ModulasTen

PROBUG

68000 SOFTWARE DEBUGGER

USER MANUAL

SBE®

PROBUG Command Summary

In this list, commands are printed in upper-case, parameters in lower-case. Optional parameters are enclosed in brackets. Commas indicate separators between parameters; a dot is the terminator.

Aa.	Assemble
Ba[,b,c,d].	Breakpoint †
Ca,b,c.	Copy Memory
D[a,b].	Disassemble
E[a].	Execute DOS Bootstrap
Fa,b,c[,d].	Fill Memory
H[a,b].	Set Haltpoint †
Ia[,b].	Inspect/Alter Memory
J[a,b,c,d].	Jump To Location †
L[a,b].	Load Program Into RAM
M.	Inspect Stack Word by Word
Ma[,b].	Inspect Memory Word by Word
N[a].	Trace Next Instruction †
O[a,b].	Set Observation Point †
P[a,b].	Print Memory
Q[a,b,c,d].	Quiet Trace †
R.	Print Register Contents
Ra[,b].	Print/Alter Register Contents
Sa,b,c[,d].	Search Memory For Pattern
T[a,b,c,d].	Trace Instructions †
Wa,b[,c].	Write Program In S-Record Format
@[a,b,c].	Enter Transparent Mode
*[a].	List/Set Program Counter

† Program Execution Command

PROBUG Command Summary

In this list, commands are printed in upper-case, parameters in lower-case. Optional parameters are enclosed in brackets. Commas indicate separators between parameters; a dot is the terminator.

Aa.	Assemble
Ba[,b,c,d].	Breakpoint †
Ca,b,c.	Copy Memory
D[a,b].	Disassemble
E[a].	Execute DOS Bootstrap
Fa,b,c[,d].	Fill Memory
H[a,b].	Set Haltpoint †
Ia[,b].	Inspect/Alter Memory
J[a,b,c,d].	Jump To Location †
L[a,b].	Load Program Into RAM
M.	Inspect Stack Word by Word
Ma[,b].	Inspect Memory Word by Word
N[a].	Trace Next Instruction †
O[a,b].	Set Observation Point †
P[a,b].	Print Memory
Q[a,b,c,d].	Quiet Trace †
R.	Print Register Contents
Ra[,b].	Print/Alter Register Contents
Sa,b,c[,d].	Search Memory For Pattern
T[a,b,c,d].	Trace Instructions †
Wa,b[,c].	Write Program In S-Record Format
@[a,b,c].	Enter Transparent Mode
*[a].	List/Set Program Counter

† Program Execution Command

6/83

M-6790
PROBUG 2.0
July 1983

PROBUG

68000 SOFTWARE DEBUGGER

USER MANUAL

SBE, Inc.
4700 San Pablo Avenue
Emeryville, California 94608

Table Of Contents

Introduction To PROBUG

PROBUG is a sophisticated interactive debugger for testing software written for the 68000. PROBUG is designed to facilitate debugging on SBE's ModulasTen M68K10 single-board computer; it can be used on other 68000-based computers as well. PROBUG offers a number of commands designed specifically to help you identify the most commonly encountered problems.

PROBUG provides commands to:

* Print, inspect, and alter memory
* Print and alter registers
* Copy and fill memory
* Search memory for a pattern
* Assemble and disassemble instructions
* Download programs from, and write programs to, a host computer or other external device
* Boot up disk operating system

In addition, there are PROBUG commands that allow you to control execution of your program. With these **program execution commands**, you can:

* Breakpoint through your program, with optional iteration count
* Trace through RAM or ROM
* Halt program execution on memory change

There is also a way to call some of PROBUG's routines from within your program; this is discussed under **PROBUG Function Calls** toward the back of the manual.

For information on installing the PROBUG PROMs and jumpering the ModulasTen M68K10 single-board computer, see Appendix A, **Getting Started: PROBUG And The M68K10.**

If you have never used a debugger, we suggest you read Appendix B, **How To Use PROBUG,** for some suggestions.

For quick reference, the 3" x 5" card provided with this manual lists all PROBUG commands.

What Happens When You Turn The Power On

When you turn on the M68K10, several lines of output will appear on the terminal connected to channel "B". The current contents of the MPU registers (data, address, and several other registers) will be displayed on the terminal. The output will look like this:

```
PROBUG 2.0 - SBE SOFTWARE DEBUGGER
COPYRIGHT SBE, INC.  1983


      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
A) FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000200
PC)  FC0028    SR) 2700    CCR) -----    USP) FFFFFFFF    .SSP) 00000200
FC0028  MOVE    #$2700,SR
>
```

An explanation of this output appears under **MPU Registers.**

The > indicates that PROBUG is ready for you to type a command. This prompt appears whenever PROBUG has completed a command, as well as at the beginning of a debugging session.

PROBUG Command Syntax

A PROBUG command is a letter followed by one or more parameters and a terminator. For example, in the command **I1000.** the I represents an instruction to PROBUG to inspect the contents of memory. The **1000** is the parameter, which in this case tells PROBUG which memory location to inspect. The . is the terminator.

Some PROBUG commands do not require any parameters, but all require a terminator. You can use either a dot (.) or a carriage return (**<CR>**) as the terminator at the end of a command.

As a protection against syntax errors, PROBUG checks your input as it is typed, not just after you type a terminator. If you type a character that is invalid in the context in which you're typing it, that character will not appear on the screen. If you are trying to type and nothing is appearing on the screen, this may be the reason. Don't panic. Refer to the command description in this manual and enter a correct value.

PROBUG evaluates all expressions to 32 bits; if an expression translates to a number shorter than 32 bits, PROBUG adds the leading zeroes. In other words, PROBUG right-justifies all strings and numbers that you input.

Parameters

A PROBUG command may have one or more parameters, separated by either a comma or a space. The first parameter follows directly after the command, with no separator. In the command **C100,200,300.** the 100 is the first parameter, separated from the other parameters with commas. The command **C100 200 300.** means the same thing as the command **C100,200,300.**.

The parameters themselves may be expressed in a variety of ways. The most common way is to use ordinary hexadecimal digits (in the above example, 100, 200 and 300). Note that PROBUG expects numeric input to be hexadecimal. Likewise, it prints most output in hex, except for some output from the Disassemble command, which it prints in decimal.

Any number may be preceded by an ampersand (&) to make it decimal instead of hexadecimal. This holds true whether inputting the number, referencing a location, etc.

Another way of expressing a parameter is to enclose an ASCII string in single quotes (e.g., 'WXYZ'); the string will be translated into its numeric ASCII code. (Note: the ASCII string must be no more than four characters.) For example, the character **S** is represented in ASCII as a hex 53; thus, within a parameter, a quote-enclosed **'S'** has the same meaning as the hexadecimal number 53.

You can also define a parameter as the contents of a register. (Registers are discussed on the following page.) To define a parameter as the contents of a register, you must precede the register name with the letter "R". For example, **RD3** represents the contents of data register D3. You can use data registers (expressed as RD0 through RD7), address registers (RA0 through RA7), and user registers (R0 through R7; see below under **User Registers**) in this manner. Thus, if address register A3 contains the hex digits E0E0E0E0, the expression **RA3** has the same meaning as the expression **E0E0E0E0.**

RS represents the contents of the system stack pointer; **RU** represents the contents of the user stack pointer.

You can express any parameter as the result of adding or subtracting terms of any type (hex or decimal numbers, ASCII strings, or register contents). For example, **I1000+FF.** and **I10FF.** are equivalent. Another example: if address register A0 contained the hex number 0100, the command **I100+RA0.** would inspect memory location 200 (100 plus the contents of A0, which is 100).

Unary minuses are allowed. For example, you can use -1 as an expression.

An asterisk (*) represents the current contents of the program counter and can be used in any expression. For example, if the program counter is 2000, the commands **I2000.** and **I*.** are equivalent. There is also an * command which allows you to list and change the program counter.

Default Parameters. With some PROBUG commands, a default is assumed if you do not specify a parameter. For example, the D (Disassemble) command allows you to specify the location at which the disassembly should begin. With no parameter, **D.** begins disassembling at the location currently in the program counter, which is the default for this command. Defaults for other PROBUG commands are documented under the detailed command descriptions.

Registers

The contents of the 68000's eight address registers and eight data registers, labelled A0-A7 and D0-D7 respectively, are available for use within PROBUG. You can define parameters using the current contents of these registers in expressions; in addition, you can change register contents with the **R** command.

In addition to the data and address registers, there are eight user registers, R0 through R7. These may be used to help you match the locations on an assembly listing with those in RAM. Suppose your program is assembled at location 0000 but loaded in RAM at location 4000. If you store the number 4000 (the relocation factor) in user register R1, you can refer to **R1** instead of adding the relocation factor to each address you specify in PROBUG commands. For example, **20+R1** would have the same meaning as **4000+20** or **4020**. This both saves time and helps prevent typing errors.

Using a register reference to define a parameter frequently involves adding an offset to the register. Unlike with other terms, you can omit the plus sign when adding to address, data, or user register references. For example, the above expression **20+R1** can be abbreviated to **20R1**. The expression **10+RA2** can be abbreviated to **10RA2**.

Be careful when using unary minuses before register references; the implied addition takes precedence over the minus. For example, the expression **-10R1** evaluates to **-(10+R1)**, <u>not</u> to **-10+R1**.

If you do not specify the number of the user register, PROBUG assumes you mean user register R0. Thus, **1F+R0** and **1FR0** and **1FR** all have the same meaning (1F plus the value of user register R0).

MPU Registers. The data and address registers, program counter, status register, condition code register, user stack pointer, and system stack pointer are known collectively as the **MPU registers**. The user registers are not included as MPU registers.

The MPU register contents are displayed on the terminal whenever PROBUG regains control after your program executes. The current instruction is displayed, in disassembled form, on the next line. The following is a sample display of register contents and the disassembled current instruction.

```
      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00B00000
PC)  001000    SR) 0704    CCR) --Z--    .USP) 00B00000    SSP) 0000076C
001000  MOVEQ.L  #20,D0
```

The first line displays the contents of data registers D0 through D7; the second line, the contents of address registers A0 through A7. The third line displays the contents of the program counter (PC), the status register (SR), the condition code register (CCR), the user stack pointer (USP), and the system stack pointer (SSP).

The dot (.) before "USP" indicates that the 68000 is in user state. In this case, the USP contains the same number as address register A7. If the 68000

is in supervisor state, the SSP (system stack pointer) will be preceded by a dot and will contain the same number as register A7. See under the **R.** command for more on the MPU register contents.

Conventions Used In This Document

For readability, a dot (.) is used in this document as the terminator for PROBUG commands. You can use either a dot or a carriage return as a terminator. Parameters are denoted as lower-case letters, in boldface (e.g., **a** is the first parameter, **b** is the second, etc.). Parameters are separated by commas; you can substitute spaces.

Square brackets ([]) around a parameter indicate that the parameter therein is optional. If there are no brackets, you must specify the parameter. For example, in the command **Ia[,b]** the first parameter is mandatory and the second parameter is optional. Note that the comma (or a space) must be typed only if you specify the second parameter.

The **I**, **M**, and **R** commands print long words one word (two bytes) at a time, with a space in between to improve readability. Accordingly, in sample output from these commands, we separate every two bytes with a space. This space should not be interpreted as a character.

In examples that show your input and PROBUG's output, your input is shown in boldface to distinguish it from PROBUG's output. The _ symbol indicates a press of the space bar.

Special Keys

When typing PROBUG commands, use the backspace or the DELETE or RUBOUT key to backspace over an incorrectly typed character. Press LINE FEED or CTRL-X (hold down the CONTROL key while pressing "X") to cancel the whole expression you're typing and start over.

Use the ESCAPE (ESC) key to interrupt the output of commands that generate large amounts of output. These commands include Print, Search, and Trace.

CTRL-L: Retrieve Previously Typed Character(s). PROBUG stores each character you type in a buffer. When you type the command **I80F0.**, for example, the "I" is in position 1 of the buffer, the "8" is in position 2, the "0" in position 3, etc. The buffer changes, position by position, as you type a new line.

You can use part or all of the previously typed line on the current line by using CONTROL-L (hold down the CONTROL key while pressing "L") for each character you want to retrieve. This is particularly useful when typing long numbers. For example, suppose you just typed **IDE0800.** and then realized you meant to type **PDE0800.** Instead of cancelling the line and starting over, you could backspace over the command, type "P" and then type CTRL-L six times to retrieve the number.

This feature is much more easily demonstrated than documented; experiment with it to see how it works.

Program Execution Commands

PROBUG offers an assortment of **program execution commands** for executing and testing your program. With these commands, you specify the way in which you want your program executed: where it should start executing, where and why it should halt, etc. PROBUG temporarily passes control to your program and lets it execute as you specify. PROBUG regains control only after the program or program segment has finished executing.

Whenever PROBUG regains control after a program execution command, it displays the current contents of the MPU registers, and prints a message about the circumstances under which the program stopped executing.

The program execution commands are as follows:

B Breakpoint
J Jump
N Next Instruction(s)
Q Quiet Trace
T Trace Instructions

In addition, the commands

H Set Haltpoint
and **O** Set Observation Point

are used in conjunction with program execution commands, as described below.

