RTE-IV Debug Subroutine

Reference Manual

HEWLETT PACKARD

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014

Library Index No.
2RTE.320.92067-90005

MANUAL PART NO. 92067-90005
Printed in U.S.A. February 1980

PRINTING HISTORY

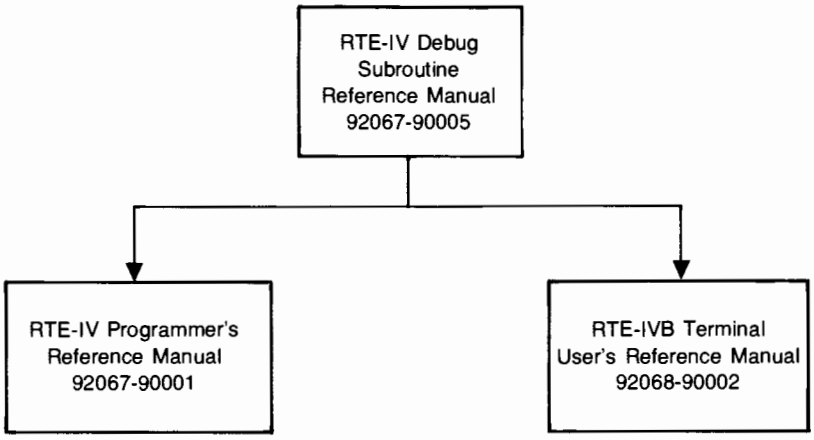
New editions are complete revisions of the manual. Update packages contain replacement pages or write-in instructions to be merged into the manual by the customer. Manuals will be reprinted as necessary to incorporate all prior updates. A reprinted manual is identical in content (but not in appearance) to the previous edition with all updates incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

Third Edition ..... Feb 1980

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

DOCUMENTATION MAP



See the *RTE-IV Programmer's Reference Manual* and *RTE-IVB Terminal User's Reference Manual* for complete documentation maps.

CONTENTS

CONTENTS (Continued)

ILLUSTRATIONS

SECTION I
INTRODUCTION

DBUGR is a utility program for debugging programs run on HP 1000 series computers. It features:

* Symbolic or octal printout
* Symbol definition
* Register examination and change
* Paper tape loading and verification
* Memory search
* Memory clear
* Breakpoints
* Map examination.

1-1. OPERATING ENVIRONMENT

DBUGR runs on any HP 1000 series computer equipped with DMS and teleprinter (or CRT) running the RTE-IV operating system. DBUGR itself is approximately 3.6K words in length including symbol table and excluding the breakpoint table. The user symbol table space is fixed at 50 locations in length. Two locations are required for symbols of one or two characters, while three locations are required for symbols of three to six characters. The breakpoint table, which uses 50 words of memory, consists of 10 breakpoints of 5 words each.

Paper tape operations are configured to dump to LU4 and read from LU5.

1-2. LOADING AND USING DBUGR

To use DBUGR it must be relocated with the program to be debugged. This may be done in one of two ways. The first is to use the DB format in the loader. LOADR will set the primary entry point of the program to DBUGR and save the program's actual start address in DBUGR. For example:

```
*RU,LOADR,,%PROG,,DB
```

Note that this is the only way that segment breakpoint can be utilized.

The second method is for the program to call DBUGR directly by the following calling sequence:

```
        Assembly                                Fortran

        EXT DBUGR                               CALL DBUGR(lu#-optional
        JSB DBUGR                                          parameter)
        DEF RTN
        DEF lu# of console (optional)
    RTN EQU *
```

When the user's program begins execution, DBUGR takes control and prints START DBUGR on the optional console logical unit or, if not included, on the logical unit passed to DBUGR through the system subroutine LOGLU. At this point the user can initiate a debug operation. All debug operations are conducted at the Assembly Language level. A load map of the program is essential. If debugging a program written in a higher level language, a mixed listing of source and assembly is required.

DBUGR has 10 breakpoints available which reside in a relocatable module named %SGBPT. If the user desires to change the number of available breakpoints, the user may write his own SGBPT module and load the new code in place of the system library routine SGBPT. This can be done by using the loader LI,%SGBPT command to search %SGBPT file before the system library.

The routine looks like:

```
ASMB,R,Q
       ENT SGBPT,SGBPE
SGBPT  DEF *+1            start of table points
       REP 10             determines number of breakpoints (10)
       OCT 0,0,0,0,0
SGBPE  DEF *              end of table pointer
       END
```

1-3. ELEMENTS OF DBUGR AND TEXT CONVENTIONS

Input to DBUGR consists of controls and expressions.

1-4. CONTROLS

Controls, consisting of special characters and letters preceded with escape or ALTMODE, act as directives to DBUGR. In the text that follows, escape will be denoted by \, carriage return by CR, control by CTRL, tab by TAB and line feed by LF. (CTRL-J is optional LF on 264x terminals.) Escape prints as a backslash (\) on a TTY or CRT. Some controls are:

```
/
!
\S
=
+
```

For muiltipoint terminals or terminals using driver DVR07, refer to Appendix D.

NOTE

Some terminals do not print the first backslash after an excape (e.g., the 2640). The control "\U" will cause DBUGR to print two backslashes for each escape.

1-5. EXPRESSIONS

An expression, which will be denoted in the text by the letter n, consists of one or more terms, combined by operators as in the following:

```
AA+10
```

1-6. TERMS

A term may be a symbol (denoted in the text by the letter s), a number, or a special notation called a mark. The following are terms:

BB — A symbol is defined as a letter or a period followed by any number of letters, digits or periods. Only the first six characters are significant. For example:

```
ABC         A5        .B
A.ZZZZZ     .4        ...
```

Each symbol may be assigned a value. The symbol and its value are equated by entries in the DBUGR symbol table. Values for symbols lie in the range of -32768 through 32767.

3775 — An octal number.

386. — A decimal number. The decimal point is used to characterize the number as decimal rather than octal. A decimal point in a number other than as the last character is not meaningful.

. and * — Period and Asterisk are marks. When used as terms in an expression their value is equal to that of the current value of DBUGR's location counter.

\Q Esc-Q is a mark. It refers to the last quantity typed.

1-7. OPERATORS

Legal operators are:

+ Add terms

Space Add terms (this is manually faster than shifting and depressing the + key.)

- Subtract term

, Inclusive OR terms (inclusive OR is used to selectively set individual bits within a word).

Operators are discussed in more detail in Appendix A.

1-8. CONVENTIONS FOLLOWED IN THE TEXT

1. What you input to DBUGR is underlined.
   ---

2. Use of a control that does not cause print-out is denoted by brackets ([]). These will not be underlined.

3. All values referenced in the text are octal unless otherwise noted.

4. Vertical and horizontal spacing shown in the examples is not exactly as it appears on a teletype or CRT printout.

1-9. LIMITATIONS

DBUGR reads and interprets each character as it is entered so that many commands may be invoked by a single keystroke. Since one character is read at a time, it is possible for the user to get ahead of DBUGR and get a system prompt. When this happens the user should increase the priority of the program being debugged with the PR command. Alternatively, through use of the \\U command, DBUGR can accept a continuous string or line of characters as would be sent from a 264x terminal by a soft key, cartridge tape unit, or a Multipoint 264x terminal using driver DVR07.

OUTPUT MODES

SECTION II

Information can be displayed to us in one of four modes:

1. Symbolic instructions
2. Numeric constants
3. Address pointers
4. ASCII characters

Although a printing mode normally remains in effect until changed, it is possible to either:

a. momentarily invoke a different mode for the display of only one number, or

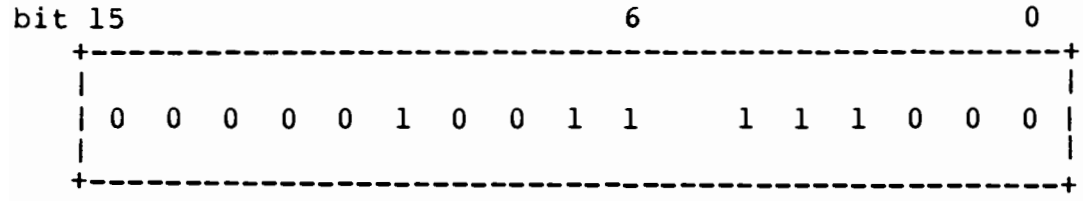
b. temporarily invoke a different mode for a series of examinations until a carriage return is entered.

2-1. PRINTOUT AS SYMBOLIC INSTRUCTIONS

DBUGR is in symbolic mode when it is loaded.

