:

BP [breaknumber]

EXAMPLES: This command starts at the current CS:IP, and stops at the procedure .FOO.
G .FOO

This execution breakpoint occurs at location CS:10.
BP 2 = 10

This breakpoint occurs when the instruction at location 1000:0000 is about to be executed.
BP 3 = 1000:000

EXAMPLES, continued

This breakpoint occurs when the memory location at TEST is read (but not necessarily executed).
BP 2 = .TEST R

This breakpoint occurs when the IO port represented by the symbol KEY is read.
BP 3 = IO .KEY R

This breakpoint occurs when the memory location TEMP is written.
BP 4 = .TEMP W

This breakpoint occurs when any data in the address range 10000 to 100FF is read.
BP 5 = 1000:00FF R

This breakpoint occurs when the value AF (hex) is written to location TEMP.
BP 6 = .TEMP W DATA AF

This breakpoint occurs when the variable at .TEMP is overwritten with a word data value of 77.
BP 7 = .TEMP W DATA 77

This breakpoint occurs when any of the cpu software vectors are read.
BP 8 = 0:3xx R

SINGLE STEP PROGRAM EXECUTION

PROBE lets you single step your program execution by 8088 assembly language instructions as well as by high level language source statements. While single stepping, a window can be defined to appear at the bottom of the display that lets you watch anything else in your program you want to see. The assembly language single step command has this format:

ST [=start address] [P] [O] [A]

Single stepping will start at the current CS:IP if you do not include **[=start address]**. The command starts by displaying the next several instructions. The cursor is positioned to the right of the instruction to be executed. Typing the enter key will cause the program to execute the displayed instruction.

There are two parameters which give you control of how the single step command operates. While you are stepping, typing **P** will treat a call procedure as a single step. Typing **O** will treat a software interrupt procedure as a single step. If the **P** or **O** option is used when the STep command is invoked, then the options apply globally to all procedures.

Typing PgDn will single step a screen full of instructions. Typing any other key will cause the program stepping to stop and will not execute the instruction to the left of the cursor. The current CS:IP location and instruction are displayed after each step. The contents of the operands for each instruction are displayed at each single step, so you don't have to exit the command to view the contents of the operands.

If the A option is specified, single stepping occurs automatically on the screen. The Ctrl S key lets you pause the stepping. Any other key bails you out.

EXAMPLE: To execute a single step starting at location 100:20 and continue until a non-enter key is typed:
-st

```
0671:0010 MOV   SI,WORD PTR [BP+0114] -- ,00E6
0671:0014 MOV   AL,BYTE PTR [SI]      -- ,8A
0671:0016 XOR   AH,AH
0671:0018 TEST  AX,AX
0671:001A JNZ   001F   ;..FTOCIO_CODE#57+0198 -will jump
0671:001C JMP   0117   ;..FTOCIO_CODE#57+0290
```

The other type of single stepping which you can do if you are operating the PC SOURCE PROBE is by high level language statement number. Chapter 3 describes the compiler controls you must use to generate the needed information for source level single stepping. The format for this command is:

SS [= start address] [A] [M modulename]

The single step operation is similar to the assembly language version as are the **[=start address]** and **[A]** parameters. If **[M modulename]** is specified in this command, only lines in **modulename** are stepped, and all other code executes at full speed. With this type of single stepping, you can use the keys below to move around in the file you are currently stepping through.

PG UP (^U)	display the previous page
PG DN (^D)	display the next page
<Ctrl> PG UP (^T)	display the top page of the file
<Ctrl> PG DN (^B)	display the bottom page of the file
HOME (^H)	reposition screen to the current CS:IP
<enter>	reposition screen to current CS:IP and take a single step
up arrow (^P)	no effect
down arrow (^L)	no effect
<any other>	exit step mode.

EXAMPLE: To start stepping the program at the source level from location .MAIN:
SS =.MAIN

WINDOWS

While you are single stepping, a window which displays anything you want can be displayed. To create the contents of this window, define a macro which contains PROBE commands and PRINT statements. (See the section entitled: PRINTING FROM WITHIN MACROS for more information.) Then simply assign the macro to the window with the command below. The window is automatically updated after each single step.

WI = macroname

REAL TIME TRACE

One of the most powerful features of PC PROBE is the real time trace. PROBE saves the program flow in real time as it occurs. The trace data (1024 cycles or about 500 instructions) provides detailed information on what the program was doing up until a breakpoint occurred. It shows you the sequence of assembly language instructions (and optionally the source code). For each instruction you see the following:

Address of the instruction
Opcodes and Operands
Data which the instruction moved in memory or IO
Data which was pushed and popped
Interrupt operation
DMA cycles on the PC backplane
Interrupt lines on the backplane or optional logic probes

The format for the Trace command is:

T [N]

The previous N instructions are displayed. If N is not specified then all trace data is displayed. There must be a space before the N. When the number of lines of trace data to be displayed is more than a screen full, the display will pause and the <enter> key will display the next page. Any other key will terminate the trace command. Trace data is only collected after a Go command.

EXAMPLES: This trace shows source code along with assembly language execution. The bold type below is not part of the trace display, but are notes which describe the information in this display.

```
-t
          OP
ADDR    CODE   OPERAND(S)
```

```
112. *c_temp = f_temp - 32;
0652D MOV AX,WORD PTR [BP+0008]      ;f_temp
    0ACE4 READ - SS - 14
    0ACE5 READ - SS - 00
06530 SUB AX,0020
06533 MOV SI,WORD PTR [BP+000A]
    6ACE6 READ - SS - D8
    6ACE7 READ - SS - 00
06536 MOV WORD PTR [SI],AX
    09CA8 WRITE - DS - F4
    09CA9 WRITE - DS - FF

113. *c_temp = *c_temp * 5;
06538 MOV AX,0005
0653B IMUL WORD PTR [SI]
      09CA8 READ - DS - F4
      09CA9 READ - DS - FF
0653D MOV WORD PTR [SI],AX
      09CA8 WRITE - DS - C4
      09CA9 WRITE - DS - FF

114. *c_temp = *c_temp / 9;
0653F MOV BX,0009
06542 CWD
06543 IDIV BX
06545 MOV WORD PTR [SI],AX
      09CA8 WRITE - DS - FA
      09CA9 WRITE - DS - FF
```

label which matches operand

source code

assembly language

data during bus cycle

segment register used

type of bus operation

address on bus

If PC PROBE is running, then the above display would not include the high level lines of source code. The high level lines of source code are included in the display only when running PC SOURCE PROBE.

In addition to the operation of the 8088, the Trace command can optionally show you what the DMA controller is doing. The DMA cycles are in the trace data. The display of the DMA cycles can be turned on or off in the trace command with the following command:

DMA ON/OFF

MACRO COMMANDS

In case the PROBE commands are not exactly what you want, you can use the MAcro command capability of PROBE to define your own custom commands. MAcro commands use the PROBE commands as primitives. Parameters can be passed to macro commands when you execute them. Parameters are then filled into the PROBE commands within the macro. The format for a macro command is given below.

```
MA macroname =
M- COMMAND[%parameternumber]
M- COMMAND[%parameternumber]
M-....
M- END
```

PROBE commands can be entered until the **END** is specified. Parameters with parameternumbers ranging from 0 to 9 may be passed to the macro. This is done by specifying **%parameternumber** in the **COMMAND** where the parameter is to be substituted. Parameters may be any ASCII string. Once defined, you can display your macros with:

MA macroname

If you do not specify a **macroname**, then all defined macros are displayed.

EXECUTING MACROS

When the macro is invoked, the parameters are passed to the macro. Each parameter is assigned a number starting on the left with 0 in the EM command. The **parameter** can be a string and is separated from the next **parameter** by a comma. The **parameters** are then substituted in the macro for the parameternumber specified in the macro definition. Now you can execute your macro commands with:

EM macroname parameter, parameter,...

EXAMPLES: This is an initialization macro called INIT which saves many user keystrokes when starting up a debugging session.

```
MAC INIT =
M-LOA %0.EXE
M-LOA S %0.MAP
M-R
M-U
M-END
```

This macro definition can be displayed by typing:
MAC INIT

This macro can now be used to start a debugging session by loading the program b:myprog.
EM INIT B:MYPROG

A much more exhaustive discussion of macros with more examples is given in Chapter 6, COMMAND REFERENCE.

While a macro executes, the commands which make up the macro are displayed. This can be suppressed or enabled with the following ECHO command:

EC ON/OFF

LOADING, SAVING, AND DELETING MACROS

Once you have defined your macros, you can save them in a file. Then you can reload the macros in subsequent debugging sessions. This is done using the following commands:

SA M filespec
LOA M filespec

If you want to delete a single macro or all macros you have loaded or defined, use one of the following commands:

DE M macroname
or
DE M ALL

PRINTING FROM WITHIN MACROS

One very useful command to put in a macro is the PRint command. The PRint command lets you create formatted screens which display information in the appropriate data type. It also lets you include labels and messages about the display. The format for the PRint command is:

PR {'string' | [type] expression},.....

The PRint command can be used much like a high-level language WRITE statement. It accepts strings in either single quotes (') or double quotes (") and echoes them to the console. It evaluates expressions which are not within quotes and prints their value on the console in one of several different data types. For a more detailed discussion of the operation of the PRint command, see Chapter 6, COMMAND REFERENCE.

CONDITIONAL MACRO EXECUTION

Macro commands can be defined which conditionally and/or repetitively execute the commands in the macro. This can be done by using LOOp and IF commands within the macro definitions. Putting a LOOp command within a macro looks like this:

```
-MA macroname =
M-[PROBE Commands]
M-LOO [count expression|While boolean expression]
M-[PROBE Commands]
M-ELO
M-[PROBE Commands]
M-END
```

The PROBE commands between **LOO** and **ELO** commands are executed repetitively as controlled by the loop expression. There are two types of loop expressions: count expressions and while boolean expressions.

Count expression. This is the number of times the loop should execute. If no number is specified in this form of the LOOp command, then a loop forever function is implemented.

While boolean expression. The commands within the loop are executed repetitively as long as the boolean expression is true. See the beginning of Chapter 6, COMMAND REFERENCE, for the definition of a boolean expression.

EXAMPLES: This macro is defined to execute until a procedure is called with a certain parameter.

```
MA L2 =
M- LOO While @W.TEMP <> %0
M- G .PROCEDURE_START
M- ELO
M- END
```

The following command executes the program until PROCEDURE_START is executed, and the word at .TEMP is equal to 100.

```
EM L2 100T
```

The other type of conditional macro execution is the **IF** condition. The IF command can be used inside a macro to allow a series of commands to be executed conditionally. The end of the IF command is specified by **EIF**. The **ELSE** portion of the IF command is optional. The format for a macro containing an IF command is as follows:

```
MA macroname =
M- [PROBE COMMANDS]
M- IF boolean expression
M- [PROBE COMMANDS]
M- [ELSE
M- [PROBE COMMANDS]]
M- EIF
M- END
```

See Chapter 6, COMMAND REFERENCE, for more examples of the IF command.

EVALUATE EXPRESSIONS

PROBE provides an on line hex calculator, base converter, and expression processor. There are two forms for the EValuate command:

```
EV expression
EV pointer
```

In the first form of this command **expression** is evaluated and displayed in the bases shown below. A non-printing ASCII character is shown as a period. This command serves as a hex calculator and base converter.

HEX DECIMAL INTEGER BINARY ASCII

In the second form of this command, pointer is evaluated to a 5 hex character number. This command serves to calculate real addresses in the same way as the 8088.

EXAMPLES: The following expressions are evaluated:

```
EV ss:sp+5
30005

EV (((AX-5)*2)/10)
0200H 512T +512T 1000000000Y '.'

EV 'a'
0061H 97T +97T 1100001Y 'a'

EV AAAA:FFFF
BAA9FH
```

CHAPTER 5
DEBUGGING APPLICATIONS

INTRODUCTION

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

A SAMPLE DEBUGGING SESSION
ADVANCED DEBUGGING TECHNIQUES.

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

A SAMPLE DEBUGGING SESSION

The commands are listed alphabetically in Chapter 6, 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 6, COMMAND REFERENCE, for more information. The program to be debugged is a C program and the source, object, and map files for the program are included on your disk. So, you can actually try the example in real time. The example is not a trivial one and demonstrates debugging a program which is interactive with the operating system and other PC facilities. The example is taken from the C Programming Manual by Kernighan and Ritchie but has been broken into two modules to demonstrate debugging a multiple module program. If you are using Assembler, Pascal, or some other language in your application, you will still find this tutorial useful from a procedural point of view. The following is a listing of this sample program.

DEMO PROGRAM LISTING

```
/*-----------------------------------------------
Main test program for:  Calculation of Fahrenheit
to Celsius Table.  Based on the Kernighan and
Ritchie text
-----------------------------------------------*/
extern Prompt ();
extern GetVal ();
typedef int temp;     /* temperatures are integers
*/
int lower, upper, step;
temp fahr, celsius;
main()
{
/*-----------------------------------------------
 Initialize lower bound, upper bound, and step.
-----------------------------------------------*/
 Init (&lower, &upper, &step);
/*-----------------------------------------------
 Print table header.
-----------------------------------------------*/
 printf ("   F     C\n");
/*-----------------------------------------------
 Print table.
-----------------------------------------------*/
 fahr = lower;
 while (fahr <= upper) {
/*-----------------------------------------------
  Compute celsius temperature.
-----------------------------------------------*/
   Compute (fahr, &celsius);
/*-----------------------------------------------
 Print line of table.
-----------------------------------------------*/
  printf("%5d %5d\n", fahr, celsius);

/*-----------------------------------------------
  Go to next line of table.
-----------------------------------------------*/
```

```
  fahr = fahr + step;
  };
return;}
/*-------------------------------------------
procedure name: Init
function:
    1) Initialize Lower bound, upper bound, and
step.
inputs:
    l -- lower bound
    u -- upper bound
    s -- step
outputs:
    l, u, s set.
called:
    init (&l, &u, &s);
-------------------------------------------*/
Init (l, u, s)
int *l, *u, *s;
{
    /*---------------------------------------
    Get lower limit.
    ---------------------------------------*/
    Prompt ("\nFahrenheit lower limit? ");
    *l = GetVal ();
    /*---------------------------------------
    Get upper limit
    ---------------------------------------*/
    Prompt ("Fahrenheit upper limit? ");
    *u = GetVal ();
    /*---------------------------------------
    Get step.
    ---------------------------------------*/
    Prompt ("Fahrenheit step value?  ");
    *s = GetVal ();
return;}
/*-------------------------------------------
```

```
procedure name: Compute
function:
    1) Compute the celsius temperature from the
input
       fahrenheit temperature.
inputs:
    f_temp -- fahrenheit temperature
outputs:
    c_temp -- equivalent celsius temperature

        c = (f-32) * (5/9)

called:
    Compute (f_temp, &c_temp);
-------------------------------------------*/
Compute (f_temp, c_temp)
temp f_temp, *c_temp;
{
    *c_temp = f_temp - 32;
    *c_temp = *c_temp * 5;
    *c_temp = *c_temp / 9;

return;}
/*-------------------------------------------
IO module for:
     Calculation of Fahrenheit to Celsius Table.
     Based on p.8 of the Kernighan and Ritchie
text.
-------------------------------------------*/
/*-------------------------------------------
procedure name: Prompt
function:
    1) Output the character string to the console.
inputs:
    str -- string to be output
outputs:
    none
called:
    Prompt ("Prompt string");
-------------------------------------------*/
```

```
Prompt (str)
char str [];
{
int i;
    i = 0;
    while (str [i] != '\0') {
     putch (str [i++]);
    };
return;}
/*-----------------------------------------------
procedure name: GetVal
function:
    1) Get decimal value from the standard input
device
inputs:
    none
outputs:
    return decimal value.
called:
    value = GetVal ();
-----------------------------------------------*/
GetVal ()
{
int in_val;
    scanf ("%d", &in_val);
    return (in_val);
}
```

EXERCISING THE DEMO

In this example, the input you provide from the keyboard is shown in bold print so you can identify it from the PROBE output. First invoke the version of PROBE you are running.

If you are running PC PROBE type:

PROBE <enter>

If you are running PC SOURCE PROBE type:

SOURCE<enter>

PROBE has macro commands which are similar to batch files. Macros let you create your own commands which you can substitute for groups of other PROBE commands. Use the LOAd Macro command to load a macro file called PROBE.MAC. This macro file has been previously created and could prove useful for many different debug sessions.

-loa m probe.mac <enter>

To take a look at the names of the macros you have just loaded from this file, use the MAcro command.

-ma <enter>
MEM
INITIALIZE
AbsWrite
PPS
PPS2
AArmsBArmsCResetDSS
21
R

Now execute the previously constructed macro INITIALIZE. This macro will assign modulenames to sourcefile names, load the demo program and its symbol table, define a symbol, and open a window. The macro displays as it executes. *Note that the ASI commands in this macro are valid only for the PC SOURCE PROBE since they assign program modulenames to sourcefile names. If the ASI commands are not in your INITIALIZE macro, then you must have loaded the PROBE.MAC macro file from the PCPROBE diskette rather than the PCSOURCE diskette.* This is the only difference in the PROBE.MAC files between these two PROBE products. To execute macro initialize, use the EM command:

```
-em initialize <enter>
-asi ftocm_code ftocm.c
-asi ftocio_code ftocio.c
-loa ftocnew.exe
-loa s ftocnew.mpl
 Reading symbols.
 Reading lines.
-symbol .start = cs:ip
-wi = r
-
```

Since this macro has loaded the symbol table for us, we can now look at symbols in the symbol table. (Note that this symbol table has been previously stripped of symbols in the C library and only program symbols remain.) The symbol table shows the segment:offset for each symbol as it now applies to the loaded program. The segment values may be different when you load the program since this depends upon the version of DOS and other drivers you may have installed in your system. The module names and high level language statement numbers are also displayed in the symbol table. To look at all symbols use the SY command.

-sy <enter>

Address	Symbol name
09BD:00D8	CELSIUS
0647:00B7	COMPUTE
09BD:00D6	FAHR
0654:0042	GETVAL
0647:006D	INIT
09BD:00D0	LOWER
0647:000C	MAIN
0654:000C	PROMPT
0624:0002	START
09BD:00D4	STEP
09BD:00D6	UPPER

Line numbers for module ..FTOCM_CODE

#21=0647:0012	#26=0647:0024	#31=0647:002F	#32=0647:0035
#36=0647:003E	#41=0647:004C	#46=0647:005F	#47=0647:0066
#49=0647:0068	#74=0647:0073	#75=0647:007E	#80=0647:0088
#81=0647:0093	#86=0647:009D	#87=0647:00A8	#89=0647:00B2
#112=0647:00BD	#113=0647:00C8	#114=0647:00CF	#116=0647:00D7

Line numbers for module ..FTOCIO_CODE

#30=0654:0012	#31=0654:0017	#32=0654:0027	#33=0654:003B
#35=0654:003D	#56=0654:0048	#57=0654:0057	

The CS:IP registers are set to the start of program execution when the program is loaded. You can now start single stepping the program from the start.

-st <enter>

```
.START:
0A0F:0002 CLI
0A0F:0003 MOV    AX,0DA8
0A0F:0006 MOV    DS,AX
0A0F:0008 MOV    AX,0E3C
0A0F:000B MOV    SS,AX
0A0F:000D MOV    SP,0080

AX=0000     CS=0A0F     SS=0E3C     DS=09FF     ES=09FF
BX=0000     IP=0002     SP=0080     SI=0000     DI=0000
CX=0000          BP=0000
DX=0000          FL=O0 D0 I0 T0 S0 Z0 A0 P0 C0
0E3C:0080   0000 0000 0000 0000-0000 0000 0000 0000
```

Note the window display for the registers and the stack data at the bottom of the screen. A single step is taken and the window is updated after each <enter key> is typed. This is happening because the INITIALIZE macro executed the PROBE WIndow command. This WIndow command assigned a macro named r to the window. The macro r displays registers and the bottom of the stack. Type any other key to terminate single stepping. Now, if you are running PC SOURCE PROBE, you can do source level single stepping. If not, then skip the next 2 commands. First, change the window to show the program variables which are printed from the MEM macro:

wi = mem <enter>

Now you can source step with the Source Step command:

```
-ss <type enter twice to get this display>

21.     Init (&lower, &upper, &step);
22.
23.     /*-----------------------------------------------------
24.     Print table header.
25.     -----------------------------------------------------*/
26.     printf (" F C\n");

Fahr= 0
Celsius= 0
Upper= 0
Step= 0
```

While you are source stepping, you can type PgUp and PgDn keys to scroll forward and backward through the source code to see what is happening. Type any other key to stop stepping.