Which Program Execution Command To Use. Breakpoints, haltpoints, trace, and quiet trace are the four main methods of controlled execution of your program. The advantages of each method are described in this section.

Breakpoints are perhaps the most valuable diagnostic tool of all the program execution commands. They let you quickly identify problems in your program by displaying the contents of processor registers after your program or program segment has executed. You can then discover and correct problems using other PROBUG commands.

When you use the Breakpoint command, PROBUG sets a breakpoint at the location you specify. Your program then starts executing from the location in the program counter, following the flow of the program (including subroutines, etc.). The program executes, in real time, until it reaches the breakpoint location. If you specify more than one location as a breakpoint, a breakpoint will occur at whichever location is encountered first.

When your program reaches the designated location, it stops executing. The processor registers are displayed and saved, and the locations you specified are no longer breakpoints. You can now examine memory, correct problems, etc.

Breakpoints can be set in RAM only, and only on the first word of an instruction. To debug programs in ROM, you must use traces or quiet traces.

no p. 6

Traces let you step through your program one instruction at a time, displaying the register contents after each instruction is executed. Before executing each instruction, PROBUG displays the processor registers and disassembles the next instruction. Because instructions are traced one at a time, the program cannot be executed in real time during a trace.

There are two trace commands: **N** and **T**. With the **N** command (**Trace Next Instructions**), you can specify the number of instructions to trace; as soon as the specified count is reached, the trace stops. With the **T** command, you can specify up to four addresses; the trace then lasts until any of the addresses is reached.

Observation points (discussed below) are monitored during traces. Haltpoints (also discussed below) are also monitored during traces, but because of the 68000's "trace interrupt" feature, PROBUG doesn't have to write the haltpoints into your program. Programs can thus be traced, with haltpoints set, whether they're in RAM or ROM. This is a major advantage of both traces and quiet traces.

The **T** command cannot be used to trace through interrupts or exceptions.

Quiet Traces provide a way of tracing through your program without generating much output on the screen. A quiet trace is much faster than a regular trace, since it prints register contents only after tracing all instructions, rather than printing them after each instruction is executed.

The **Q** command offers several advantages over other program execution commands. First, as mentioned above, you can use it to trace through either ROM or RAM, with or without haltpoints set. Second, when you use it with the **O** command, you can monitor a particular location in RAM (called an observation point) to find out exactly when that location changes. See below in this section for more on observation points.

Another major advantage of quiet traces is that they can be used to follow the flow of program control. When the address you specify on the **Q** command is reached, PROBUG prints the previously executed instruction as well as the current instruction, in disassembled form.

During a quiet trace, the program will execute much more slowly than in real time, but not as slowly as with the **T** command.

Jumps are the simplest way of executing your program from PROBUG. No fuss, no muss, just execute. Using the **J** command, you can start the program either at the program counter (the default), or at another memory location (by specifying one parameter). If you specify more than one parameter, execution starts at the location specified in the first parameter, and PROBUG sets breakpoints at the locations you specified as the other parameters.

Observation Points are used in conjunction with quiet traces, traces, haltpoints, and breakpoints to monitor locations in RAM. When the contents of the specified location change, PROBUG prints a message telling you of the change. Observation points can be used either to report any change at the location, or to print a message only when the contents of the location change to a

particular value.

Sometimes a program erroneously changes a location in memory that is supposed to remain constant. Or, a location is being set to an incorrect value. Observation points are very useful in locating the instruction that is changing the contents of a particular location in memory.

When you use observation points with the trace modes (usually quiet trace), the memory locations being observed are checked after each instruction. When a change is detected, program execution stops and PROBUG prints the address of the observation point, the location's previous contents, its current contents, and the address of the instruction that changed the location.

Observation points can also be used with breakpoints and haltpoints, but since the program runs in real time, the observation points are checked only when a breakpoint or haltpoint is reached. PROBUG reports any change that occurred between the starting location of the jump/breakpoint and the location at which execution stopped.

When haltpoints with iteration counts are used with observation points, memory is checked each time the haltpoint is encountered. This can be used to combine the advantages of observation points with real-time execution. (See under Haltpoints, below.)

Note that the **O** command only sets observation points; it does not automatically generate the quiet trace necessary to monitor the location(s) in question. Observation points remain set until you remove them.

Haltpoints are similar to breakpoints, but are kept in a table where they stay until you remove them. The table will hold up to 8 haltpoints. You set locations in this table using the **H** command.

Haltpoints are especially useful when there are particular locations you always want to designate as breakpoints. Instead of specifying the locations every time you issue a breakpoint command, you can set them as haltpoints. This is also useful if you want to specify more than 4 locations as breakpoints (four is the limit on a breakpoint command).

You can set haltpoints at error exits or exit points to block exit routes. This is useful for keeping control of your program as it executes.

Another advantage of haltpoints is that you can set them in RAM or ROM. When used with RAM, haltpoints are similar to breakpoints: when your program encounters a haltpoint, a breakpoint occurs and control is returned to PROBUG.

Only the **T**, **Q**, and **N** commands will recognize haltpoints set in ROM. With these commands, PROBUG checks the haltpoint table between each instruction, and stops program execution when a haltpoint is reached.

After reaching a haltpoint, you can continue program execution from that location without clearing it from the haltpoint table. In other words, if the program is currently at a haltpoint, that haltpoint is ignored when program execution continues.

Iteration Count With Haltpoints. To assist in monitoring loops, you can specify an **iteration count** as part of a haltpoint command. This count is the number of times that the instruction at the haltpoint location should be executed before control returns to PROBUG. Once the iteration count is decremented to zero, control will be returned to PROBUG each time the haltpoint is encountered.

When observation points are set, they are monitored each time the haltpoint is reached, and also after the instruction at the haltpoint is executed. This allows some of the usefulness of observation points to be retained while running in real time. By setting several haltpoints with large iteration counts at strategic points in the program, you can narrow down the general area in which the observation point is being changed.

If you do not want to set an iteration count, trace through ROM, or use the haltpoint table to set permanent breakpoints, breakpoints may be a more useful debugging technique than haltpoints.

Error Messages

Any time an exception occurs, PROBUG prints an error message enclosed in asterisks, giving the reason for the exception. The current contents of MPU registers are also printed. All register contents are preserved, and you are returned to command mode (the > prompt appears).

The errors that occur most commonly are **bus errors** and **address errors**. Both types of errors can occur from within PROBUG as well as from within your program; they are handled differently in the two cases. When they occur from within PROBUG, the registers that were previously on the stack remain unchanged on the stack. When the errors occur from your program (as a result of a program execution command), the register contents are printed along with the diagnostic message.

Bus errors occur when a program execution command or your program itself tries to reference a location where there's no RAM or ROM. For example, suppose there is no RAM above the location 1FFFF and you typed a **P** command to print the contents of locations 1FFE0 through 20010. The results would look something like this:

```
>P1FFE0,20010.
01FFE0  3D7C 0001 006A 2D6F   0001 006C 026F 7FFF   =|___j-o___l_o__
01FFF0  6120 0C50 4AFB 6602   3080 5341 66F0 426E   a _PJ{f_0_SAfpBn
020000  *** BUS ERROR ***
>
```

Address errors occur when a program execution command (including **O** and **H**), or your program itself, tries to access an odd location such as 1001. Here is a sample of what PROBUG might print if your program tried to access location C01.

```
>J48000.

*** ODD ADDRESS ERROR ***
OCCURRED AROUND 04800E WHILE ACCESSING 00000C01
INSTRUCTION CODE 5378    FUNCTION CODE 5

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 0000076C
PC) 04800E    SR) 2704    CCR) --Z--    USP) 00B00000    .SSP) 0000076C
04800E  CLR.W   D1
>
```

This tells you that the error occurred around location 04800E while the program was trying to access location 0C01. The opcode of the instruction that caused the error is 5378. You could now use a PROBUG command to inspect the problem area, looking for instruction 5378 around location 04800E.

If you try to access an odd location using a non-program execution PROBUG command, PROBUG will either round down to the next even location or print the message *** ODD ADDRESS ERROR *** with no other information. If a command rounds down when you try to access an odd location, its command description will say so.

NMI (Non-Maskable Interrupt) Button

You can safely interrupt any currently executing program that is non-timing-dependent by pressing the NMI button (called the ABORT button on some machines). This is useful for determining why a program is hung, regaining control of a program that never reached a breakpoint, etc.

PROBUG will display the contents of the registers at the point of the interrupt and will return a > prompt. The contents of the MPU registers are saved.

The following is a sample of output resulting from a press of the NMI button:

```
*** NON-MASKABLE INTERRUPT ***

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00B00000
PC)  001000    SR) 0704    CCR) --Z--    .USP) 00B00000    SSP) 0000076C
001000   ADDQ.W   #2,D0
>
```

After the message and register contents are printed, you can execute any PROBUG command. To continue program execution at the point at which you pressed the NMI button, enter a program execution command (e.g., jump, breakpoint, etc.).

RESET Button

The RESET button resets the M68K10; it is equivalent to a power-up. If the processor halts, or if your program mistakenly overwrites PROBUG's RAM area, RESET is the button to push.

RESET differs from the NMI button in the following ways:

1) RESET resets all the M68K10's I/O components, and generates a RESET for the iSBX connectors and the Multibus.

2) RESET loads the system stack pointer with PROBUG's initial system stack pointer. (NMI leaves the stack pointer as it was before.)

3) RESET sets the program counter to the beginning of PROBUG. (NMI leaves the program counter as it was before.)

4) RESET initializes the PROBUG variables and sets up all exception vectors.

Like NMI, RESET restores breakpoints and haltpoints that were set in RAM and prints the contents of the registers.

The following is a sample of output resulting from a press of the RESET button:

```
PROBUG 2.0 - SBE SOFTWARE DEBUGGER
COPYRIGHT SBE, INC.  1983

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00000200
PC)  FC0028     SR) 2700     CCR) -----     USP) 00B00000     .SSP) 00000200
FC0028  MOVE   #$2700,SR
>
```

PROBUG COMMAND DESCRIPTIONS

Aa. ASSEMBLE

The **A** command allows you to patch programs using assembly language instructions. PROBUG converts these instructions into hex and puts them in RAM at the location you specify. PROBUG accepts all standard Motorola 68000 mnemonics and addressing modes, plus the DC (Define Constant) directive.

To enter instructions into RAM, specify the desired starting location as a parameter; for example, type **A1000.** to begin entering instructions at location 1000. PROBUG responds by displaying address 1000 as a prompt. Enter each instruction followed by a carriage return; PROBUG assembles the source line and prompts you with the RAM address of the next instruction. Press carriage return alone on a line to exit assemble mode and return to PROBUG command mode.

In the following example, your actions are shown in boldface, PROBUG's in lightface:

```
>A1000.
001000  MOVEQ  #2,D0 (carriage return)
001002  MOVE.W D0,$0C00 (carriage return)
001006  (carriage return)
>
```

In order to detect and flag errors, PROBUG parses each instruction as you type it, and ignores invalid characters. Most syntax errors are thus detected while the line is being entered. Those that are not detected until you press carriage return cause PROBUG to print an error message. The address prompt is overwritten with asterisks and is then re-issued.

Immediate data constants that you enter are assumed to be decimal. To define a hex value, precede the value with a $ - e.g., **$F000**. To enter an ASCII literal, use single quotes around the character string.

You can type any number of spaces before the mnemonic or between the fields. To document patches, you can add comments to instructions; however, comments are not saved anywhere and are thus useful only if you're using a printing terminal. To add a comment to an instruction, type a space after the operand field and comment away.

To facilitate skipping over existing instructions, or overwriting instructions that have already been assembled, the + and - keys can be used to step forward and backward in memory (respectively). For example:

```
00200A  -
002008  -
002006  NOP
```

As in PROBUG command mode, you can use addition, subtraction, and user register references within expressions. For example, if user register **R0** contains 1000 (hex), the source lines **BEQ $100C+R0** and **BEQ $200C** have the same meaning. You can also use * to reference the current program counter (the address of the current instruction).

no p. 14

Whenever possible, PROBUG converts your instructions to a more compact form. For example, **ADD #1,D0** becomes an **ADDQ #1,D0**; a **MOVE.L #1,D0** becomes a **MOVEQ #1,D0**; etc.

The Define Constant (**DC**) assembly directive is allowed with bytes, words, and long words (**DC.B**, **DC.W**, and **DC.L**). The operands may be expressions or ASCII literals enclosed in single quotes (e.g., **DC.B 'TEST STRING',$04**).

The following are examples of use of the **DC** directive, and the resulting ASCII code in RAM (in hex). Each pair of hex numbers represents one byte. An underscore (_) indicates that PROBUG will skip over that byte to align the next instruction on a word boundary. (This occurs when you use **DC.B** with an odd number of characters or numbers in the operand.)

Directive & Operand	Resulting ASCII Codes (Hex)
DC.B 1,2,3,4,5	0102030405__
DC.W 'ABCDE'	414243444500
DC.L 0+'AB','CDE'	0000414243444500
DC.B 'TEST STRING',$04	5445535420535452494E4704

Note that strings of ASCII literals are left-justified and padded with zeroes. Expressions are right-justified.