In symbolic mode the contents of a memory location are printed as symbolic instructions; the operation codes are represented as mnemonics. The address field of one-word memory reference instructions will be printed as octal numbers with the page number merged with the page offset. Only the first word of multi-word instructions will be interpreted.

It is important to note that in symbolic mode every location is printed as a symbolic instruction whether that location contains an instruction, data or an address. The bit pattern in memory is simply interpreted as an instruction. For example, the octal value 1767 would appear in memory as:



Notice that bits 6 through 15 correspond to the bit pattern for the ALF instruction. The lower 6 bits are similarly treated as instructions. The printout appears as:

```
ALF,CLE,ALF
```

Also notice that not all shift rotate instructions are defined. In those cases DBUGR will print the valid opcodes merged with the remaining bits. For example, the octal value 1747 will be printed as ALF,CLE,7.

DBUGR's symbol table does not contain instructions referencing the overflow register, therefore overflow instructions will be printed as I/O instructions with a select code of 1.

The operand field for I/O and double shift instructions will be printed correctly except when the operand is zero, in which case a blank is printed instead of a zero. Therefore STF 0 will be printed as STF, and RRL 16 (actual operand is 0) will be printed as RRL.

"I" and "C" are special symbols in DBUGR's symbol table with the octal values of 100000 and 1000 respectively. Therefore 100000 and 1000 will be reverse assembled as I and C.

From this discussion, it is obvious that we must know whether we are looking at data or instructions. It is also obvious that it is desirable to be able to print data or address words as octal numbers. DBUGR provides this capability.

2-2. PRINTOUT AS NUMERIC CONSTANTS

We can either momentarily change the printing mode to octal so the last item printed will be reprinted as octal, with subsequent printouts continuing in the master mode; or we can change the master mode to octal so that all subsequent printouts will be in octal.

Conversely, if we change the master mode to octal, we have the capability of momentarily switching back to symbolic; it works both ways. The examples below demonstrate printing modes. In these examples the following controls are used:

n/ Prints the contents of memory location n. These contents can be modified at this point, but that is discussed later.

\C Set master print mode to numeric constant. Remember, C means constant.

= Print the last quantity (typed by either us or DBUGR) as a numeric constant. This is a momentary mode change.

\= Same as =, except constant mode remains in effect until a carriage return. This is a temporary mode change.

\S Set master print mode to symbolic instruction. Remember, S means symbolic.

! Print the last quantity typed as a symbolic instruction. This is a momentary mode change.

\! Same as !, except symbolic mode remains in effect until a carriage return. This is a temporary mode change.

2-3. CHANGING THE RADIX OF NUMERIC PRINTOUT

An interesting feature of DBUGR is that the numeric output (in any mode) does not have to be octal. The radix of output values can be from 2 to 33. Thus, it is possible to print data as a decimal value or examine memory in hexadecimal. If the radix is decimal, numbers are followed by a period. You can change the output radix by using the control. DBUGR interprets all values as 16- bit unsigned numbers, regardless of radix.

n\R where n can be any decimal value from 2 to 33 (or octal value from 2 to 41. Remember, R means radix.

For example:

```
2420/  ADA 2745  \C [LF]                    Select numeric constant
-----            --                         master mode
2421/  3004 2\R  [LF]                       Select binary output
           ---
10100010010/  10000010000 10\R [CR]         Select octal output*
                          ----
\S   [LF]                                   Select symbolic master
--                                          mode
2423/  JMP 2714,I  [LF]

2424/  SZA  [LF]

2425/  JMP 2714,I  2\R  [LF]                Select binary output
                    ---
        .
        .
        .
        .
```

*The radix could have been specified in decimal as 8.\R.

```
10100010110/ JMP 10111001101,I 20\R [CR] Select hexadecimal output
                                ----
\S [LF]                                     Select symbolic master
--                                          mode
517/ JSB 331,I [LF]
518/ CLA,SSA,SLA,SZA  \C [LF]               Select constant master
                      --                    mode
519/  51A  [LF]
51A/  9B3E 10\R  [CR]                       Select octal output
           ----
2471/  3004  10.\R [LF]                     Select decimal output
-----        -----
1338./ 1040.  [LF]
1339./  44464.  [LF]
1340./ 1026. [LF]
```

2-4. ALPHANUMERIC (ASCII) PRINTOUT

It may be necessary to examine alphanumeric information. If so, DBUGR allows us to change the printing mode so that the contents of each memory location are printed as two ASCII characters. In this mode DBUGR will interpret the 16 bits as two 8-bit ASCII codes, whether the memory location contains data, an instruction, or an address. The controls are:

\H	Sets master printing mode to ASCII characters.
′	Prints the last quantity typed as two ASCII characters. Then prints a double quote.
\′	Same as ′, except ASCII mode remains in effect until a carriage return.

Study the following example carefully:

```
2514/ 115762            \H [LF]  Select ASCII printout
-----                   --
2515/PV"=50126          ′ PV"  [LF]
        -               -
2516/NE"=47105          ′ NE"  [LF]
        -               -
2517/@."=40056          ′ @."  [LF]
        -               -
2520/-"=2655            ′ -"   [LF]
        -               -
2521/*"=115452          ′ *"   [LF]
        -               -
2522/S"=2523            ′ S"   [LF]
        -               -
2523/-0"=126660         ′-0"   [LF]
        -               -
2524/ >"=115476         ′>"    [LF]
        -               -
2525/+0"=25460          ′+0"   [LF]
        -               -
.
.
.
2532/-"=2655            ′-"    [LF]
        -               -
2533/ "=12              ′"     [LF]
        -               -
2534/ "=5               ′"     [LF]
        -               -
2535/ *"=115452         ′*"    [CR]
        -               -
```

From an examination of the octal equivalents of the printed characters, several observations can be made:

1. A character will be printed only if the upper and/or lower 8 bits of a word contain bit patterns that represent legitimate ASCII characters.

2. If either 8-bit field does not represent an ASCII character, it is not printed. There is no way to determine whether a single printed character fell in the upper or lower half of the word without looking at the octal equivalent. At location 2522 above, only the lower half of the word converts to the character S.

3. If neither half word contains an ASCII code, but does contain binary information, nothing is printed (location 2534 above). If a non-printing ASCII character is encountered, DBUGR performs the function that ASCII code represents. For example, encountering a 007 causes the bell to sound, and encountering a 012 causes a line-feed.

2-5. ADDRESS POINTERS

There is one more printing mode, It allows DBUGR to print a 16-bit memory location as an address pointer. Keep in mind that in HP 1000 computer programming, many words are set up to contain a full 15-bit address plus a bit to indicate direct or indirect use of that address. In the following code:

```
3000          LDA         POINT,I
                .
                .
                .
3010 POINT    DEF         DATA
                .
                .
                .
7000 DATA     BSS         1000
```

the DEF pseudo operation generates a full 15-bit address at location 3010. The address generated is that of the symbol DATA; i.e., 7000, and this address is used as a pointer by instructions such as the LDA at 3000 to access the data stored at DATA. The contents of location 3010 look like:

```
bit 15                                              0
   +-------------------------------------------------+
   |0  0  0  0  1  1  1  0  0  0  0  0  0  0  0  0| =7000
   +-------------------------------------------------+      8
```

Memory locations containing address words can easily be examined in numeric constant mode. In fact, if we switch to the printing mode that allows us to print the contents of memory locations as addresses, there appears to be no difference in output. The example below demonstrates the apparent identity. The control <- (left arrow or underline on some CRTs) is used to print the last quantity typed in address pointer mode.

```
\C      [CR]
--
1510/  60247    <- 60247    [LF]
-----           --
1511/  3004     <- 3004     [LF]
                --
1512/  102000   <- 2000,I   [LF]
                --
```

The two modes return almost identical results! What then is the use of address pointer mode?

Printing in address pointer mode is useful only in conjunction with another feature of DBUGR; i.e., the capability of associating symbols with numeric addresses.

2-6. DEFINING SYMBOLIC ADDRESSES IN DBUGR

Remember that when a program is in memory for execution, all symbols associated with program locations at assembly time have been converted to absolute addresses; they are gone! As DBUGR maintains its own symbol table, we can associate arbitrary symbols with specific locations. This is useful because we may wish to see locations and address references printed as symbols rather than words.