Since this program is going to write to the screen, you should isolate the PROBE screen where you are now typing from the applications screen which the program will use. This is done with the screen switching command SC.

```
sc on <enter>
```

Now you can go from the current CS:IP and set a breakpoint at the instruction located at COMPUTE. The program prompts you for Upper Limit, Lower Limit, and Step values. Supply the values shown in the display below.

```
-g .compute <enter>
Execution begun.
FAHRENHEIT LOWER LIMIT? 0
FAHRENHEIT UPPER LIMIT? 100
FAHRENHEIT STEP VALUE ? 10
  F      C

Software breakpoint encountered at 09E2:00B7=.COMPUTE.

Fahr=   0
Celsius= 0
Upper=   100
Step=    10
```

The breakpoint at COMPUTE has been encountered. Note that the window pops up after the breakpoint and displays the information specified by the "MEM" macro. This program takes the input you supplied to its prompts, converts the data, and stores the results in program symbols. It might be a good idea at this point to see if the conversion happened correctly. One example of doing this is to look at the contents of the variable UPPER. You can do this with the display WOrd command with a starting address of UPPER.

```
-wo .upper <enter>
0D58:00D2 0064 000A 000A FFEF-2020 4620 2020 2020
```

Since no length or end address was given to the WOrd command, the default display is one line full of words. Since the starting address of this command was UPPER, the first word in the display is the value of UPPER. You see that the value of UPPER is now 64 hex. However, this program interpreted the keyboard input as an ASCII decimal number. Let's see if they are the same. The value you typed in for UPPER limit was 100 decimal. Use the EValuate command to display 100 decimal in several different bases. Note the t subscript indicates base ten or decimal to PROBE.

```
-ev 100t <enter>
0064H 100T +100T 1100100Y 'd'
```

This command shows the 100 decimal in hex, decimal, integer, binary and ASCII. Note that you could also have evaluated the contents of UPPER with the following command. (To see an explanation of the @ operator see Chapter 6, COMMAND REFERENCE.)

```
-ev @.upper <enter>
0064H 100T +100T 1100100Y 'd'
```

Next try using the hardware breakpoint capability of PROBE to trap a memory overwrite. Go till the variable .fahr is written.

```
g .fahr w <enter>
Execution begun.
  0   -17

Hardware breakpoint encountered at
09E2:0035=..FTOCM_CODE#32
```

The display shows that the hardware breakpoint has been encountered as well the current location of the CS:IP. Now you can see how the program executed in real time before the breakpoint was encountered. To do this, use the Trace command shown here to display the last 5 assembly language instructions from the trace data:

```
-t 5 <enter>
        OP
ADDR  CODE   OPERAND(S)
-------------------------------------------

0682B POP BP
        0ACDC READ - SS - 18
        0ACDD READ - SS - 11
0682C RET FAR
        0ACDE READ - SS - 5D
        0ACDF READ - SS - 00
        0ACE0 READ - SS - 47
        0ACE1 READ - SS - 06
        064CD MOV SP,BP

46.  fahr = fahr + step;

064CF MOV AX,WORD PTR [00D4] ;.STEP
        09CA4 READ - DS - 0A
        09CA5 READ - DS - 00
064D2 ADD WORD PTR [00D6],AX ;.FAHR
        09CA6 READ - DS - 00
        09CA7 READ - DS - 00
        09CA6 WRITE - DS - 0A
        09CA7 WRITE - DS - 00
```

First look at the following line of trace data on the screen:

```
46.     fahr = fahr + step;
```

This is the last C source statement which occurred before the end of the trace data. The executed assembly language is shown below this source statement. The first instruction shows that the variable STEP was moved into the AX register.

```
064CF MOV AX,WORD PTR [00D4] ;.STEP
```

The 064CF is the address of this MOV instruction. The .STEP is shown as a comment to the right of the instruction meaning the operand WORD PTR [00D4] is the symbol STEP. The details of instruction execution are shown under the instruction.

```
09CA4 READ - DS - 0A
09CA5 READ - DS - 00
```

Note that the address of the the word STEP is 09CA4. A value of 0A was read from memory and put into the AX register. The next instruction is:

```
064D2 ADD WORD PTR [00D6],AX ;.FAHR
```

The execution of this instruction shows that the variable FAHR was read with a value of 0 from address 0D656. It was then added to the AX register (which now contains STEP) and then written back to FAHR as 0A. This write cycle caused the breakpoint which was set on a write to the variable FAHR.

Also note the extra instruction after the breakpoint in the trace. This occurs because the prefetch queue in the 8088 has already loaded the instruction into the execution unit before the write cycle which caused the trap happened.

Next you can try macro commands. First define a macro which sets a breakpoint and prints a variable in an integer base after break a breakpoint. Macros are simply groups of PROBE commands which can be invoked by executing the macroname. Parameters can be passed to macros by including a %number as a place holder in the macro definition. A %0 in this macro will be used to pass a breakpoint to the macro when it is executed. The %1 will be used to pass a location in memory to be printed. Here is the macro.

```
-mac s = <enter>
M- g %0 <enter>
M- print "VALUE = ",%i @ w %1 <enter>
M- end <enter>
```

The **%i** in the PRint command is not a parameter to be passed to the macro but an indication of the data type to be printed. It indicates the data is to be printed in an integer format. The **@ w** in the PRint command indicates that the data to be printed is at the word pointed to by the **%1** parameter. The "VALUE =" in the PRint command will

print a label. Since this is a short explanation of one of the more complex commands in PROBE, you should review the PRINT command in Chapter 6, COMMAND REFERENCE, for more details.

If you were to execute this macro now, you would see its commands during execution. You normally want to do this when you are first testing a macro. Since this one already works, you can suppress the display of the macro commands while the macro executes. Do this by turing off the ECho with this command:

-ec off <enter>

Since the window is still on, it will display when the breakpoint is encountered. This is also not needed at this point. Therefore, close the window.

-wi c <enter>

The previous two commands were not necessary to execute this macro, they simply reduce what is happening while this macro is executing to make things simpler for this example.

Now use this macro to set a breakpoint when the variable .FAHR is written. Then print the contents of FAHR as an integer.

-em s .fahr w, .fahr <enter>

```
Hardware breakpoint encountered at
09E2:0035=..FTOCM_CODE#32
VALUE = 20
```

As you can see, the breakpoint was encountered, and the current value of FAHR was printed. Now use the macro to set a breakpoint at linenumber 46 and print the contents of FAHR again.

-em s #46, .fahr <enter>

```
Software breakpoint encountered at
09E2:005F=..FTOCM_CODE#46
VALUE = 20
```

This macro can be repetitively executed by simply typing **<F3><enter>**. Since the screen switching is still on, the program is writing to the other virtual screen. You can view what the program is doing to this screen by using the SCreen command. When the application screen is displayed, type any key to return to the PROBE screen.

-sc s <enter>

All of the above breakpoints were non-sticky, that is, they have to be specified each time you execute a Go command. Another way to define a breakpoint is with the BP command. This allows a breakpoint to be defined which is automatically included in each Go command. It also lets you define breakpoints without starting program execution. Define the following breakpoint which occurs when the data written to .FAHR is 60 decimal.

-bp 1 = .fahr w data 60t <enter>

Now execute a Go command from the current CS:IP. The sticky breakpoint is automatically set.

-g <enter>
Hardware breakpoint encountered at
09E2:0035=..FTOCM_CODE#32

As you can see, the breakpoint occurred. You can verify that this event really happened by displaying the contents of FAHR. Rather than just looking at it in one base, use the evaluate command to look at it in several bases as explained earlier.

-ev @.fahr <enter>
003CH 60T +60T 111100Y '<'

As you can see 60 decimal is stored in FAHR and therefore the hardware breakpoint on this event is proven. Macros can also be defined which include LOOp commands. LOOp commands let you execute groups of commands repetitively. Define a macro which stops on the Nth occurrence of the breakpoint. The number N will be passed to the macro as parameter %0. The breakpoint will be passed to the macro as parameter %1.

```
-MAC L = <enter>
M- LOO %0
M- G %1
M- ELO
M- END
```

Now execute this macro and pass parameters to stop on the 3rd occurrence of writing to the variable FAHR. This type of operation is sometimes referred to as "pass count on break."

```
-EM L 3, .FAHR W <enter>

Execution begun.
Hardware breakpoint encountered at
09E2:0035=..FTOCM_CODE#32
Execution begun.
Hardware breakpoint encountered at
09E2:0035=..FTOCM_CODE#32
Execution begun.
Hardware breakpoint encountered at
09E2:0035=..FTOCM_CODE#32
```

When programs execute, procedures may call other procedures to several levels of nesting. It is useful to know what the procedure calling sequence has been when you are in a given procedure. PROBE can show you the calling sequence of the procedures by analyzing the stack. Try the NEst command at this point to show the stack nesting.

```
-NE S <ENTER>
CS:IP is 09E2:0035 near ..FTOCM_CODE#32
Called from 0A33:01CE near .GETVAL
```

Another common need is to determine the values of local variables on the stack which are only active when a procedure is invoked. Since the DOS languages currently do not pass stacked based symbols and automatic variables to the symbol table, you must look at the stack to figure this out for yourself. The macro command capability of PROBE simplifies this process by letting you define a complex command to reference variables on the stack. Even though you must create the macro first, which can be difficult for complex arrays and chained pointers, the macro saves you a lot of time when you want to look at the variable several times during a debugging session. The

macro which prints the automatic variable i on the stack is defined as:

```
MAC I=
M-PR 'I=', %I@WSS:SP+4
END
```

Now assign this macro to the window so you can see the variable after each step or breakpoint.

WI =I <enter>

To get back to the point where this variable is activated in the program, reload the program.

-loa ftocnew.exe <enter>

Now single step again by source code statements as you did at the start. If you are not running PC SOURCE PROBE, use the STep command instead.

-ss <enter>

Note that once the procedure which activates i on the stack is entered, that i changes as you step through the While structure in the program.

If you are running PC SOURCE PROBE, you can make use of the on-line editor to note changes to your program. The editor lets you open a file for display and then edit lines of the file. However, the changed lines do not get stored in the file. They go instead to a side file called a log file. You can then use your favorite text editor after the debugging session to move the changes from the log file into your source files. This procedure helps minimize the exposure of your source files during debugging. It also gives you a history of changes you have made to your program since the log file was updated. First invoke the editor. Note that since you will not specify a filename to be edited, the default is the file which matches the current CS:IP. PROBE finds this file by finding which module is associated with the current linenumber. Then PROBE finds which source file has been assigned to this modulename (done in the ASIgn command).

-ed <enter>

```
101.    outputs:
102.    c_temp -- equivalent celsius temperature
103.
104.    c = (f-32) * (5/9)
105.
106.    called:
107.    Compute (f_temp, &c_temp);
108.
-----------------------------------------------*/
109.    Compute (f_temp, c_temp)
110.    temp f_temp, *c_temp;
111.    {
112.    *c_temp = f_temp - 32;
113.    *c_temp = *c_temp * 5;
114.    *c_temp = *c_temp / 9;
115.
116.    return;}
```

When the editor is entered, it is in Display mode, and the menu window shows you the keys for scrolling through the source file. Press the PgUp key to move to the start of the file. Now type the ESC key which puts you in the Command mode of the editor. A new menu of editor commands is displayed in the menu window. You can note a change to a line of code by using the Change command. In this case change line 114.

FTOCM.C-c 114 <enter>
Enter file for log of changes: **a:log.tmp**

Since this is the first time the editor has been entered, it prompts you for the file to put the logged changes in. The file name which has been chosen is a:log.tmp. This editor provides line editing using the basic edit keys as defined for DOS. Recall line 114 into the edit area by typing the F3 key. Use the ins, del, rubout, and DOS edit keys to make the desired changes to this line. Typing <enter> will log the changes into the log file. Type E to exit the editor. Note that the source file has not been changed; the change has only been recorded in the log file. This is only a sample of using the PROBE commands and more examples are contained throughout the remainder of this manual.

ADVANCED DEBUGGING TECHNIQUES

This section shows more advanced examples of using 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, use the following procedure to debug it. The primary objective in using PROBE to debug a boot load sequence is to replace the the needed PROBE vectors at locations 0:4, 0:8, and 0:C after they have been modified by the boot loader. Proceed as follows.

1. First boot up DOS, and load PROBE.
2. Look at locations 4 thru 7 and C thru F with the byte commands:
   BY 0:4 L 4
   0000:0004 3F 08 41 89
   BY 0:C L 4
   0000:000C 42 01 00 80
3. Create and save a macro to restore these locations to their current contents. (The values shown here may not be the same because of the version of PROBE software you have. Use current values.)
   MAC FIX=
   M- BY 0:4= 3F 08 41 89
   M- BY 0:C= 42 01 00 80
   M- END
   SAVE M FIX.MAC
4. Put an INT 19 instruction somewhere in free memory.
   BY 3000:0 =CD,19 (or ASM an INT 19 at location 3000:0)
5. Insert the diskette with the new boot loader to be debugged.
6. It may be necessary to switch to an external console. This should be done if the system locks up when the STOP button is pressed later.
7. Issue the Go command to execute the INT 19.
   G = 3000:0 [OPTIONAL HARDWARE BREAKPOINTS]

8. If it is desired to trap somewhere in the boot process, then it may be advisable to set a hardware breakpoint. An execute breakpoint will not work because the boot loaded program may write over the top of the breakpoint trap. If you want to stop at an instruction somewhere in the boot loader, use the Fetch verb in the hardware breakpoint. At this point, the system will perform a new boot load.
9. Return control to PROBE by pressing the STOP button.
10. Execute the fix macro to patch the modified vectors.
    EM FIX
11. Since DOS does not know that PROBE is running at this point, PROBE commands LOAd, SAve, Source STep, and EDit cannot be used. If you want symbols attached to your boot loader, see the section LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM later in this chapter.

DEBUGGING A DEVICE DRIVER WHICH INSTALLS ITSELF

You can debug a device driver which installs itself during the system boot up process using the following procedure.

1. Load PROBE.
2. If it is desired to have the symbols for the device driver, then follow the instructions in the section LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM later in this chapter.
3. Do the boot load sequence as described in the application DEBUGGING A BOOT LOAD SEQUENCE.
4. Press the STOP button.
5. The device driver is now installed and can be debugged by PROBE. If a jump to self loop has been put into the device driver, then the driver segment register can be located and control can be transferred to PROBE with the STOP button before the driver starts to execute. Otherwise, the STOP button will interrupt the driver while it is executing.

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

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

1. Load PROBE, and then execute a quit and stay resident - Q R.
2. You are now in DOS. Press the STOP button.
3. You are now back in PROBE running as a quit and stay resident program. You may load the symbol table using the procedure LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM later in this chapter.
4. Go back into DOS with the Go command. You may want to set a hardware breakpoint first. (Don't set an execute breakpoint which will get overwritten by the load of the device driver or quit and stay resident program - use a Read or Fetch breakpoint instead.)
5. Now invoke the device driver while you are in DOS.
6. The breakpoint or STOP button will get you back into PROBE.

LOADING THE SYMBOL TABLE WHEN PROBE DOES NOT LOAD THE PROGRAM

Some applications such as boot loaders and installed device drivers must be loaded by DOS or the system and not by PROBE. To get the symbol table into PROBE for these types of programs, you must use the absolute form of the load symbols command.

LOA S symboltablefile segmentvalue:0 A

This will adjust symbol values to match the loaded program before they are stored in the PROBE symbol table. Since PROBE did not load the program, it will not know the segment value for the load address. To find this value, use the following procedure.

Put a jump to self loop at the start of the program to be debugged for debugging purposes. When the program is loaded, it will not begin to execute but will stay in its current code segment. Next press the STOP button.

If your program was written in Assembly language and the ES register has not been changed by your program between the time it was loaded and the time the STOP button was pressed, then:

segmentvalue is = ES+10

If your program was written in a high level language like C, then the program initializations which come from the language run-time libraries will execute between the time your program is loaded and the STOP button is pressed. This cannot be prevented. First put a public label called TEMP in your program at the jump to self loop. When you press the STOP button look at the CS register. Next look at the segment value specified by the linker in the .MAP file for the public label you inserted into your program. The segmentvalue which is used in the PROBE command is:

segmentvalue = CS - Segment value of TEMP in Link Map

Now the symbol table can be loaded using:

LOA S filespec segmentvalue:0 A

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 the 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 PROBE or using concurrent process debugging.

Locking out all Interrupts While in PROBE

When first bringing up interrupt routines, it is most useful to lock out all interrupts while in the PROBE software. This is because an interrupt from a device which happens while in the PROBE software could potentially never return or crash the system if it is allowed to be serviced. To lock out interrupts while in the PROBE software use the INTerrupt command as described in Chapter 6, COMMAND REFERENCE. This allows masking any or all of the interrupts from the system interrupt controller from being processed while in the 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 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 PROBE software has control of the system. For debugging this type of system, use the INTerrupt command as described in Chapter 6, COMMAND REFERENCE to allow some interrupts to take control of the system whenever they happen.

DEBUGGING ON A NON-DOS OPERATING SYSTEM

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

1. First boot DOS and load the 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 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 PROBE can get control.
5. You cannot use PROBE LOAd or SAve commands 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 PROBE to external console and execute the INT ALL OFF command before rebooting.

DEBUGGING MEMORY OVERWRITES

One of the most difficult problems to isolate is that of a memory overwrite. PC PROBE greatly simplifies the detection of this problem by allowing you to set a breakpoint on writing to a memory location or range of memory locations. The breakpoint can be further qualified to detect a data pattern along with the write. Overwrite breakpoints can find bugs such as writing or "stacking" into a code segment. Overwrite breakpoints can also find bugs which change a variable to a known data value which is known to be wrong. However, some bugs are more subtle, and cause an overwrite to a data variable which is not "out of range" but is coming from another area of the program which should not be writing to this variable. This type of bug can be detected by trapping the overwrite condition, then looking at the CS register to see if the overwrite is coming from an area of the program which is valid to be changing this variable. An example of this might be the user program writing into DOS or DOS writing into the user program. This problem can be found by creating a macro which:

1. Detects the overwrite.
2. Stops and checks the CS register when the overwrite occurs.
3. Continues if the CS register is valid for this overwrite.

Assume that it is OK to write to the variable FOO if the code segment is 0C9A, but not OK if the CS is anything else. The following macro would detect this condition.

```
MAC L =
G .FOO W
LOO W CS = 0C9A
G .FOO W
ELO
END
```

DEBUGGING A SYSTEM WHICH CRASHES AND TAKES 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 PROBE to work. These cases are best debugged on an external console. You should also lock out all interrupts while in the PROBE software with the INT ALL OFF command. This ensures that interrupts in the system do not take control away from PROBE while you are in the PROBE software.

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

USING PROBE WITH TOPVIEW

For more information on using PROBE with TopView, see Appendix H.

ADDITIONAL APPLICATIONS INFORMATION

For additional applications information, see Appendix K.

CHAPTER 6
COMMAND REFERENCE

INTRODUCTION

This chapter contains a detailed description and examples for each PROBE command. The commands are listed alphabetically. This chapter also defines the common terms and parameters that are used in the PROBE command text.

COMMON PARAMETERS AND DEFINITIONS

The following definitions apply to the commands found in PC PROBE.

PARAMETER DEFINITION

value — A numerical value. This can be:
-- a numeric constant (e.g. 1234)
-- a register (e.g. AX)
-- dereferenced data (e.g. @w 1234:5678)
-- a symbol

expression — A value calculated by combining a series of values with operators.

{+ , - , * , / , ~, & , | , % }.

Normal precedence of operators is assumed: (*, / ,% ,&) are higher than (+, -, |) and evaluation proceeds left to right on operators with equal precedence. Precedence may be overridden by the use of '(' and ')'. Note that & is AND, | is OR, ~ is NOT bit by bit, % is modulo.

address — An address can be represented as:

expression:expression
or
expression

In the latter case, the expression is assumed to be the offset, and the default segment is used.

PARAMETER DEFINITION

range — A range of memory values which can be specified by either of two forms:

start address L length
start address end address

where start address can be any of these forms:

segment expression:offset expression
default segment:offset expression

and where end address is:

offset expression (start address seg assumed)

When using the first form for range, a space must be placed after the L. The maximum length for range is FFFF.

boolean expression — An evaluated 16 bit expression resulting in TRUE or FALSE if bit 0 is 1 or 0. Or, an evaluated 16 bit expression of the form:
expression { >, <, =, <=, >=, <> } expression
resulting in TRUE or FALSE.

base — The default base for numbers is hex. The base can be decimal if a T subscript is used. Use quotes for ASCII characters.

filespec — [drive] [path] [name of file] If these are not specified, then the default drive/path is used.

[ ] — The brackets indicate that this is an optional part of the command and may be left out.

{ } — The curly brackets enclose a number of choices for the command and one choice must be made.

module — A module is a single unit of compilation and produces an object file. The module typically has a name which is assigned through a compiler control.

PARAMETER DEFINITION

modulename A modulename can be assigned to the module by some compilers. This associates a modulename with a group of linenumbers in the MAP file of the linker.

linenumber A linenumber is a number passed from the compiler to the linker as a symbol which points to the first instruction in a line of executable source code. Not all line numbers generate executable code, and therefore, they are not passed to the linker.