Branch Labels. To aid in resolving branch addresses, four special labels are allowed: **:A**, **:B**, **:C**, and **:D**. Only branch instructions and decrement-and-branch instructions may reference these labels. Each label's value is defined when the label appears at the beginning of the line.

You can make references to as-yet undefined labels. When you define the label, the address is resolved. Backward references are also allowed.

If you attempt to leave assemble mode with a label undefined, you will get the message *** UNDEFINED LABEL *** and you will be re-prompted with the most recent address. In such cases, you need not re-type the entire line at which you forgot to define the label. Just go back to the line (using - if necessary) and type the label alone on the line. The source line will remain as well as the newly attached label.

This example shows the use of labels (the labels are boldfaced here):

```
001000      CMP.B  #'0',D0
001004      BLO    :B
001006      CMP.B  #'9',D0
00100A      BLS    :A
00100C      CMP.B  #'A',D0
001010      BLO    :B
001012      CMP.B  #'F',D0
001016      BHI    :B
001018      SUB.B  #7,D0
00101A  :A  AND.B  #$0F,D0
00101E      BRA    :C
001020  :B  MOVEQ  #-1,D0
001022  :C  BRA    $1000
001024  (carriage return)
```

When using the **A** command, you can substitute certain mnemonics for others:

BLO is allowed for **BCS**
BHS is allowed for **BCC**
DBLO is allowed for **DBCS**
DBHS is allowed for **DBCC**
DBRA is allowed for **DBF**

Ba[,b][,c][,d]. **BREAKPOINT** (Program Execution Command)

Before using this command, please read the section **Program Execution Commands** above in this manual to compare breakpoints with haltpoints, traces, and quiet traces.

The **B** command causes your program to execute until (one of) the location(s) you specify is reached. The program always starts executing at the location in the program counter. (If you want to breakpoint starting at a location other than the one in the program counter, use the **J** command.)

The command **B1000.** says to execute the program from the current location and stop at location 1000. The contents of various registers are displayed when location 1000 is reached. Here's a sample:

```
>B1000.
BREAKPOINT AT 001000

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00B00000
PC)  001000    SR) 0704    CCR) --Z--    .USP) 00B00000    SSP) 0000076C
001000   SUBQ.W    #1,$0C00
>
```

If you specify more than one parameter, the program stops executing as soon as it encounters any one of the locations specified by the parameters. For example, with the command **B1000,409C,2000,FF00.** the program would stop executing as soon as location 1000, 409C, 2000, or FF00 had been reached. Register contents are displayed as soon as program execution stops.

If you have set haltpoints (with the **H** command) and they are encountered before the breakpoint location(s), program execution will halt before the breakpoint location(s). If you have set observation points and they change during program execution, a message will be displayed to indicate this.

Note: Do not use locations as breakpoints if you have already set them as haltpoints.

As soon as breakpoint output is displayed, all the breakpoints you specified are cleared. The location at which program execution stopped is now the current location.

If the program never reaches a breakpoint, press the NMI button to regain control.

Ca,b,c. **COPY MEMORY**

This command copies the contents of location **a** through **b** (called the source block) into locations **c** through **c + (b - a)** (called the destination block). Since the **C** command by itself generates no output, several **P** (print) commands appear in the following example to illustrate the effects of a Copy.

```
>P1000,100F. (Print contents & ASCII representation of locations 1000-100F)

001000  5678 9ABC 9999 9999   FFFF FF60 FFFF 12FF   Vx_<_____`____

>C1000,1003,2000.  (Copy contents of 1000 through 1003 to location 2000)

>P2000,200F. (Print contents & ASCII representation of locations 2000-200F)

002000  5678 9ABC 3D7C 0001   006E 2D6F 0072 006C   Vx_<=|___n-o_r_l

>P1000,100F. (Print contents & ASCII representation of locations 1000-100F)

001000  5678 9ABC 9999 9999   FFFF FF60 FFFF 12FF   Vx_<_____`____

>C1000,1003,1002.  (Copy contents of 1000-1003 to location 1002)

>P1000,100F. (Print contents & ASCII representation of locations 1000-100F)

001000 5678 5678 9ABC 9999   FFFF FF60 FFFF 12FF   VxVx_<_____`____
>
```

D[a][,b]. DISASSEMBLE

The **D** command displays assembled code in its original state (assembly language) for your perusal. Typically, you use **D** to examine the next instructions to be executed by a breakpoint or jump command.

The **D** command with no parameters starts the disassembly at the location in the program counter, and disassembles ten instructions. If you specify one parameter (**a**), the disassembly will start at that location, and disassemble continuously. For example, **D1000.** starts the disassembly at location 1000 and keeps going until you type a character. If you specify two parameters, the disassembly will start at location **a** and stop after disassembling the instruction at location **b**.

To freeze output on the screen, press the Escape key; to start it up again, press the Escape key again. To terminate the disassembly, type any other character on the keyboard.

Output From The D Command. For each instruction, the **D** command prints the address, hexadecimal instruction code, standard mnemonic for that code, and operand(s).

Illegal instructions are disassembled as **DC.W** directives. The **D** command resolves branch instructions to their absolute addresses. Instructions using the program counter with the displacement mode of addressing are printed with the displacement resolved to an absolute memory location.

Note: Normally, PROBUG displays operands in hex, preceded by a **$**. PROBUG displays immediate operands in the range -31 to +31 in decimal.

The following is a sample **D** command and its output:

```
>D48000.
048000  7014            MOVEQ.L    #20,D0
048002  31C0 0C00       MOVE.W     D0,$0C00
048006  4241            CLR.W      D1
048008  5440            ADDQ.W     #2,D0
04800A  5378 0C00       SUBQ.W     #1,$0C00
04800E  66F8            BNE.S      $048008
048010  4E71            NOP
048012  60EC            BRA.S      $048000
048014  4E71            NOP
048016  FFFF            DC.W       $FFFF
>
```

E[a]. EXECUTE DISK OPERATING SYSTEM BOOTSTRAP

This command starts execution of the bootstrap program which loads the disk operating system into RAM. You can specify the disk drive (0, 1, 2, or 3) from which the system should be loaded. For example, the command **E1.** would load the system from drive 1. If you do not specify the drive, the system will be loaded from drive 0.

If an error occurs while the system is being loaded, control returns to PROBUG and the error codes are displayed in the registers. Your disk operating system documentation should contain a description of the bootstrap program, including the codes for possible errors. See that document for a translation if an error code appears in the registers.

Fa,b,c[,d]. FILL MEMORY

This command writes the value **c** into locations **a** through **b**. Depending on the size of **c** (1 byte, 2 bytes, or 4 bytes long), PROBUG will choose byte, word, or long word mode.

Since the **F** command by itself generates no output, several **P** (print) commands appear in the following example to illustrate the effects of a Fill.

```
>F1000,101F,4C. (Fill locations 1000 through 101F with 4C, byte by byte)
>P1000,101F. (Print contents & ASCII representation of locations 1000-101F)

001000  4C4C 4C4C 4C4C 4C4C   4C4C 4C4C 4C4C 4C4C   LLLLLLLLLLLLLLLL
001010  4C4C 4C4C 4C4C 4C4C   4C4C 4C4C 4C4C 4C4C   LLLLLLLLLLLLLLLL

>F1000,101F,004C. (Fill locations 1000-101F with 004C, word by word)
>P1000,101F. (Print contents/ASCII of locations 1000-101F)

001000  004C 004C 004C 004C   004C 004C 004C 004C   _L_L_L_L_L_L_L_L
001010  004C 004C 004C 004C   004C 004C 004C 004C   _L_L_L_L_L_L_L_L
>
```

The optional parameter **d** indicates the amount by which PROBUG should increment value **c** at each consecutive location:

```
>F1000,101F,E0,02. (Fill 1000-101F with E0, incrementing each byte by 02)
>P1000,101F.  (Print contents/ASCII of locations 1000-101F)

001000  E0E2 E4E6 E8EA ECEE   F0F2 F4F6 F8FA FCFE   `bdfhjlnprtvxz/~
001010  0002 0406 080A 0C0E   1012 1416 181A 1C1E   ________________
>
```

If the size of the increment differs from the size of the value itself, PROBUG will switch modes to accommodate the larger of the two. For example, the command **F1000,2000,E0,100.** would force PROBUG into word mode. Similarly, if the value is larger than the increment, the value will determine the mode.

When operating in word or long word mode, the **F** command will begin a fill only at an even location. If you specify an odd starting location it will be rounded down to make it even. Thus, the commands

```
F1001,2000,F0F0,02.
F1001,2001,F0F0,02.    and
F1000,2000,F0F0,02.
```

all perform the same function.

Ha[,b]. **SET HALTPOINT(S)** (With Program Execution Commands)

Before using this command, you should read the section **Program Execution Commands** earlier in this manual.

Use the **H** command to set up to eight haltpoints. The haltpoints will remain in the haltpoint table until you remove them (or until the computer is turned off). Note that the **H** command only allows you to set the haltpoints; it does not automatically cause the program to execute. After setting haltpoints, you must use a program execution command to execute the program.

Haltpoints are treated much like breakpoints when encountered by program execution commands other than traces. After setting haltpoints in ROM, you must use a trace (**T**, **N**, or **Q** command) to monitor those haltpoints.

To set a haltpoint at a location, type **Ha.** where **a** is the location. For example, the command **H1000.** would set a haltpoint at location 1000. To display the table of all the haltpoints you have set so far, type **H.** with no parameter.

Iteration Count. To assist in monitoring loops, you can specify an iteration count as the second parameter of a haltpoint command. (The iteration count is assumed to be hexadecimal unless you precede it with an &). For example, suppose you entered the command **H1000,&100.** and then executed a program execution command. The instruction at location 1000 would be executed one hundred times before control was returned to PROBUG.

Type **-H.** to remove all haltpoints. Type **-Ha.** to remove only the haltpoint at location **a**. The following is an example of output from the **H** command:

```
>H10F0.    (Set location 10F0 as a haltpoint)
>H90C,4.   (Set location 90C as a haltpoint with count of 4)
>H.        (List all current haltpoints)
HALT POINTS
00090C  COUNT=0004
0010F0
>-H10F0.   (Remove location 10F0 from the haltpoint table)
>H.        (List all current haltpoints)
HALT POINTS
00090C  COUNT=0004
>
```

Ia[,b]. INSPECT/ALTER MEMORY

This command lets you inspect and alter memory locations one byte at a time, beginning at location **a**. For example, the command **I1000.** displays memory address 1000 and its contents. You can alter the contents by typing in the new value followed by a terminator (see below). To retain the existing contents, simply type a terminator without typing a new value. Whether you continue to inspect memory locations or exit the I command depends on the kind of terminator you use.

Terminators. To exit the I command, use the conventional . or **<CR>** as a terminator. (They serve this function whether you alter memory locations or not.) The space bar and comma also serve as terminators within the I command. They allow you to continue to inspect/alter memory locations. Press the space bar to get to the next location; type a comma to go backward one location. Both of these terminators can be used successively to skip forward or backward more than one location.

The following is an example of the use of the I command. The _ symbol represents a press of the space bar. Your actions are shown in boldface.

```
>I1000.           (Inspect/alter memory location 1000)
001000  00 23_    (Replace 00 with 23; go to next location)
001001  00 _      (Leave as is; go to next location)
001002  FF 33,    (Replace FF with 33; go to previous location)
001001  00 ,      (Leave as is; go to previous location)
001000  23 .      (Leave as is; exit I command)
>
```

Amount Of Memory Per Line. The I command accepts a parameter that specifies the amount of memory to be displayed at a time. To display every other byte (see the next paragraph), specify "1"; to display 2 bytes (1 word) at a time, specify "2"; to display 4 bytes (1 long word) at a time, specify "4". For example, the command **I1000,2.** says "inspect 2 bytes (1 word) of memory beginning at location 1000." If you do not specify this parameter, it is assumed you want every byte displayed, one byte at a time.

Sometimes it is useful to examine only the odd or only the even bytes. For this purpose, you can specify "1" for byte mode which will display only every other byte, one byte at a time. The result of specifying "1" depends on whether the address given with the command is odd or even. For example, the command **I1001,1.** would display

```
001001  E0E0 E0E0_
001003  E000 0100_
001005  12FF FF05_     etc.  (_ represents a press of the space bar)
```

Each time you press the space bar, the contents of the next odd address are displayed; the contents of even addresses are never displayed. Similarly, the command **I1000,1.** displays the contents of even address 1000, 1002, etc. (with spaces typed by you in between), and never displays the contents of odd addresses.

Use the parameter 2 (for word mode) to display 2 bytes at a time; use 4 (for long word mode) to display 4 bytes at a time. Example: the command **I1000,4.** would display

```
001000  E0E0 E0E0_
001004  E0E0 E0E0_
001008  E0E0 E0E0_     etc.
```

J[a][,b][,c][,d]. **JUMP TO LOCATION** (Program Execution Command)

Please read the section **Program Execution Commands** earlier in this manual before using this command.

The J command jumps to location **a** of memory and begins executing at that point. If you do not specify a location, J starts execution at the current program counter.