Before we see how symbols are manipulated, it is necessary to learn how DBUGR sequences through a program. For example, to begin debugging at location 2500, we would type:

```
2500/
-----
```

DBUGR will respond by typing the contents of location 2500 (assume we are in numeric constant print mode). The output would look like:

```
2500/  171572
-----
```

If we strike LF or CTRL-J on a 264x, the next sequential location in memory is displayed:

```
2500/  171572 [LF]
-----
2501/     723 [LF]
2502/    5241
```

Although it was not explicitly mentioned, in all previous examples advancing to the next line of print was accomplished by LF.

Obviously, DBUGR maintains a location counter similar to the P-register of the computer, and LF is the control for sequentially examining memory. Later we will see that the contents of the memory location just printed can be changed before executing the LF.

If we wish to associate a symbol with the address 2500, we can type the following:

n<s: where n is an expression resulting in a 16-bit value and s is a symbol. The symbol is then said to be defined.

Thus, "less than" (<) is a control to designate a value, and "colon" (:) is a control to define a symbol.

```
2500<SYM:
```

will associate the symbol SYM with the address 2500.

In the following example, notice how symbols replace numeric notation after location 2500. Assume we are in numeric constant mode, and SYM has been defined as above.

```
2476/   126720   [LF]
-----
2477/   115476   [LF]
SYM/    2734     [LF]
SYM+1/  115471   [LF]
SYM+2/  25460    [LF]
SYM+3/  126666   [LF]
   .
   .
   .
   .
   .
   .
   .
```

A strange thing happens after location SYM+10:

```
SYM+10/  165715   [LF]
2511/    7004     [LF]
2512/    77113
```

Addresses are associated relative to a particular symbol for 10 octal locations. If we wish to continue using symbolic addressing, we have to define another symbol:

```
SYM+130/    165715 [LF]
--------
2631/ 7004  [CR]
2631/ SYM2: [LF]
-----------
SYM2+1/ 77113
```

Notice the use of CR above. Unlike LF it only returns us to the next print line; the location counter does not change. Actually it is generally not necessary to strike a CR before typing a new address or using a new control. You can continue on the same line.

The exception to the 10 rule above is for symbols of one or two characters. For these symbols, the numeric offset has no limit. Thus:

```
2631<S:  [CR]
-------
S+5000/ 7004 [LF]
-------
S+5001/ 165715 [LF]
   .
   .
   .
```

This mode is useful with relocatable code where we might define a symbol for the origin of each module. The offsets would then be the same as those printed on the assembly listings.

Another form of symbol definition that would have been more efficient at location 2631 above is illustrated below:

```
SYM+130/    165715    [LF]
--------
2631/       7004  SYM2:    [LF]
                  -----
SYM2+1/     77113
```

When no value is explicitly designated for a symbol definition, the current value of the location counter is used.

Now let us look more closely at the printout in symbolic instruction mode (Column A of Figure 2-1). Notice that we may specify the address at which to begin printing by using its symbolic form:

A SYMBOLIC INSTRUCTION		B ADDRESS POINTER	C OCTAL CONSTANT
SYM+1/	JSB 1471,I	<-15471,I	=115471
SYM+2/	JMP 1460	<-25460	=25460
SYM+3/	JMP SY+35,I	<-26666,I	=126666
SYM+4/	JSB 1454,I	<-15454,I	=115454
SYM+5/	CLA,CLE,INA,SZA,RSS	<-SYM+7	=2507
SYM+6/	CLE,SEZ,SSA	<-2120	=2120
SYM+7/	JMP 2716,I	<-26716,I	=126716
SYM+10/	JSB 1453,I	<-15453,I	=115453
SYM+11/	CLA,CLE,SSA,SZA,RSS	<-SY+13	=2513
SYM+12/	CLE,SEZ,SSA	<-2120	=2120
SYM+13/	JMP 2716,I	<-26716,I	=126716
SYM+14/	JSB 1762,I	<-15762,I	=115762
SYM+15/	CLA,CLE,SSA,RSS	<-SY+21	=2521
SYM+16/	2746	<-SY+115	=2746

Figure 2-1.COMPARISON OF SYMBOLIC, ADDRESS POINTER, AND OCTAL PRINTOUT

Notice that not only are the locations printed in symbolic form, but some addresses referenced by memory reference instructions also appear using the offset limits; i.e., 10 for multicharacter symbols or no limit (but >0) for one or two character symbols.

Referring to the octal printout in Column C, notice that the entire word is printed as an octal constant, even if that constant represents an address that can be represented symbolically, as at location SYM+7.

In symbolic instruction mode, symbolic addresses appear only if that word is a one-word memory reference instruction. Address pointers will not be printed symbolically. They will appear either as octal constants, as locations SYM+16 or as meaningless instructions as at location SYM+11.

A glance at Column B in Figure 2-1 immediately reveals that address pointer mode is the only mode that will print 15-bit addresses in symbolic form. Notice that the control <- is used to momentarily invoke address pointer mode. To set the master printing mode to address pointer, use the control:

\A Sets the master printing mode to address pointer. Remember A means address.

The control \ <- can be used to temporarily invoke address pointer mode until a carriage return is executed.

2-7. DELETING SYMBOLS FROM THE SYMBOL TABLE

Using the control \K will kill all symbol definitions that we have defined. Symbols defined by DBUGR remain. Remember K means kill.

CAUTION

Do not use the following to define address labels:

1. Instruction mnemonics; if a symbol is redefined, the original value is lost.
2. The letters C and I; these are defined as 1000 and 100000 octal, respectively.
3. Special Register Symbols.

2-8. ASSIGNING A SYMBOL TO AN ADDRESS JUST PRINTED BY DBUGR

In addition to being able to define a symbol equal in value to a specific address or to the current value of the location counter, we can equate a symbol equal in value to an address just printed by DBUGR:

`s&` Defines symbol s equal in value to the address of the last quantity typed, if an instruction.

For example:

```
1476/   JMP 305 XFER&   [CR]
-----
305/    SLA+17          [CR]    The symbol XFER is defined
----                            as location 305
XFER/   SLA+17          [LF]
-----
XFER+1/ SLA+20
```

2-9. HOW TO PRINT THE LAST QUANTITY TYPED IN MASTER MODE IF YOU HAVE JUST CHANGED MODES

`; Prints last quantity typed in master mode.`

For example:

```
1700/ HLT 77 =102077 ;HLT 77 [CR]
-----        -       -
1777/ 40502 'AB" ;40502
-----       -    -
```

MEMORY EXAMINATION AND MODIFICATION

SECTION III

3-1. CONTROLS TO EXAMINE MEMORY ONLY — LOCATION COUNTER DOES NOT CHANGE

We have already used two controls to examine memory locations:

n/ To open and print the contents of location n.

LF To open and print the contents of the next sequential location.

Actually these controls do more than print the contents of the memory location; they "open" the location for possible modification. Before we explore that possibility, let us look at some other controls for examining memory:

/ Opens and prints the contents of the location pointed to by the last quantity typed. The quantity typed is used as a 15-bit address, regardless of the printing mode.

For example:

```
2007/  20602  /  67  [LF]   Opens and prints contents
-----         -             of 20602

2010/  30501  /  115723     Opens and prints contents
              -             of 30501
```

Notice that DBUGR's location counter does not change. We can continue sequential examination of memory simply by using LF.

\/ Opens and prints the contents of the location pointed to by the last quantity typed. Unlike /, the last quantity typed is interpreted as a memory reference instruction and only the 11-bit address field is used as the effective address. The location counter does not change.

\L Prints the contents of the next 16 sequential locations; the last printed location is left open.

n\L Same as the \L, the quantity n replacing 16 as the number of locations listed.

For example:

```
5010/    ELA    5\L
-----           ---
5011/    ADA 2042
5012/    CMA,INA
5013/    SSA,RSS
5014/    JMP 3066
5015/    JSB 2076
```

3-2. CONTROLS THAT EXAMINE MEMORY AND SET THE LOCATION COUNTER TO A REMOTE ADDRESS

The following two controls are similar to / and \/, but notice that the location counter is set to the address of the memory location being examined.

CTRL-I (TAB) (compare with /)

Opens and prints the contents of the location pointed to by the last quantity typed. This quantity is used as a 15-bit address pointer. DBUGR's location counter is changed to contain this address.

For example:

```
2312/   2031    [CTRL-I]   Location counter
-----
2031/   126313  [CTRL-I]   change to 2031
26313/  43734              then to 26313
```

\CTRL-I (\ TAB) (Compare with E/)

Same as CTRL-I except that the last quantity typed is interpreted as an instruction and only the 11-bit address field is used as the pointer.