DATA REFERENCING WITH THE @ OPERATOR

The @ operator works on an address by using the format:

@ {W|P} address

The @ operator indicates that **address** contains the value to be used. **{W|P}** indicates the type of value contained at address. **P** is a full 32 bit pointer and **W** is a 16 bit value. If **W** or **P** is not specified, then a byte is assumed with the upper byte set to 0. The @ operator can be used for several levels of indirection with the indirection evaluated right to left.

In the examples below, a command along with its results are shown. You may get a feel of how to use this operator in commands by making the assignments shown below, then trying out the command examples which use the @ operator. More examples of this operator are shown in the PRint command examples.

ASSIGNMENTS

```
BYTE 8000:10 = 20,00,00,30,30,00,97,42
BYTE 3000:20 = 05,32
BYTE 3000:24 = 72
BYTE 2000:3205 = 22
BYTE 4297:30 = AA
BYTE 2000:24 = 34
SY .START = 8000:10
R DS = 2000
```

COMMAND/RESULT	EXPLANATION
BYTE .START 8000:10 20	where .START = 8000:10 8000:10 = 20
BYTE @P .START 3000:0020 05	where .START = 8000:10 8000:10 = 3000:20 3000:20 = 05
BYTE (@W.START+2):(@W.START) 3000:20 05	where .START = 8000:10 8000:12 = 3000 8000:10 = 20 3000:20 = 05
BYTE @P .START+4 4297:30 = AA	where .START = 8000:10 8000:14 = 4297:30 4297:30 = AA
BYTE (@W.START+2):(@W.START)+4 3000:24 = 72	where .START = 8000:10 8000:12 = 3000 8000:10 = 20 3000:24 = 72
BYTE DS:@W .START 2000:0020 34	where .START = 8000:10 8000:10 = 20 the current DS register is 2000 2000:20 = 34
BYTE DS:@W@P.START 2000:3205 22	where .START = 8000:10 8000:10 = 3000:20 3000:20 = 3205 the current DS register is 2000 2000:3205 contains 22

ASSEMBLE

PURPOSE: To enter instructions in 8088 assembly language mnemonics into memory.

FORMAT: **ASM [start address] <enter> <assy language stmt>**
<enter> only to stop assembly

REMARKS: You can enter instructions into memory starting at the **start address.** If **[start address]** is not specified, then the assembly starts where a previous assemble command ended. The default segment for **start address** is CS: unless another segment or value is specified. The standard assembly language syntax is supported with the guidelines below.

1. Numbers are in hex (decimal if suffixed by the letter T). Symbols in the symbol table may also be used in expressions. 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.

   EXAMPLE:
   ASM .LOOP
   ASM 4000T:0

2. 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

3. Segment override mnemonics are cs:, ds:, es:, and ss:

   EXAMPLE:
   0642:0000 MOV AX,CS:[.LOOPSTART]

4. 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 intersegment jumps or calls, otherwise, the current segment is used.

   EXAMPLE:
   0642:0000 JMP .LOOP ;A 2 BYTE SHORT JUMP
   0642:0005 JMP FAR .START ;A FAR JUMP

5. 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.

   EXAMPLE:
   0642:0004 DEC WORD PTR [SI]

6. 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]

7. All forms of register indirect commands are supported.

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

8. All opcode synonyms are supported.

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

ASIGN*

PURPOSE: To assign a filespec to a module name for use in single STep, Trace, and EDit commands. This command is found in PC SOURCE PROBE.

FORMAT: **ASI ..modulename filespec**

REMARKS: To do source level single stepping, the symbol table must have line numbers and each group of line numbers must have a unique modulename. The **modulename** is assigned when the program is compiled. (See chapter 3.) To find the lines of source code from the modulenames which are in the symbol table, PROBE must know the filespec for the source code which generated the module. This command assigns **modulenames** to **filespecs.**

FORMAT: **ASI**

REMARKS: Displays all modulename/filespec assignments.

FORMAT: **ASI ..modulename**

REMARKS: Displays the filename assigned to **modulename.**

FORMAT: **ASI ..modulename NULL**

REMARKS: This effectively deletes the filespec assignment to the **modulename.**

*NOTE: Yes, we know it is spelled wrong - but it has a less offensive abbreviation.

EXAMPLES: To assign modulename FTOCM_CODE to the file B:FTOCM.C:

```
ASI ..FTOCM_CODE B:FTOCM.C
```

To check assignment of module FTOCM_CODE:

```
ASI ..FTOCM_CODE
Module ..FTOCM_CODE is assigned to file B:FTOCM.C
```

To remove assignment of module MY_MODULE:

```
ASI ..MY_MODULE NULL
```

To check assignment of all modules:

```
ASI
Module ..FTOCM_CODE is assigned to file B:FTOCM.C
Module ..FTOCIO_CODE is assigned to file
B:FTOCIO.C
Module ..MY_MODULE is not assigned to a file
```

BREAKPOINT

PURPOSE: To define or display sticky breakpoints.

FORMAT: **BP breaknumber = breakpoint condition**

REMARKS: **Breaknumber** is from 1 to 8 and **breakpoint condition** matches the definitions shown next in breakpoint formats. Breakpoints defined with the BP command are sticky breakpoints and are automatically included with each Go command. Sticky breakpoints are deactivated with the DElete BP command. Defining a sticky breakpoint which already has a definition will redefine the **breakpoint condition** and display the old definition. Symbols, registers, etc. are interpreted as absolute values and are substituted into the breakpoint definition when program execution starts. In other words, BPs are evaluated when the Go command is executed. 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. Breakpoints set with the Go command are not sticky breakpoints.

The first format for **breakpoint condition** is:

[IO] address [verb] [DATA datavalue]

Address is a memory address for the breakpoint unless preceded by the key word **IO** which indicates that the address is for IO. If no segment register is specified for **address**, then the CS register value is assumed. **Address** may be a range of addresses by replacing a hex character in the address with the character X. The X indicates a "don't care" condition for this hex character of the address. Range breakpoints cannot be used for execution breakpoints. **[verb]** is one of the following:

R for read
W for write
A for all (i.e read, write, or execute)
D1 for DMA transaction on channel #1
D2 for DMA transaction on channel #2
D3 for DMA transaction on channel #3

If **[verb]** is not included in the breakpoint, then break on instruction execution is assumed.

The key word **DATA** indicates that the data portion of the operation must also match **[datavalue]** to cause the break at **address**. If **DATA** is not included, then data is ignored in the breakpoint. **Datavalue** may only be a byte quantity. Also, **datavalue** is not allowed for execution breakpoints.

The second format for **breakpoint condition** is:

L # [H|L]

This type of breakpoint causes a trap when a hardware line represented by # is either high (H) or low (L). The hardware lines are defined differently depending on whether or not the logic probe cable is plugged into the PC PROBE. See Appendix I LOGIC SIGNALS for the definition of each line.

FORMAT: **BP [breaknumber]**

REMARKS: Displays the definition of the breakpoint assigned to **breaknumber**. If **[breaknumber]** is left out then all sticky breakpoints are displayed.

EXAMPLES: This breakpoint occurs when the instruction at the location specified by the symbol TEST is about to be executed.
BP 1 = .TEST

EXAMPLES, continued

This execution breakpoint occurs at location CS:10.
BP 2 = 10

This breakpoint occurs when the instruction at location 1000:0000 is about to be executed.
BP 3 = 1000:000

This breakpoint occurs when the memory location at TEST is read (but not necessarily executed.)
BP 4 = .TEST R

This breakpoint occurs when the IO port represented by the symbol KEY is read.
BP 5 = IO .KEY R

This breakpoint occurs when the memory location TEMP is written.
BP 6 = .TEMP W

This breakpoint occurs when any data in the address range 10000 to 100FF is read.
BP 7 = 1000:0000 TO 1000:00FF R

This breakpoint occurs when the low memory vector table is read.
BP 8 = 0:3xx r

This breakpoint occurs when the value AF (hex) is written to location TEMP.
BP 8 = .TEMP W DATA AF

This breakpoint occurs when a dma transaction on DMA channel 2 moves data anywhere in memory. See the Special Notes on Breakpoints section for a caution on setting breakpoints on DMA operations.
BP 8 = XXXX:XXXX D2

This breakpoint occurs when a logic level HI is detected on line 4.
BP 8 = L 4 h

EXAMPLES, continued

Now these breakpoints can be displayed with the BP command:

```
BP
BP 1=.TEST
BP 2=10
BP 3=1000:000
BP 4=.TEST R
BP 5=1000:0057 R
BP 6=.TEMP W DATA AF
BP 7=.TEMP W WDATA 77AF
BP 8=L 4 H
```

Special Notes on Breakpoints

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

A parity error will cause a breakpoint.

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 8088 queue and cannot always be stopped from executing when the breakpoint occurs.

Three hardware breakpoints are available in PROBE. The remaining five allowable breakpoints are implemented via a software interrupt of type 3. Since these breakpoints are implemented in software, they may only be "execute" type breakpoints (i.e. no verb specified). A maximum of three non-execution type breakpoints and five execution type breakpoints may be active when program execution begins including both STICKY and NON-STICKY breakpoints. PROBE will report an error if an attempt is made to activate more than three non-execution type breakpoints or five execution breakpoints at once.

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 read breakpoint on the address in prom. This will cause a break when the instruction is fetched but not necessarily executed. This is because the 8088 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 set an instruction execution breakpoint in prom memory.

To set a breakpoint on a DMA cycle which was the result of a DOS system call (for example, function codes F or 3D), then the external console must be used. This is because the interrupts caused by the use of the local keyboard and console confuse the DOS DMA interrupt service routines. DMA operations initiated directly by the user do not have this restriction.

The PC 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. If the applications program is going to modify the NMI vector when it runs, you must follow the sequence described in "USER PROCESSED NMI" in Appendix D to restore the vector. If the applications hardware uses the IOCHK line, the PC PROBE can determine which source the NMI came from. Example situations when the application may modify the NMI vector are:

1. If the applications program uses a numeric run-time library which changes the NMI vector at run-time.
2. If an 8087 is plugged into the system and the user has written routines to process the 8087 exceptions which cause an NMI.

BYTE

PURPOSE: To display or change the bytes in memory.

FORMAT: **BY [range]**

REMARKS: The bytes in memory specified by **range** are displayed in hex and ASCII. If no segment register or segment value is specified in **range**, then the segment value from the previous BYte command will be used. If no length is specified in **range** then the default is the previous length. If no **range** is specified at all, then a block of bytes starting at the end of the previous block is shown with a length equal to the previous length. The PgDn key will also display the next screen full of bytes.

FORMATS: **BY start address = value [,] value [,] value...**
or
BY start address = <enter>

REMARKS: In the first format, bytes in memory are changed to the list of values to the right of the equal sign. In the second format, the current byte at the address is displayed and a prompt waits for a new data value at this location. After the new data is entered, the next location is displayed and can be changed. Typing <enter> alone on a line ends the changes. When data is written to memory it is read back for verification unless the NOVerify condition is Noread. Errors are reported accordingly.

EXAMPLES: To store a list of bytes:
BY .START = 50,51,52

To store the same list of bytes but to look at each one before changing it:
BY .START =
3000:0000 40 - 50 <enter>
3000:0001 43 - 51 <enter>
3000:0002 44 - 52 <enter>
3000:0003 40 - <enter>

EXAMPLES, continued

To display the data:
BY .START L 3
3000:0000 50 51 52 *PQR*

The next 3 bytes can be displayed with the following command. Since no length was specified, PROBE uses the previously specified length of 3.
BY <enter>
3000:0003 40 41 42 *@AB*

Registers can also provide the start address.
BY SS:SP = 0, 0, 0, 0
BY SS:SP+BP L 3

ASCII strings can also be stored.
BY .START = 'THIS IS A LINE'

COMPARE

PURPOSE: To compare the contents of two blocks of memory.

FORMAT: **COM range destinationaddress**

REMARKS: The contents of the block of memory specified by **range** is compared to the same size block starting at the location specified by **destinationaddress**. The entire block is compared and the mismatches are displayed in the following format:

Source address byte byte Destination address

The Source address corresponds to the range block of memory and the first byte is the mismatched byte in the range block. The second byte is the contents of the destination address.

The COMpare command uses the DS segment value unless there is another segment register or segment value specified.

EXAMPLES: To compare the 100 locations starting at DS:200 to the locations at 1000:100:
COM 200 L 100T 1000:100

To compare the 32 locations starting at SS:SP to the locations starting at ES:DI:
COM SS:SP L 20 ES:DI

To compare the 257 locations starting at TEMP to NEWTEMP:
COM .TEMP .TEMP+100 .NEWTEMP

To compare the 32 locations starting at LOOP to 100H in the CS segment:
COM .LOOP L 20 CS:100

CONSOLE

PURPOSE: To change the console for the PROBE.

FORMATS: **CON Local**
CON Remote
CON Remote [1|2]
CON Other

REMARKS: **CONsole Local** restores the communication to the PROBE from the PC keyboard and monitor. This is the default when 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 command.

CONsole Remote uses the serial RS232 port on the PC PROBE board for commands. This frees up the keyboard and the monitor on the PC 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 PROBE software. This is very useful for debugging routines which steal DOS interrupts.

Remote 1 specifies using the COM1 port and **Remote 2** specifies using COM2. The function is the same as the CONsole Remote case.

The connection to the PC PROBE, COM1, or COM2 RS232 ports is described in Appendix C. The external CRT configuration parameters are also described in Appendix C. When this command is executed and PROBE begins to use the external CRT, the menu is not "ON" for the external CRT. The user must execute the "Menu On" command to display the menu window. The user must ensure that the PROBE.CFG file contains the proper CRT configuration parameters before turning on the MENU window, since sending cursor motion strings which do match the CRT may lock it up.

CONsole Other allows the screen display for PROBE commands to appear on an alternate video monitor driven from an alternate video display controller board plugged into the PC. 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 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 PROBE output is put on the monochrome monitor.

If Console Other is being used, the user should not invoke screen switching with the SCreen switching command since this will cause needless switching of the screen.

DELETE

PURPOSE: To delete symbols, macros, or breakpoints.

FORMATS: **DE S .symbolname**
or
DE S ALL

REMARKS: Deletes the symbol described by **symbolname** or deletes all symbols in the symbol table.

FORMATS: **DE M macroname**
or
DE M ALL

REMARKS: Deletes the macro described by **macroname**, or deletes all macros in the macro table.

FORMATS: **DE B breaknumber**
or
DE B ALL

REMARKS: Deletes the breakpoint referenced with **breaknumber**, or deletes all the breakpoints.

EXAMPLES: To delete the symbol TEST:
DE S .TEST

To delete the macro INIT:
DE M INIT

To delete breakpoint number 7:
DE B 7

DIR

PURPOSE: To display the names of files in a directory. This command is only available in the PC SOURCE PROBE.

FORMAT: **DI [filespec]**

REMARKS: If **filespec** is not listed, '*.*' is used. All standard DOS wildcard options for filenames are available with the DIr command.

EXAMPLES: To display all files in the current directory of drive A:

```
DI A:*.*
COMMAND.COM
SOURCE.EXE
SRMNHW.EXE
PROBE.CFG
```

To display all macro files in the current directory of drive B with a .MAC extension:

```
DI B:*.MAC
INIT.MAC
SOURCE.MAC
```

To display all files in the current drive and path of \USER\PROG with a .C extension:

```
DI \USER\PROG\*.C
FTOCM.C
FTOCIO.C1
```

DMA

PURPOSE: To include or remove the display of DMA cycles with the trace display.

FORMAT: **DMA channelnumber ON/OFF,...**

REMARKS: The display of DMA cycles can be included or removed when the trace data is displayed. This applies to all DMA channels including refresh.

CONDITION	RESULT
0 ON	DMA channel 0 cycles displayed
0 OFF	DMA channel 0 cycles not displayed
1 ON	DMA channel 1 cycles displayed
1 OFF	DMA channel 1 cycles not displayed
2 ON	DMA channel 2 cycles displayed
2 OFF	DMA channel 2 cycles not displayed
3 ON	DMA channel 3 cycles displayed
3 OFF	DMA channel 3 cycles not displayed

FORMAT: **DMA**

REMARKS: The status of all DMA display cycles is displayed.

EXAMPLES: To enable display of channel 0 and 2 DMA cycles in the trace data and disable all others:
DMA 0 ON, 2 ON, 1 OFF, 3 OFF

To display conditions:
DMA
0 ON, 1 OFF, 2 ON, 3 OFF

ECHO

PURPOSE: To display or suppress the commands inside a macro on the screen.

FORMAT: **EC ON/OFF**

REMARKS: If echo of macro commands is **ON**, then the commands are displayed on the screen as they are executed. If echo is **OFF**, then only the output of the commands within the macro are displayed on the screen and the commands are not displayed. Turning off the command display makes the macro execute faster and leaves a "cleaner" looking screen.

FORMAT: **EC**

REMARKS: The status of echo will be displayed.

EXAMPLES: To enable display of macro commands:
EC ON

To disable display of macro commands:
EC OFF

To display state of macro command echoing:
EC
Do Not Echo Macro Commands.

EDIT

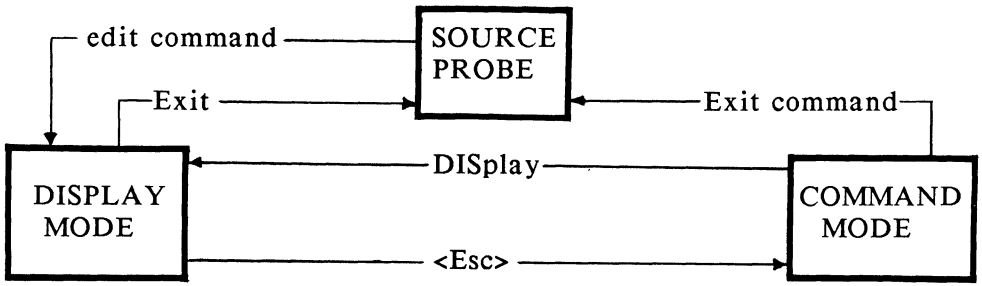
PURPOSE: Edit lets you display files and real time trace data. Changes to files can be noted in a log file. This command greatly reduces the need for program listings. It also lets you search trace data for events. This command is only available in the PC SOURCE PROBE.

There are two different edit modes: a DISPLAY mode and a COMMAND mode. A submenu of commands is displayed in the menu window while in each mode.

DISPLAY mode. This mode lets you display and page through a file. Edit begins in DISPLAY mode. To move from the DISPLAY mode to the COMMAND mode, press the <esc> key.

COMMAND mode. Edit commands operate on the displayed file. In addition, the standard DOS edit keys are used to operate on specified lines in the displayed file or Trace data.

The following diagram illustrates how to move between PC SOURCE PROBE command level, edit Display Mode, and edit Command Mode.



PG UP
PG DN
<Ctrl>PG UP
<Ctrl>PG DN
HOME
up arrow
down arrow

Change command
DIR command
Insert command
Line command
Search command

FORMAT: **ED [filespec]**

REMARKS: The file is opened and displayed in Display mode. The display starts at the same point where the previous display of this file ended from the previous edit command. If this file has never been edited, it is displayed starting at the top page. If **filespec** is not specified, the file addressed by the current CS:IP is displayed on the screen. The cursor is placed on the line associated with the current CS:IP.

The editor keeps track of the line number position in 10 different files. This provides quick repositioning of displayed files when switching between several files for display. If a request is made to display an eleventh file, you are prompted for a file to be "closed" so the the new file may be "opened." Note that this open/closed status applies only to the internal file/linenumber table and does not reflect whether the file is actually an open DOS file. The currently displayed file is the only file that is an open DOS file.

The currently displayed file, the log file, and the list file are the only files that are ever opened by PC SOURCE PROBE. This means that PC SOURCE PROBE will have a maximum of 3 files open under DOS at one time.

FORMAT: **ED TRACE [Lines|Hardware][# lines/instructions]**

REMARKS: This command is only available in SOURCE PROBE. Trace data is treated like a file and is displayed in DISPLAY mode. Displaying the trace data under control of the editor allows it to be paged through and searched with the editor in COMMAND mode. The options in this command for the display format for the trace data and the number of lines to be displayed are identical to the options in the Trace command. If a number of lines is specified when this command is invoked, then only those lines can be processed by the Edit command. Specifying more lines increases the time required to process this command. Note TRACE must be spelled out to minimize conflicts with other filenames.

EXAMPLES: To edit the source file B:FTOCM.C:
ED B:FTOCM.C

To edit trace data displayed with source lines:
ED TRACE L

To edit low-level trace:
ED TRACE

Display Mode

PURPOSE: This mode opens a file for display and scrolling. You are in this mode when you invoke the editor.

REMARKS: While in display mode, the following keys have effect:

PgUp	-- display the previous page
PgDn	-- display the next page
Ctrl PgUp	-- display the top page of the file.
Ctrl PgDn	-- display the bottom page of the file.
HOME	-- move to the line that was first displayed in this file. This will either be the first line or the line containing the current CS:IP.
up arrow	-- display the page ending one line before the line currently at the bottom of the screen (up one line).
down arrow	-- display the next line (down one line).
E	-- exit edit mode.
<esc>	-- move to edit COMMAND mode.