When you specify two or more parameters, the J command is interpreted as a breakpoint command. That is, the first parameter is the point at which PROBUG is to start executing the program; the other parameters are interpreted as breakpoints. For example, the command **J1000,1F00,2000,C090.** jumps to location 1000 and executes until one of the three locations 1F00, 2000, or C090 is reached. See the **B** command for details on breakpoints.

The following is a sample of the use of the J command. In this sample, a haltpoint had been set at location 90C.

```
>J900.  (Jump to location 900 and begin executing)

HALTPOINT AT 00090C

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00B00000
PC)  00090C    SR) 0704    CCR) --Z--    .USP) 00B00000    SSP) 0000076C
00090C   BNE.S    $000940  (FALSE)
>
```

L[a][,b]. LOAD PROGRAM INTO RAM

The **L** command loads a program into RAM from an S-record formatted object tape or from a host computer that produces output in S-record format. The program is loaded according to the load addresses in its S-records. Data is not displayed as it is being read.

With a parameter, **L** loads the specified (hexadecimal) number of files into memory: for example, **L2.** loads two files into memory. Each of the files to be loaded must end with an S8 or S9 record (see below for a description of S-record format).

You can specify the starting location in RAM (in hex) using a second parameter: for example, **L2,1000.** loads two files starting at location 1000. That is, the first program record is loaded at location 1000 and subsequent records are loaded relative to that location.

To cancel a Load from tape, type "S1" (upper-case "S") followed by a line-feed. To cancel a Load from a host computer, type any character while the data is being loaded, and the load will abort.

Select Port. Before using the **L** command, you may want to select the port using the **@** command. See under that command for a description of how to do this.

Transparent Mode. Each time you use the **L** (or **W**) command, PROBUG automatically enters a special "transparent" mode which allows your terminal to communicate with the host computer or other external device. While PROBUG is in transparent mode, the characters you type at the terminal are sent to the other port, and output from the host computer goes to your terminal.

You should run both ports at the same baud rate. (If your only task is to download programs, the two ports may not need to be at the same speed.)

Once you are in transparent mode, you can type the appropriate command telling the host computer to start sending data. The command line would typically end in a carriage return, but do **not** end the command line as you normally would on the host computer. Instead, type the "exit character" (explained below) to terminate the command line. PROBUG will send the substitution character to the host computer; this terminates the command line and starts the transmission of S1 records into RAM.

Exit Character. When you type a Load command, a message like the following appears:

```
EXIT CHARACTER = $01 = CONTROL-A
```

This indicates what character to type when you want to begin the actual transfer of data into RAM. The exit character causes an exit from transparent mode. While you are in transparent mode, the exit character is the only character you can type that does not get sent to the host computer. Instead, when you type the exit character, a "substitution character" is sent to tell the host computer to begin the load.

Because a carriage return signifies the end of a line in most computer systems, the substitution character has been set to carriage return. In other words, when you type the exit character, the host computer receives a carriage return. If your host computer expects a character other than carriage return at the end of a command line, be sure to change the substitution character to that character using the **@** command. The exit character can also be set using the **@** command.

When PROBUG exits transparent mode to start the load process, the host computer, having received the substitution character ending the line, begins sending the S1 records. Typing the exit character synchronizes the host computer with the M68K10.

Note that while PROBUG is in load mode (loading S1 records from the alternate port), typing any character at the terminal will abort the load process.

S-Record Format

The host computer's output, or the object tape, consists of a series of data records and an end-of-file record. Each record consists of pairs of ASCII characters representing hexadecimal digits; each pair of characters represents one 8-bit byte. Data records include a header, the number of bytes in the record (in hex), the program block's address, the program data itself, and a checksum for the record. The end-of-file record includes all of these fields except that it does not contain program data.

There are two types of data records: S1 and S2. The data record type is marked at the beginning of the record, in ASCII. If the program loads into the first 64K bytes of RAM, its load address is two bytes long, and the data record starts with the ASCII characters "S1". If not, the record starts with "S2". The only difference between S1 and S2 records is a 2-byte versus a 3-byte load address.

The first byte of the record, after the "S1" or "S2", contains the number of bytes to follow in the record. This number is given in hex and includes the starting address and checksum. The next two bytes (three bytes if it is an S2 record) show the address at which the record is to be stored in memory. Next follows the actual program/data bytes and, after that, a checksum byte. The checksum is the one's complement of the summation of all previous bytes in the record, including the record length and address bytes. See the example below.

The end-of-file record is labelled according to the type of data record in the group. If they were S1 records, the end-of-file record is marked "S9". If they were S2 records, it is marked "S8". The end-of-file record does not contain any program data. Its address is the address at which the program will start executing.

The following is a sample of S-record format. Since there is no output from the **L** command, you see this type of record only when using the **W** command.

The records take up one line each. In the first line, **S1** is the record

type, **0B** is the number of bytes, **1000** is the address, **FEDCBA9876543210** is the data, and **AC** is the checksum. In the second line, **S9** is the record type, **03** is the number of bytes, **1000** is the address, and **EC** is the checksum. Note that there is no data in the **S9** record.

```
S10B1000FEDCBA9876543210AC
S9031000EC
```

PROBUG does not control the start/stop functions of the tape, so you must manually start the tape after typing the **L** command. Stop the tape as soon as the PROBUG prompt (>) appears; otherwise, any characters following the end of the program on the tape will be interpreted as PROBUG commands.

All characters between records are ignored by the **L** command; thus **<CR>** and line feed may be used to terminate each record. If PROBUG detects an error (e.g., invalid character or checksum failure) during the loading process, it will display an appropriate error message.

The **L** command puts the address given in the S9 or S8 record into the program counter. Thus, if you execute a **J** command directly after executing an **L** command, the program will start executing at its start address.

N[a]. TRACE NEXT INSTRUCTION(S) (Program Execution Command)

The **N** command is identical to **T** (Trace) but it executes only the next instruction. See the section **Program Execution Commands** and the description of the Trace command for more details on tracing.

The command **N.** with no parameters traces the next instruction - that is, the one at the current value in the program counter. Control is then returned to PROBUG. After using **N**, you can trace subsequent instructions by pressing carriage return or typing a dot (.). You can continue using carriage return or dot to single-step through the program until you type another command.

```
>N.

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00B00000
PC)  04100C    SR) 0704    CCR) --Z--    .USP) 00B00000    SSP) 0000076C
04100C  NOP
>
```

The **N** command with a parameter traces the number of instructions you specify; for example, **N3.** traces three instructions starting at the next instruction and following the flow of the program.

Suppose you set observation points and then use an **N** command to trace the next ten instructions. If an observation point changes before ten instructions have been traced, the trace will stop after the instruction that changed the observation point; all ten instructions will not be traced.

M. INSPECT STACK WORD BY WORD

The **M** command with no parameters allows you to inspect/alter the contents of either the user stack or the system stack, one word at a time, depending on whether the 68000 is in user or supervisor state. Use the same terminators as with the I command to move forward and backward in the stack and to exit the **M** command. The following is a sample of the use of the **M** command:

```
>M.                 (Inspect stack word by word)
0007B6  0000 1F_    (Replace 0000 with 001F; go to next location)
0007B8  1021 _      (Leave as is; go to next location)
0007BA  FFEB FFEC,  (Replace FFEB with FFEC; go to previous location)
0007B8  1021 ,      (Leave as is; go to previous location)
0007B6  001F .      (Leave as is; exit M command)
>
```

Ma[,b]. INSPECT MEMORY WORD BY WORD (2 bytes at a time)

This command is identical to the **Ia[,b].** command except that its default mode is word mode. That is, if you don't use the second parameter (b) to specify the amount of memory you want displayed per line, the system displays 2 bytes at a time. The following example assumes memory location 928 contains 4E71.

```
>M928.              (Inspect 1 word of memory starting at location 928)
000928  4E71 _      (Leave as is; go to next word)
00092A  BE06 0A_    (Change BE06 to 000A; go to next word)
00092C  FE02 .      (Leave as is; exit M command)
>
```

Use the same commands as with the I command to travel through memory.

The **M** command accepts even addresses only.

The **M** and I commands offer you the convenience of inspecting memory either in bytes or in words, without having to specify the size.

Oa[,b]. SET OBSERVATION POINT (With Program Execution Commands)

Observation points are used in conjunction with quiet traces, traces, halt-points, and breakpoints to monitor locations in RAM. When the contents of the specified location change, PROBUG prints a message telling you of the change. Observation points can be used either to report any change at the location, or to print a message only when the contents of the location changed to a specific value. Please read **Program Execution Commands** earlier in this manual before using observation points.

You can set an observation point in two ways: with one or with two parameters after the **O** (the letter O, not zero). With one parameter (**Oa.**), PROBUG reports whenever the contents of location **a** change. When you specify a second parameter (**Oa,b.**), PROBUG prints a diagnostic message only when the contents of location **a** change to target value **b**.

Specify only word addresses as observation points. Observation points apply to whole words, not to individual bytes; for example, if you specify location 1000, both 1000 and 1001 are observed.

PROBUG keeps a table of up to 8 observation points and their corresponding target values, if any. The locations remain in the table until you remove them or until the computer is turned off.

Note that the **O** command only <u>sets</u> observation points; it does not automatically generate the program execution command necessary to track down the problem. PROBUG reports changes that occurred in the location(s) being observed only after you do a quiet trace, trace, or breakpoint.

To remove an observation point from the table, precede the **O** with a minus sign (-). For example, the command **-O1000.** removes location 1000 as an observation point.

To display a list of all observation points you have set, type **O.** (with no parameter). The command **-O.** with no parameter removes **all** observation points from the table. The following is a sample that includes various uses of the **O** command.

```
>O100.       (Set location 100 as an observation point)
>OC00,F100.  (Set location C00 as observation point with target value F100)
>O.          (Display all observation points and their contents)

OBSERVATION POINTS
000100  47E7
000C00  F000   TARGET VALUE = F100

>-O100.      (Remove location 100 as an observation point)
>O.          (Display all observation points and their contents)

OBSERVATION POINTS
000C00  F000   TARGET VALUE = F100
>
```

Note that the **O** command with no parameters displays the observation points and their <u>current</u> contents. If you change the contents of an observation

point (for example, use the **M** command on it), a subsequent **O** command will reflect the change.

Observation Points With Quiet Trace. Suppose you did a quiet trace after setting an observation point with a target value. If the specified change occurred in location 000C00, the following output would appear:

```
>0C00,F004.  (Set location C00 as observation point with target value F004)
>Q.          (Quiet trace starting at location in current program counter)

INSTRUCTION AT LOCATION 048002 CHANGED DATA AT
LOCATION 000C00 FROM F000 TO F004

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 0000076C
PC)  048006     SR) 2704     CCR) --Z--     USP) 00B00000     .SSP) 0000076C
048002   ADD.W      #4,$C00
048006   MOVEQ.L    #15,D0
>
```

When you trace through your program after setting observation points, program execution stops after the instruction that changed the observation point. This is true whether you use **Q**, **N**, or **T** to trace through the program. In the above example, if the contents of location C00 had not changed to F004, no message would have been printed.

If you ran a breakpoint after setting the same observation point and target value, and the instruction changed as you specified, program execution would stop at the breakpoint and PROBUG would report that the observation point changed. The output might look like this:

```
>0C00,F004.  (Set location C00 as observation point with target value F004)
>B480A0.     (Breakpoint starting at location in current program counter)

INSTRUCTION BETWEEN 048000 AND 0480A0 CHANGED DATA AT
LOCATION 000C00 FROM F000 TO F004

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 0000076C
PC)  0480A0     SR) 2704     CCR) --Z--     USP) 00B00000     .SSP) 0000076C
0480A0   MOVEQ.L   #20,D0
>
```

Note: the reference to an instruction "between" locations 48000 and 480A0 means that during program execution between those points, the data changed. It does not necessarily mean that the guilty instruction's address falls between 48000 and 480A0.

P[a][,b]. PRINT MEMORY

This command displays the contents of memory within the locations you specify. For example, **P1000,2000.** displays the contents of locations 1000 through 2000. If you specify only one location, PROBUG displays 256 bytes of memory (one screenful) starting at that location. Note: if you specify an odd address, it will be rounded down; each line can begin only with an even byte. For example, **P1000.** and **P1001.** have the same meaning.

The **P** command with no parameters displays 256 bytes of memory starting at the address contained in either the system stack pointer or the user stack pointer. The system stack pointer has the starting address if the 68000 is in supervisor state; the user stack pointer has the starting address if the 68000 is in user state.

Each line of the display shows the contents of eight words (16 bytes) one word at a time, preceded by the address of the line's first byte. To the right, the ASCII representation of the bytes appears, or, for any byte that does not translate to an ASCII character, an underscore (_) appears. (The Print command masks the high-order bit, so ASCII values A0-FE are printed as 20-7E.)

After using the Print command, you can print subsequent screenfuls (256-byte blocks) of memory by pressing carriage return or typing a dot (.). You can continue to use carriage return or dot to print subsequent screenfuls until you type another command.

To freeze output on the screen as it is printing, press the Escape (ESC) key. Press it again to allow output to continue. To abort printing and return to PROBUG command mode, press any character at the keyboard (besides Escape). PROBUG will finish displaying the current line before the abort takes effect.