For example:

```
5002/  JMP  5007  \  [CTRL-I]  Location counter
-----             -
5007/  SLB+56                  Changes to 5007
```

3-3. CONTROLS TO CHANGE THE CONTENTS OF MEMORY

To change the contents of a memory location, simply open that location, then type the quantity desired followed by one of the controls discussed below. The quantity you type is designated by n.

3-4. MEMORY CHANGE CONTROLS THAT MAINTAIN THE LOCATION COUNTER WITHIN THE CURRENT SEQUENCE

nCR
nLF
n^

n is the quantity that we want stored in the open location. The control that follows this quantity executes as previously described.

For example:

```
5010/   1600    7777  [CR]  Contents of 5010
-----
*/ 7777                     changed to 7777. Verify by
--                          re-examining 5010.
```

Note that an asterisk is used to imply an address equal in value to the current value of the location counter. A period can be used in place of an asterisk.

```
5012/  27006   6655  [LF]   Contents of 5012
-----          ----
5013/  2060    ^             changed to 6655.  Verify
                -
5012/  6655          [CR]    by sequencing backward.

5012/  6655    44444  ^      Contents of 5012 changed
-----          -----  -
5011/  7777    [LF]          to octal 44444.  Verify
5012/  44444                 by sequencing forward.
```

IMPORTANT: The use of CR will close the currently open memory register!

For example:

```
5010/  1600    7777  [CR]   Contents of 5010 becomes
-----          ----
3333   [CR]                 7777.  Attempt to change
----
*/ 7777                     again, but 5010 still con-
--                          tains 7777.
```

NOTE: Note that a period is used above to refer to an address equal in value to the current value of the location counter.

DBUGR provides a control for storing one or two ASCII characters in a word. It has the form s" where s is a symbol.

Type the ASCII input and the quote followed by any of the memory change controls discussed above:

```
1700/   20532  AB"     [CR]
-----          ---
*/  40502    'AB"
--           -
```

Only letters, digits, or the period, may be used; blanks and special characters are illegal.

The next two controls allow us to store a new address in an open memory location and then open and examine the contents of the location pointed to by that address.

n% Stores the quantity n in the currently open memory location; then opens and prints the contents of the location pointed to by n. N is used as a 15-bit address pointer. The location counter does not change.

Notice that this control is similar to / with the added capability of changing the currently open memory location.

For example:

```
5011/  7777 55%  102055  [CR]
-----       ---
```

The contents of 5011 are changed to contain the quantity 55; then location 55 is opened and its contents are printed.

n\% Same as n% except that n is interpreted as an instruction, and only the 11-bit address field is used as a pointer to the location to be opened and printed.

For example:

```
4011/ ADB 215 ADB 312\% 4445 [CR]
-----         ---------
312/ 4445 [CR]
----
4001/ ADB 312
-----
```

The contents of 4001 are changed to point to location 312. Location 312 is then opened and its contents printed. It is then opened again for comparison. Notice that we reprint location 4001 to verify the change.

3-5. MEMORY CHANGE CONTROLS THAT CHANGE THE LOCATION COUNTER TO A REMOTE SEQUENCE OF CODE

The following controls allow us to store a new address in the currently opened location and then open, print and set the location counter to the address into which we have just stored.

nCTRL-I Stores n in the currently open register and then opens and prints the contents of the location pointed to by n. Remember that the quantity is interpreted as a 15-bit address. The location counter is set to this address.

For example:

```
337/  STB 1772,I    / ALF+73 =1773  1660 [CTRL-I]
----                -               ----
1660/ ISZ 126 =34126  [CR]
              -
1772/ ELA+60 =1660
-----        -
```

Location 337 contains an indirect reference to location 1772, which contains 1773. Before 1660 CTRL-I is typed, location 1772 is the open location. Typing 1660 CTRL-I changes the contents of 1772 to the value 1660 and then opens and prints the contents of location 1660, with the location counter set to this address.

n\CTRL-I Same as nCTRL-I except that n is interpreted as an instruction.

For example:

```
330/  JMP  461  JMP  500 \[CTRL-I]
----            -------- -
500/  JMP  207,I [CR]
330/  JMP  500
----
```

Location 330 originally contains a JMP 461. This is changed to a JMP 500, and location 500 is opened, printed and the location counter is set to location 500.

This control is excellent for making in-core patches. For example, suppose it is necessary to insert instructions between locations 3737 and 3740:

```
3737/  ADA  200
3740/  SSA
```

The inserted code must be stored at locations remote from the in-line code, therefore, we must jump to the added instructions from location 3737 and return to location 3740.

Assuming that memory is available at location 3745, we can use DBUGR as follows:

```
3737/  ADA  200  JSB  3745  \[CTRL-I]
-----          ------------
3745/  0  [LF]
3746/  0  ADA 200   [LF]
          --- ---
3747/  Inserted code
         .
         .
         .
         .
37xx/  0  JMP 3745,I
          ----------
```

Note that the ADA 200 was moved to the beginning of the inserted code to make room for the JSB instruction.

3-6. SPECIAL REGISTER MODIFICATION AND EXAMINATION

To examine user or system maps, modify or examine O-, E-, X-, Y-registers, or examine the DMS status as of the last breakpoint, special modes are required.

3-7. SPECIAL REGISTERS

\M [CR] will cause the following two lines to be displayed:

```
AREG BREG XREG  YREG EOREG MASK   (CBVAL-CBADDR,I) CBMASK = CBTEST
11   237  11376 5775 2     177777 7      22042     177777   SZA
```

AREG	A-register
BREG	B-register
XREG	X-register
YREG	Y-register
EOREG	E-, O-registers in bits 1 and 0 respectively
MASK	Search Mask
CBVAL	Conditional Breakpoint Value
CBADDR	Conditional Breakpoint Address
CBMASK	Conditional Breakpoint Mask
CBTEST	Conditional Breakpoint Test Instruction

The special registers are accessible by the symbol names given above, the same way user defined symbols may be accessed. The EOREG holds the extended bit status in bit 1, and the overflow bit status in bit 0. For example, EOREG = 3 means extend and overflow are set. EOREG = 1 means extend register clear and overflow register set.

Two other symbols available to the user are:

WRTLU The EXEC control word of the device for DBUGR output. (Refer to the RTE-IV or RTE-IVB Programmer's Reference Manual regarding EXEC calls.)

BRFLG 0 means check for break in debug loops

1 means no check will be made for break (speeds up loop processing)

3-8. MEM STATUS SPECIAL MODE

The following special mode displays the MEM status.

\? displays the MEM status as of the last breakpoint in the following format:

```
MS =  X15 X14 X13 X12 X10 YYYY
```

where X15, X14, X13, X12, X11 and X10 = MEM status register bits 15, 14, 13, 12, 11, and 10 respectively. YYYY is the four digit octal number representing the base page fence.

3-9. MAP EXAMINATION SPECIAL MODE

The following special mode allows examination of the system and user maps and cross load.

\J puts DBUGR into this special mode and responds with a CR LF and three spaces.

UM Displays the user map.

SM Displays the system map.

XL Sets up this special mode to cross load from an address in the alternate map.

PA Displays the port A maps.

PB Displays the port B maps.

A Aborts the special mode with no change.

```
UM:
--

 0 = MR0      20 = MR20
 1 = MR1      21 = MR21
 2 = MR2      22 = MR22
 3 = MR3      23 = MR23
 4 = MR4      24 = MR24
 5 = MR5      25 = MR25
 6 = MR6      26 = MR26
 7 = MR7      27 = MR27
10 = MR10     30 = MR30
11 = MR11     31 = MR31
12 = MR12     32 = MR32
13 = MR13     33 = MR33
14 = MR14     34 = MR34
15 = MR15     35 = MR35
16 = MR16     36 = MR36
17 = MR17     37 = MR37
```

The system map registers routine operates the same as the user map registers routine except "UM" is replaced by "SM".

The cross load routine begins by outputting a CR, LF and six spaces with XL followed by three more spaces. The operator then enters an address and a slash to display the contents of that address in the alternate map. DBUGR will return to allow additional cross loads.

For example:

```
XL
--

      XL   ADDRESS/  (old contents)
           --------
      XL
```

An LF will also increment the address counter as in the case of examining memory locations.