EXAMPLES: To edit the file which is pointed to by the current CS:IP:
ED

To go to the end of this file:
Ctrl PgDn

To move back to the line which was displayed when this file was first edited:
HOME

To switch to command mode:
<esc>

Command Mode

While in the COMMAND mode an alternate menu of edit subcommands is displayed. A description of each of the following subcommands is given in this section:

Change
Insert
Line
Search
DIR
DISplay
Exit.

FORMAT: **Change [linenumber]**

REMARKS: Perform line editing on a line from the current file. If a linenumber is listed, it may be edited using the standard DOS edit keys F1-F5. The editing area for the line to be edited is the line below the Change command on the screen. Use the function keys to copy the line from the screen to this area. When the line has been edited with the function keys, the enter key will write the line to a log file. If no log file has been previously specified, the PC SOURCE PROBE will prompt the user for the name of a log file. Commands append output to the end of the log file. Each change writes the following information to the log file:

1. A blank line.
2. The current file name.
3. The edited linenumber and line.

EXAMPLE: Assume the line of text at line number 116 looks like this:
116. *c_temp = f_temp - 35;

To go to line 116:

c 116 <enter>

(The file is positioned to display line 116 on the screen.)

Command Mode, continued

To recall the line into the line editing area:
<F3>

To replace the 35 with 32 move the cursor left with the left arrow key twice then type 2. Then type <F3> again to display:
116. *c_temp = f_temp - 32;

FORMAT: **Insert [linenumber]**

REMARKS: Copies keyboard input directly to the log file. The input lines may be edited using the standard F1-F5 keys. Line editing is terminated by typing <enter> as the only character on a line.

If this is the first Change/Insert command, you are prompted for the log file name. Every insert command from this point on, puts its output into this file. Each insert writes the following information to the log file:

1. A blank line.
2. The current file name, followed by the linenumber if listed.
3. The input lines.

Command Mode, continued

FORMAT: **Line [linenumber]**

REMARKS: The requested linenumber is displayed in the middle of the screen. If linenumber is omitted, the line at the bottom of the screen is moved to the middle of the screen.

FORMAT: **Search [linenumber] "string"**

REMARKS: The text file is searched for "string" starting at linenumber. If linenumber is omitted, the search begins at the line at the bottom of the screen. If "string" is not found from linenumber to the end of the file, the search continues from the beginning of the file. The "string" is case-less. That is, lower and upper-case strings will match.

EXAMPLE: To search for the function COMPUTE:
S "Compute"
Found in line 154

FORMAT: **DIR**

REMARKS: Same as DIr command at PC SOURCE PROBE command level.

FORMAT: **DIS [filespec]**

REMARKS: Switches back to edit DISPLAY mode, and displays the specified file. If filespec is omitted, the current file is redisplayed.

FORMAT: **Exit**

REMARKS: Exits editing mode and returns to the PC SOURCE PROBE command menu.

EVALUATE

PURPOSE: To evaluate an expression or pointer in several bases.

FORMATS: **EV expression**
EV pointer

REMARKS: In the first form of this command, **expression** is evaluated and displayed in the bases shown below. A non-printing ASCII character is shown as a period. This command serves as a hex calculator and base converter.

HEX DECIMAL INTEGER BINARY ASCII

In the second form of this command, **pointer** is evaluated to a 5 hex character number. This command serves to calculate real addresses in the same way as done by the 8088.

EXAMPLES: The symbol PORT5 is evaluated.

```
EVA .PORT5
35H 53T +53T 110101Y '5'
```

Here are several more evaluation examples:

```
EV ss:sp+5
30005

EV (((AX-5)*2)/10)
0200H 512T +512T 1000000000Y '.'

EV 'a'
0061H 97T +97T 1100001Y 'a'

EV 10T
000AH 10T +10T 1010Y '.'

EV -1
FFFF 65535T -1T 1111111111111111Y '..'
```

EXAMPLES, continued

```
EV AAAA:FFFF
BAA9FH

EV ..LIST#25
0775
```

FILL

PURPOSE: To fill a range of memory locations with values in a list.

FORMAT: **FI range list**

REMARKS: If the number of items in **list** is fewer than the number of bytes in **range**, then **list** is repeated until the end of **range** is reached. If **list** contains more items than there are bytes in **range**, then the excess **list** items are ignored. The FIll command uses the DS segment value unless there is another segment register or segment value specified. The data which is written is read back for verification unless the NOVerify condition is set to Noread.

EXAMPLES: To fill the 100 locations starting at DS:200 with A5:
FI 200 L 100T A5

To fill the 33 locations starting at CS:FF00 with 100:
FI CS:FF00 FF20 100T

To fill the 257 locations starting at TEMP with the string "ZERO":
FI .TEMP .TEMP+100 "ZERO"

To zero out 256 bytes on the stack:
FI SS:SP-100 L 100 0

FLAGS

PURPOSE: To display or change 8088 flags.

FORMAT: **FLA**

REMARKS: The following 8088 flags are displayed.

FLAG NAME	SET	CLEAR
Overflow	O1	O0
Direction	D1	D0
Interrupt	I1	I0
Sign	S1	S0
Zero	Z1	Z0
Aux carry	A1	A0
Parity	P1	P0
Carry	C1	C0

FORMAT: **FLA flagname = flagvalue**

REMARKS: The flagnames as listed above are set equal to 0 or 1.

EXAMPLES: To display flags:
FLA
O0 D1 I0 T0 S0 Z1 A1 P1 C1

To set the interrupt flag:
FLA I = 1

FLOAT

PURPOSE: To display or change floating point numbers in memory. Note that this command is only available in the /87 versions of PROBE.

FORMAT: **FLO [type] [range]**

REMARKS: **[type]** specifies one of the following:

TYPE	DESCRIPTION	#BYTES
I	32 BIT INTEGER	4
L	LONG 64 BIT INTEGER	8
P	PACKED DECIMAL	10
S	SHORT 32 BIT REAL	4
R	REAL 64 BITS	8
T	TEMP REAL 80 BITS	10

If not specified, **[type]** defaults to the type used in the last FLOat command. The initial default is R. If no register or segment value is specified in **range**, then the segment value of the previous FLOat command will be used. If no length is specified in **range**, then the default length is 15.

FORMATS: **FLO [type] start address = value [,] value [,] ...**
FLO [type] start address = <enter>

REMARKS: In the first format, floating point numbers are changed to the list of values to the right of the equal sign. In the second format, the current number at the address is displayed and a prompt waits for a new data value at this location. After the new data is entered, the next location is displayed and can be changed. Typing <enter> alone on a line ends the changes. When data is written to memory it is read back for verification unless the NOVerify condition is Noread. Errors are reported accordingly.

EXAMPLES: This command stores a list of floating point numbers.

```
FLO .START = 150 E 25,1234.3334 E -66
```

This command stores a list of floating point numbers but looks at each one before changing it.

```
FLO .START =
3000:0000  9.999 e 13      - 1000.0 e 10 <enter>
3000:0009  1.0000062 e -14 - <enter>
```

These commands display floating point numbers based at the address of the variable .fahr. The number is shown in different formats.

```
-flo .fahr 1 1
0D58:00D6    6.3879640332419039 E 29
-flo i .fahr
0D58:00D6    1.7040260000000000 E 6
-flo p .fahr
0D58:00D6    2.0462020200020006 E 17
-flo s .fahr
0D58:00D6    0.0238784901696956 E-37
-flo r .fahr
0D58:00D6    6.3879640332419039 E 29
-flo t .fahr
0D58:00D6    0.4314508391370397 E-2456
```

GO

PURPOSE: To execute the program being debugged. Stops execution when a specified breakpoint occurs.

FORMAT: **G [=address,] breakpoint, breakpoint,...**

REMARKS: **[=address,]** indicates the location to start program execution. The comma is required if **[=address,]** is specified. If **[=address,]** is not included, then program execution starts at the current CS:IP. If **[=address,]** specifies only an offset, then the current code segment value is used. If a breakpoint is set at the current CS:IP, the PROBE single steps the first instruction then sets the breakpoint. Breakpoint definitions are described in the BreakPoint command. A maximum of 8 breakpoints can exist.

EXAMPLES: This command starts in the current module at line 25 and sets a breakpoint at line 37.
G =#25, #37

This command starts at the current CS:IP and stops at CS:100.
G 100

This command starts program execution at the location specified by the symbol START and sets three non-sticky breakpoints.
g =.start , io .key r, .temp w, io .porta r data 'q'

This command starts at the current CS:IP and sets a breakpoint after TEST has been read.
G .TEST R

Sticky and Non-sticky Breakpoints

Breakpoints which are defined in the Go command are activated for the duration of program execution and then are removed once control has been returned to the PROBE. These are commonly referred to as "NON-STICKY BREAKPOINTS." Breakpoints which are previously defined and assigned a break number by the BP command remain activated when program execution is terminated. This means these breakpoints will still be in effect when a new Go command is issued, even if they are not specified in the new Go command. These are commonly called "STICKY BREAKPOINTS." Sticky breakpoints are deleted with the DElete B command.

IF

PURPOSE: To allow conditional execution of macro commands.

FORMAT:
MA macroname =
M- [PROBE Commands]
M- IF boolean expression
M- [PROBE Commands]
M- [ELSe
M- [PROBE Commands]]
M- EIF
M- [PROBE Commands]
M- END

REMARKS: The **IF** command can be defined inside a macro to allow a series of commands to be executed conditionally. The end of the **IF** command is specified by **EIF**. The **ELSe** portion of the IF command is optional.

Boolean expression is defined at the start of this chapter in the section called Common Parameters and Definitions. **ELS** and **EIF** must be in the same macro which invoked **IF**.

EXAMPLE: This macro prints a message if a procedure is called with an out-of-range value.

```
MA I =
M- G .PROCEDURE_START
M- IF @W.TEMP > %0
M- PR "Temp is out of range!  Temp = ", @W.TEMP
M- EIF
M- END
```

To execute the macro:
EM I 1000

If TEMP is greater than 1000, the following warning message is issued:
Temp is out of range! Temp = 1010

INITIALIZE

PURPOSE: To restore register values which were set by the most resent program load.

FORMAT: **INI**

REMARKS: When a .EXE file is loaded, the SS:SP, CS:IP, DS and ES registers are set by the loader and operating system. It may be desirable to reinitialize the registers to these conditions when restarting program execution if a program or procedure has left the stack and registers in an indeterminate state. If a high level language program has executed its last executable statement, then it has informed the operating system it has finished, and memory has been deallocated for the program. In this case, if an INItialize command is issued, an error condition will be reported. The program must be reloaded before it can be executed. This will not be a problem if the program is stopped before executing its last executable statement or the program was written in assembly language which does not deallocate memory with the last executable statement. However, an assembly language operating system call to EXIT (INT 20H) will deallocate memory.

This command may not work properly for all programs, since only registers are re-initialized. This means that programs with load-time initialized data (such as C which guarantees that all variables are initialized to 0) will not have their data values restored. For these cases, the program must be re-loaded even if it has not yet terminated.

INTERRUPT

PURPOSE: To enable or disable interrupt masks in the 8259 interrupt controller while in PROBE software.

FORMAT: **INT [#] [state]**

REMARKS: **state** = OFF means the interrupt is masked
state = ON means the interrupt is unmasked

is the level of the interrupt on the 8259
or
is ALL for all interrupt levels

Individual interrupt lines on the system 8259 programmable interrupt controller can be masked or unmasked while the PROBE software has control of the PC. The background interrupts such as the real time clock, disk controller operations, and keyboard servicing will continue to request servicing while the PROBE software is executing. However, if the routines which service these requests are not working, they could prevent PROBE from operating. This could happen if they never return to the PROBE software. This could also happen if 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 PROBE. To prevent this from occurring, use the INTerrupt command to mask off the selected interrupts after entry into PROBE software. You would normally execute this command after you had just loaded your program and before executing it for the first time.

The INTerrupt command sets the state of the 8259 interrupt mask register while in the PROBE software. The mask change occurs as soon as the INTerrupt command is executed. This does not affect your program since the mask is always restored to the value it had when PROBE was entered from a breakpoint. 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 not changed.

2. ENTERING PROBE SOFTWARE. When PROBE software is entered from a breakpoint in the users program, it sets the mask to the state which was specified by the INTerrupt command. If no INTerrupt command has ever been issued, it sets the mask to the state it was in when the 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 cannot be executed. The following are the interrupt #'s which you use for this command:

Interrupt Level	Description
0	Timer
1	Keyboard
2	Reserved
3	Com2
4	Com1
5	Fixed disk
6	Floppy disk
7	LPT1

FORMAT: **INT**

REMARKS: Displays the state of the interrupt mask. 'On' means the level is unmasked, 'Off' means masked.

EXAMPLES: To display the state of the interrupt mask:
INT
0 On, 1 On, 2 Off, 3 Off, 4 Off, 5 On, 6 On, 7 Off

To unmask the COM1, interrupt while in PROBE with:
INT 4 ON

To mask all interrupts off while in PROBE:
INT ALL OFF

To reset the interrupt mask to the state it was in when PROBE was first started:
INT ALL Default

LIST

PURPOSE: To copy the output of PROBE commands to another list device. This gives you a history of your debugging session.

FORMAT: **LI**

REMARKS: Displays the current list device, if one is active.

FORMAT: **LI = {filespec | LPT1: | COM1:}**

REMARKS: All PROBE output is copied to LPT1, COM1, or a file. The output continues to be sent to the current PROBE console.

FORMAT: **LI Close**

REMARKS: Stop sending a copy of the PROBE output to the list device. If the list device was a file, it must be closed before exiting PROBE and going back to DOS or the information in this file will be lost.

EXAMPLES: To set the list device to the line printer:
LI = LPT1:

To display the current list device:
LI
LPT1:

To end list output:
LI C

LOAD

PURPOSE: This command loads application programs, symbol tables, or macro files.

FORMAT: **LOA filespec [optional parameters]**

REMARKS: This command loads the applications program named by **filespec** into PC memory. If the file is a .EXE file, the PROBE establishes the load address and initializes the necessary CPU registers. If the file is one produced by the SAve command, then the file is not in the .EXE format and is loaded at CS:100. **[Optional parameters]** in the load command are passed to the loaded program in the program prefix segment of the loaded program in the normal manner. *The load command should be used to load the program before the LOAd S command is used to load the symbol table.*

FORMAT: **LOA Symbols filespec [address [Absolute]]**

REMARKS: This command loads the internal symbol table space of the PROBE with information from **filespec**. This file is one produced by the linker and is the link map. When the symbols are loaded, their values are adjusted to match the previously loaded applications program. *Therefore, the applications program should normally be loaded before the symbol table is loaded.* If the symbol table is loaded before a program is loaded, the symbol's segment value will be set as though the program was loaded at 0. If the symbol table already contains symbols, loading a new symbol file will add to the current symbols. If it is desired to remove these symbols, use the DElete Symbols ALL command.

If **[address]** is included in this LOAd command, then this is also added to the symbols at load time. This form of the LOAd symbol command should be used for .COM files. Each symbol is determined by the equation:

program load address + address + symbol

If **[address [A]]** is included in this LOAd command, then the address is interpreted as an absolute location. Here, the symbols are determined by:

address + symbol

This is very useful for attaching symbols to loaded device drivers, quit and stay resident programs, or programs running on non-DOS operating systems.

FORMAT: **LOA Mac filespec**

REMARKS: This command loads the internal macro table space of the PROBE with macros which are defined in the file. This file could have been created with the PROBE previously and saved with the SAve command or could have been created with a text editor. The format of the macro file is the same as that of the macros themselves. Loading macros adds to the macros which have already been defined or loaded into PROBE.

EXAMPLES: These commands load a standard .EXE file and pass a parameter called LPT1 to the program. Then a symbol table file is loaded.

```
LOA DUMP.EXE LPT1:
LOA S DUMP.MAP
```

This command loads a .COM file. When a symbol table for a .COM file is loaded, it must be adjusted for a relative offset of negative 100H bytes.

```
LOA DUMP.COM
LOA S DUMP.MAP 0-10:0
```

This command loads the symbol table for an installed device driver or a resident program in memory which has its starting code segment at 0632.

```
LOA S MYDRIVER.MAP 0632:0 A
```

This command loads the standard PROBE macro file.

```
LOA M PROBE.MAC
```

LOGIC

PURPOSE: To change the heading for logic signals in the display of the trace command.

FORMAT: **LOG [I/E]**

REMARKS: If the I is selected the heading for the trace indicates bus interrupt request lines. If the E is selected the heading for the trace indicates external logic signals. If nothing is selected the current setting for the heading is displayed.

LOOP

PURPOSE: To allow command sequence loops to be executed within a macro. The macro can be exited conditionally.

FORMAT:
-MA macroname =
M-[PROBE Commands]
M-LOO [count expression|While boolean expression]
M-[PROBE Commands]
M-ELO
M-[PROBE Commands]
M-END

REMARKS: The LOOp command can only be executed inside a macro. All of the PROBE Commands between **LOOp** and **ELO** are executed. Each time the LOO command is executed, the expression for continuing execution of the loop is tested. There are two types of loop expressions: count expressions and while boolean expressions.

count expression. This is simply a number. If no number is specified, then the loop continues forever until the STOP button is pressed.

while boolean expression. Boolean expressions were defined previously in the section called "Common Parameters and Definitions." The loop continues while the boolean expression is True.

Loops can be nested 5 deep; however, ELO must be within the macro which invoked the loop and not in a nested macro.

EXAMPLES: This is an example macro that counts the number of breakpoints before stopping. First define the loop inside the macro.

```
MA L =
M- LOO %0
M- Go %1
M- ELO
M- END
```

EXAMPLES, continued

To execute the program until the 10th time it hits the breakpoint at line 58:
EM L 10T, #58

This macro causes program execution to continue as long as the value of TEMP is not equal to the value you pass the macro as a parameter.

```
MA L2 =
M- LOO While @W.TEMP <> %0
M- G .PROCEDURE_START
M- ELO
M- END
```

To execute the program until PROCEDURE_START is executed and the word at .TEMP is equal to 100:
EM L2 100T

This macro displays the value of TEMP and the first 5 bytes of the stack each time the procedure start is executed.

```
MAC L3 =
M- LOO
M- GO .PROCEDURE_START
M- W .TEMP
M- W SS:SP L 5
M- ELO
M- END
```

Now each time PROCEDURE_START is executed, the word at TEMP and the 5 words on the top of the stack are output.

MACRO COMMANDS

PURPOSE: User commands can be created through the definition of macros. These macros can be saved and included in future debugging sessions.

FORMAT:
MA macroname =
M- COMMAND[%parameternumber]
M- COMMAND[%parameternumber]
M-....
M- END

REMARKS: **Macroname** has a maximum of 20 ASCII characters. PROBE commands can be entered until the END terminator is specified. Up to 10 parameters with parameter numbers ranging from 0 to 9 may be passed to the macro by specifying **%parameternumber** in the COMMANDS where the parameter is to be substituted. Parameters may be any ASCII string. Macros may be nested within macros up to a level of 5. See notes on nesting later in this description to determine how to pass parameters to nested macros. Since macro names are distinguished from symbol names by the way they are used, there is no conflict in defining macros which have the same name as a symbol. Maximum length of a macro is 255 characters. Note that the M- in the macro definition is printed as a prompt by the PROBE and only the **COMMANDS** are entered to define the macro.

FORMATS:
MA macroname
or
MA <enter>

REMARKS: The first command displays the definition for **macroname**. The second command displays all macro names.

FORMAT: **EM macroname parameter, parameter,....**

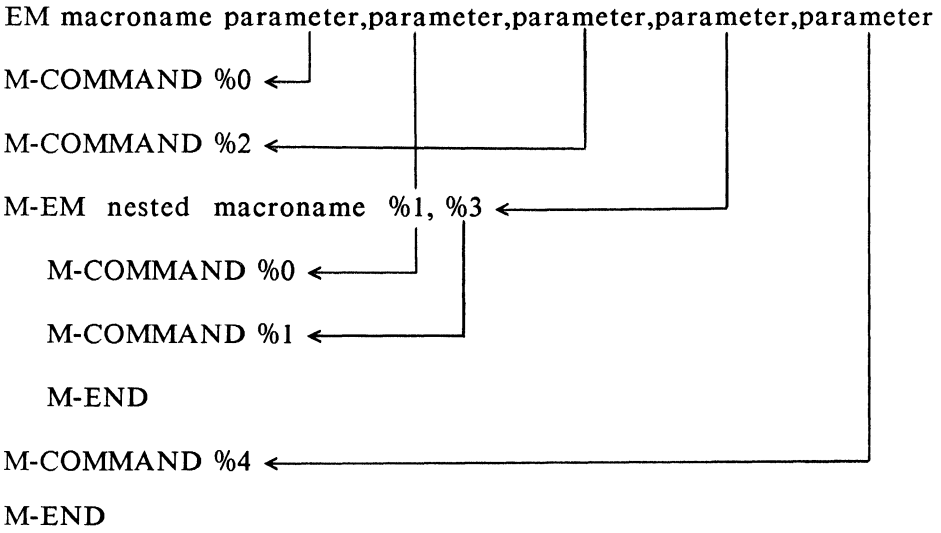
REMARKS: This command invokes the macro and passes parameters to the macro if they are included in the command. The parameters are assigned parameter numbers of 0 to 9 from left to right in the EM command. The **parameter** can be a string and is separated from the next parameter by a comma. The parameters are then substituted in the macro for the parameter number specified in the macro definition.

