COPYRIGHT

Copyright © 1978 and 1981 by Digital Research. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual or otherwise, without the prior written permission of Digital Research, Post Office Box 579, Pacific Grove, California, 93950.

This manual is, however, tutorial in nature. Thus, the reader is granted permission to include the example programs, either in whole or in part, in his own programs.

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CP/M is a registered trademark of Digital Research. ASM, DDT, MAC and SID are trademarks of Digital Research. Intel is a registered trademark of Intel Corporation.

The "SID User's Guide" was prepared using the Digital Research TEX Text Formatter and printed in the United States of America by Commercial Press/Monterey.
Foreword

SID, T.M. the CP/M® symbolic debugger, expands upon the features of the CP/M standard debugger described in the "CP/M Dynamic Debugging Tool (DDT) User's Guide" and provides greatly enhanced facilities for assembly level program checkout. Specifically, SID includes real-time breakpoints, fully monitored execution, symbolic disassembly, assembly, and memory display and fill functions. Further, SID operates with "utilities" that can be dynamically loaded with SID to provide traceback and histogram facilities.

Section 1 of this manual describes the command forms that initiate SID and the command lines that direct the actions of the SID program. Section 2 describes SID's ability to reference absolute machine addresses through symbolic expressions. Section 3 describes the commands that direct the debugging process. The SID utilities, described in Section 4, provide additional debugging facilities. Section 5 contains several examples of SID debugging sessions.
Table of Contents

1 SID Operation Under CP/M
   1.1 Starting SID .............................. 1
   1.2 SID Command Input .......................... 5

2 SID Symbolic Expressions
   2.1 Literal Hexadecimal Numbers ................ 9
   2.2 Literal Decimal Numbers .................... 9
   2.3 Literal Character Values ................... 10
   2.4 Symbolic References ...................... 11
   2.5 Qualified Symbols .......................... 11
   2.6 Symbolic Operators ....................... 12
   2.7 Sample Symbolic Expressions ............... 13

3 SID Commands
   3.1 The Assemble (A) Command ................ 15
   3.2 The Call (C) Command ..................... 17
   3.3 The Display Memory (D) Command .......... 17
   3.4 The Fill Memory (F) Command ............. 20
   3.5 The Go (G) Command ...................... 20
   3.6 The Hexadecimal Value (H) Command ...... 22
   3.7 The Input Line (I) Command ............. 23
   3.8 The List Code (L) Command .............. 27
   3.9 The Move Memory (M) Command ............ 28
   3.10 The Pass Counter (P) Command .......... 28
   3.11 The Read Code/Symbols (R) Command ..... 31
   3.12 The Set Memory (S) Command ............ 35
   3.13 The Trace Mode (T) Command ............ 36
Table of Contents (continued)

3.14 The Untrace Mode (U) Command .................. 39
3.15 The Examine CPU State (X) Command ............. 40

4 SID Utilities

4.1 Utility Operation .................................. 43
4.2 The HIST Utility ................................... 44
4.3 The TRACE Utility ................................ 46

5 SID Sample Debugging Sessions ....................... 51
Section 1
SID Operation Under CP/M

1.1 Starting SID

Type one of the following commands to start the SID program.

(a) SID
(b) SID x.y
(c) SID x.HEX
(d) SID x.UTL
(e) SID x.y u.v
(f) SID * u.v

In each case, SID loads into the Transient Program Area (TPA) and relocates itself to the top of the TPA, overlaying the Console Command Processor portion of CP/M. Figure 1-1 shows memory organization before SID is loaded while Figure 1-2 shows the memory configuration after SID is loaded and relocated. Due to the relocation process, SID is independent of the exact memory size that CP/M manages in a particular computer configuration.

![Figure 1-1. Memory Configuration Before SID Loads](image)

All Information Presented Here is Proprietary to Digital Research
After loading and relocating, SID alters the BDOS entry address to reflect the reduced memory size, as shown in Figure 1-2, and frees the lower portion of the TPA for use by the program under test. Note that although SID occupies only 6K of upper memory when operating, the self-relocation process necessitates a minimum 20K CP/M system for initial setup, leaving about 10K for the test program.

Command form (a) above loads and executes SID without loading a test program into the TPA. Use this form to examine memory or write and test simple programs using the built-in assembly features of SID.

Form (b) above is similar to (a) except that the file given by x.y is automatically loaded for subsequent test. Note that although x.y is loaded into the TPA, it is not executed until SID passes program control to the program under test using one of the following commands: C (Call), G (Go), T (Trace), or U (Untrace). It is your responsibility to ensure that there is enough space in the TPA to hold the test program as well as the debugger. If the program x.y does not exist on the diskette or cannot be loaded, SID issues the standard "?" error response. If no load error occurs, SID responds as follows:

```
NEXT  PC  END
nnnn pppp eeee
```

where nnnn, pppp, and eeee are hexadecimal values that indicate the next free address following the loaded program, the initial value of the program counter, and the logical end of the TPA, respectively. Thus, nnnn is normally the beginning of the data area of the program under test; pppp is the starting program counter (set to the beginning of the TPA), and eeee is the last memory location available to the test program, as shown in Figure 1-3. Although x.y usually contains machine code, the operator can name an ASCII file, in which case these program addresses are less meaningful.