3-10. CONTROLS TO TEMPORARILY CHANGE THE PRINT MODE OVER A SERIES OF EXAMINATIONS

There are several controls that we can use in place of n/ and n\/ (refer to paragraph 3-1). As with \!, \=, \' and \<- they temporarily change the print mode until we execute a carriage return, thereby returning to master mode. However, unlike \!, \=, \' and \<-, they open and print the contents of a register.

n$ Same as n/ but sets temporary print mode to symbolic instruction.

n# Same as n/ but sets temporary print mode to numeric constant.

n@ Same as n/ but sets temporary print mode to address pointer.

n) Same as n/ but sets temporary print mode to ASCII.

n\$ Same as n\/ but sets temporary print mode to symbolic instruction.

n\# Same as n\/ but sets temporary print mode to numeric constant.

n\@ Same as n\/ but sets temporary print mode to address pointer.

These controls can be used without a preceding address expression (n), in which case they can be used in place of / and \/. However, the mode change will be momentary only. Below are given two examples.

Example #1:

```
\C                                                    Begin in constant mode
--
2537/  421  [CR]
-----
2540$  LDA  2534,I  / LDA 62572 / 234 [LF]            Change to symbolic
-----               -           -                     temporary mode
2541/  ADA  2740    [LF]
2542/  SSA          [LF]
2543/  JMP  2657    [LF]
2544/  CLA          [LF]
2545/  STA  2663    [LF]
2546/  LDB  2661    [LF]
2547/  STB  2662    [CR]                              Return to constant mode
[LF]
2550/       125503  [LF]
2551/       46664   [LF]
2552/       6020    [CR]
4000$       SZA     [LF]                              Change to symbolic
-----                                                 temporary mode
4001/  JMP  4016    [LF]
4002/  LDA  1717    [LF]
4003/  STA  5235  / CPA  72525  / JSB 34 [LF]
               -              -
4004/  STA  5236    [CR]                              Return to constant mode
[LF]
4005/  53266        [LF]
4006/  73237        [LF]
4007/  17575        [LF]
```

Example #2:

```
\S                                                    Begin in symbolic mode
--
4012/  LDB  1717    [LF]
4013/  ADB  5273    [LF]
4014/  CLA          [LF]
4015/  STA  1,I     [LF]
4016/  LDA  4103    [LF]
5000#  2003 /  72053 [LF]                             Change to constant
-----       -                                         temporary mode
5001/  126775       [LF]
5002/  73240        [LF]
5003/  12306        [LF]
5004/  73242        [LF]                              Return to master mode
5005/  2400         [CR]
[LF]
5006/  STA  1,I     [LF]
5007/  LDA  5240    [LF]
```

CONTROLS TO LOAD, PUNCH AND VERIFY PAPER TAPE

SECTION IV

4-1. PROGRAM LOAD

DBUGR provides a control for loading absolute binary paper tape:

\Y Loads absolute binary paper tape, as prepared by the HP Assembler. Mount tape to be loaded before typing \Y. Remember, Y means yank. Loading will terminate at end of tape, or upon any attempt to load beyond the upper limit. DBUGR will report an error for any attempt to load over itself.

4-2. PUNCHING BINARY TAPES AND PRODUCING PATCHES

Memory locations can be changed and punched on paper tape one by one, or blocks of memory can be punched.

As the tape is punched in absolute format readable by HP absolute program loaders, we can use these controls to punch entire programs or binary patches.

n1<n2>\D Punches locations n1 and n2 inclusive

n\D Stores the value n into the open register, if any; then punches the register last opened (with the new contents, if stored).

Let's say we have have discovered a bug and must make the following changes:

Location	Instruction	Change
2050	RBR	RAR
2051	SSB	SSA

Using DBUGR in symbolic instruction mode:

```
2050/  RBR  RAR  \D  [LF]
-----       ---  --
2051/  SSB  SSA  \D
            ---  --
```

One record is punched for each word.

4-3. TAPE VERIFICATION

To verify a tape, we use the following control:

\V Verifies paper tape against memory. Remember V means verify. Mount tape prior to typing \V. Discrepencies are listed as follows:

Address/ [contents of memory] [contents of tape]

If the word on tape is zero, no discrepancy is listed. Verification will terminate at end of tape or upon any attempt to verify beyond the upper limit. DBUGR will report discrepancies within itself.

MEMORY SEARCH AND CLEAR

SECTION V

5-1. MEMORY SEARCH

We can search memory within specified addresses for words that are either equal or not equal under mask to a specified word.

Actually, memory is searched for a quantity that is equal or not equal to a quantity formed by taking the logical product (AND) between the mask and each memory location examined. This allows us to search for only a specified group of bits within each word.

5-2. THE SEARCH MASK

The mask is stored at a location referenced by the symbol "MASK". Initially it is set to 177777. Now let us open and print location MASK:

```
MASK/ 177777        We can now change the mask.
-----
```

5-3. LOGICAL PRODUCT REVIEWED

If we take the logical product between a mask and another word, we end up with a result that saves the original value (either 0 or 1) of all bits in the word masked by 1's and clears or extracts all bits in the word masked by 0's. Suppose we wish to search on the lower eight bits in a word:

```
   Mask = 000377  = 0000000011111111
                8                   2
   Word = 073123  = 0111011001010011
                8                   2
          -------   ----------------

Logical = 000123  = 0000000001010011
Product         8                   2
```

The result of the operation is to clear out all bits in which we are not interested and to save only the lower eight bits for comparison against some 8-bit value.

5-4. SEARCH LIMITS

The following controls establish search limits:

n< Sets the lower limit to n.

n> Sets the upper limit to n.

The limiting addresses are included in the search.

5-5. EQUALITY SEARCH

n\W Searches within the limits specified for all words equal under mask to n. Remember, W means word.

Example #1: Search within locations 2000 and 3000 inclusive for all halt instructions (1020xx).

```
MASK/   177777    177700    [LF]  Change mask to ignore
-----             ------          bits 0-5

2000<3000>102000\W
------------------

2234/     102002
2337/     102002
2361/     102011
2572/     102002
2641/     102002
2710/     102002
2743/     102002
2756/     102002
```

Example #2: Search within locations 77700 and 77777 inclusive for all halt instructions in the Basic Binary Loader (BBL).

```
MASK/   177700
-----
77700<77777>102000\W
--------------------
77715/  102077
77740/  102000
77747/  102011
77751/  102055
```

5-6. INEQUALITY SEARCH

n\N Searches within the limits specified for all words under mask not equal to n. Remember, N means not equal to word.

For example: Print memory within locations 2270 and 2315 inclusive. Do not print words equal to zero.

```
MASK/   177777   [CR]        Verify mask
-----
2270<2315>0\N
-------------
2270/   50262
2271/   26303
2272/   26255
2273/   64124
2274/   6002
2275/   26246
2276/   34242
2277/   170155
2311/   70155
2312/   160156
2313/   70243
2314/   30374
2315/   170574
```

We infer that locations 2300 through 2310 contain zero.

5-7. CLEAR OR SET MEMORY

n\\Z Zero, or set memory within the limits specified to n. Remember, Z means zero.

Example #1: Set memory locations 2300 to 2310 inclusive to 177777 (octal); then print these locations to verify the operation.

```
2300<2310>177777\\Z   [CR]
------------------
2270<2310>177777\W
------------------
2300/     177777
2301/     177777
2302/     177777
2303/     177777
2304/     177777
2305/     177777
2306/     177777
2307/     177777
2310/     177777
```

Example #2: Zero locations 2300 to 2310 inclusive.

```
2300<2310>0\\Z
--------------
```

5-8. EFFECTIVE ADDRESS SEARCH

The effective address is the location an instruction must reference to acquire or store data. Many times in HP1000 programming addresses are referenced indirectly:

```
LDA   1772,I
```

The above instruction must further reference location 1772 to obtain an effective address from which to load the A-register. If the contents of 1772 = 02340, the LDA instruction would acquire data from location 02340.

n\E — Finds all instructions within the limits specified that effectively address location n. Indirect chains are followed to a depth of 16. Normally the mask is set to all 1's, but if not, n is treated as specifying a range of addresses, and all instructions effectively referencing addresses within that range are printed. Remember, E means effective address.

Example #1: Find all instructions within 2000 and 3000 inclusive that reference location 70567.

```
MASK/   177777      [CR]
-----
2000<3000>70567\E
-----------------

2050/   165775          2050 references 70567 indirectly
                        through 1775.

1775/   70567           Print location 1775 to verify
                        operation.
```

Example #2: Find all instructions within 2000 and 3000 inclusive that effectively address any location in the range 4060 to 4067.

```
MASK/   177777      177770      [CR]
-----               ------
2000<3000>4065\E
----------------
2500/   JMP         1500,I   All instructions indirectly
2550/   JMP         1503,I   reference locations in the
2600/   ADA         1503,I   range 4060 to 4067.
2700/   JMP         1503,I
\C
--
1500/   4060        [LF]     Verify by printing the
-----                        contents of the pointers.
1501/   4062        [CR]
1503/   4065
-----
```

Note the effect of the mask. By placing a 0 as the lowest octal digit of the mask, no specific address is specified for this digit position, therefore the entire range of addresses from 4060 to 4067 is processed.