INITIALIZATION MACROS

When PROBE is first loaded, the current path name is searched for the file INIT.MAC. If found, this file is loaded automatically. If the file defines a macro called INITIALIZE, then that macro is automatically executed. It is useful to create this custom file and macro to set up for a debugging session. A sample initialization macro is included on the PROBE diskette.

NESTING MACROS

Macros can call other macros or can be recursive and call themselves to a depth of 5. The user can create a modular set of macro primitives which can be used in the definition of other macro commands. This can also be used to save space in the macro table if many different macro commands have common parts. The parameters within a macro have a local scope and are substituted from the command line invocation of the macro. Parameters can be passed to nested macros by specifying the parameter number of the outer macro in the invocation line of the inner macro. The following diagram describes the flow of parameters between macros.



MACRO COMMAND EXAMPLES

INITIALIZATION MACRO

This is an initialization macro called INIT which could save many user keystrokes when starting up a debugging session.

```
MAC INIT =
M-LOA %0.EXE
M-LOA S %0.MAP
M-R
M-U
M-END
```

This macro definition can now be displayed by typing:

```
MAC INIT
```

This macro can now be used to start a debugging session by loading the program b:myprog.

```
EM INIT B:MYPROG
```

GO FROM STACK ADDRESS

This macro starts program execution from the current return address on the stack.

```
MAC QW=
M- G @PSS:SP
END
```

PRINT A STRUCTURE

This macro prints out an element of a complex array of structures. Here is an example C structure.

```
# define NameSize 50
struct E_Rec {
Int EmpNum;
char EmpName [Name Size];
};
#define NumEmpRecs 1000
struct E_Rec EmpRecs [NumEmpRecs];
```

This macro is defined to print the nth element of the structure.

```
Mac EMPRECS =
M- PRI "Employee name=",%S .EmpRecs+(%0t*52)+2
M- PRI "Employee # =", %D @W .EmpRecs + (%0t*52)
M- END
```

This macro can now be used to print the 10th employee record.

```
EM EMPRECS, 10
```

TRAP A MEMORY OVERWRITE

This macro solves a very common debugging problem. The problem is that a memory overwrite occurs in a data area which is commonly written to. The chore is to find the overwrite which is not supposed to be happening. This can be done by checking the code segment of the instruction which did the overwrite to ensure that it is in a part of the program which is allowed to write to this data area. This macro starts program execution and sets a breakpoint on writing to the variable .FOO. The macro then checks the CS register to see if it is equal to the 0C9A. If it is, then program execution continues. If it is not, then program execution stops.

```
MAC OVERWRITE =
M- G .FOO W
M- LOO W CS = 0C9A
M- G .FOO W
M- ELO
M- END
```

NESTING MACROS

This example shows passing parameters between macros.

```
MAC MAC1 =
BY %0
BY %2
EM MAC2 %1, %3
BY %4
END

MAC MAC2 =
BY %0
BY %1
END
```

To execute MAC1 and pass parameters to it:
EM MAC1 0,1,2,3,4

This macro expands as follows:
BY 0
BY 2
BY 1
BY 3
BY 4

STANDARD MACROS SUPPLIED BY ATRON

The following list of macros are contained on your PC PROBE diskette in a file called PROBE.MAC. Following the list is a an explanation of each macro and how it is used.

SECTOR READ MACRO
SECTOR WRITE MACRO
DISPLAY PROGRAM PREFIX SEGMENT MACRO
CHAINED BREAKPOINTS
DETECT A STACK SEGMENT CHANGE BREAKPOINT
DEMO SET UP MACRO
DISPLAY REGISTERS AND STACK DATA MACRO
TRAP INT21 MACRO

SECTOR READ MACRO

This macro reads 1 sector from the disk into system memory at location 2000:100. You may want to change this address or make it a parameter.

```
MAC AbsRead=
r cs=2000
r ip=0
r ds=cs
asm cs:ip
mov al, 0%0
mov dx, 0%1
mov cx, 1
mov bx, 100
int 25
jmp d

g cs:d
by ds:100 1 100
END
```

This macro is executed with:

```
em AbsRead DriveNumber, SectorNumber
```

DriveNumber is a parameter which specifies which drive is to be read. SectorNumber specifies which sector is to be read.

SECTOR WRITE MACRO

This macro writes 1 sector from the disk into system memory at location 2000:100. You may want to change this address or make it a parameter.

```
MAC AbsWrite=
r cs=2000
r ip=0
r ds=cs
asm cs:ip
mov al, 0%0
mov dx, 0%1
mov cx, 1
mov bx, 100
int 26
jmp d

g cs:d
END
```

The macro is executed with:

```
em AbsWrite DriveNumber, SectorNumber
```

DISPLAY PROGRAM PREFIX SEGMENT MACRO

This macro displays the data in the Program Prefix Segment. The ES register must be set to the value it had when the program was loaded (start of PPS).

```
MAC PPS=
ec off
pr "MemTop              = ", @wes:2
pr "Terminate Address  = ", @wes:c,':',@wes:a
pr "CtrlC Address      = ", @wes:10,':',@wes:e
pr "Hard Error Address = ", @wes:14,':',@wes:12
em PPS2
END
MAC PPS2=
pr "FCB 0 ="
by es:5c 1 10
pr "FCB 1 ="
by es:6c 1 10
pr "Command line = "
by es:81 1 @bes:80
ec on
END
```

This macro is executed with:

```
em PPS
```

CHAINED BREAKPOINTS

This macro lets you detect the sequential execution of three separate breakpoints. This macro is set up to restart the sequence detection when a fourth breakpoint is detected. The parameters passed to the macro are the segments and offsets for each of the breakpoints. For example Aseg and AOffset are the segment and offset values respectively for the first breakpoint in the chain.

```
MAC AArmsBArmsCResetD=
sy .AtC=0:0
loop w .AtC=0
        g %0:%1
        g %2:%3, %6:%7
        if cs=%2
        if ip=%3
                g %4:%5, %6:%7
                if cs=%4
                if ip=%5
                        sy .AtC=0:FFFF
                eif
                eif
        eif
        eif
elo
END
```

The breakpoint is executed with:

```
em AArmsBArmsCResetD ASeg,AOffset, BSeg,BOffset,
CSeg,COffset, DSeg,DOffset
```

DETECT A STACK SEGMENT CHANGE BREAKPOINT

This macro single steps a program until the stack segment is about to be loaded.

```
MAC SS=
ec off
st
loop w ((@wcs:ip) & 38ff) <> 108e
st
elo
ec on
END
```

It is executed with:

```
em ss
```

DEMO SET UP MACRO

This macro loads the demo program and does the proper assignments for source level debugging.

```
MAC INITIALIZE=
asi ftocm_code %0ftocm.c
asi ftocio_code %0ftocio.c
loa %0ftocnew.exe
loa s %0ftocnew.mpl
symbol .start = cs:ip
wi = mem
END
```

It is executed with:

```
em initialize
```

DISPLAY REGISTERS AND STACK DATA MACRO

This macro displays all registers and stack data. It is useful to assign this macro to the window during assembly language single stepping.

```
MAC RR =

pr '-------------------------------------------------------------'
r
wo ss:sp
END
```

This is done with the following command:

```
WI = RR
```

TRAP INT21 MACRO

This macro sets a breakpoint at the first instruction of the INT 21 system call.

```
MAC 21=
loo %0
go @p0:21*4
r
elo
END
```

It is executed with:

```
em 21
```

MENU

PURPOSE: To enable or disable the display of commands in the menu window.

FORMATS: **ME ON**
or
ME OFF

REMARKS: The first command prompts the user with a display of PROBE commands in the menu window. The second command disables the display of the menu window. The menu window is defined as lines 23 and 24 on the console. When switching to an external console, the menu is initially off, and must be turned on.

MODULE

PURPOSE: Assigns the current default module name prefix for line numbers in the symbol table.

FORMAT: **MOD modulename**

REMARKS: The module command assigns the default **modulename** for line numbers. If a line number is used as a symbol and no modulename is specified then the default **modulename** is used. The default modulename after a LOA S command is the first modulename in the symbol table. Specifying the modulename along with the line number overrides the default module name.

FORMAT: **MOD**

REMARKS: The current default module selection and status is displayed followed by the names of all modules in the symbol table. A module is the result of a single compilation. See Chapter 3 and the ASIGN command in this chapter for more information on naming modules.

EXAMPLES: To assign the default modulename to MAIN:
MOD MAIN

Now line numbers in the module MAIN may be described without specifying the modulename as follows:
SY #233

To override the default modulename specify the module as demonstrated by this general format and specific example:
SY ..modulename#linenumber

SY ..SECOND#233

MORE

PURPOSE: To display alternate fields of commands in the menu window.

FORMAT: **MOR**

REMARKS: There are two sets of commands which may appear in the menu window. The MORe command pulls up the other set of commands for display in this window. This set continues to be the default set of displayed command prompts until another MORe command is executed. This is simply a display function and all commands are available from either menu.

MOVE

PURPOSE: To move a block of memory to a new location.

FORMAT: **MOV range destinationaddress**

REMARKS: The block of memory specified by **range** is moved to a location starting at **destinationaddress.** The memory is moved on a byte by byte basis starting with the first address of range. Each location is read back after it has been written and error conditions are reported unless the NOVerify condition is set to NOread. The MOVe command uses the DS segment value unless there is another segment register or segment value specified.

EXAMPLES: To move the 100 locations starting at DS:200 to the locations at 1000:100:
MOV 200 L 100T 1000:100

To move the 32 locations starting at SS:SP to the location starting at ES:DI:
MOV SS:SP L 20 ES:DI

To move the 257 locations starting at TEMP to NEWTEMP:
MOV .TEMP .TEMP+100 .NEWTEMP

To move the 32 locations starting at LOOP to 100H in the CS segment:
MOV .LOOP L 20 CS:100

NEST

PURPOSE: To display the calling sequence of procedures by analyzing the stack.

FORMAT: **NE [S]**

REMARKS: The BP is assumed to point to its old value. If the [S] option is not used, the return address is assumed to be located directly under the old BP value on the stack (i.e. each procedure begins with PUSH BP; MOV BP,SP). If the [S] option is used, the procedure is assumed to start with PUSH BP; SUB SP,xx; MOV BP,SP. In this mode, memory is searched from CS:IP to a PUSH BP instruction. The SUB SP,XX instruction is emulated to get back to the return address. The memory search continues backwards from the return address to another PUSH BP instruction. Do not use the S option with the Microsoft Pascal or C compilers. Use the S option with the Lattice C compiler.

EXAMPLE:
```
NE
CS:IP IS 0624:01CC NEAR ..MEMORY_TESTER#103
CALLED FROM 0624:020D NEAR
..MEMORY_TESTER#114
CALLED FROM 0780:00FC NEAR ..START
```

NOBREAK

PURPOSE: This command is used to disable the termination of execution when a break condition has been encountered. This applies only to hardware versions.

FORMATS: **NOBreak Stop**
NOBreak Continue

REMARKS: If NOBreak = Continue when a non-execution break condition is encountered, a pulse is transmitted on test pin 1 on the PC PROBE and program execution continues. This provides an external trigger for an oscilloscope or logic analyzer when the program is executing in a loop. The nobreak condition is always Stop for all execution breakpoints. This command applies to the 3 hardware breakpoints as a group. It is not possible to have some hardware breakpoints in the Stop mode while others are in the Continue mode.

FORMAT: **NOBreak**

REMARKS: Displays the current status of the NOBreak.

EXAMPLES: To set the nobreak condition to Stop:
NOB S

To set the nobreak condition to Continue:
NOB C

CAUTION

The NOBreak condition can only be used if the following conditions are true while the applications program is running.

1. No execution breakpoints have been set.
2. The program may not do an exit system call.
3. No parity errors occur.
4. No erroneous access to PC PROBE memory occurs.
5. No user generated NMIs occur.

NOVERIFY

PURPOSE: To disable or enable the read after write verification for certain commands.

FORMATS: **NOV Read**
NOV Noread

REMARKS: The following commands are affected by the NOVerify command:

BYte
WOrd
PT
MOVe
FIll

If the NOVerify condition is **Read** (which is the default), then a read after write verification is performed when the commands above, change memory. If the value read does not match the value written, then an error is reported. If the NOVerify condition is Noread, then no check is performed. This command is useful if memory mapped Input/Output devices are being addressed with these commands. They may function incorrectly if the read after write is performed; therefore, you may want to set the NOVerify condition to **Noread** for these cases.

FORMAT: **NOV <enter>**

REMARKS: Displays the current status of the NOVerify condition.

EXAMPLES: To set the NOVerify condition to Noread:
NOV N

To set NOVerify condition to Read:
NOV R

NUMERIC FLAGS

PURPOSE: To display or change 8087 flags. The 8087 must be present for this command to operate. This command is only available in the /87 versions of PROBE.

FORMAT: **NF [C|S]**

REMARKS: This command displays all the 8087 flags, which include control and status words, instruction address, opcode, and operand address. If the **[C]** or **[S]** option is specified, then only the control word or status word respectively are displayed.

FORMAT: **NF {C|S} = [value]**

REMARKS: If **value** is specified, the selected control or status register is given the new value. If **value** is not specified, the current status or control word is displayed, and you are prompted for a new value.

EXAMPLES: To display all 8087 flags:

```
NF
Control word=03FF
Status word=4100
Instruc addr=00000.  Opcode=0000
Operand addr=00000
```

To change the control word to 0FFF:

```
NF C = 0FFF
```

NUMERIC REGISTERS

PURPOSE: To display or change 8087 registers. The 8087 must be present. This command is only available in the /87 versions of PROBE.

FORMAT: **NR [{S | T} (#)]**

REMARKS: **S** or **T** selects the stack or tag register # of the 8087 for display. If the **[{S | T} (#)]** is not specified, then all 8087 registers are displayed.

FORMAT: **NR {S|T} (#) = [value]**

REMARKS: The specified 8087 stack or tag register is changed to **value**. If **[value]** is not specified, then the register is displayed along with a prompt. A value may be entered at this point.

EXAMPLES: To display all 8087 registers:

```
NR
ST (0)= 3.141592 E 00 Tag 0=Valid=0
ST (1)= 0.000000 E 00 Tag 1=Zero=1
ST (2)= ************ Tag 2=Empty=3
ST (3)= ************ Tag 3=Empty=3
ST (4)= ************ Tag 4=Empty=3
ST (5)= ************ Tag 5=Empty=3
ST (6)= ************ Tag 6=Empty=3
ST (7)= ************ Tag 7=Empty=3
```

To set tag register 4 to a 2:

```
-nr t (4) =2
```

To display tag register 4:

```
-nr t (4)
Tag 4=NAN/Infinity/Denormal=2
```

To display stack register 0:

```
-nr s 0
ST (0)=-0.1859660090  E 16363
```

POINTER

PURPOSE: To display or change 32 bit pointers in memory.

FORMAT: **PT [range]**

REMARKS: The pointers in memory specified by **range** are displayed in hex. If no segment register or segment value is specified in **range**, then the segment value from the previous PoinTer command will be used. If no length is specified in **range** then length defaults to the previous length. If no **range** is specified at all, then a block of pointers starting at the end of the previous block is shown with a length equal to the previous length. The PgDn key will also display the next screen full of pointers.

FORMATS: **PT start address = value [,] value [,] value...**
or
PT start address = <enter>

REMARKS: In the first format, pointers in memory are changed to the list of values to the right of the equal sign. In the second format, the current pointer at the address is displayed, and a prompt waits for a new data value at this location. After the new data is entered, the next location is displayed and can be changed. Typing <enter> alone on a line ends the changes. When data is written to memory it is read back for verification unless the NOVerify condition is Noread. Errors are reported accordingly.

EXAMPLES: This command stores a list of pointers.
PT .START = 0000:0000, FFFF:0000, AEF0:0008

To display this data:
PT .START L 3
0624:0003 0000:0000 FFFF:0000 AEF0:0008

To display the next 3 pointers:
PT
0624:000F AACC:DDDD AEFF:AACE AEFF:C0C0

Registers can also provide the start address.
PT SS:SP = 3000:AEF0

PORT

PURPOSE: To display or modify the contents of an IO port.

FORMAT: **PO [Word] portnumber**

REMARKS: Display the byte contents of the IO port addressed by **portnumber**. If **Word** is specified in the command, then a word read occurs.

FORMAT: **PO [Word] portnumber = value**

REMARKS: The byte value is written to the port addressed by **portnumber**. If **Word** is specified, then a word write occurs.

EXAMPLES: To display the port addressed by the symbol PORT5:

```
PO .PORT5
AE
```

To change the port:

```
PO .PORT5 = AA ;in hex
PO .PORT5 = 'A' ;in ASCII
PO .PORT5 = 10T ;in decimal
```

PRINT

PURPOSE: To print information in a formatted fashion on the console.

FORMAT: **PR {'string' | [type] expression},.....**

REMARKS: The PRint command can be used much like a high-level language WRITE statement. It accepts strings in either single quotes (') or double quotes (") and echoes them to the console. It evaluates expressions which are not within quotes and prints their value on the console in one of the data types specified below:

%a	display expression as ASCII
%b	display expression as a hex byte
%w	display expression as a hex word ** Default**
%d	display expression as an unsigned decimal
%i	display expression as a signed decimal integer
%s	expression is an address. The data at the address is displayed as ASCII until a terminating 0.

Note that **expression** is treated as a value in all cases above except the %s case. The @ operator can be used to get the expression which is pointed to by an address. The examples will demonstrate this.

EXAMPLES: To print the word pointed to by the symbolname .FAHR:

```
PR "FAHR = ", @w .Fahr
FAHR = 2000
   where:    .FAHR = 1000:10
             1000:10 contains the word 2000
```

To print the word pointed to by the pointer at the symbolname .FAHR:

```
PR "FAHR = ", @w@p .Fahr
FAHR = 0010
   where:    .FAHR = 1000:10
             1000:10 contains the pointer 2000:20
             2000:20 contains the word 0010
```

EXAMPLES, continued

To print the integer pointed to by the pointer at the symbolname .FAHR:

```
PR "FAHR = ", %i @w@p .Fahr
FAHR = 16
   where:    .FAHR = 1000:10
             1000:10 contains the pointer 2000:20
             2000:20 contains the integer 16
```

To print the word pointed to by the symbolname CELCIUS as an integer:

```
PR 'CELSIUS = ', %i @w.celsius
CELSIUS = -17
   where:    celcius = 3000:30
             3000:30 contains the word FFEF
             FFEF expressed as an integer is -17
```

To print the 0 terminated string which begins at BUFFER:

```
PR "The buffer is:  ", %s .buffer
The buffer is:  Hello, world.
   where .buffer is 3000:30
             3000:30 contains "Hello,world"

PR 'THE OTHER BUFFER IS: ',%s @p.buffer
THE OTHER BUFFER IS: Whole nuther world
   where:    .buffer =2000:10
             2000:10 contains 4000:87
             4000:87 contains "Whole nuther world"
```

To print the third byte of a structure, where the structure is pointed to by the pointer in .STRUCTURE:

```
SY .TEMP = @P.STRUCTURE
PR "STRUCTURE BYTE=",%b @b.TEMP+2
STRUCTURE BYTE IS = 22
   where:    STRUCTURE = 8000:50
             8000:50 contains 3000:10
             3000:12 contains 22
```

EXAMPLES, continued

To print the ASCII char at .START:

```
PR "START =',%a
p
```

where: .START = 8000:10
8000:10 contains the byte 50

QUIT

PURPOSE: To leave the PROBE and return to the operating system.

FORMAT: **Q**

REMARKS: PROBE restores the user's screen and returns control to the operating system. This also removes the PROBE program prefix segment and restores all interrupt vectors to their original value.

FORMAT: **Q R**

REMARKS: PROBE returns control to the operating system but the program prefix segment and interrupt vectors 1, 2, and 3 remain in memory. Note that there is a space between the Q and R.

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

Q R is useful whenever the PROBE 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 PROBE LOAd command cannot be used to load a program after a Q R. It can, however, be used to load a symbol table.

Sticky breakpoints are not set with the Q R. They are only set with the Go command. Therefore, after doing a Q R, press the STOP button then do a Go command to continue execution with sticky breakpoints set.

REGISTERS

PURPOSE: To display or change cpu registers.

FORMAT: **R [registername]**

REMARKS: The contents of the specified 8088 register as described by the registernames below are displayed. If no **registername** is specified, then, all registers are displayed.