The following is a sample of possible output from a **P** command, interrupted by pressing any key.

```
>P2000. (Print screenful of contents in hex & ASCII, starting at 2000)

002000  3D7C 0001 006A 2D6F   0042 006C 026F 7FFF   =|___j-o_B_l_o__
002010  6120 0C50 4AFB 6602   3080 5341 66F0 426E   a _PJ{f_0_SAfpBn
002020  0000 0600 0000 2020   0000 0600 6000 0012   ______ ____`___
(any key)
>
```

To reference the system stack pointer in an expression, use **RS**; to reference the user stack pointer, use **RU**. Thus, if the 68000 is in supervisor state and you want to print the user stack, type **PRU**.

Q[a][,b][,c][,d]. **QUIET TRACE** (Program Execution Command)

Please read the section **Program Execution Commands** before using this command.

The **Q** command provides a way of tracing through your program without generating a lot of output on the screen. Quiet traces print register contents only after tracing all instructions, rather than printing them after each instruction is executed. This makes quiet traces much faster than traces.

Like the Breakpoint (**B**) command, **Q** takes up to four addresses as parameters. Quiet traces always begin at the location in the program counter. Use the ***** command to change the program counter if necessary before using this command.

The command **Q1000.** does a quiet trace from the location in the program counter to location 1000. If you specify more than one parameter, the quiet trace will stop at whichever location is encountered first. **Q** with no parameters runs continuously.

The **Q** command offers several advantages over other program execution commands. First, you can use it to "breakpoint" through either ROM or RAM. Second, when you use it with the **O** command, you can monitor a particular location in RAM (called an observation point) to find out exactly when that location changes.

Third, quiet traces can be used to follow the flow of program control. When the address you specify on the **Q** command is reached, PROBUG prints the previously executed instruction as well as the current instruction, in disassembled form.

The following sequence includes the use of the **O** command, and assumes the current program counter is F000.

```
>O100.     (Set location 100 as an observation point)
>QF200.    (Quiet trace from current program counter to F200)

INSTRUCTION AT LOCATION 00F1F2 CHANGED DATA AT
LOCATION 000100 FROM 0014 TO 0000

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00000B00
PC)  00F1F6     SR) 0704     CCR) --Z--    .USP) 00000B00     SSP) 0000076C
00F1F2   CLR.W     $0100
00F1F6   SUBQ.W    #1,$0C00
>
```

Note that although the **Q** command said to trace to F200, it never got there because it stopped after the instruction that changed the observation point.

If location 100 had not changed, the trace would have continued to F200, and the MPU registers would have been printed without a message.

As mentioned above, quiet traces can be used to follow the flow of program control. Suppose your program crashes with an unimplemented instruction at 40020. Run the program using a **Q** to that location:

```
>Q40020.   (Quiet trace from current program counter to 40020)

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00000B00
PC)  040020    SR) 0704    CCR) --Z--    .USP) 00000B00    SSP) 0000076C
001420   RTS
040020   DC.W    $FFFF
TRACE STOPPED AT 040020
>
```

001420 is the address of the instruction that caused the program to jump to 40020.

Note: After using **Q**, you can trace the next instruction by pressing carriage return or typing a dot (.). You can continue to use carriage return or dot to single-step through the program (as if you were using the **N** command) until you type another command.

There are two ways that the **Q** command is commonly used: (1) **Q** with one parameter with neither haltpoints nor observation points set; and (2) **Q** without a parameter and with no haltpoints, but with one observation point set. These two uses of the **Q** command have been optimized to run much faster than **Q** commands with multiple parameters and multiple haltpoints and observation points.

R. PRINT REGISTER CONTENTS

The **R.** command prints the contents of the MPU registers and disassembles one line starting at the current program counter. The following is an example of this command:

```
>R.  (Print register contents)

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00B00000
PC)  04100C    SR) 0704    CCR) --Z--    .USP) 00B00000    SSP) 0000076C
04100C   BEQ.S   $041056    (TRUE)
>
```

The first line (line **D**) displays the contents of the data registers, D0 through D7. The second line (line **A**) displays the contents of the address registers, A0 through A7. The third line displays the contents of the program counter (**PC**), the status register (**SR**), the condition code register (**CCR**; see below), the user stack pointer (**USP**), and the system stack pointer (**SSP**). Either the USP or the SSP will be preceded by a dot (.) to indicate the state of the 68000 (user or supervisor, respectively).

Condition Code Register (CCR) Codes: Each of the five positions in this field tells whether a particular bit is on or off. The positions are as follows:

Position	Meaning	If On	If Off
1	Extend Bit	X	-
2	Negative Bit	N	-
3	Zero Bit	Z	-
4	Overflow Bit	V	-
5	Carry Bit	C	-

In the above example, "--Z--" appears in the **CCR** field, indicating that the extend and negative bits are off, the zero bit is on, and the overflow and carry bits are off. If all of these bits were on, "XNZVC" would appear in the **CCR** field.

After the MPU registers are displayed, the next instruction to be executed is disassembled. If the instruction is a conditional (**Bcc**, **DBcc**, or **Scc**), the condition codes are examined and **(TRUE)** or **(FALSE)** is printed to indicate whether or not the condition occurred.

Ra[,b]. PRINT/ALTER REGISTER CONTENTS

The **Ra[,b].** command prints the contents of the register you specify and allows you to alter them. You can use this command to print/alter a data, address, or user register. As with the I command, you can go on to the next location (or back to the previous one) after printing or altering the contents of a register.

There are eight data registers, eight address registers, and eight user registers. When used with the **R** command, these are expressed as **RD0** through **RD7**, **RA0** through **RA7**, and **R0** through **R7**, respectively. For example, the command **RA1.** would display the contents of address register A1; the command **R3.** would display the contents of user register R3. As with the I command, type . or **<CR>** to exit; press the space bar to go on to the next address register; type , to go back to the previous register. To alter the contents, type the new value followed by a terminator.

The order in which the registers are displayed (if you type **RD0.** and keep typing the space bar to go on to the next register) is as follows: RD0 through RD7, RA0 through RA6, USP, SSP, SR, PC, then back to RD0, etc.

To change the SSP or USP, type **RA7.** from the PROBUG prompt. This will display the contents of either the USP or the SSP, depending on the state of the 68000 (user or supervisor). Note: If you change the SSP, the registers that PROBUG pushed on the stack are moved to the new stack location.

You can change register contents without displaying them at all by typing the command followed by the desired value and a terminator. For example, suppose you want to change the contents of address register A4 to 1F. Typing **RA4,1F.** changes the register's contents without displaying the contents either before or after the command.

The following sequence shows the use of the R command with one parameter; a "before" and an "after" **R.** command is included to illustrate the changes.

```
>R.

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 0000FE00 0001C000 0000076C
PC)  04100C    SR) 2704    CCR) --Z--    USP) 0001C000   .SSP) 0000076C
04100C   CLR.W   D1
>

>RA5.                (Print/alter address register A5)
RA5  0000 FE00 1000_ (Replace 0000FE00 with 00001000; go to next register)
RA6  0001 C000 .     (Leave as is; exit R command)
>
```

```
>R.

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00001000 0001C000 0000076C
PC)  04100C     SR) 2704     CCR) --Z--     USP) 0001C000     .SSP) 0000076C
04100C   CLR.W   D1
>
```

Note that address register A5 now contains 00001000, the new value. The values in the other registers remain the same.

Sa,b,c[,d]. SEARCH MEMORY FOR PATTERN

This command searches memory locations **a** through **b** for pattern **c**. The pattern you specify may be one byte, one word (2 bytes), or one long word (4 bytes). As with other expressions, the pattern may be a hex number, a result of addition/subtraction, or an ASCII character enclosed in single quotes.

All locations between **a** and **b** (inclusive) that match the pattern will be displayed, along with their contents. If no matches are found, nothing is printed.

The **S** command will not find a pattern that is split between two words. For example, the pattern "34" would **not** be found in the long word "0123 4567".

You can specify a fourth parameter (**d**) representing a bit mask. Both memory and the value of the pattern are logically ANDed with the mask before being compared with each other. PROBUG will compare only those bits that have a corresponding "1" bit in the mask. For each "0" bit in the mask, the corresponding bit in memory is ignored during the search for a match. If you do not specify a mask, the mask is implicitly FFFFFFFF.

If the size of the pattern differs from the size of the mask, PROBUG will switch modes to accommodate the larger of the two. For example, the command **S1000,2000,E0,FFFF.** would force the search into word mode, adding leading zeroes to the pattern E0. PROBUG would search for the word 00E0.

To interrupt output from the Search command, type ESCAPE, and type it again to resume output printing. To abort a search in progress and return to PROBUG command mode, type any character besides ESCAPE.

The following is a sample of the **S** command.

```
>S800,900,10.  (Search locations 800 through 900 for 10)
00085F  10
0008F0  10
0008F4  10
>
```

Search For Non-Match: -S. To search for a non-match, precede the **S** command with a minus sign (-). For example, the command **-S1000,2000,0000.** would print only those words between locations 1000 and 2000 that do **not** consist of all zeroes.

T[a][,b][,c][,d]. **TRACE INSTRUCTIONS** (Program Execution Command)

Traces allow you to step through your program instruction by instruction and examine register contents after each instruction. You can use traces on ROM as well as RAM. Please read the section **Program Execution Commands** before using the **T** command.

T always begins at the location in the program counter (the current location). Use the * command to change the program counter if necessary before using this command. PROBUG prints the register contents after each instruction is executed.

You can specify up to four addresses in the **T** command; the trace then lasts until any one of the addresses is reached. For example, the command **T1000,2000,2F00,3C00.** begins by executing the current instruction and continues until it reaches location 1000, 2000, 2F00, or 3C00.

The **T** command often generates a considerable amount of output as the program runs. Press the ESCAPE (ESC) key at any time to freeze the output on the screen. Press ESC again to continue the trace.

To abort printing and return to PROBUG command mode, press any character at the keyboard (besides ESCAPE). PROBUG will finish displaying this instruction's register contents before the abort takes effect.

Note: After using **T**, you can trace the next instruction by pressing carriage return or typing a dot (.). You can continue to use carriage return or dot to single-step through the program (as if you were using the **N** command) until you type another command.

The following is a sample of possible trace output when the program counter is 4800A. This example assumes you pressed a key while the third instruction was being traced.

```
>T.  (Trace continuously beginning at the current program counter)

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000036 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A) 00000820 00000000 00000000 00000000 00000000 00000000 00000000 00000770
PC)  04800A    SR) 2700    CCR) --Z--    USP) FFFF5FFF   .SSP) 00000770
04800A  SUBQ.W   #1,$0C00

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000036 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A) 00000820 00000000 00000000 00000000 00000000 00000000 00000000 00000770
PC)  04800E    SR) 2700    CCR) -----    USP) FFFF5FFF   .SSP) 00000770
04800E  BNE.S    $048000  (TRUE)

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000036 00000000 00000000 00000000 00000000 00000000 00000000 00000000
A) 00000820 00000000 00000000 00000000 00000000 00000000 00000000 00000770
PC)  048000    SR) 2700    CCR) -----    USP) FFFF5FFF   .SSP) 00000770
048000  TST.W    (A0)+
TRACE STOPPED AT 048000
>
```

The trace was stopped because of the key you pressed. Note that PROBUG prints the address at which the trace was stopped.

Note: If you use the **T** command after setting observation points, and the trace causes a change in an observation point, the trace will stop after the instruction that changes the observation point.

To trace only the next or next few instructions, use the **N** command.

Wa,b[,c]. WRITE PROGRAM IN S-RECORD FORMAT

The **W** command writes the contents of location **a** through **b** to a host computer, or to tape, in S-record format. Please read the descriptions of the **L** and **@** commands for details on how communication takes place between PROBUG and the host computer or external device.

S1 and S9 records are used if the program is in the first 64K bytes of memory, S2 and S8 if not. Refer to the description of S-record format under the **L** command for more information.

If you specify only parameters **a** and **b**, the end-of-file record (an S8 or S9 record) will have location **a** as its starting address. For example, the command **W48000,48013.** would assign 48000 as the end-of-file record's address, indicating that program execution should begin at 48000 when the program is loaded again.

If you specify a third parameter (**c**) in addition to the first two, the end-of-file record will have location **c** as its address. The program will begin executing at that location when it is loaded again.

The following is a sample of S-record formatted output generated by a **W** command. The records take up one line each. In the first line, the record type is **S1**, the number of bytes is **0B**, the address is **1000**, the data is **FEDCBA9876543210**, and the checksum is **AC**. In the second line, the record type is **S9**, the number of bytes is **03**, the address (set by the third parameter of the **W** command) is **2000**, and the checksum is **DC**. Note that there is no data in the **S9** record.

```
>W1000,1007,2000.
S10B1000FEDCBA9876543210AC
S9032000DC
```

PROBUG does not control the start/stop functions of the tape, so you must manually start the tape after typing the **W** command. Stop the tape as soon as the PROBUG prompt (>) appears.

Before using the **W** command, be sure both of the M68K10's ports are running at the same baud rate.

@[a][,b][,c]. ENTER TRANSPARENT MODE

The @ command is used to initiate communication with a host computer or other external device. Using the @ command causes you to enter "transparent mode," from which you can "talk" directly to the host computer (type commands, etc.).