BREAKPOINT/TRACE DEBUGGING

SECTION VI

6-1. INTRODUCTION

Breakpoint trace debugging is simply a matter of forcing the computer to halt at a particular point in its program execution so we can look at what is happening. We can examine operational registers, dump memory and make memory patches. Of course, we must have a means of resuming execution after a breakpoint and the ability to control the number of times we actually break within a programmed loop.

Combining memory examination and change controls, search and print controls, and binary patching with the breakpoint trace capability makes DBUGR the powerful tool that it is.

To aid in the tracing operations of a program, a breakpoint may be set at most instructions. When control reaches that instruction, it is not immediately executed; DBUGR gains control and prints out the following:

```
ADDRESS(INSTRUCTION)   A   B    EO STATUS
```

1. Program location counter (P-register) (in address mode)
2. The "broken" instruction (in instruction mode)
3. Contents of the A- and B-registers (starts in constant mode may be changed - see Section 3-7)
4. As one word the extend and overflow registers (in constant mode - see Section 3-7).

Segment breakpoints exist for segmented programs with DBUGR appended to them by the RTE-IV loader. Breakpoints may be set in four different modes:

1. ["A]\B Break at entry to ALL segments
2. ["N]\B Break at entry to NO segments
3. [xxxxx]\B Break at entry to a specific segment
4. n[xxxxx]\B Break at a defined address within a specified segment.

A breakpoint will be set when the specified segment is loaded into memory. Thereafter, breakpoints remain in effect until a new segment is loaded. At that time breakpoints associated with the newly loaded segment will be set. At such time as the previous segment is re-loaded, the breakpoints associated with that segment will be set, and the breakpoints associated with its predecessor will become dormant.

A segment entry breakpoint displayed just after segment load would look like:

```
SEGMENT  XXXXX  BREAK
ADDRESS(INSTRUCTION)   A   B   X   Y   EO STATUS
```

When a segment entry breakpoint is defined, DBUGR makes no check for the validity of the segment name. Therefore, segment names should not begin with ("A) or ("N).

6-2. CONDITIONAL BREAKPOINT

A conditional breakpoint can be set that will only break when a particular memory location is equal (or not equal) to a particular value after being masked. Location CBVAL, CBMASK, CBADDR, and CBTEST are used to set up the conditions required for the breakpoint to occur. The conditions are set up as follows:

CBVAL — Compare Value

CBMASK — Mask Value

CBADDR — Memory Address to be tested (A-register =0, B-register =1)

CBTEST — Condition required for break. For equality enter "SZA" or 2002; for inequality enter "SZA, RSS" or 2003. Note that the comparison is logical and not arithmetical.

The conditional breakpoint may be invoked only upon an existing breakpoint. Moreover, the program must be proceeding from that breakpoint. Conditions will thereafter be tested only at that particular breakpoint.

The conditional breakpoint is invoked by the double escape P command (\\P). Each time the breakpoint is encountered, the contents of the address specified by CBADDR will be XOR'ed with the contents of CBVAL. The result will be AND'ed with CBMASK and this final result will be tested with the instruction in CBTEST (SZA or SZA,RSS). The equation for this test appears in the "\M" display:

```
(CBVAL-CBADDR,I)^CBMASK = CBTEST
```

If the test instruction "skips", the conditional breakpoint will "break". If not, DBUGR will execute the instructions and continue. The conditional breakpoint may be counted with the n\\P command. In this case the conditions will be checked before the breakpoint is counted. A count is made only if a break would have been made. Therefore, n\\P will count n breaks before printing a break message.

In order to delete a breakpoint from the breakpoint table, give the \B command. Then the breakpoint table will be listed:

```
0          M+347      breakpoint in main memory
1 SEG1     77777,I    segment entry breakpoint
2 SEG2     BETA       breakpoint in SEG1
3 SEG3     PHI+5      breakpoint in SEG2
ENTER INDEX OF BP TO DELETE, A to END 1 [CR]
                                      - ----
ENTER INDEX OF BP TO DELETE, A to END A [CR]
                                      - ----
```

Breakpoint 1, the segment entry breakpoint for SEG1, was deleted.

A faster command for deleting breakpoints is \\B which deletes all breakpoints at once.

If an attempt is made to enter more breakpoints than the capacity of the breakpoint table, an error message will be displayed.

```
NO MORE ROOM FOR BREAKPOINTS
```

The user should either delete some breakpoints or create a larger breakpoint table module to be loaded with the DBUGR in the future.

6-3. RESTRICTIONS

We are, however, restricted in the instructions at which a breakpoint can be set. This is because DBUGR gains control at the breakpoint by replacing the programmed instruction with a jump to DBUGR itself. After the break has been accomplished, the instruction is either executed from its temporary location within DBUGR or execution resumes at another address we specify.

DO NOT break at the following:

1. Instruction used as a constant.

2. Instruction that is used as an address pointer in an indirect chain of an instruction.

3. Instruction that is program modified (e.g. a configured I/O instruction).

4. EIG or DMS instructions listed in Figure 6-1

CAUTION

In DBUGR, an attempt to JSB to a point below the MP fence (except in an EXEC call) as the result of a trace or proceed command causes DBUGR to reject the command and to print the break message for the violating instruction followed by "MP?". You can get around this problem with a trace. If the offending instruction is no longer a breakpoint, you can proceed. The solution then is to move the breakpoint to a point after the system call and then proceed.

Not Allowed				
DJP	DJS	SJS	SJU	UJP
UJS	XCA	XCB	XSA	XSB

Figure 6-1. Untraceable Instructions

6-4. CONTROLS

Breakpoint controls in DBUGR are:

n\B	Set a breakpoint at n. Remember, B means breakpoint.
\B	Enter remove breakpoint mode.
\\B	Remove all breakpoints.
["A]\B	Break at entry to ALL segments.
["N]\B	Break at entry of NO segments (remove segment breakpoint).
[SEG1]\B	Break at entry to a segment named SEG1.
n[SEG1]\B	Break at n within a segment named SEG1.
\P	Proceed with program execution after a break. Remember P means proceed.
\\P	Proceed with conditional breakpoint invoked.
n\P	Proceed; do not trap until n breaks from now. All breakpoints encountered contribute to the count.

\O	Change break point and trace register print mode to the current master print mode.
n\G	Go to location n; begin execution with flags and accumulators as they were at break. Remember, G means go.
n\X	Execute the instruction n, then return control to DBUGR. If the instruction is a jump, DBUGR loses control. DBUGR prints a carriage return/line-feed before and after executing the instruction. If the instruction performs a skip, DBUGR prints an additional carriage return/line-feed. Remember, X means execute.
\T	Trace one line of code. DBUGR simulates the instruction printed in the last break message and then prints a new break message with the new location, instruction and registers. Two-word instructions also print the address used by the instruction. Tracing data is unpredictable.
n\T	Trace n instructions. Causes n break messages to be printed.
\\T	Trace through a subroutine call. Unlike a \T trace of a JSB, a one-shot breakpoint is placed at "..+1". Control passes to the subroutine and is regained upon return with a break. The one-time breakpoint is thus removed. This command should not be used when the JSB is followed by an argument list. Also, it should not be used when the called subroutine may return to somewhere other than "..+1".