FORMAT: **R registername = [value]**

REMARKS: The specified register is changed to **value**. If **value** is not specified, then the register is displayed along with a prompt to let you change the value. The 8088 register names are as follows:

AX	CS	SS	DS	ES	AH	AL
BX	IP	SP	SI	DI	BH	BL
CX	BP			CH	CL	
DX	FL			DH	DL	

EXAMPLES: To display all registers:
R

```
AX= 1234   CS= 0000   SS= 1000   DS= 0011   ES= 0100
BX= 0104   IP= 1000   SP= 5000   SI= 0000   DI= 0000
CX= 0002              BP= 4000
DX= ABAB              FL= O0 D1 E1 S0 Z1 A1 P0 C0
```

To change the IP register to 2000:
R IP =
IP = 1000 - 2000 (you enter the 2000 in response)

To display the AX register:
R AX
AX = 1234

To move the value of a register into another register:
R AX = BX

To put an ASCII value into a register:
R AL = 'Q'

SAVE

PURPOSE: To save blocks of memory or macros in a file.

FORMAT: **SA range filespec**

REMARKS: The block of memory specified by **range** is written to the file specified by **filespec**. The SAve command uses the CS segment register unless there is another segment register or segment value specified. The file is saved as a direct memory image and is not a .EXE or .HEX file.

FORMAT: **SA M filespec**

REMARKS: The macros which are currently active for this debugging session are saved in filespec. The definition of active macros are those which were created with the MAcro command or loaded with the LOAd command and are recognized when their macro names are specified during the debugging session. If it is desired to save symbols, a macro can be created that defines the symbols and this macro can be saved.

EXAMPLES: To save the 100 locations starting at CS:200 to a file:
SA 200 L 100T B:myprog.hex

To save the 32 locations starting at DS:FF00 to a file:
SA DS:FF00 FF1F myprog.hex

To save the 256 locations starting at TEMP:
SA .TEMP .TEMP+FF myprog.hex

To save the 32 locations starting at LOOP:
SA .LOOP L 20 myprog.hex

To save the on-line macros in a file:
SA M myprog.mac

SCREEN

PURPOSE: Screen switching isolates the application screen from PROBE's screen by creating two virtual screens.

FORMAT: **SC**

REMARKS: Displays the current state of screen switching.

FORMAT: **SC {ON | OFF}**

REMARKS: Enables or disables screen switching in Go, STep, and Source Step commands. If screen switching is enabled, the application screen is displayed whenever your program is executing. In this manner, output from the program is not confused with output from PROBE. When PROBE has gained control from the application program after a breakpoint or after each single step, the application screen is saved and the PROBE screen is displayed.

FORMAT: **SC Show**

REMARKS: Shows the current user screen. After a breakpoint has been reached, this command may be used to display the current state of the application screen. The application screen remains displayed until any key is pressed.

Screen switching commands 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 PC console.

EXAMPLES: To enable screen switching in Go and STep commands:
SC ON

To display screen switching state:
SC
Switch screens

This command displays the application screen. Once it is displayed, type any key to return to the PROBE screen.
SC S

SEARCH

PURPOSE: To search a block of memory for a list of values.

FORMAT: **SEA range list**

REMARKS: The block of memory indicated by **range** is searched for a list of values. The list of values may be in any data type. If no segment register or segment value is specified for **range**, the DS segment value is used. All locations within **range** which match **list** are displayed. If no match is found the message "NO MATCH FOUND" is reported.

EXAMPLES: To search the 100 locations starting at DS:200 for A5:
SEA 200 L 100T A5

To search the 32 locations starting at CS:FF00 for 100T:
SEA CS:FF00 FF1F 100T

To search the 256 locations starting at TEMP for the string "ZERO":
SEA .TEMP .TEMP+FF "ZERO"

To search the stack for the string 'STACK':
SEA SS:SP-FF L FF 'STACK'

SELECT

PURPOSE: To limit the number of linenumbers loaded from the MAP file into the PROBE symbol table to those in selected modulenames.

FORMAT: **SEL [..modulename1, ..modulename2...]**

REMARKS: The linenumbers for the specified **modulenames** are selected during the loading of the symbol table. All linenumbers for all other modules are not loaded. If **modulename** is not specified in this command, then the command displays all currently selected modules. This command limits only linenumbers and not symbolnames since they are associated with a modulename.

EXAMPLE: Using the following command, the symbol table will only receive lines from the three specified modules.

SEL ..main, ..consoleio, ..intprocedure

The selected modules can now be displayed.

SEL <enter>
..main
..consoleio
..intprocedure

FORMAT: **SEL ALL**

REMARKS: The linenumbers for all modules are selected. This is the initialization default.

EXAMPLE: SEL ALL

SOURCE STEP

PURPOSE: To single step program execution by source statements. This command applies to PC SOURCE PROBE.

FORMAT: **SS [= start address] [A] [M modulename]**

REMARKS: Source lines are displayed on the screen and executed one at a time as the <enter> key is pressed. If start address is not specified, then the current CS:IP is used. Typing <enter> will cause the program to execute the displayed line.

Before source level stepping of a program is started, the ASIgn command should have been used to assign modulenames to sourcefiles. You only need to make the assignments for files you want to step through. Assignments can normally be done with an initialization macro, while single stepping. If the current modulename is not assigned to a file with the ASIgn command, the source line is not displayed but the symbol value (..modulename#linenumber) will be displayed.

Several lines of upcoming source code are displayed with each single step. If the **A** option is specified, the stepping will occur automatically. Single stepping can be terminated by pressing any key (except <enter>) or the STOP button. If **M modulename** is specified, only lines in the specified modulename are stepped, all other code executes at full speed.

While you are in the source step command, you can use the keys below to move around in the file you are currently in.

PG UP (^U)	display the previous page
PG DN (^D)	display the next page
<Ctrl> PgUp (^T)	display the top page of the file
<Ctrl> PgDn (^B)	display bottom page of the file
HOME (^H)	reposition screen to the current CS:IP
<enter>	reposition screen to current CS:IP and take a single step
up arrow (^P)	no effect
down arrow (^L)	no effect
<any other>	exit step mode.

EXAMPLES: To start stepping the program at the source level from location .MAIN:
SS =.MAIN

To source step only through module FTOCM_CODE:
SS M ..FTOCM_CODE

To automatically step each line:
SS A

Notes on Source Level Single Stepping

For source level single stepping, the modulenames and linenumbers for each module must be in the symbol table for the module which is to be single stepped. See Chapter 3 for a discussion of the compiler and linker controls which generate symbol table information.

If the line numbers for a particular module are not in the symbol table, and source level stepping enters that module, then the program will run real time until it returns to a module which has linenumbers and modulenames in the symbol table. When source level single stepping through portions of a runtime library or assembly language which also does not have linenumbers, the program will run real-time until it returns to an area which has the linenumbers.

If the program to be single stepped is written in Lattice C, review Single Stepping by C Source Statements in Chapter 2 before proceeding.

STEP

PURPOSE: To single step program execution at the assembly language level.

FORMAT: **ST [=start address] [P] [O] [A]**

REMARKS: If **[=start address]** is not specified, then the first step starts at the current CS:IP. The command starts by displaying the next several instructions. The cursor is positioned to the right of the instruction to be executed. Typing the enter key will cause the program to execute the displayed instruction.

Typing **P** will treat a call procedure as a single step. Typing **O** will treat a software interrupt procedure as a single step. If the **P** or **O** option is used when the STep command is invoked, then the options apply globally to all procedures. The **P** and **O** options are implemented by setting a software breakpoint after the next instruction to be stepped. If while stepping with the **P** or **O** option, the program starts running and single stepping fails, it is probably because the program did not return to the instruction after the single stepped instruction.

Typing PgDn will single step a screen full of instructions. Typing any other key will cause the program stepping to stop and will not execute the instruction to the left of the cursor. The current CS:IP location and instruction are displayed after each step. The contents of the operands for each instruction are displayed at each single step so you don't have to exit the command to view the contents of the operands.

If the **A** option is specified, single stepping occurs automatically on the screen until any key is pressed.

A hardware breakpoint on reading CS:IP is used to take each single step.

If the Window command has a window currently assigned, the window is updated after each single step.

EXAMPLE: To execute a single step starting at location 100:20 and continuing until a non-enter key is typed:

```
       -st
0671:0010 MOV   SI,WORD PTR [BP+0114] -- ,00E6
0671:0014 MOV   AL,BYTE PTR [SI]      -- ,8A
0671:0016 XOR   AH,AH
0671:0018 TEST  AX,AX
0671:001A JNZ   001F   ;..FTOCIO_CODE#57+0198 -will jump
0671:001C JMP   0117   ;..FTOCIO_CODE#57+0290
```

SYMBOL

PURPOSE: To define or display the values of the symbols which were loaded into the PROBE symbol table space.

FORMATS: **SY .symbolname = symbolvalue**
SY [..modulename]#linenumber = linenumbervalue

REMARKS: Symbols can be used in place of address values in PROBE commands and expressions. Symbols are variable length ASCII strings up to a maximum of 80 characters. If no segment value is specified when defining a symbol, then the DS segment value will be used. The symbols are stored in protected memory on the PC PROBE card. The number of symbols which can be stored is determined by the sum total of the length of each symbol as well as how many macros exist since they are stored in the same area in memory. Symbols may be defined in terms of other symbols. If a symbol already exists, its value is changed in the symbol table, and the previous symbol value and a warning message are displayed. A symbol can be set to a 16 bit value by defining it with a segment value of 0.

If the symbol table overflows, the user may allocate more symbol table space in system memory with the T parameter in PROBE.CFG. See Appendix C.

FORMATS: **SY .symbolname**
or
SY [..modulename]#linenumber

REMARKS: The value of the symbol represented by the symbol name or the line number is displayed. The line numbers of a high level language program as they are passed from the LINK.MAP file are also symbols in the symbol table. Since all modules may have duplicate line numbers, the modulename is also stored in the PROBE symbol table. Two periods preceding the module name specifies the name as a module name, otherwise the default module name is used. Lower case ASCII is treated as uppercase in symbol table searches. A wildcard character * may be added to the end of a

symbol name for symbol table searches. An * may be added to the end of a module name to display all line numbers in a module.

FORMAT: **SY [address]**

REMARKS: If **address** is not specified, all symbols in the symbol table are displayed. If address is specified then the symbol which matches that address is displayed.

EXAMPLES: This command displays all symbols.

```
SY
VALUE           SYMBOL NAME
FBFC:EF48       BUFFER
FBFC:EEE2       FILENAME
FBFC:F1D6       I

LINE NUMBERS FOR MODULE TESTER

#15=0634:003C   #16=0634:0045   #17=0634:005B
#20=0634:0089   #21=0634:0092   #23=0634:00A3
```

This command displays the symbol start.
SY .START
.START= 1000:00AE

This command defines the symbol LOOP.
SY .LOOP = 2000:0050

This command displays the line 122 of module PAS1.
SYM ..PAS1#122
..PAS1#122=0624:003C

To display the symbol which matches the current CS:IP:
SY CS:IP
..PAS1#19

To define a symbol based on the current stack segment:
SY .SS = 0:SS

To define a symbol in terms of this symbol:
SY .MY_VAR = .SS:BP+6

EXAMPLES, continued

To list all symbols starting with PAS:
SY .PAS*

To list all line numbers for the module FTOC:
SY ..FTOC*

TRACE

PURPOSE: To display information stored by the PC PROBE in real time during program execution.

FORMAT: **T [N]**

REMARKS: The previous N instructions are displayed. If N is not specified then all trace data is displayed. Note that there must be a space before N. Trace data is only collected after a Go command. When the number of lines of trace data to be displayed is more than a screen full, the display will pause and the <enter> key will display the next page. Any other key will terminate the trace command. The format of the display is as follows.

ADDRESS OPCODE OPERAND

1. Address is for the cycle conducted on the bus.
2. Opcode shows the instruction executed.
3. Operand shows the operands used in the instruction. During the execution portion of the instruction, the operand portion of the display shows the bus activity during execution including the type of cycle. The prefetch cycles into the 8088 queue are not displayed in this trace format. Targets of short jumps/calls are displayed symbolically if the current code segment can be used to address the 20 bit real address of the instruction. Data operands are converted to symbols if there are no base or index registers involved.

4. If the address matches a symbol in the symbol table, the symbol is shown as a label on the line. If the address matches a ..modulename#linenumber in the symbol table, then this is shown as a label. If the latter modulename is assigned a file name, then the source code of the high level language is shown previous to the assembly language execution (if the SOURCE version is running). This allows viewing of the local variables embedded in the source line.
5. The information displayed from the trace can be qualified to display or suppress the DMA cycles (see the DMA command). If no complete instructions can be interpreted from the 1048 cycles of trace information, then a message to this effect is displayed, and all trace cycles are dumped to the screen. For example, this could occur if a long repeat string instruction was executed and the opcode fetch cycles had been cycled out of the trace memory. An exhaustive search is performed on the trace data before this occurs, which could take several minutes.
6. The segment register used during a data read or write operation is shown in the trace display. Here is a summary of the type of bus cycles and segment registers which will appear in the display.

TYPE OF BUS CYCLE	TYPE OF SEGMENT REGISTER
R - READ	DS - DATA SEGMENT
W - WRITE	CS - CODE SEGMENT
I - INPUT	SS - STACK SEGMENT
O - OUTPUT	ES - EXTRA SEGMENT
D - DMA	

FORMAT: **T H [N]**

REMARKS: If the H option is specified then 5 additional logic signals which are traced during instruction execution are displayed. These five logic signals are shown as Interrupt lines 2 thru 6 in the trace display. However, if the external logic probes are attached to the PC PROBE, then the five external logic signals are traced instead. The logic signals may be attached to logic in the system which need to be traced during program execution. See Appendix I, Logic Signals for more information. This trace format also displays the prefetch cycles in the 8088 queue, the state of the interrupt enable flag, and the non-maskable interrupt. If the H option is specified in the Trace command then the format is as follows:

```
ADDRESS  OPCODE  OPERAND  INTERRUPT IE NMI
                          2 3 4 5 6
```

The additional signals are clocked into PC PROBE trace memory on each bus read or write. Therefore, the resolution of these signals is much greater than that of instruction execution. The information displayed from the trace can be qualified to display or inhibit the activity of individual DMA channels. This removes mundane trace information such as DMA refresh cycles from the trace or allows activity from a DMA channel of interest to be displayed. See the DMA command for more details on how to enable/disable DMA cycles from the trace display. If no complete instructions can be interpreted from the 1024 cycles of trace information, then a message to this effect is displayed and all trace cycles are dumped to the screen. This could occur, for example, if a long repeat string instruction were executed and the opcode fetch cycles had been cycled out of the trace memory.

FORMAT: **T L [N]**

REMARKS: For the PC SOURCE PROBE, real time trace data is matched through the symbol table to the associated source file, and the previously executed N lines of source code are displayed. The symbol table for the source code must contain the line numbers which match the lines of source code in the source file.

EXAMPLE: In this trace display, only source code is displayed.

```
T L 6
LINE    SOURCE or MODULE
NO.     LINE        NAME
31.  fahr = lower;
32.  while (fahr <= upper) {
36.  Compute (fahr, &celsius);
112. *c_temp = f_temp - 32;
113. *c_temp = *c_temp * 5;
114. *c_temp = *c_temp / 9;
```

EXAMPLE: This trace shows source code along with assembly language execution. Note that the bold type below is not part of the trace display, but rather notes which describe the information in this display.

```
-t
          OP
ADDR      CODE   OPERAND(S)
```

```
112.  *c_temp = f_temp - 32;
0652D MOV AX,WORD PTR [BP+0008]      ;f_temp
      0ACE4 READ - SS - 14
      0ACE5 READ - SS - 00
06530 SUB AX,0020
06533 MOV SI,WORD PTR [BP+000A]
      6ACE6 READ - SS - D8
      6ACE7 READ - SS - 00
06536 MOV WORD PTR [SI],AX
      09CA8 WRITE - DS - F4
      09CA9 WRITE - DS - FF

113.  *c_temp = *c_temp * 5;
06538 MOV AX,0005
0653B IMUL WORD PTR [SI]
      09CA8 READ - DS - F4
      09CA9 READ - DS - FF
0653D MOV WORD PTR [SI],AX
      09CA8 WRITE - DS - C4
      09CA9 WRITE - DS - FF

114.  *c_temp = *c_temp / 9;
0653F MOV BX,0009
06542 CWD
06543 IDIV BX
06545 MOV WORD PTR [SI],AX
      09CA8 WRITE - DS - FA
      09CA9 WRITE - DS - FF
```

label which matches operand

source code

assembly language

data during bus cycle

segment register used

type of bus operation

address on bus

If PC PROBE is running, then the above display would not include the high level lines of source code data. The high level lines of source code are included in the display only when running PC SOURCE PROBE.

EXAMPLE: In this trace command, logic signals and prefetch cycles are shown in the trace data.

```
-T H 2
```

ADDR	OP CODE	OPERAND(S)	LOGIC 2	3	4	5	6	DMA ACK 1	2	3	INT ENB	NMI
46.		fahr = fahr + step;										
064CF	MOV	AX,WORD PTR [00D4]										;.STEP
064D0	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
064D1	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
064D2	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
09CA4	READ - DS - 0A		1	1	1	1	0	0	0	0	1	0
09CA5	READ - DS - 00		1	1	1	1	0	0	0	0	1	0
064D2	ADD	WORD PTR [00D6],AX										;.FAHR
064D3	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
064D4	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
064D5	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
064D6	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
09CA6	READ - DS - 00		1	1	1	1	0	0	0	0	1	0
09CA7	READ - DS - 00		1	1	1	1	0	0	0	0	1	0
064D7	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
064D8	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0
09CA6	WRITE - DS - 0A		1	1	1	1	0	0	0	0	1	0
09CA7	WRITE - DS - 00		1	1	1	1	0	0	0	1	0	
064D9	INST. BYTE FETCH		1	1	1	1	0	0	0	0	1	0

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 8088 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 PROBE cannot display accurate instruction execution, the following is printed followed by a dump of all trace cycles.

Unable to match data cycles with prefetched instructions in trace. This may be caused by uninitialized trace memory (no Go command has been entered or a step command has been done since the last Go), 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.

UNASSEMBLE

PURPOSE: To unassemble memory into 8088 assembly language instructions (or near assembly language when more clarity is needed in the disassembly).

FORMAT: **U range**

REMARKS: The specified **range** of memory is unassembled to the next highest whole instruction. If no length is specified, then the default is the length specified in the previous U command. If no segment value or segment register is specified in **range**, then the segment value from the previous U command is used. The PgDn key will unassemble the next screen full of instructions. The start of **range** must correspond to the first byte of an instruction or all of the unassembled instructions which follow may be incorrect. All indirect references to memory are specified as word or byte in the unassembly to simplify the understanding of the data type which is being addressed. All relative jumps and calls show the absolute offset rather than the relative offset for ease of determining the target address. The target addresses are also shown with a symbol comment at the end of the instruction. If there is no matching symbol, then the nearest previous symbol plus an offset is used. Data operands also show symbols as a comment if there are no base or index registers involved.

EXAMPLE: To unassemble four instructions starting at the symbol .START: (Note that Length 4 in this command refers to lines of code not bytes of code.)

U .START L 4

```
..MEMORYTESTER#82
0624:0000 0B1EB2F5      MOV BX,WORD PTR [F5B2]   ;MEMADDER+002
0624:0004 268B2F        MOV BP,WORD PTR ES:[BX]
0624:0007 7E05          JLE 000D                 ;MEMTEST#99+36
0624:0009 9A00000020    CALL 2000:0000           ;TESTMEMORY
```

WINDOW

PURPOSE: To open a window to display user defined data after each Go, STep, or Source Step command.

FORMAT: **WI = macroname**

REMARKS: Opens the window and executes the macro **macroname** after each Go command and after each step in the STep and Source Step commands. The output of the macro command is displayed in the window. The size of the window is automatically adjusted to fit the amount of data output by the macro.

FORMAT: **WI**

REMARKS: Displays the name of the current window macro.

FORMAT: **WI Close**

REMARKS: Closes the window at the bottom of the screen.

EXAMPLE: The following commands define a macro to display all register values and the contents of a buffer, and then open a window to show this data after each single step.

```
MAC R =
-R
-BY .BUFFER L .BUFFERLENGTH
-END

WI = R
```

WORD

PURPOSE: To display or change the words in memory.

FORMAT: **WO [range]**

REMARKS: The words in memory specified by **range** are displayed in hex. If no segment register or segment value is specified in **range**, then the segment value from the previous WOrd command will be used. If no length is specified in **range**, then the default is the previous length. If no **range** is specified, then a block of words starting at the end of the previous block is shown with a length equal to the previous length. The PgDn key will display the next screen full of words.

FORMATS: **WO start address = value [,] value [,] value...**
or
WO start address = <enter>

REMARKS: In the first format, words in memory are changed to the list of values to the right of the equal sign. In the second format, the current word at the address is displayed and a prompt waits for a new data value at this location. After the new data is entered, the next location is displayed and can be changed. Typing <enter> alone on a line ends the changes. When data is written to memory it is read back for verification unless the NOVerify condition is Noread. Errors are reported accordingly.