All Information Presented Here is Proprietary to Digital Research
Command form (c) is similar to form (b) except that the test program is assumed to be in Intel "hex" format, as directly produced by ASM or MAC. In this case, the initial value of the program counter is obtained from the terminating record of the hex file unless this value is zero, in which case the program counter is set to the beginning of the TPA. As the ASM and MAC manuals discuss, the program counter value can be given on the "END" statement in the source program. Again, it is your responsibility to ensure that the hex records do not overlay portions of the SID debugger or CP/M Operating System. If the hex file does not exist or if errors occur in the hex format, SID issues the "?" response. Otherwise, the principle program locations shown in the previous paragraph are listed at the console.

Use command form (d) when a SID utility function is to be included. In this case, SID is first loaded and relocated as above. The utility function is then loaded into the TPA. Utility functions are also self-relocating and immediately move to the top of the TPA, placing themselves directly below the SID program. The BDOS entry address is changed to reflect the reduced TPA, as shown in Figure 1-4. Generally, the utility program prints sign-on information and may or may not prompt for input from the console. Exact details of utility operation are given in Section 4, "SID Utilities."
Command form (e) is similar to (c), except that the symbol table given by u.v is loaded with the program x.y. Symbol information is loaded from the current top of the TPA downward toward the program under test, as shown in Figure 1-5.

The symbol table is in the format produced by the CP/M Macro Assembler. In particular, the symbol table must be a sequence of address and symbol name pairs, where the address consists of four hexadecimal digits, separated by a space from the symbol that takes on this address value. The symbol consists of up to 15 graphic ASCII characters terminated by one or more tabs (\|) or a carriage-

All Information Presented Here is Proprietary to Digital Research
return line-feed sequence. Note that you can create or alter a symbol table using the CP/M editor, as long as this format is followed.

The response following program load is as shown in command form (b) above, giving essential program locations. When SID begins symbol load, it displays the following message:

SYMBOLS

This message indicates that any subsequent error is due to the symbol load process. In particular, the "?" error following the SYMBOLS response is due to a non-existent or incorrectly formatted symbol file.

Command form (f) is similar to (e), except that no program is loaded with the symbol file u.v.

Examples of typical commands that start the SID program are shown below.

(a) SID
(b) SID DUMP.COM
(b) SID DUMP.ASM
(c) SID SAMPLE.HEX
(c) SID DUMP.HEX
(d) SID TRACE.UTL
(d) SID HIST.UTL
(e) SID DUMP.COM DUMP.SYM
(e) SID DUMP.HEX DUMP.SYM
(e) SID TEST.COM TEST.ZOT
(f) SID * DUMP.SYM

1.2 SID Command Input

Command input to SID consists of a series of "command lines" that direct the actions of the SID program. These commands allow display of memory and CPU registers, and direct the execution and breakpoint operations during test program debugging.

When SID is ready to accept the next command, it displays a "#" at the console. Each command is based upon a single letter, followed by optional parameters, and terminated by a carriage return. Note that all standard line editing features of CP/M are available, with a maximum of 64 command characters. The following table lists the CP/M line editing functions.
Table 1-1. CP/M Line Editing Controls

<table>
<thead>
<tr>
<th>Control Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑C</td>
<td>CP/M system reboot, return to CCP</td>
</tr>
<tr>
<td>↑E</td>
<td>Physical end-of-line</td>
</tr>
<tr>
<td>↑H</td>
<td>Delete last character and backspace cursor</td>
</tr>
<tr>
<td>↑P</td>
<td>Print console output (on/off toggle)</td>
</tr>
<tr>
<td>↑R</td>
<td>Retype current input line</td>
</tr>
<tr>
<td>↑S</td>
<td>Stop/start console output</td>
</tr>
<tr>
<td>↑U</td>
<td>Delete current input line</td>
</tr>
<tr>
<td>↑X</td>
<td>(Same as ↑U)</td>
</tr>
<tr>
<td>Rubout</td>
<td>Delete and echo last character</td>
</tr>
</tbody>
</table>

The ↑ character indicates that you must simultaneously hold down the control key while depressing the particular function key. Note that the ↑R, ↑U, and ↑X keys cause CP/M to type a "#" at the end of the line to indicate that the line is being discarded.

Various SID commands produce long typeouts at the console (see the "D" command which displays memory, for example). In this case, you can abort the typeout before it completes by typing any key at the console (a "return" suffices).