With the @ command, you can also set the exit and the substitution characters, which are used in communicating with the host computer. In addition, you can select the port for loading/writing. Please read the description of the **L** command for an introduction to transparent mode and related concepts.

Enter Transparent Mode. Type @ to enter transparent mode, which allows your terminal to communicate with a host computer or other external device. While PROBUG is in transparent mode, the characters you type at the terminal are sent to the other port.

To enter transparent mode without doing an L or W, type **@.** with no parameters. As soon as you do this, the system will respond with a message similar to the following:

EXIT CHARACTER = $01 = CONTROL-A

This message indicates the character you must type to exit transparent mode (as with the **L** and **W** commands, the exit character is by default CTRL-A). The ASCII code for the exit character (in this case, 01) is included in the message, preceded by a **$**. To exit transparent mode, type the exit character and you will be returned to PROBUG command mode.

Set Exit Character. To set the exit character to be used with the **L** and **W** commands (as well as with the @ command itself), use @ with a parameter. You can do this either in hex, or in ASCII surrounded by single quotes: for example, type **@02.** to set the exit character to hex 02 (which is a CTRL-B). The command **@'\'.** would set the exit character to backslash (\).

Set Substitution Character. To set the substitution character, use a second parameter with the @: for example, the command **@02,04.** would set the exit character to ASCII 02 (CTRL-B) and the substitution character to 04 (CTRL-D). To set only the substitution character, use a comma before the exit character: for example, the command **@,04.** sets the substitution character to 04 and leaves the exit character as it is.

Select Port Address. On the M68K10, there are two ports, channel A and channel B. Channel B generally connects to the user terminal. Channel A can be connected to a printer, to a host computer, etc. By default, the port used to download programs from the host computer (using the **L** command), and to write them back to the host computer (using the **W** command), is channel A.

If this default is not desired, use the @ command to select the port before using the **L** or **W** command. The port selection is the third parameter of the @ command. Specify "1" for channel A or "0" for channel B. For example, the command **@02,04,1.** sets the exit character to 02 and the substitution character to 04, and selects channel A.

To select the port without selecting the other two characters, use commas: **@,,1.** in this example. (The command **@02,,1.** would set the exit character to 02, leave the substitution character as it is, and select channel A.)

Select "1" (the default) if you want the channel A to be used to load and write files. Select "0" if you want channel B to be used to load and write files (this is generally used when loading files from an S-record formatted object tape).

You can also select a port other than channel A or B by specifying the hex address of the port instead of specifying "1" or "0". For example, the command **@,,FD0020** specifies address FD0020 for the port (and leaves the exit and substitution characters as they are). See under **Custom I/O** later in this manual for a description of custom port specifications.

Be sure to run both ports at the same baud rate.

Note: You cannot set the exit or substitution character to null using the @ command; this is the same as leaving it out and will cause the old value to be retained. (For example, the commands **@0,0,1.** and **@,,1.** are equivalent.)

*[a]. LIST/SET PROGRAM COUNTER

The * command sets the program counter to the value you specify. For example, the command ***1000.** sets the program counter to 1000. With no parameter, *. lists the current program counter as a prompt for you to enter a new value. Press carriage return or type a dot (.) after typing the new value. To exit without changing the program counter, press carriage return or type a dot.

If you press the space bar or comma after typing *., this command works like the R command which allows you to print/alter register contents. The space bar causes an advance to data register **D0**; press it again to advance to **D1**, etc. The comma backs up to **SR** (status register). See under the R command for the order in which the registers appear when you use the space bar or comma.

PROBUG Function Calls

Certain subroutines within PROBUG can be accessed from your programs. Most of these subroutines perform I/O functions such as input and output of characters, character strings, hexadecimal numbers, and decimal numbers. You can also extend the available functions with your own routines (see **Adding Your Own Functions**, below).

PROBUG functions are accessed using the TRAP 14 instruction. Use the following calling sequence:

```
MOVE.B #<Funtion Number>,D7
TRAP   #14
```

The function number is passed in the low order byte of D7; thus, up to 256 functions may be selected.

Except where otherwise noted, all registers are preserved by these routines.

Function Number	Meaning

255 **RESTART - Restart PROBUG**

PROBUG is re-initialized as if the reset button had been pressed. Control is not returned to the user program. This function is necessary only when PROBUG's RAM has been altered.

254 **REENTER - Re-enter PROBUG**

PROBUG is entered and the MPU registers are printed. You can now execute any PROBUG command. Typing "J." will continue the program at the next instruction.

253 **SELECT - Select I/O Port**

ENTER: A0 contains the address of the I/O port.

This function allows I/O to be redirected to other devices. The redirection applies to function calls only, not to PROBUG command I/O.

All of the I/O function calls communicate through the I/O port selected by this routine. By default, the I/O port is set to the same terminal that PROBUG uses for its I/O. See the section **Custom I/O** below.

252 **INITIO - Initialize I/O Port**

This routine is used to initialize the I/O port. This is usually necessary only after the I/O port is selected for the first time. Note: This routine destroys the contents of register D0.

Function Number	Meaning

251 CREADY - Check For Character Ready

EXIT: The NOT EQUAL condition is set if a character is waiting.

This routine checks the status of the I/O port to see if a character is waiting to be read. If a character is waiting, the NOT EQUAL condition is set. The lowest byte of register D0 may be altered by this routine.

The location of the I/O port is determined by the "Select I/O Port" function.

250 INCHAR - Get Character From I/O Port

EXIT: Low order byte of D0 contains the character.

Like the GETCHAR routine, INCHAR gets a single character from the keyboard, but INCHAR does not echo the character. The character is returned in the lowest byte of D0. The high order bit of the character is always cleared.

The location of the I/O port is determined by the "Select I/O Port" function.

249 CBUSY - Check Whether Output Busy

EXIT: The EQUAL condition is set if the I/O port is busy.

This routine checks the status of the I/O port to see whether the previous character has been output. The EQUAL condition is set if the output is still busy. Note that this routine only returns the status of the port; most devices will report a not-busy condition before actually outputting the character. The lowest byte of register D0 may be altered by this routine.

The location of the I/O port is determined by the "Select I/O Port" function.

248 OUTCHR - Write Character To I/O Port

ENTER: D0 contains the character to write.

This routine writes the character in the low order byte of D0 to the I/O port selected with the "Select I/O Port" function.

Function Number	Meaning

247 **GETCHAR - Get and Echo Character From I/O Port**

EXIT: Character in low order byte of D0.

This routine gets a single character from the keyboard and echoes the character. The character is returned in the lowest byte of D0. The location of the I/O port is determined by the "Select I/O Port" function.

246 **GETBUFF - Input String From Terminal**

ENTER: A0 contains the address of a buffer;
D0 contains the length of the buffer.

EXIT: A0 points one past the last character typed;
D0 contains the number of characters typed.
CARRY set if error or line cancelled.

This routine inputs a line from the keyboard into the character buffer to which register A0 points. On entry, the low order byte of register D0 contains the length of the buffer. Any backspace and delete characters are checked and processed.

To cancel the line, use CTRL-X. All other control characters, except carriage return (CR), are ignored. When CR is entered, control is returned with register D0 containing the number of characters typed and register A0 pointing one character past the last character typed. If the line is cancelled, or the buffer is overflowed, or too many backspaces are entered, control is returned with the carry bit set. The buffer pointer used by CHKNEXT, GETNEXT, GETHEX, and GETDEC is set pointing to the first character of the buffer.

245 **SETBUFF - Set Buffer Pointer**

ENTER: A0 contains the address of a buffer.

This routine sets the buffer pointer used by CHKNEXT, GETNEXT, GETHEX, and GETDEC to the address in register A0.

244 **CHKNEXT - Check Next Character In Buffer**

EXIT: A0 contains the current buffer pointer;
D0 contains the next character in the buffer.

The character pointed to by the buffer pointer is returned in the lowest byte of register D0. This routine does not advance the buffer pointer; this allows the next character in the buffer to be previewed before being read by the GETNEXT, GETHEX, or GETDEC functions. CHKNEXT can also be used to get the current buffer pointer.

Function Number	Meaning

243 GETNEXT - Get Next Character In Buffer

EXIT: D0 contains the next character in the buffer.

The character pointed to by the buffer pointer is returned in the lowest byte of register D0. The buffer pointer is advanced to the next character in the buffer unless the character returned is a CR. The buffer pointer is not advanced past the CR at the end of the buffer.

242 GETHEX - Get Hexadecimal Number From Buffer

EXIT: D1 contains the 32-bit value of the number;
D0 contains the next character in the buffer.
EQUAL condition set if no number found.

The string of ASCII characters pointed to by the buffer pointer is converted to a 32-bit binary number and returned in register D1. The conversion process is terminated by the first non-hexadecimal digit in the string. The buffer pointer is left pointing to the terminating character which is returned in register D0. If no valid hexadecimal digits are found, the EQUAL condition is set in the condition codes and register D1 is set to zero.

241 GETDEC - Get Decimal Number From Buffer

EXIT: D1 contains the 32-bit value of the number;
D0 contains the next character in the buffer.
EQUAL condition set if no number found.
CARRY set if number overflow.

The string of ASCII characters pointed to by the buffer pointer is converted to a 32-bit signed binary number and returned in register D1. The conversion process is terminated by the first non-decimal digit in the string. The buffer pointer is left pointing to the terminating character which is returned in register D0. If no valid decimal digits are found, the EQUAL condition is set in the condition codes and register D1 is set to zero. If a number is converted that cannot be contained in 32 bits, the CARRY bit is set and the value in register D1 is undefined.

240 CONVERT - Convert Hex Byte To ASCII

ENTER: D0 contains the byte to convert.
EXIT: D0 contains 2 ASCII characters.

The lowest byte of register D0 is converted to two ASCII characters and returned in the lower word of register D0.

Function Number	Meaning

239 PCRLF - Print End-Of-Line Sequence

This routine prints a carriage return, line feed, and four nulls on the terminal. The four nulls are included to give printing terminals enough time to perform a carriage return.

238 PSTRING - Print String

ENTER: A1 contains the address of a string.

The string of characters pointed to by register A1 is printed on the terminal. The string is terminated by a $00 byte. Register A1 is left pointing one character past the $00 byte. If this routine encounters a $01 byte embedded in the string, an end-of-line sequence will be written in its place. The lowest byte of register D0 is destroyed by this routine.

237 PHEX8 - Print 8-bit Hex Number

ENTER: D0 contains the byte to write.

The lowest byte of register D0 is printed as a two-digit hexadecimal number. The contents of register D0 are preserved.

236 PHEX16 - Print 16-bit Hex Number

ENTER: D0 contains the word to write.

The lowest two bytes of register D0 are printed as a four-digit hexadecimal number. The contents of register D0 are preserved.

235 PHEX24 - Print 24-bit Hex Number

ENTER: D0 contains the value to write.

The lowest three bytes of register D0 are printed as a six-digit hexadecimal number. The contents of register D0 are preserved.

234 PHEX32 - Print 32-bit Hex Number

ENTER: D0 contains the value to write.

The contents of register D0 are printed as an eight-digit hexadecimal number. The contents of register D0 are preserved.

Function Number	Meaning

233 PBYTE - Print Hex Byte At A0

ENTER: A0 contains the address of the byte to write.
EXIT: A0 points to the next byte in memory.

The byte pointed to by register A0 is converted to two hexadecimal digits and printed. Register A0 is advanced to point to the next byte in memory.

232 PDEC - Print Decimal Number

ENTER: D1 contains the value to write.

On entry, register D1 contains a 32-bit two's complement binary number. The number will be printed as a signed decimal number with leading zeroes suppressed. To print leading zeroes or leading blanks, see below.

231 PDECF - Print Decimal Number With Field Size

ENTER: D1 contains the value to write;
D0 contains the field size.

On entry, register D1 contains a 32-bit two's complement binary number. The number may be printed as a signed decimal number with leading zeroes or leading blanks in a field from 1 to 127 characters long. The field size is given by the lowest byte of register D0. A positive field size means pad with leading spaces. A negative field size means pad with leading zeroes. If the number overflows the field specified, the field size is ignored.

For example, printing 100 with a field size of 5 causes " 100" to be printed. With a field size of -5, "00100" would be printed; with a field size of 0, "100" would be printed.

230 LINK - Link User Function Table

ENTER: A0 contains the address of the user's function code table.
EXIT: A0 contains the address of the previous table.

The LINK function is called with the address of the new function table in register A0. The address of the previous table is returned in register A0. For linked tables, the last entry in the table should have $FE as a function code followed by the address of the next table. If the address of the next table is unknown at the time of the LINK, the address returned in register A0 should be stored as the last entry. The high order byte of register A0 will be $FE and the next three bytes will be the address of the previous table. Register A0 will contain $FFFFFFFF if no tables have been linked.

Adding Your Own Functions

The PROBUG function processor can be used to call functions that you define, as well as the functions listed in the above table. To define new function codes, you must provide a table giving the new codes and the addresses of the processing routines to implement them. Use the LINK function (code 230) to do this. You can provide more than one such table if desired.

Each entry in the code table must be four bytes long, of which the first byte is the function code number and the subsequent three bytes are the address of the processing routine.