Example #1: Break at location 77710.

```
77710\B                              Set breakpoint at 77710.
-------

77700\G                              Begin execution at 77700.
-------

77710 (CLA)  173775   13  14 0       DBUGR prints this.

EOREG/       0                       Examine EOREG for flags.
------
```

Example #2: Breakpoint within a loop.

```
LOOP+15\B                          Set breakpoint at LOOP+15.
---------

LOOP\G                             Begin execution at LOOP.
------

LOOP+15 (CLE) 4324  17  0 41  3    DBUGR prints this.
-----

EOREG/       3                     Examine EOREG for flags.
------

10\P                               Proceed
----
LOOP +15 (CLE) 57211 5  2 24  3    DBUGR prints this
\P                                 Continue execution.
--
```

Example #3:

```
*RU,MAIN                          Execute program loaded
                                  with DBUGR
   START DBUGR
16002<M:       23456<S:    [CR]   Set start address of MAIN and SEG
--------       --------
S+5\B                             Set Breakpoint in Segment AREA
--------
\P
--
SEGMENT SEG1 BREAK
S  (0)  17542 5608 17702  22  2
S+5[SEG2]\B                       Set Breakpoint in SEG2
-----------
\P
--
SEGMENT SEG2 BREAK
S+5 (0) 17542 5606 0 45 22
M+50\B                            Set Breakpoint in MAIN
------
[SEG4]\B                          Set entry Breakpoint in SEG 4
--------
\P
--
M+50(LDA M+700) 0 2234 54 72 1    Break in MAIN
\P
--

SEGMENT SEG4 BREAK
S  (0)  17445 5562 7422 3322 1
                                  Break at segment load.
\P
--
S+10 (JSB 112,I)  24 0 177777 55 2
["N]\B                            Clear segment Breakpoint except
-------                           for specified segments.
\P
--
END DBUGR
```

Example #4:

```
:RU,PROG                                Execute program loaded
                                        with DBUGR
START DBUGR

34221\B                                 Set breakpoint
-------
\P                                      Proceed to breakpoint
--
34221(LDA 1) 43 243 0 77 1

CBVAL/  0      130[LF]                  Set compare value
------         ---
CBMASK/  177777 377[LF]                 Set mask to lower byte
                ---
CBADDR/   0      1 [LF]                 Test address (B-register)
                 -
CBTEST/   SZA    [CR]                   Test for equality

\\P                                     Proceed with conditional break
---                                     invoked

34221 (LDA 1) 32562                     23530 2372 0 0

                                        The lower byte of B-register
                                        is equal to test value.

1/23530 23400 [CR]                      Change lower byte of B-register
--      -----                           to 0

\\P                                     Proceed with conditional break
---                                     invoked
34221 (LDA 1) 177777  730 2372 0 0

\P                                      Proceed without conditions
--
34221 (LDA 1)   44     25 3372 0 0

\P
--
END DBUGR
```

6-5. MAGIC SYMBOLS

DBUGR has two magic symbols "." and "..". A magic symbol is one of DBUGR's working registers (variables) that is also in its symbol table. We already know about "." which is DBUGR's location counter. Normally "." is set to some value with one of the many commands that open a register, but it also may be set this way:

```
201 < .:
--------
.= 201    [LF]   verify "." was set
------
202/  CLA
```

".." is defined, whenever DBUGR is entered or processing a break point message, to be the next instruction's address. Thus if we have just broken:

```
3011  (CLA)   10   4010   12  77  2   ..=  3011
                                       ---
```

".." is move each trace and/or break. This provides a convenient way of opening the broken location but is even more powerful when we use it to move the trace location. Thus:

```
3011  (CLA)   10   4010   12  77  2 [CR]

..-1/  STA   3040  JMP 3022 [CR] patch previous instruction
-----        --------

.<..:  \T                        set ".." to current location
-----  --                        and trace one instruction

3022  (LDA)  3045  10  4010  12  77  2    (after the jump)
```

MORE ABOUT OPERATORS

APPENDIX A

Now that we have used most of the controls in DBUGR let's take a closer look at the use of operators (refer to Section 1-7.) within address expressions.

A-1. PLUS, BLANK AND MINUS

The minus sign (-) performs subtraction. The operators plus (+) or blank are used to perform addition. Suppose we wish to change the jump at location 2513:

```
2513/   JMP   2522   JMP 2522+3     Type jump address as a relative
-----                ----------     address and verify by
                                    reexamining 2513.
./      JMP   2525
--
```

Or, let's print the contents of SYM+5, a symbol we have defined as being equal to 2500:

```
SYM+5/   INA     Type the symbolic address and verify by printing
------           the contents of the same address expressed in
                 octal.
2505/    INA
-----
```

A-2. INCLUSIVE OR

Now we use the operator comma (,) to inclusive OR a CLA with an INA, thereby creating the combined register operation CLA, INA:

```
62572/   CLA    CLA,INA   [CR]   Change register 62572.
------           -------

.        2404                    Verify new contents in octal.
--
```

or in octal:

```
\C
--

62572/   2400   002400,002004
------          -------------
```

A-3. THE MARK \Q

Note that in the previous example we had to retype the CLA to use it in the inclusive OR expression. This is not really a lot of trouble, but we do have a mark that, when in an expression, implies the last quantity typed. Thus,

```
62572/   CLA   CLA,INA        can also be accomplished by the
------         -------        mark, EQ:

62572/   CLA   \Q,INA
------         ------

./       CLA,INA   =2404
--                 -
```

The mark \Q implies the last quantity typed, in this instance CLA. Remember that Q means last quantity typed.

For example:

```
2050/   2004   \Q+\Q  = 4010   Q+100000 = 104010
-----          -----  -        -------- -
```

A-4. USING DBUGR TO DO SIMPLE ARITHMETIC

We can use DBUGR to evaluate simple expressions, and to convert decimal or octal numbers to other radices. The following sections illustrate this capability.

A-5. OCTAL ADDITION AND SUBTRACTION

```
[CR]
234+766=1222
--------

5444-45=5377
--------

4564+567-34+125=5444
----------------
```

A-6. OTHER CONVERSIONS

```
2.\R        Change to binary output
----

23=1001     Convert octal to binary
---

16\ R       Change to hexadecimal output
-----

14=C        Convert octal to hexadecimal
----

78.=4E      Convert decimal to hexadecimal
----
```

A-7. DECIMAL ADDITION AND SUBTRACTION

```
10.\R
-----

45.+56.=101.
--------

578.+9788.=10366.
-----------

400.-45.-26.-78.=251.
-----------------
```

A-8. OCTAL TO DECIMAL CONVERSION

```
10.\R
-----

245=165.
----

256+77+654-147=562.
---------------
```

A-9. DECIMAL TO OCTAL CONVERSION

```
89.=131
----

7859.=17263
------

78.+59.=211
--------

998.-997.=1
----------
```

ERROR MESSAGES

APPENDIX B

DBUGR will recognize various types of errors. The messages and their meanings are as follows:

X	You pressed the RUB OUT key to delete a typing mistake. DBUGR will ignore any prior partial expression.
?	You used an unassigned control. Any prior expression is ignored. Input Error in special mode.
U	The symbol last used was undefined and a definition was required. The entire preceding expression is ignored.
P?	Page error: You caused a memory reference instruction to reference an address not in the current page or the base page. The expression is ignored. DBUGR's conception of the "current page" can be changed by examining any location in the desired page.
CHK	A checksum error has occurred.
MP?	DBUGR detected a possibly legal instruction that it cannot trace (or proceed with the break point at) without violating memory protect. Move the break point and proceed.
IN?	An instruction that is legal in the 21XX base set but not executable by DBUGR was detected and DBUGR cannot trace (or proceed with the breakpoint at). Move the breakpoint and proceed. User attempted to set a breakpoint on an instruction DBUGR cannot proceed from. Execution of an instruction using the (instruction)EX feature of DBUGR was attempted using a two or more word instruction (not supported - only one word instructions can be executed using this feature).
TP?	An attempt to trace, set breakpoint, or paper tape load into DBUGR. Loading, tracing, or setting of breakpoint is terminated.
O?	Symbol table overflow.
DM?	An attempt to access memory that is beyond the background partition.

DBUGR AT A GLANCE

APPENDIX C

Table C-1. Quick Reference To Frequently Used Commands

COMMAND	EXPLANATION
n/	print contents of location n
\L	list 16 locations
n\L	list n locations
n\R	set Radix to n
\S	Symbolic mode
\C	Constant mode
\H	hollerith mode
\A	address mode
n<sym:	define sym as n
n\X	execute n
n\G	Go to n
\P	proceed
\\P	proceed with conditional breakpoint
\T	trace an instruction
n\P	proceed for n breaks
n\\P	proceed with cond. breakpoint for n breaks
n\T	trace n instructions
n\B	set breakpoint at n
[x]\B	set entry breakpoint for seg x
n[x]\B	set breakpoint in seg x at n
\\B	clear all breakpoints
n1<n2>n3\W	search n1 to n2 for n3 under mask
n1<n2>n3\N	search n1 to n2 for not equal to n3 under mask
\M	display special register
\J	display map register

C-1. MODE CONTROL

DBUGR has several basic printing modes:

- As symbolic instructions. This is the mode DBUGR is in when loaded. Controls are:

  \S Set the master printing mode to symbolic instruction.