EXAMPLES: This command stores a list of words.
WO .START = 0000,0001,0002

This command stores the same list of words but looks at each one before changing it.
WO .START =
3000:0000 0000 - 0001 <enter>
3000:0002 0001 - 0002 <enter>
3000:0004 0002 - 0003 <enter>
3000:0006 0004 - <enter>

EXAMPLES, continued

To display the data:
WO .START L 3
3000:0000 0000 0001 0002

The next 3 words can be displayed with the following command. Since PROBE remembers the last length, the length defaults to 3 in this case.
WO <enter>
3000:0006 0004 AAAA BBBB

Registers can also provide the start address.
WO SS:SP = 0, 0, 0, 0
WO SS:SP+BP L 3

APPENDICES

APPENDIX A
PROBE ERROR MESSAGES

Initialization error messages

"Cannot open file: filename
Enter new drive specification:"
Printed at probe initialization if it could not find the file PRMNHW.EXE or SRMNHW2.EXE in the specified directory.

"Invalid invocation."
The command line after SOURCE was invalid. Probably missing the "\" before the path specification.

"Config file must be: "A=x0000"
where x = {1,2,3,4,5,6,7,8,9,C,[D]}
The first line of the PROBE.CFG file must be A=x0000 where x0000 is the base address of the board.

"No memory at specified address."
No memory could be found at the address listed in the PROBE.CFG file. This could be caused by:
1. The board jumpers not matching the base address in the PROBE.CFG file.
2. An attempt to re-load PROBE software while PROBE is still running.

"Source Probe hardware incorrectly installed."
or
"PC Probe hardware incorrectly installed."
The software was not able to communicate with the PROBE. This could be caused by:
1. The board jumpers not matching the base address in the PROBE.CFG file.
2. The board being write protected and still running. The STOP or RESET button will correct this.

"Cannot allocate memory for screen swapping"
"Cannot allocate memory for symbol table"
This indicates that DOS did not have enough memory to allocate to PROBE to satisfy the W or T parameters in the PROBE.CFG file. See Appendix C.

General

"Syntax Error."
The command syntax is not recognizable.

"DOS Critical Error #xx"
A critical DOS error occurred.

"Unrecognized command."
"Unrecognized edit command."
The first 2 or 3 letters typed do not match a command.

"Invalid expression."
The expression is invalid. (e.g. no operator between operands, 2 operators between operands,...)

"Attempted division by 0."
An attempt was made to divide by 0 in the expression.

"Too many ('s."
There were more '(' than could be parsed.

"Symbol not found."
The symbol in the command line could not be found in the symbol table.

"File not found."
The requested file could not be found.

"Path not found."
An element of the path does not exist.

"Too many open files in DOS."
There are too many DOS files open for SOURCE PROBE to open another file. Source opens a maximum of 3 files.

"Access denied to file."
The file could not be opened for reading or writing. This occurs during Edit trace if a syntax error exists in Trace options.

"Disk is full."
The disk is full and the file could not be written.

"Must enable disk interrupt to use disk."
The disk interrupts must be enabled to use the disk. See the INTERRUPT command.

"File system error."
A DOS file system error occurred.

BP Command.

"Must be: 1 <= BP number <= 8"
Breakpoint numbers are 1 to 8.

"BP > 40 characters."
Breakpoints must be less than 40 characters long.

"Warning -- BP already exists: "
The BP already existed and has been redefined. Its old value is printed as a warning.

BYTE, WORD, POINTER, MOVE, FILL commands.

"Cannot write in location xxxx:yyyy"
The location xxxx:yyyy is not RAM. This message will only occur if NOVerify = Read.

CONSOLE Command.

"Must enable keyboard to go to local console."
The keyboard must be enabled in order to use the local console.

DIR Command.

"File not found."
The requested directory or target file was not found.

DMA Command.

"Must be: 0 <= Dma channel number <= 3"
Dma channels are 0 to 3.

EDIT Command.

"No filename assigned to current CS:IP"
There is not a filename assigned to the module associated with the current CS:IP. Use the ASIgn command to make the association or use the EDIT FILENAME command.

"Enter file for log of changes:"
Enter the file that will contain a log of changes. This prompt will occur only for the first CHANGE command.

"Enter file to be "closed" (<enter> for none): "
The SOURCE PROBE file table is full and a file must be "closed" in order to "open" a new file.

"File not in table."
The file to be "closed" so that another may be "opened" is not a currently "open" file.

EMACRO Command.

"Macro nesting > 5."
Macros may only nest to a level of 5.

"Loop nesting > 5."
Loops may only nest to a level of 5.

GO Command.

"Parity Error."
A parity error occurred in the IBM PC RAM memory and caused an NMI.

"8087 Co-processor error, status=xxxx"
An 8087 error occurred and caused an NMI.

"Cannot set breakpoint at xxxx:yyyy"
The location xxxx:yyyy is not RAM and an execution breakpoint cannot be written there.

"Too many execution breakpoints."
There are 8 execution breakpoints.

"Too many hardware breakpoints."
There are 3 hardware breakpoints.

"Verb must be R, W, A"
These are the only allowable verbs.

"I/O verb must be R, W, or A"
Must have Read, Write, or All verb for IO operations.

"I/O address cannot contain segment"
May not have a segment value on the port number.

IF Command.

"Loop/If only allowed in macro."
IF is only allowed from within a macro.

INIT Command.

"Program has terminated. Must be re-loaded."
The program has terminated and released its memory back to the DOS memory manager. It must be re-loaded.

INTERRUPT Command.

"Cannot disable keyboard from local console."
The keyboard interrupt may only be disabled from the remote console.

LOAD Command.

"File not found."
The file to be loaded could not be found.

"Insufficient memory"
The file to be loaded was too large to fit in available memory.

"Invalid Filename"
The filename to be loaded is not a valid DOS filename.

"Internal error during load"
There was a DOS internal error during the load of the file.

"Could not find 'Publics by' in file."
The line 'Publics by' denotes the file as a valid symbol file when loading symbols.

LOOP Command.

"Loop/If only allowed in macro."
Loop is only allowed from within a macro.

"Loop nesting > 5."
Loops may only nest to a level of 5.

MACRO Command.

"Macro does not exist."
The macro to be displayed does not exist.

"Macro already exists."
The macro to be defined is already defined in the symbol table.

"Symbol table full."
The symbol table is full and the macro definition cannot be saved.

"Macro definition too long for *macroname*"
Macros must be less than 255 characters long.

QUIT Command.

"Have already done a Q R. Must now do a GO."
May only do Q R once.

"Warning -- List file has been closed. Open another one."
The list file is closed for a Q R. You must open another file if you wish listing to continue.

REGISTER Command.

"Unknown register name."
The only valid register names are:
ax, bx, cx, dx, ah, al, bh, bl, ch, cl, dh, dl
si, di, sp, bp, cs, ds, es, ss

"Unknown flag name."
The only valid flag names are:
O, D, I, T, S, Z, A, P, C

SAVE Command.

"Invalid file name."
The filename to be saved is not a valid DOS filename.

SCREEN SWITCH Command.

"Must be at local console for screen switching"
Cannot perform SCreen Show from remote console since data is already on PC console.

SYMBOL Command.

"Symbol not found."
The symbol to be displayed is not in the symbol table.

"Symbol table full."
The symbol table is full and no more symbols can be defined. See T option in Appendix C.

"Warning: Redefining symbol. Old value was "
The symbol already existed and has been redefined.

TRACE Command.

"Trace data invalid - Breakpoint not found"
The end of execution could not be found in the trace.

"Trace Overrun in GetNextInstLoc"
An internal error occurred while processing trace data.

"Unable to find instructions in trace. Trace data cycles follow:"
The last 1024 bus cycles did not contain an instruction, most probably due to uninitialized trace memory (no Go command has been entered) or the execution of a long string move. The data cycles are output.

APPENDIX B
SOFTWARE/MAINFRAME COMPATIBILITY

The PROBE is compatible with the software and versions shown in Table B-1.

Table B-1. Software Compatibility

SOFTWARE	VERSION	MANUFACTURER
PCDOS/MSDOS	2.0/2.1/3.0/3.1	IBM/MICROSOFT
PASCAL		IBM/MICROSOFT
C		MICROSOFT/LATTICE
		COMPUTER INNOVATIONS
		AZTEC with DOS linker
		Wizard with DOS linker
ASSEMBLER		IBM/MICROSOFT
FORTRAN		MICROSOFT

The PROBE is compatible with the systems shown in Table B-2.*

Table B-2. Hardware Compatibility

SYSTEM	MANUFACTURER
PC	IBM
XT	IBM
COMPAQ PORTABLE	COMPAQ
PC 1200	TANDY
PC 150,151	ZENITH
NCR PC	NCR
SPERRY PC	SPERRY
LEADING EDGE	LEADING EDGE
FARADAY BOARD	FARADAY ELECTRONICS

*Compatibility with other systems will be added in the future. Contact Atron for additional information.

APPENDIX C
CONFIGURATION FILE AND EXTERNAL CONSOLE CONNECTION

The configuration file is an ASCII file named PROBE.CFG, and the file provides PROBE system information to be used during some commands. PROBE.CFG specifies:

1. The base address of PROBE memory.
2. Console configuration parameters for the external console.
3. Whether the symbol table space has been expanded.
4. Whether screen switching with a color monitor will be used.

If the PROBE.CFG file is not present in the default pathname, then the default setting is the same as the VT100.CFG file. The PROBE.CFG parameters are described later.

CONNECTING AN EXTERNAL CRT TO PC PROBE'S SERIAL PORT

PC PROBE assumes the following transmission parameters for transmitting and receiving data via the on-board serial IO channel.

2400 baud (this is the default)
8 data bits
2 stop bits
no parity

The interface for the PC 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. Note that your terminal may require you to connect pins 4 to 5 (RTS - CTS) and 6 to 20(DSR - DTR) at the terminal.

Pin 1,7 - ground
Pin 3 PROBE transmit
Pin 2 PROBE receive

Figure C-1. RS232 Signals

CONNECTING AN EXTERNAL CRT TO COM1 OR COM2 PORT

For the COM1 and COM2 ports, the transmission parameters are set from DOS software using the MODE command and PROBE uses these parameters as they are set.

PROBE.CFG PARAMETERS

When PROBE writes to the screen it uses cursor and screen control parameters for editing and for the Menu window. When an external CRT is used, these parameters must be put into the PROBE.CFG file to tell PROBE how to work with the CRT you are using. A description of these parameters is shown below. (The A parameter has a different use.)

A segment address of PC PROBE board. This must be first.

H Home cursor and clear screen sequence.

L Clear line from cursor to end.

M Move cursor to row Y, column X. (See X, and Y defined below.)
Move cursor to row W, column V. (See W, and V defined below.)

O Set offset to be added to V or W (definitions below).

T Select n 64k blocks of memory from the system's memory for use when the PROBE symbol table overflows. If not specified, the default is 0.

W If set to Y for yes (not the same Y as defined below), then 12K of system memory is used to save the screen memory for a color graphics adapter when the screen switch command is used. The default does not reserve memory in system memory space.

The format for each parameter is:
<id letter> = HexChar [V|W|X|Y]* HexChar [V|W|X|Y]*
HexChar...00
(id letter is A,H,L,M,O,T, or W)
Hex Characters specify ASCII characters in sequence for example, the sequence <esc>H is represented as 1B 48.

DEFINITIONS OF X,Y,V,W

X an ASCII X. PROBE will translate the column number to ASCII and place the two resulting ASCII characters at this position in the sequence.

Y an ASCII Y. PROBE will translate the row number to ASCII and place the two resulting ASCII characters at this position in the sequence.

V an ASCII V. PROBE will add the column number to the input offset and send the resulting ASCII character at this position in the sequence.

W an ASCII W. PROBE will add the row number to the input offset and send the resulting ASCII character at this position in the sequence.

Below are some examples of configuration files for the indicated terminals. Some of these are included on the PC PROBE diskettes so that they can be easily copied to PROBE.CFG.

The first example is for using a PC as a terminal which uses the TERMCOM terminal emulator supplied with the PC PROBE diskettes.

PC TERMINAL EMULATOR
A=D0000
H=1B 31
L=1B 32
M=1B 33 W V
O=00
R=9600

VT100: (factory default)
A=D0000
H=1B 5B 31 3B 48 1B 5B 32 4A 00
L=1B 5B 30 4B 00
M=1B 5B Y 3B X 48 00
O=00

WY-30
H=1E 1A 00
M=1B 3D W V 00
L=1B 54 00
O=1F

HP
H=1B 48 1B 4A
M=1B 26 61 X 63 Y 59
L=1B 4B
O=FF

VT52:
H=1B 48 1B 4A 00
L=1B 4B 00
M=1B 59 W V 00
O=1F

QUME QVT102
H=1E 1B 79 00
M=1B 5B W 1B 5D V 00
L=1B 54 00
O=1F

ADDS:
A=D0000
H=1B 59 20 20 0C 00
L=1B 4B 00
M=1B 59 W V 00
O=1F

ADM-3A:
A=D0000
H=1E 1A 00
L=00
M=1B 3D W V 00
O=1F

TELEVIDEO 925
A= D0000
H= 1E 1A 00
L= 1B 54 00
M= 1B 3D W V 00
O= 1F

HAZELTINE 1510:
A=D0000
H=7E 12 7E 18 00
L=7B 0F 00
M=7E 11 Y X 00
O=00

DUMB TERMINAL
A= D0000
H= 0
L= 0
M= 0
O= 0

IBM 3101
A= D0000
H= 1B 48 1B 4C
L= 00
M= 1B 59 V W 00
O= 1F

EXAMPLE: This shows what PROBE will send to the CRT when it wants to move the cursor to row 5 column 10 for two different types of CRT's (i.e. two different PROBE.CFG files).

On a VT100:<esc> [05 ; 10 H
On an ADM-3A:<esc> = $){ 5 + 1F = 24 = '$'10 + 1F = 29 = ')' }

NOTE: Terminals which do not support L will not clear the line on the screen when the line is being edited. However, the image of the line in PROBE memory is cleared. To avoid the command line recall use the Dumb terminal configuration.

USING THE PC AS A TERMINAL EMULATOR

Another PC can be used as an intelligent terminal by using the Termcom program provided on the PROBE diskette. In this case connect pins 5-6-20 on the PC. Termcon is a terminal emulator program which makes the COM1 port of a second PC look like an intelligent terminal to PROBE.

1. First copy PC.CFG to PROBE.CFG to configure PROBE to work with Termcom.
2. On the PC which is acting like a terminal, use the mode command to set the baud rate.

   mode com1: 2400,n,8,2

3. Connect an RS232 cable between the PROBE com port and the COM1 port of the PC terminal emulator.
4. On the PC terminal emulator type: **termcom**
5. Once PROBE has been brought up, switch to the external console with the CONsole R command.
6. The menu is not on when the switch is made to the terminal emulator. To get the menu back, type: **ME ON**

EDITING KEYS ON REMOTE CONSOLE

The previous DOS editing keys can be mapped to the external CRT. The special keys on the IBM PC keyboard can all be mimicked from the external CRT by using the following conversion. Note that the character, '^', is used as a special lead-in for these keys. This character may be obtained on the IBM PC keyboard as <shift>6 but may be at a different location on your remote terminal. If you are using a PC as a remote console, it would be useful to use Borland International's Superkey to remap these keys to the function they would have on a PC keyboard.

IBM Key	Function	Special two-key sequence
F1	Same as DOS.	^1
F2	See above.	^2
F3	See above.	^3
F4	See above.	^4
F5	See above.	^5
Ins	See above.	^I or ^i
Del	See above.	^K or ^k (Kill character)
PgUp	Up one page	^U or ^u
PgDn	Down one page	^D or ^d
<Ctrl>PgUp	Top page	^T or ^t
<Ctrl>PgDn	Bottom page	^B or ^b
HOME	Home page	^H or ^h
Up arrow	up one line	^P or ^p
Down arrow	down one line	^L or ^l

Commands which dump data to the screen can be terminated with the Ctrl C key or the STOP button on the external switch box. The screen dump will pause with the Ctrl S key.

APPENDIX D
USER PROCESSED NMI

The PROBE uses the non-maskable (NMI) interrupt to generate the breakpoint. This will cause special processing to be needed when the NMI is used by other system elements such as user generated NMI on the IOCHK line on the bus or software modifications to the NMI vector at location 8.

If the applications program intends to change the NMI vector at location 8 then control will not return to PROBE when a hardware breakpoint occurs. However, this can be remedied by the following procedure.

Start the user program and set an execution breakpoint at the point in the applications program sometime after the NMI vector has been changed by the applications program. When this breakpoint has been detected and control has returned to PROBE, the PROBE will inspect the vector to see if it has changed. If it has, PROBE stores this new vector away and then restores location 8 to its breakpoint vector. Hence forward, when an NMI occurs, PROBE checks to see what type of NMI has occurred: hardware breakpoint, parity error, or user NMI. If a user NMI or parity error, PROBE continues executing the user NMI routine. If the NMI was caused by a hardware breakpoint, then PROBE retains program control. The sequence of stopping the processor with an execution breakpoint and fixing up the PROBE vector is necessary each time the vector changes. However, this normally occurs only once at the start of the user's program such as in the initialization code of a numeric run-time library.

APPENDIX E
FILES ON YOUR PROBE DISKETTES

There are several files on your PROBE diskettes which may or may not be needed depending upon what you are doing. Only those used for "EXECUTING PROBE SOFTWARE" are required. A list of these files and their purpose is given below:

DESCRIPTION OF FILE	FILES ON PC PROBE	FILES ON PC SOURCE PROBE
CURRENT VERSION	2.13	1.13
EXECUTING PROBE SOFTWARE	PROBE.EXE	SOURCE.EXE
EXECUTING PROBE SOFTWARE	PRMNHW.EXE	SRMNHW.EXE
EXECUTING PROBE SOFTWARE		
PROBE DIAGNOSTIC SOFTWARE	PROBEDIA.EXE	
PROBE DIAGNOSTIC SOFTWARE		
STRIP SYMBOLS FROM MAP	STRIP.EXE	
STRIP DATA FILE V1.O PASCAL	STRIP10.PAS	
STRIP DATA FILE V3.11 PASCAL	STRIP31.PAS	
STRIP DATA FILE LATTICE C	STRIP.C	
STRIP DATA FILE MICROSOFT C	STRIP.MSC	
STRIP SYMBOLS FROM PLINK86	STRIPPE.EXE	
A FILE OF SAMPLE MACROS	PROBE.MAC	PROBE.MAC
EXECUTABLE DEMO FILE	FTOCNEW.EXE	FTOCNEW.EXE
PRESTRIP SYMTABLE FOR DEMO	FTOCNEW.MP1	FTOCNEW.MP1
SOURCE FILE DEMO MODULE #1	FTOCM.C	FTOCM.C
SOURCE FILE DEMO MODULE #2	FTOCIO.C	FTOCIO.C
PROBE CONFIGURATION FILE	PROBE CFG	PROBE.CFG
CFG FILE FOR VT100 TERMINAL	VT100.CFG	
CFG FILE FOR ADM3A TERMINAL	ADM3A.CFG	
CFG FILE FOR TV925 TERMINAL	TV925.CFG	
CFG FILE FOR DUMB CRT	DUMB.CFG	
CFG FILE FOR TERMCOM	PC.CFG	
PC TERMINAL EMULATOR	TERMCOM.EXE	
PRODUCE C LIST FILE	CLIST.EXE	

DESCRIPTION OF FILE	FILES ON PC PERF ANAL	FILES ON PC PROBE/PL
CURRENT VERSION	1.13	2.13
EXECUTING PROBE SOFTWARE	PERF.EXE	PPL.EXE
EXECUTING PROBE SOFTWARE	PAMNHW.EXE	PRMNHWPL.EXE
EXECUTING PROBE SOFTWARE		PMN2PL.EXE
PROBE DIAGNOSTIC SOFTWARE		PROBEDIA.EXE
PROBE DIAGNOSTIC SOFTWARE		
STRIP SYMBOLS FROM MAP		STRIP.EXE
STRIP DATA FILE V1.O PASCAL		STRIP10.PAS
STRIP DATA FILE V3.11 PASCAL		STRIP31.PAS
STRIP DATA FILE LATTICE C		STRIP.C
STRIP DATA FILE MICROSOFT C		STRIP.MSC
STRIP SYMBOLS FROM PLINK86		STRIPPE.EXE
A FILE OF SAMPLE MACROS	PERF.MAC	PROBE.MAC
EXECUTABLE DEMO FILE	MEMTEST.EXE	FTOCNEW.EXE
PRESTRIP SYMTABLE FOR DEMO	MEMTEST.MP1	FTOCNEW.MP1
SOURCE FILE DEMO MODULE #1	MEMTEST.PAS	FTOCM.C
SOURCE FILE DEMO MODULE #2		FTOCIO.C
PROBE CONFIGURATION FILE	PROBE.CFG	PROBE.CFG
CFG FILE FOR VT100 TERMINAL		VT100.CFG
CFG FILE FOR ADM3A TERMINAL		ADM3A.CFG
CFG FILE FOR TV925 TERMINAL		TV925.CFG
CFG FILE FOR DUMB CRT		DUMB.CFG
CFG FILE FOR TERMCOM		PC.CFG
PC TERMINAL EMULATOR		TERMCOM.EXE
PRODUCE C LIST FILE		CLIST.EXE

FILES ON PC SOURCE PROBE/PL	FILES ON PC PROBE/87	FILES ON PC SOURCE PROBE/87	FILES ON PC PROBE PERF ANAL/87
1.13	2.13	1.13	V1.13
SOURCEPL.EXE	PROBE87.EXE	SOURCE87.EXE	PERF87.EXE
SRMNHWPL.EXE	PRMNHW87.EXE	SRMNHW87.EXE	PAMNHW87.EXE
	PROBEDIA.EXE		
	STRIP.EXE		
	STRIP10.PAS		
	STRIP31.PAS		
	STRIP.C		
	STRIP.MSC		
	STRIPPE.EXE		
PROBE.MAC	PROBE.MAC	PROBE.MAC	PERF.MAC
FTOCNEW.EXE	FTOCNEW.EXE	FTOCNEW.EXE	PROBE.CFG
FTOCNEW.MP1	FTOCNEW.MP1	FTOCNEW.MP1	MEMTEST.PAS
FTOCM.C	FTOCM.C	FTOCM.C	MEMTEST.EXE
FTOCIO.C	FTOCIO.C	FTOCIO.C	MEMTEST.MP1
PROBE.CFG	PROBE.CFG	PROBE.CFG	
	VT100.CFG		
	ADM3A.CFG		
	TV925.CFG		
	DUMB.CFG		
	PC.CFG		
	TERMCOM.EXE		
	CLIST.EXE		

APPENDIX F
PROBE/MS DOS INTERFACE DESCRIPTION

These interrupt vectors are taken over at all times by PROBE. They are restored to their original value after a Quit command.