The last entry in the table must consist of either $FE or $FF as the function code byte. If you are creating multiple function code tables, use $FE for the last entry in all but the last table, and give the address of the next function code table as the subsequent three bytes. PROBUG will then chain to the next table after going through the current table.

Use $FF as the function code number for the last entry in the last table, or, if you are creating only one function code table, for the last entry in that table.

Function codes 230 through 255 are implemented in PROBUG, as defined in the above table, and cannot be re-defined. Codes 0 through 229 are available for user functions.

Function code numbers 128 through 255 are processed differently from function codes 0 through 127. Function codes 128 through 255 execute in the supervisor state. For codes 128 through 229, control is passed to the subroutine with both the status register and the return address pushed on the system stack. The routine must end in a RTE instruction.

For function codes 0 through 127, control is passed to the subroutine with just the return address pushed on the stack. If the program is in the supervisor state when the function is called, the return address will be on the system stack and control is passed to the routine with the processor in the supervisor state. If the program is in the user state when the function is called, the return address will be on the user stack and control is passed in the user state. In either case, the routine must end with a RTS instruction.

What Happens When TRAP 14 Instruction Is Executed. When a TRAP 14 instruction is executed, the current program counter and status register are pushed onto the system stack, the supervisor state is entered, and control is passed to the address given in the TRAP 14 exception vector.

The TRAP 14 exception vector points to the PROBUG function code processor, which first checks the range of the function code number in data register D7. If the number is in the range 230 to 255, control is passed to the appropriate function within PROBUG. If the number is in the range 0 to 229, the user function code table is searched. If the function is not found in that table, or if no table has been linked, an error message as printed and control is returned to PROBUG. Otherwise, control is passed to the address given in the function code table. The contents of all registers are preserved.

Custom I/O

You can interface an MPSC, ACIA, or custom I/O device to PROBUG. To select the port to be used for downloading programs, use the @ command. To select the I/O address, use PROBUG function call 253 (SELECT I/O PORT).

An MPSC is the default device to connect to the M68K10's terminal or printer port. To connect an MPSC to the terminal port, specify 0 as follows:

For Downloading:	To Select I/O Port:
@,,0.	MOVE.L #0,A0 MOVE.B #253,D7 TRAP #14

To connect an MPSC to the printer port, specify 1 as follows:

For Downloading:	To Select I/O Port:
@,,1.	MOVE.L #1,A0 MOVE.B #253,D7 TRAP #14

If you do not want to use the terminal or printer port, you need to specify the port type and address in four bytes:

The first byte is the code for the port type. Specify $00 for MPSC, $01 for ACIA, or $FF for a custom device. For custom devices, you will need to create special subroutines; see below.

The subsequent three bytes are the port address.

For example:

TASK	FOR DOWNLOADING	TO SELECT I/O PORT
Connect MPSC at Address FD0002	@,,00FD0002.	MOVE.L #$00FD0002,A0 MOVE.B #253,D7 TRAP #14
Connect ACIA at Address FD0500	@,,01FD0500.	MOVE.L #$01FD0500,A0 MOVE.B #253,D7 TRAP #14

For custom devices, you will need routines to perform the following five functions:

* Initialize I/O port;
* Check for a character ready;
* Read the character from the device;
* Check to see whether the output is busy; and
* Write a character to the device.

To supply PROBUG with the addresses of these five functions, you must create a table consisting of five long branch instructions (one to each of the subroutines). Then PROBUG will only need to know the address of this table.

no p. 54

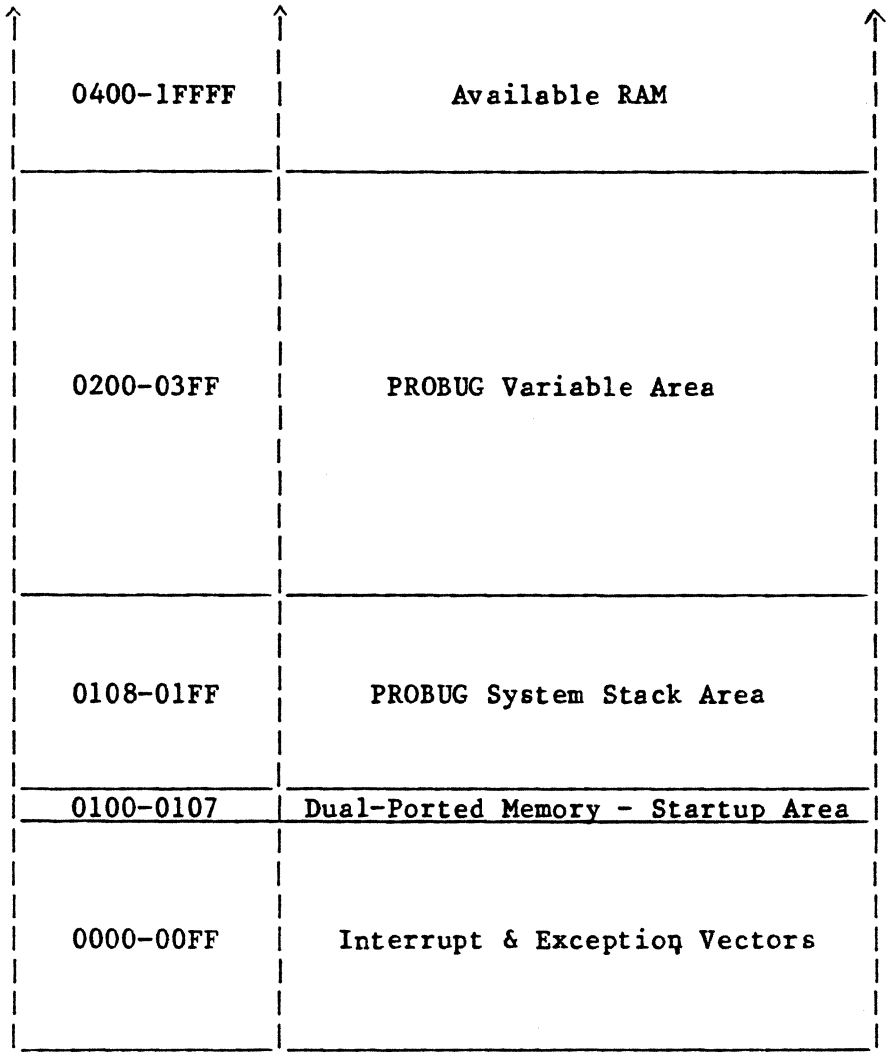
Suppose you have written the five routines and called them INITIO, CREADY, INCHAR, CBUSY, and OUTCHAR, respectively. The following is a sample program for selecting the routines for the custom device. In the LEA instruction, the $FF indicates that this is a custom device; BRTBL is the address of the branch table.

```
            LEA     $FF000000+BRTBL,A0
            MOVE.B  #253,D7             SELECT I/O PORT
            TRAP    #14
             .
             .
             .
BRTBL       EQU     *
            BRA.L   INITIO              INITIALIZE I/O PORT
            BRA.L   CREADY              CHECK FOR CHARACTER READY
            BRA.L   INCHAR              READ CHARACTER FROM DEVICE
            BRA.L   CBUSY               CHECK WHETHER OUTPUT BUSY
            BRA.L   OUTCHAR             WRITE CHARACTER TO DEVICE
```

See the section on PROBUG function calls to determine what values are returned by these routines.

PROBUG Memory Map

The following chart shows the memory addresses that PROBUG uses.



Special Considerations With PROBUG

This collection of notes offers an assortment of information for the sophisticated, the curious, and the unlucky.

Stack Usage. Whenever PROBUG is in command mode, it saves all register contents on the system stack; thus PROBUG itself uses up about 110 bytes (decimal) of system stack space. During certain NMI presses, up to 100 additional bytes may be used. Be sure to take this into consideration when you set up your system's stack area.

What Happens When You Set A Breakpoint Or Haltpoint. When you type a breakpoint command, or start executing your program after setting a haltpoint in RAM, PROBUG saves the instructions at the breakpoint/haltpoint location(s) and temporarily replaces each one with a special instruction. This instruction is $4AFB for breakpoints and $4AFA for haltpoints (Motorola's standard codes for illegal instructions).

Your program now begins to execute. When it reaches a breakpoint/haltpoint, it doesn't recognize the $4AFB or $4AFA as executable code. Control is returned to PROBUG, which restores the original instruction and displays register contents and a message indicating why the program stopped executing.

When a haltpoint with an iteration count is reached, the count is decremented, and the original instruction is restored and executed. The haltpoint's location is again replaced with $4AFA, and program execution continues.

Because PROBUG needs to write the breakpoint/haltpoint illegal instruction code into the location you specify, you can use breakpoints on RAM (writeable memory) only, and not on ROM.

Exception Processing. Unless you set up routines to handle exceptions, and vectors pointing to these routines, PROBUG will process all exceptions. Assuming PROBUG is the error processor, whenever an exception occurs, PROBUG prints the reason for the exception and prints the contents of the MPU registers. The registers are saved on the stack, and PROBUG returns to command mode.

Exceptions include interrupts, attempts to divide by zero, traps, etc.

TRAP Exceptions In Trace Mode. The **TRAP, TRAPV, CHK,** and **Zero Divide** exceptions all have lower priority than a **TRACE** exception. As a result, PROBUG traces the first instruction of the exception processing routine of these four types of exceptions. Also, PROBUG does **not** trace the first instruction after the one that caused the exception.

Suppose a TRAP #0 routine is written and the TRAP #0 exception vector is pointing to the routine. When the TRAP #0 instruction is encountered it causes an exception. Because of the way TRAP exceptions are handled, you could not trace or quiet-trace to the address of the instruction after the TRAP #0. Also, a haltpoint set at that address would not be recognized as a haltpoint when you traced or quiet-traced through it.

To allow these instructions to be traced, you can write your exception processing routine as follows. At the end of the routine, check to see whether the 68000's trace bit was on when the exception was taken. If it was on, turn the trace bit on before the RTE instruction:

```
        TST.B   (A7)
        BPL     EXIT
        OR      #$8000,SR
EXIT    RTE
```

Limitations Of Trace Mode. You cannot trace through an instruction that turns trace mode off. For example, you should use the instruction **OR #$2700,SR** wherever you would use **MOVE #$2700,SR.**

You can use the **N.** command on instructions that turn trace mode off.

Timing. PROBUG affects the timing of programs. For non-timing-dependent programs, there is no difference between executing the program outside of PROBUG and executing it with PROBUG program execution commands.

When used on timing-dependent programs, however, PROBUG may affect the program's pathways. For example, tracing through your program takes much longer than executing it in real time. If you use program execution commands such as trace and breakpoint for timing-critical code, the results will not necessarily reflect what would happen in real time.

Start Program On M68K10 From Another Processor

The M68K10's dual-ported memory allows another processor on the Multibus to access PROBUG's RAM. The external processor can use the eight bytes of PROBUG's memory at locations 100-107 to communicate with PROBUG. Using these eight bytes, the external processor can cause PROBUG to load a new stack and program counter and begin execution of a program. These bytes are used as follows:

When the high-order bit of location 100 is set to 1, PROBUG goes into a loop monitoring the high-order bit of location 104. As long as the high-order bit of location 104 is 0, PROBUG remains suspended and any memory location may be altered (even PROBUG's stack space or its variables).

When the high-order bit of location 104 is set to 1, the bytes at both locations 100 and 104 are cleared. The contents of locations 100-103 are loaded into the system stack pointer, and the contents of locations 104-107 are loaded into the program counter. Program execution will begin at the address that was in locations 104-107.

When you first power the system up, PROBUG sets locations 100 and 104 to be its initial stack and initial program counter, respectively.

Example. Suppose you wanted to load a 32K-byte program through dual-ported memory into the M68K10's RAM at location 0, and then execute the program. In this program, the stack resides at location $800 and the program starts executing at location $1000.

In order to prevent PROBUG from using the RAM that your program will occupy, you need to set location 100 to be a number whose high-order bit is on. For example, you could put $FF at location 100, followed by the 3-byte address at which the stack will reside. Since the stack will reside at location $800, locations 100 through 103 should contain $FF000800.

Set locations 104 through 107 to be the address at which the program should start executing: in this case, $00001000.

If memory is copied starting at location 0 and the data that will be at locations 100-107 is set up correctly, the communication with PROBUG and the copying of program data can be done in the same step. When the program is copied into RAM at locations 0 through $7FFF, the data at locations 100 through 107 should be:

FF00080000001000

The $FF at location 100 signals PROBUG to wait until the high-order byte of location 104 changes to a 1. To start the program, you must now set location 104 to be a number whose high-order bit is on. Again, you can use $FF.

The stack is now set to $800 and the program starts at location $1000.

APPENDIX A: Getting Started - PROBUG And The M68K10

If you are using PROBUG on SBE's ModulasTen M68K10 single-board computer, read this section for information on installing the PROBUG PROMs, jumpering the M68K10, and serial port usage. If you are using an MPU board other than the M68K10, refer to that MPU's documentation for relevant information. In addition, the following information on PROM handling may be useful.

PROM Installation. PROBUG is contained in two 2764-type, 28-pin PROMs (8K x 8-bit). You will receive these PROMs in special anti-static treated tubing or packaging. Keep them in their package until you are ready to install them.