  ! Print the last quantity typed as an instruction.

  \! Set temporary printing mode to instruction, and print the last quantity typed as an instruction.

- As address pointer.

  \A Set the master printing mode to address. If only initial symbols are defined, this is equivalent to constant.

  <- Print the last quantity typed as an address pointer.

  \<- Set temporary printing mode to address, and print last quantity typed as an address.

- As constants, in a specified radix.

  \C Set the master printing mode to constants, in the current radix.

  = Print the last quantity typed as a constant.

  \= Set the temporary printing mode to constant, and print the last quantity typed as a constant.

  n\R Set the output radix to n.

- As ASCII characters in halfwords.

  \H Set the master printing mode to ASCII characters in half-words.

  ´ Print the last quantity typed as two ASCII characters. Then print ".

  \´ Set the temporary printing mode to ASCII, and print the last quantity typed as ASCII.

  s" The symbol s, of one or two characters right-adjusted, is taken as a term on input.

- To print a quantity in the current master print mode:

  ; Print last quantity typed in current mode.

- To change the breakpoint register print mode to current master print mode:

  \O

- To operate in character or block mode (multipoint terminals). Refer to Appendix D.

  \\U Switch from character to block mode or vice versa.

- To change a print mode temporarily:

  $ Same as bar (/), but set termporary print mode to symbolic instruction. Temporary mode is in effect over a series of examinations, until carriage return is typed by you, then the master mode becomes in effect again.

  \$ Same as escape bar, but set temporary print mode to symbolic instruction.

  @ Same as bar, but set temporary print mode to address pointer.

  \@ Same as escape bar, but set temporary print mode to address pointer.

  # Same as bar, but set temporary print mode to constant.

  \# Same as escape bar, but set temporary print mode to constant.

) Same as bar, but set temporary print mode to ASCII.

  \) Same as escape bar, but set temporary print mode to ASCII.

C-2. SYMBOL MANIPULATION

s: Define the symbol s as having value specified by the location counter.

s& Define symbol s as having value equal to the address of the last quantity typed, if an instruction.

\K Kill all symbol definitions other than the initial symbol table.

C-3. LOCATION EXAMINATION

n/ Print the contents of location n, in the current master print mode, and open the location for possible modification. The location counter is set to n.

/ Open and print the contents of the location pointed to by the last quantity typed, taken as a direct 15-bit address. The location counter is not changed.

\/	Open and print the contents of the location pointed to by the last quantity typed, taken as an memory reference address instruction. The location counter is not changed.
\L	Open and print the contents of the next 16 sequential locations starting with the current location. The location counter is advanced by 16.
n\L	Same as \L, except list n lines.
[CR]	Close any open location. No change is made.
n[CR]	Store the quantity n in the open location, if any.
[LF]	Open and print the contents of the next sequential location as determined by the location counter. The location counter is advanced by one.
n[LF]	Store the quantity n in the open location, if any; then open and print the contents of the next sequential location. The location counter is advanced by one.
^	Open and print the contents of the previous sequential register. The location counter is decremented by one.
n^	Store the quantity n in the open location, if any; then open and print the contents of the previous sequential location. The location counter is decremented by one.
CTRL-I (TAB)	Open, set the location counter to, and print the contents of the location pointed to by the quantity typed, taken as an address pointer.
nCTRL-I (nTAB)	Store the quantity n in the open location, if any. Then open, set the loction counter to, and print the contents of the location pointed to by n, taken as an address.
\TAB	Open, set the location counter to, and print the contents of the location pointed to by the last quantity typed, taken as an instruction.
n\TAB	Store the quantity n in the open location, if any. Then open, set the location counter to, and print the contents of the location pointed by n, taken as an instruction.
n%	Store the quantity n in the open location, if any. Then open and print the contents of the location pointed to by n, taken as an address. The location counter is not changed.
n\%	Store the quantity n in the open location, if any. Then open and print the contents of the location pointed to by n, taken as an instruction. The location counter is not changed.

C-4. PROGRAM LOAD AND VERIFY

\Y	Load absolute binary paper tape, as prepared by HP Assembler.
\V	Verify paper tape against core.

C-5. PUNCHING

n\D	Store n into the open location, if any. Then punch the last register (with the new contents n, if stored).
n1<n2>\D	Punch locations n1 to n2, inclusive.

C-6. MEMORY SEARCH AND CLEAR

n<	Set the lower limit for search or clear to n.
n>	Set the upper limit for search or clear to n.
n1<n2>n3\W	Search between the limits n1 and n2 for all words equal under mask to n3.
n1<n2>n3\N	Search between the limits n1 and n2 for all words not equal under mask to n3.
n1<n2>n3\E	Effective address search: find all instructions between the limits n1 and n2 which, under mask, effectively address location n3.
n1<n2>n3\\Z	Zero, or clear core between the limits n1 and n2, to n3. If n3 is omitted, it is taken as zero.
\MASK	Examine/modify search mask.

C-7. BREAKPOINTS AND PROGRAM CONTROL

n\B	Set a breakpoint at n. If n is in segment space, use last segment loaded as the associated segment name. Error if no segment loaded yet.
\B	Enter remove breakpoint mode.
\\B	Remove all breakpoints
n[NAME]\B	Set a breakpoint in segment NAME at location n.
[NAME]\B	Break at entry to Segment NAME.
["A]\B	Break at entry to ALL Segments.

["N]\B	Break at entry to NO Segments except for those specified.
\P	Proceed with program execution after a break trap.
\\P	Proceed with conditional breakpoint invoked.
n\P	Proceed; do not trap until n breakpoints from now.
n\\P	Proceed; do not trap until n breakpoints, including conditional breakpoints.
n\G	Go to location n; begin execution with flags and accumulators as saved.
n\X	Execute the instruction n, then return control to DBUGR.
\T	Trace one instruction.
n\T	Trace n instructions.
\\T	Trace an entire subroutine call with no argument list or alternate returns.

Trace/Breakpoint simulation:

DBUGR simulates instructions when it:

a) Proceeds from the current breakpoint location

b) Traces

c) Executes an instruction

DBUGR will correctly simulate all instructions including EXEC calls in its symbol table when tracing, with the exceptions noted in Figure 6-1. A breakpoint may not be set on any of these instructions.

A method to get around instruction exceptions if to set up a subroutine containing the desired instruction(s) and execute a JSB <subroutine>\X. DBUGR will give up control to the subroutine and execute it, regaining control on the return.

C-8. SPECIAL REGISTERS

```
\M        Display contents of the special registers.
AREG/     Examine & Modify A-Register
BREG/     Examine & Modify B-Register
XREG/     Examine & Modify X-Register
YREG/     Examine & Modify Y-Register
EOREG/    Examine & Modify EO-Register
MASK/     Examine & Modify Search Mask
CBVAL/    Examine & Modify conditional breakpoint value
CBMASK/   Examine & Modify conditional breakpoint mask
CBADDR/   Examine & Modify conditional breakpoint address
CBTEST/   Examine & Modify conditional breakpoint test
WRTLU/    EXEC control word for DBUGR output device
BRFLG/    Break Flag; 0= check for break, 1= no check
```

C-9. MAP REGISTERS

```
\J:  SM  Examine system maps.
     UM  Examine user maps.
     PA  Examine port A maps.
     PB  Examine port B maps.
     XL  Cross load A- or B-registers.
\?   Display MEM status register.
```

APPENDIX D

APPENDIX
D

DBUGR can be operated in block mode. Multipoint (DVR07) terminals run in block mode only.

Escape, return and line feed cannot be used in block mode.

The following are the differences between character and block mode.

CHAR MODE	BLOCK MODE
<escape>	\
<cr>	]
<lf>	<enter for multipoint>
	<return for non-multipoint>

In order to invoke block mode operation on a non-multipoint terminal enter the following command:

```
\\U
```

When in block mode this command will set DBUGR back to character mode for non-multipoint terminals.

Since DBUGR does not receive any character until the enter or return is entered, a series of commands may be entered on one line.

For example:

```
200/201/202/<enter>
-------------------
   JSB 4,I   203   245
200/<enter>
-----------
   JSB 4,I   <enter>
             -------
201/ 203     <enter>
             -------
202/ 245     ]<enter>
             --------
```

MANUAL PART NO. 92067-90005
Printed in U.S.A. February 1980

HEWLETT-PACKARD COMPANY
Data Systems Division
11000 Wolfe Road
Cupertino, California 95014