INTERRUPT LEVEL 1

Operation	When Used
NOT USED	NOT USED

INTERRUPT LEVEL 2

Operation	When Used
NMI	STOP button Hardware breakpoints Single step

INTERRUPT LEVEL 3

Operation	When Used
SW trap	Execution breakpoints

These interrupt vectors are taken over only when PROBE is accepting command input. They are restored to their original value for each Go, STep, and Quit commands.

INTERRUPT LEVEL 1BH

Operation	When Used
<Ctrl> Break sense	Whenever <Ctrl> Break is pressed while in PROBE command mode

INTERRUPT LEVEL 24H

Operation	When Used
Critical error	Whenever a DOS Critical error occurs while in PROBE command mode

The following DOS V2.0 calls are made by PROBE only at initialization.

INTERRUPT LEVEL 21H

Function code AH=	Operation
01H	Console input
02H, 09H	Console output
3DH	Open a file
3EH	Close a file
3FH	Read from a file
4AH	Set block size
4BH	Load a program (AL = 3 = load overlay)
4CH	Terminate process (exit)

These DOS calls are used only for console communication to Remote console 1 or 2.

INTERRUPT LEVEL 14H

Function code AH=	Operation
01H	Remote 1 or 2 console input
02H	Remote 1 or 2 console output
3DH	Remote 1 or 2 console input

The following DOS V2.0 / IBM ROM BIOS calls are made by PROBE while it is running.

INTERRUPT LEVEL 10H

Function code AH=	Operation	When used
0	Set video state	Screen switching and init
2	Move cursor	LOCAL console I/O
3	Read cursor	LOCAL console I/O
5	Select page	Screen switching and init
6	Scroll up	LOCAL console I/O
9	Write attribute & character	LOCAL console I/O
14	Write character	LOCAL console output
15	Get video state	Screen Switching and init

INTERRUPT LEVEL 11H

Operation	When used
Get equipment type	Screen switching and init If bits 4 and 5 are set, then assume video memory is at B000:0, else at B800:0.

INTERRUPT LEVEL 16H

Function code AH=	Operation	When used
0	Read key	LOCAL console input
1	Key ready check	LOCAL console input

INTERRUPT LEVEL 20H

Operation	When used
Program terminate	Quit command

INTERRUPT LEVEL 21H

Function Code AH=	Operation	When used
1AH	Set DTA	DIr command
29H	Parse filename	LOAd command
30H	Get DOS Version	LOAd command
31H	Terminate process & remain resident	Quit Remain command
3CH	Create file	SAve command SAve Macros command EDit - CHANGE command LIst command (unless LPT1: or COM1: which use DOS default handles)
3DH	Open file	LOAd command LOAd Macros command LOAd Symbols command EDit - DISPLAY command STep command Trace command
3EH	Close file	SAve command SAve Macros command EDit - CHANGE command LOAd command LOAd Macros command LOAd Symbols command EDit - DISPLAY command STep command Trace command LIst command (unless LPT1: or COM1:

INTERRUPT LEVEL 21H, continued

Function Code AH=	Operation	When used
3FH	Read from file	LOAd command LOAd Macros command LOAd SYMBOLS command EDit - DISPLAY command STep command
40H	Write to file	SAve command SAve Macros command EDit - CHANGE command LIst command active
42H	Seek in file	EDit - DISPLAY command (AL = 0; 0:0) STep command (AL = 0; 0:0) Trace command (AL = 0; 0:0) EDit - CHANGE command (AL = 2; 0:0)
48H	Allocate memory	LOAd command
49H	Deallocate memory	LOAd command
4BH	Load file	LOAd command (AL = 1)
4CH	Terminate program	LOAd command
4DH	Retrieve exit code	End of program.
4EH	Find first file	DIr command
4FH	Find next file	DIr command

OTHER RESOURCES USED BY PROBE

1. 58H paragraphs of memory just above DOS are used when PROBE is first being loaded.
2. 10H paragraphs (PROBE's program segment prefix) are used while PROBE is running.
3. When a user enters a LOAd command, PROBE loads the program and sets its termination address to point to PROBE.

APPENDIX G
SYMBOL TABLE MAP FORMATS

This is a description of the format of the MAP file which is read into PROBE by the LOAd Symbol filespec command. The MAP file is used to create the PROBE symbol table and must conform to this format to be interpreted properly.

In the following description this short hand notation is used:
<S> means any number of spaces (0-80)
<C> means any number of characters

Characters in quotes must be typed as is, including their upper/lower case.

1. Any number of lines of data.
2. A line of the form:
   <C> 'Publics by' <C>
3. A blank line.
4. Any number of symbol lines of the form:
   <S> segment:offset <S> symbol name <CR>
5. A blank line. This denotes the end of symbols.
6. Any number of lines of data.
7. A line of the form:
   <C> 'Line n' <C> modulename. (Note: Modulename must start in column 18)
8. A blank line.
9. Any number of line number lines of the form:
   <S> line number <S> segment:offset <S> line number <S>...
10. A blank line.
11. Steps 7, 8, 9, and 10 may be repeated any number of times.

SPECIAL NOTES:

If the symbol '_INIXQQ_' (the spaces inside the quotes are required) is found before the line containing 'Publics by' then it is assumed to be a Microsoft Pascal module which will move DGroup. Therefore, the following information is used on the lines before 'Publics by':

If '_STACK', '_DATA', '_COMADS', or '_CONST' exist on the line, then the size of the segment is added to DGroupSize. The size of the segment is assumed to be a hex number starting in column 16 of this line.

When a line is found with '_DGROUP' on it, the DGroupStart address is picked up from the same line. The start address must be in column 2.

Once the 'Publics by' line is read, the following formula is evaluated for symbols whose segment value is the same as DgroupStart:

SSV == Symbol's new segment value
DGS == DGroupStart
SS == 'STACK' class size
DS == 'DATA' class size
CS == 'CONST' class size
MS == 'COMADS' class size
MT == Top of memory in PC

$$SSV := DGS + 1 - \frac{(SS + DS + CS + MS)}{16}$$

if (SSV + 1000H) > MT then SSV := MT - 1000H

APPENDIX H
USING PROBE WITH TOPVIEW

PROBE can be used to debug programs which are running on IBM's TopView with the following procedure. The differences between this procedure and the normal way of using PROBE include:

1. When and how you gain control over the start of execution of your program.
2. How the symbol table is loaded.

TopView does not provide a system call which loads the program and then returns. Instead, it loads and goes. In order for PROBE to gain control, you must put a jump to self loop at the start of your program. Then after control has been transferred to your program from TopView, PROBE can regain control when you press the STOP button. Next you can load the symbol table in an absolute format. This procedure is described as follows:

1. Start PROBE.
   -CON R -- use remote console so TopView has screen control.
   -Q R -- return to COMMAND.COM for further DOS processing.
2. At the beginning of your program, install a jump to self loop.
   Assembly language:

```
Start:  jmp Start
          ...
```

   C:

```
main ()
{
while (1) {};
...
  }
```

3. Translate and link this module to form a .EXE or .COM file. Generate a standard link map using the /m/l controls.

4. Using a text editor or the TYPE command, look at the output MAP file to find the segment value of the start address of the program. (This may be done using the EDIT command of Source Level PROBE products.) Write down the xxxx portion of this address.
   Assembly language:

```
        ...
    xxxx:yyyy   START
        ...

  C:
        ...
    xxxx:yyyy   MAIN
        ...
```

5. Start TopView and select your program from the "Start a program" menu.
6. After your program has reached the jump to self loop, press the STOP button.
7. Load the symbols into the probe symbol table. Use the value for xxxx that was determined in step 4 above.
   -LOA S filename.MAP CS-xxxx:0 A
8. Change IP to a point after the jump to self loop.
   -U CS:IP L 4
   -R IP=....
9. Normal debugging may continue from this point. Steps 2 through 9 may be repeated as often as necessary. Do not Quit from the PROBE product. Always do a Go and have TopView Exit. Also remember to do aDElete Symbols ALL command before loading the next symbol table.

ASSEMBLY LANGUAGE SHORT-CUT.

If the DOS loader transfers control directly to your jump to self loop in assembly language, you may omit step 4. In step 7 type: -LOA S filename.MAP ES+10:0 A

NOTE: High-level languages have the DOS loader transfer control to a run-time library which then calls the user program. Therefore, this short-cut will not work for high-level languages.

NOTE: Overlays used under PLINK86 have the DOS loader transfer control to the OVERLAY.LIB which then calls the user program. Therefore, this short-cut will not work for PLINK86 overlaid programs.

APPENDIX I
LOGIC SIGNALS

Logic signals are lines which are connected to the breakpoint and trace logic of the PC PROBE. These signals are different depending on whether or not the Logic PROBE is plugged into the PC PROBE. Below is a summary of what is represented by the logic signals L # in a breakpoint command or a display of the trace information using the T H # form.

L # in BP command	With Logic Probes	Without Logic Probes
0	Log probe 0	Bus interrupt request 2
1	Log probe 1	Bus interrupt request 3
2	Log probe 2	Bus interrupt request 4
3	Log probe 3	Bus interrupt request 5
4	Log probe 4	Bus interrupt request 6

This logic data is written into trace memory during any I/O read, I/O write, memory read, or memory write whether from the cpu or dma controller.

APPENDIX J
USING PLINK86 WITH PROBE

If the PLINK86 linker is used instead of the standard DOS linker, the SYMTABLE option must be specified in the link command to produce symbolic debugging records. If the modules to be linked have been produced with Microsoft Pascal or Fortran, the users compiled program will move the DGroup to higher memory. In order to work correctly, the "G" flag must be specified in the MAP command to PLINK86 for these compilers.

This linker places the symbolic debugging information into the .exe file instead of the .MAP file. There are two methods for PROBE to get the symbolic information from the .exe file instead of the .MAP file. If you are using the standard versions of PROBE software, the STRIPPE utility described next should be used. If you are using the /PL versions of PROBE software, go to the next section.

A utility called STRIPPE.EXE is included on the PROBE diskette to take the symbolic debugging information from the .exe file and create a standard DOS .MAP file. The syntax for this utility is:

strippe.exe [/] inputexefile, outputlinkmap, datafile

The inputexefile is the .exe file produced by PLINK86. The outputlinkmap file created by this utility is in the standard DOS .MAP format and will be read in by PROBE as the symbol table using the LOA S command. If the symbol table is too large, symbols may be stripped before they are placed in the outputlinkmap. The symbols in the datafile are stripped from the outputlinkmap file. See "STRIPPING SYMBOLS FROM THE MAP FILE" in Chapter 3 for information on how to create this datafile.

SUPPORT FOR PLINK86 USING /PL VERSIONS

Special versions of PROBE software called /PL extensions are available which support debugging programs linked with PLINK86. Contact Atron for information on these versions. The /PL versions track the overlay loader of PLINK86 so that breakpoints can be set in overlays which are not yet loaded. These versions wait until the overlay is loaded before setting the actual breakpoint. Descriptions

of the changes in the commands for the /PL versions are given in the following sections.

LOAD COMMAND for PLINK86

The format for the LOAd command does not change. The difference is that the symbol table load is from the .exe file.

```
EXAMPLE:  loa test.exe    ; loads program
          loa s test.exe  ; loads symbol table
```

SYMBOL COMMAND for PLINK86

The format of the SYmbol command which displays symbols does not change. The difference is that the display also shows the overlay number for the overlay which contains the symbol.

EXAMPLES: This command displays all symbols:

```
SY
Overlay     Address          Symbol name
0           3186:02D0        $LDEX$
1           31FE:0000        FOO
2           31FE:0000        FOO1
3           31FF:0000        FOO2
```

To display the symbol .START:

```
sy .START
.START=3182:0000
```

(Note that if the symbol is in the root overlay the overlaynumber is not displayed.)

To display the symbol .TEMP which is in overlay #1:

```
sy .TEMP
.TEMP=31FE:0000, overlay #1
```

The format of the SYmbol command which defines symbols has been changed to include the overlay number associated with the symbol.

FORMAT: **SY .symbolname = address [, overlay_number]**

REMARKS: **Symbolname** is the name of the symbol and may also include the ..modulename. **Overlaynumber** assigns the symbol to the specified overlay. If no **overlaynumber** is specified, then the default is 0 (root).

EXAMPLE: Define the symbol .FOO in overlay 3:
sy .FOO = 3128:0030 , 3

Define the symbol .FOO1 in the root overlay. Note that defining symbols in the root overlay does not require an overlay number.
sy .FOO1 = 3128:0030

INVOKING THE /PL VERSIONS

To start up the /PL version of software which you have, type the following:

VERSION OF PROBE SOFTWARE	START UP COMMAND
PC PROBE/PL	PROBEPL
PC SOURCE PROBE/PL	SOURCEPL

GENERAL ERROR MESSAGES FOR PLINK VERSIONS

A new error message has been added in the /PL versions of PROBE. If you use a symbolic reference in a command to PROBE for a symbol which is in an overlay that has not yet been loaded, the following error message will appear:

Symbol not loaded.

APPENDIX K
TECHNICAL REPORTS

This section provides technical information, potential problem areas, and bug reports for the PROBE. It will be updated periodically if you send in your registration card.

This appendix contains the following Technical Reports:

INTERRUPT 3
STACK USAGE DURING BREAKPOINT
GETTING TO DOS COMMANDS FROM PROBE
JUMPING INTO THE PROBE SOFTWARE FROM THE APPLICATIONS PROGRAM
INTERRUPTING CRITICAL CODE SECTIONS IN PC DOS
ASSORTED COMMON QUESTIONS AND ANSWERS.

Last revised: 4/25/86

INTERRUPT 3

Software interrupt 3 is used by the 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, 24 bytes of information are pushed onto the stack. However, after control is received by 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 PROBE

To execute DOS commands under the watchful eye of PROBE, do the following:

1. Load PROBE.
2. Now do a quit and stay resident command from PROBE - Q R.
3. PROBE can now be reentered by using the STOP button as long as its vectors have not been modified.

JUMPING INTO THE PROBE SOFTWARE FROM THE APPLICATIONS PROGRAM

The memory on the hardware versions of PROBE is write protected and also cannot be accessed when the applications program is being executed. This means that if the applications program tries to write to the memory area of the PROBE it will not change the memory but will generate a breakpoint and return control to the PROBE. In fact, any type of access to the PROBE memory will cause the breakpoint to be generated. When this occurs, the user can inspect the trace memory to determine what caused the access to PROBE memory. This does not apply to the non-hardware versions of PROBE.

INTERRUPTING CRITICAL CODE SECTIONS IN PC 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 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 PROBE software uses BIOS calls to do console IO. To eliminate this from happening, switch to the external console where no BIOS calls are made by PROBE software since the external console routines are in the PROBE software.

ASSORTED COMMON QUESTIONS AND ANSWERS

"WHY DOESN'T MY EXTERNAL CONSOLE WORK?"

> Ensure the connector which is plugged into the PC PROBE RS232 port is not offset. It will plug in, in either direction and work. If connected to the COM1 port, ensure that 5-6-20 are connected on the COM1 connector.

"How can I force single stepping and avoid the screen update from the instruction preview feature of the single step command?"

> Use the ST A (step automatic) option and then use Ctrl S to pause the screen.

"PC PROBE will not run with Superkey."

> Some programs take over vectors 1, 2, and 3 which PC PROBE needs to function.

"My program doesn't work when I run it, but it works when I invoke PC PROBE."

> The problem is probably a memory overwrite problem where your code is overwriting itself or DOS. When PC PROBE is loaded, a program prefix segment of 256 bytes is loaded just above DOS. When your program is loaded it will be 256

bytes higher in memory which may be enough to cause the overwrite to not cause the same problem. To find this, use the memory overwrite procedure as described in Chapter 4. Another way of attacking the problem is to first load PC PROBE, then do a reboot of DOS by assembling and then executing an INT 19 in system memory. DOS reboots. Now load your program and produce the same error. When the error occurs, press the STOP button which will put you back into PROBE so you can trace the problem. In this case, the reboot has eliminated the program prefix segment used by PROBE, but PROBE can still get control because of the STOP button.

"Why does it take a long time for the program to start running after a Go command when range breakpoints are set?"

Range breakpoints require a lot of set up on the PC PROBE hardware. The longer the range, the longer it takes to set them up.

"I got an invalid trace on a known working program, why?"

The trace data is only gathered after a Go command is executed.

"@ in a symbolname does not give me the right value for the symbol."

The @ in the symbolname is interpreted as the indirection operator.

INDEX

Other Entries

A

B

C

D

E

F

N

O

P

Q

R

S

T

U

V

W

ATRON REPAIR SERVICE POLICY

Atron will provide service repair of the PC PROBE on the following basis.

PC PROBE within 90 days hardware warranty period

Atron will send a new board to customer and customer will return failed board to Atron. Atron will pay for UPS surface freight to customer. (Customer pays for upgraded freight service.) Customer pays for return freight of failed board to Atron. A PO # will be required in advance of sending new board to customer for the price of a new system and is automatically cancelled upon arrival of the failed board to Atron.

PC PROBE outside 90 day hardware warranty period

Atron will send a new board to customer and customer will return failed board to Atron. Customer pays for freight service in both directions. A PO # will be required in advance of sending new board to customer for the price of a new system. Upon receipt of the returned board, Atron will invoice customer in the amount of Atron's then fixed repair cost for the board. If the failed board is not received by Atron within 15 days after sending customer a new board, the PO is due and payable.

LIMITED WARRANTY

Atron Corporation warrants this product to be in good working order for a period of 90 days from the date of purchase from Atron or an authorized Atron dealer. Should this product fail to be in good working order at any time during this 90 day warranty period, Atron will, at its option repair or replace this product at no additional charge except as set forth below. Repair parts and replacement products will be furnished on an exchange basis and will be either reconditioned or new. All replaced parts and products become the property of Atron. This limited warranty does not include service to repair damage to the product resulting from accident, disaster, misuse, abuse, or non-Atron modifications of the product.

Limited warranty service may be obtained by delivering the product during the 90 day period to Atron. If this product is delivered by mail, you agree to insure the product or assume the risk of loss or damage in transit, to prepay shipping charges to Atron and to insure the product is adequately packed.

ALL WARRANTIES FOR THIS PRODUCT, WHETHER EXPRESS OR IMPLIED, ARE LIMITED IN DURATION TO A PERIOD OF 90 DAYS FROM THE DATE OF PURCHASE, AND NO WARRANTIES, WHETHER EXPRESS OR IMPLIED, WILL APPLY AFTER THIS PERIOD.

ATRON HEREBY DISCLAIMS ALL OTHER EXPRESS AND IMPLIED WARRANTIES FOR THIS PRODUCT INCLUDING THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

IF THIS PRODUCT IS NOT IN GOOD WORKING ORDER AS WARRANTED ABOVE, YOUR SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. IN NO EVENT WILL ATRON BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH PRODUCT.

SOFTWARE LICENSE AGREEMENT

All Atron software is protected by both United States Copyright Law and International Treaty provisions. Therefore, you must treat this software just like a book with the following exception: Atron Corp authorizes you to make archival copies of the software for the sole purpose of backing up your software and protecting your investment from loss.

This means that this software may be used by any number of people and may be freely moved from one computer location to another so long as there is no possibility of it being used at one location while it is being used at another - just like a book.