There are two pairs of 28-pin PROM sockets on the M68K10 module: U16 & U17, and U5 & U35. The two PROBUG PROMs must be installed in sockets U16 and U17. One PROM will be labelled "U16" and the other "U17", indicating the appropriate socket for each.

When installing PROMs, which are static-sensitive, have the M68K10 nearby. Handle the PROMs one at a time. If possible, use benchtop pads and wrist straps grounded through a high-resistance resistor. Put on a wrist strap before unpacking the PROMs from their antistatic packaging.

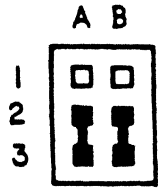
If benchtop pads and wrist straps are not available, you can reduce the risk of PROM damage as follows: Touch one finger to the surface on which you are working before handling the PROM. Keep the finger on that surface while handling the PROM. Similarly, when you get ready to load the PROM into the socket, first touch the M68K10 board with one finger and keep it touching the board while you load the PROM.

In general, avoid wearing nylon, polyester, or other static-generating fabrics, and avoid nylon and other carpet, while installing PROMs. Don't install PROMs in areas that are high in static electricity. Avoid excess handling of PROMs; handling and movement generates static. Carry PROMs only in conductive black foam or in the special anti-static treated tubes in which the PROMs were delivered.

Turn the M68K10 board, with the component side toward you, so that the NMI button is in the upper right-hand corner. Note that U16 and U17 are the middle two PROM sockets in the group of four on the board. Install each PROM so that its left edge is aligned with the left edge of the socket, and its notch is on the right, matched with the socket notch.

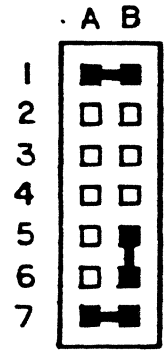
After installing the PROMs, make sure none of the pins got bent under during the installation.

Jumpering For Power Supply. The six-pin jumper area called J26 is located just to the right of area J25, which is just to the right of PROM socket U5. To make sure the M68K10 can provide power to the PROMs, be sure that jumper area J26 is set as shown here:



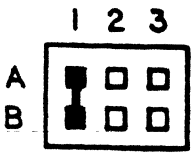
J26 Jumpers

Jumpering For 2764 PROMs. Just to the right of each PROM socket is a 14-pin jumper area for setting up the socket for the type of PROM you are installing. The jumper area for U16 is labelled J24; the area for U17 is labelled J22. Their jumpering is identical. Make sure that J22 and J24 are jumpered as shown here:



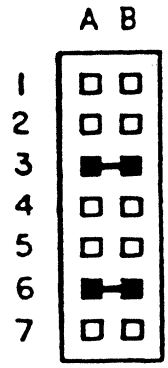
J22 and J24 Jumpers

Jumpering For PROM Size. Just below socket U6 on the right-hand side of the module is area J21, which is set according to the size of the PROMs being installed. Make sure J21 is jumpered as shown here:



J21 Jumpers

Jumpering For Access Time. Just to the right of socket U45 is area J4, which is set according to the access time of the PROMs being installed. (Note: the jumper across position 3 corresponds to an access time of 450 nanoseconds, the speed of the PROBUG PROMs. 250 nanosecond parts should be jumpered across position 1 instead of position 3.) Note that a jumper is already installed in position 6; do not move this jumper. J4 should be jumpered as shown here:



J4 Jumpers

Serial Port And Interrupt Processing. PROBUG uses channel "B" of the M68K10 to perform I/O. If you are using the "B" channel of the MPSC controller to connect your terminal to the M68K10, your program's output and the output from PROBUG will both appear on the screen: your program and PROBUG share the same serial I/O port.

PROBUG initializes both channels "A" and "B" to transmit and receive 8-bit characters with no parity and 2 stop bits. The standard shipping configuration of the M68K10 has both channels running at 9600 BAUD.

If your program configures the port to generate interrupts, you should be aware that PROBUG does not rely on interrupts from the serial port to do I/O. This can cause problems when control passes from PROBUG to your program: the interrupts that PROBUG has been ignoring will still be pending. You should take this into consideration when constructing your interrupt processing routine.

Power-Up

Whether you are using the M68K10 or another single-board computer to run PROBUG, several lines of output will appear on the terminal when you first turn the power on. The current contents of the MPU registers (data, address, and several other registers) will be displayed. The output will look like this:

```
PROBUG 2.0 - SBE SOFTWARE DEBUGGER
COPYRIGHT SBE, INC. 1983


      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
A) FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF 00000200
PC)  FC0028    SR) 2700    CCR) -----    USP) FFFFFFFF    .SSP) 00000200
FC0028  MOVE    #$2700,SR
>
```

An explanation of this output appears under **MPU Registers.**

The > indicates that PROBUG is ready for you to type a command. This prompt appears whenever PROBUG has completed a command, as well as at the beginning of a debugging session.

APPENDIX B: How To Use PROBUG - Some Debugging Suggestions

This section offers some suggestions on debugging 68000 software. Use this information as a supplement to the information in the remainder of the manual. The examples in this section show some typical problems and some of the ways you might use PROBUG to solve them.

Before using PROBUG, get a listing of the program you want to debug. Turn the power on, or follow the appropriate procedure for starting PROBUG. If your program is on tape, use the **@** command to select the port through which the program will be loaded. Now, load your program into RAM using the **L** command as documented in the command descriptions.

Debugging your program will require that you isolate each section in which problems exist, and attack them one at a time. **Breakpoints** allow you to execute segments of your program to track down problems. Breakpoints are perhaps the most frequently used of the program execution commands (see the section **Program Execution Commands** early in the manual).

Typically, after loading your program into RAM, you breakpoint or jump to a location and examine the MPU registers (which are displayed at the end of the breakpoint or jump). Use the **D** command to disassemble the group of instructions directly following the breakpoint location.

It is useful to breakpoint through a few instructions at a time so that if there's an incorrect value in one of the registers, it's easy to locate the guilty instruction.

Sometimes, it may be useful to breakpoint through each instruction rather than through several instructions at a time. In such cases you can single-step through the instructions in question with a **trace**. The **T** (Trace) command prints the contents of the MPU registers after each instruction is executed.

The **T** command generates a distractingly large amount of output as compared with the **B** command, which prints the MPU registers only after the program stops executing and control is returned to PROBUG. The **Q** (Quiet Trace) command is useful with observation points for locating an instruction that's changing memory.

Sometimes, all of the registers are correct, but memory is incorrect. If your program changed memory, use the **I**, **M**, or **P** command to inspect memory.

Forgotten Immediate Sign (#): Suppose location 1000 of your program contains a MOVE.L instruction. You had intended for this instruction to copy the value $20 into register D3. You breakpoint to location 1000 of your program, and then trace the next instruction using the **N** command:

```
>B1000.
BREAKPOINT AT 001000

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 0000E000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00000200
PC) 001000    SR) 0704    CCR) --Z--    .USP) 00000200    SSP) 0000076C
001000   MOVE.L    $20,D3
>N.

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 08F00FEE 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00000200
PC) 001004    SR) 0704    CCR) --Z--    .USP) 00000200    SSP) 0000076C
001004   MOVE.B    D3,$0C00
>
```

You notice that register D3 does not contain the immediate value 20, but rather 08F00FEE. Looking at the disassembled instruction (MOVE.L $20,D3), you realize that it has copied the contents of location 20 into register D3, rather than copying the immediate value $20 into D3. You can verify this by using the **M** command to inspect location 20.

You now have several options. If the error you found need not be fixed immediately, you can make a note on the listing to go back and correct the source, and continue on to the next bug. If the error must be fixed before you can continue debugging, you can fix the instruction with the **A** command (see below).

The contents of the registers have been set incorrectly because the instruction was incorrect. To fix this, change the value in register D3 as follows:

```
>RD3,20.
```

(See under the **R** command for details.) Note that the "20" is assumed to be hex in this command.

Make a note on your program listing to correct the source, and type a breakpoint, jump, or other program execution command to continue.

Now for a slightly more complicated example: suppose the next instruction also contained a reference to register D3, and that both instructions had been executed before you noticed that D3 should contain $20.

```
1000     MOVE.L    $20,D3
1004     MOVE.B    D3,$0C00
```

Correct the contents of register D3 as explained above. To correct the damage resulting from instruction 1004, you must change location 0C00 to contain $20. Use the **I** command as follows:

```
>I0C00.
000C00  FD 20.
>
```

If you are an experienced user of PROBUG's mini-assembler, you could change (patch) the program in RAM using the **A** command to replace the instruction:

```
>A1000.
001000    MOVE.L  #$20,D3
001002    (carriage return)
>
```

Note: In using the Assemble command, you will need to compensate for any size difference between the instruction you are replacing and the new instruction. For example, if the new instruction is a word shorter than the original instruction, add a NOP after the new instruction. If the new instruction is a word or more longer than the original one, you will need to insert a BSR and create a subroutine elsewhere in memory.

After assembling the patched code, you can start executing the program from the beginning again to reset the registers with proper values, or use the **R** command (as described above).

To continue debugging, disassemble the next group of instructions; or use your program listing to determine the next location at which there may be a problem, and breakpoint to that location.

Memory Location Being Changed: Sometimes a memory location is being changed when you did not intend for it to be changed. This can affect your program instructions, variables, etc., and can be a difficult problem to track down. However, by setting observation points at locations that are being set incorrectly, you can greatly speed up your search.

Suppose a subroutine of your program isn't working properly. You breakpoint to the beginning of the subroutine, and disassemble some instructions:

```
>B48000.
BREAKPOINT AT 048000

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00C01500 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00B00000
PC)  048000    SR) 0704    CCR) --Z--    .USP) 00B00000    SSP) 0000076C
048000  MOVEQ.L   #20,D0
>D.
048000  7014            MOVEQ.L   #20,D0
048002  31C0 0C00       MOVE.W    D0,$0C00
048006  4241            CLR.W     D1
048008  0000 5378       ORI.B     #$78,D0
04800C  0C00 66F8       CMPI.B    #-8,D0
048010  4E71            NOP
048012  60EC            BRA.S     $048000
048014  4E71            NOP
048016  FFFF            DC.W      $FFFF
>
```

You notice that this is not the original code. Location 48008 was supposed to

contain 5440 instead of 0000; the value was changed by mistake during program execution. You need to restore the original code, either by reloading the program using the **L** command, or by using the **A** command to re-enter the correct instruction as follows:

```
>A48008.
048008  ADDQ.W   #2,D0
04800A  (carriage return)
>
```

Now set location 48008 as an observation point:

```
>048008.
>
```

Supposing your program begins at location 40000, set the program counter to that location and then begin a quiet trace through the program. The quiet trace will run merrily along until the observation point changes:

```
>*40000.
>Q.

INSTRUCTION AT LOCATION 041020 CHANGED DATA AT
LOCATION 048008 FROM 5440 TO 0000

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00000200
PC)  041026    SR) 0704    CCR) --Z--   .USP) 00000200    SSP) 0000076C
041020   CLR.W     $048008
041026   SUBQ.W    #1,D0
>
```

The output from the quiet trace reveals that the CLR.W at location 041020 is the guilty instruction.

Conditional Branch Not Being Taken: You will find it useful to breakpoint to conditional branches to see whether the program takes the branch. If something is wrong, there's a problem with either the conditions leading up to the branch, or the branch instruction itself.

Suppose you breakpoint to a BEQ instruction to make sure the program takes the branch. When the BEQ instruction is reached, the MPU register contents are printed, followed by the disassembled instruction. The disassembled instruction includes a (TRUE) or (FALSE) label which indicates whether or not the branch will be taken.

You expected the branch to be taken, but the label says (FALSE), indicating that the branch will not be taken. However, the conditions leading up to the branch have all been set correctly.

The BEQ instruction should really have been a BNE. (This is easy to identify using the (TRUE) or (FALSE) label as a guide.) Make a note on your listing to go back and correct the source. Then use the **A** command to change the instruction to a BNE; or jump (use the **J** command) to the address to which it

was supposed to go.

Program Control Passing To Wild Address: Another difficult problem to track down is when program control passes to some wild address outside the program. This is often caused by careless stack maintenance and usually shows up as an illegal instruction or unimplemented instruction.

In such cases, it is useful to quiet-trace to the location of the invalid instruction, to find out how the MPU got to that address. With the **Q** (quiet trace) command, PROBUG traces through each instruction without generating any output. When the specified address is reached, PROBUG disassembles both the current instruction and the previously executed instruction, allowing you to follow the flow of program control.

For example, if your program is crashing with an unimplemented instruction at 40020, run the program using a **Q** to that location:

```
>Q40020.

      (0)      (1)      (2)      (3)      (4)      (5)      (6)      (7)
D) 00000012 0000FE00 00001000 00000000 00000000 00000000 00000000 FFFFFFFF
A) 00123455 00000000 555FF300 999F0000 00000000 00000000 0000FE00 00000B00
PC)  040020    SR) 0704    CCR) --Z--   .USP) 00000B00    SSP) 0000076C
001420   RTS
040020   DC.W    $FFFF
TRACE STOPPED AT 040020
>
```

001420 is the address of the instruction that caused the program to jump to 40020.

no p. 72