The single letter commands that direct the actions of SID are typed at the beginning of the command line. You can enter commands in upper- or lower-case. Table 1-2 summarizes the valid commands.
Table 1-2. Command Letters

<table>
<thead>
<tr>
<th>Letter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Assemble directly to memory</td>
</tr>
<tr>
<td>C</td>
<td>Call to memory location from SID</td>
</tr>
<tr>
<td>D</td>
<td>Display memory in hex and ASCII</td>
</tr>
<tr>
<td>F</td>
<td>Fill memory with constant value</td>
</tr>
<tr>
<td>G</td>
<td>Go to test program for execution</td>
</tr>
<tr>
<td>H</td>
<td>Hexadecimal arithmetic</td>
</tr>
<tr>
<td>I</td>
<td>Input CCP command line</td>
</tr>
<tr>
<td>L</td>
<td>List 8080 mnemonic instructions</td>
</tr>
<tr>
<td>M</td>
<td>Move memory block</td>
</tr>
<tr>
<td>P</td>
<td>Pass point set, reset, and display</td>
</tr>
<tr>
<td>R</td>
<td>Read test program and symbol table</td>
</tr>
<tr>
<td>S</td>
<td>Set memory to data values</td>
</tr>
<tr>
<td>T</td>
<td>Trace test program execution</td>
</tr>
<tr>
<td>U</td>
<td>Untrace (monitor) test program</td>
</tr>
<tr>
<td>X</td>
<td>Examine state of CPU registers</td>
</tr>
</tbody>
</table>

Although the details of each of the commands are given in later sections, nearly all of the commands accept parameters following the letter that governs the command actions. The parameters can be counters or memory addresses, and can appear in both literal and symbolic form, but eventually reduce to values in the range 0-65535 (four hexadecimal digits).

As an example, the "display memory" command can take the following form:

```
Dssss,eee
```

where D is the command letter, and ssss and eeee are "command parameters" that give the starting and ending addresses for the display, respectively. In their simplest form, ssss and eeee can be literal hexadecimal values, as shown below.

```
D100,300
```

These values instruct SID to print the hexadecimal and ASCII values contained in memory locations 0100H through 0300H.

Although you can usually refer to program listings to obtain absolute machine addresses, SID supports more comprehensive mechanisms for quick access to machine addresses through program symbols. In particular, the command parameters can consist of "symbolic expressions" which are described fully in the following section.
Section 2
SID Symbolic Expressions

An important facility of SID is the ability to reference absolute machine addresses through symbolic expressions. Symbolic expressions can involve names obtained from the program under test that are included in the "SYM" file produced by the CP/M Macro Assembler. Symbolic expressions can also consist of literal values in hexadecimal, decimal, or ASCII character string form. These values can then be combined with various operators to provide access to subscripted and indirectly addressed data or program areas. This section describes symbolic expressions so that you can incorporate them as command parameters in the individual command forms that follow this section.

2.1 Literal Hexadecimal Numbers

SID normally accepts and displays values in hexadecimal. The valid hexadecimal digits consist of the decimal digits 0 through 9 along with the hexadecimal digits A, B, C, D, E, and F, corresponding to the decimal values 10 through 15, respectively.

A literal hexadecimal number in SID consists of one or more contiguous hexadecimal digits. If you type four digits, then the leftmost digit is most significant, while the rightmost digit is least significant. If the number contains more than four digits, the rightmost four are taken as significant, and the remaining leftmost digits are discarded. The examples below show the corresponding hexadecimal and decimal values for the given input values.

<table>
<thead>
<tr>
<th>INPUT VALUE</th>
<th>HEXADECIMAL</th>
<th>DECIMAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>0100</td>
<td>256</td>
</tr>
<tr>
<td>ffef</td>
<td>FFFE</td>
<td>65534</td>
</tr>
<tr>
<td>10000</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>38001</td>
<td>8001</td>
<td>32769</td>
</tr>
</tbody>
</table>

2.2 Literal Decimal Numbers

Although SID's normal number base is hexadecimal, you can override this base on input by preceding the number with a "#" symbol, which indicates that the following number is in the decimal base. In this case, the number that follows must consist of one or more decimal digits (0 through 9) with the most significant digit on the left and the least significant digit on the right. Decimal values are padded or truncated according to the rules of hexadecimal numbers, as described above, by converting the decimal number to the equivalent hexadecimal value.

All Information Presented Here is Proprietary to Digital Research
The input values shown to the left below produce the internal hexadecimal values shown to the right below:

<table>
<thead>
<tr>
<th>INPUT VALUE</th>
<th>HEXADECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>#9</td>
<td>0009</td>
</tr>
<tr>
<td>#10</td>
<td>000A</td>
</tr>
<tr>
<td>#256</td>
<td>0100</td>
</tr>
<tr>
<td>#65535</td>
<td>FFFF</td>
</tr>
<tr>
<td>#65545</td>
<td>0009</td>
</tr>
</tbody>
</table>

### 2.3 Literal Character Values

As an operator convenience, SID also accepts one or more graphic ASCII characters enclosed in string apostrophes (') as literal values in expressions. Characters remain as typed within the paired apostrophes (i.e., no case translation occurs) with the leftmost character treated as the most significant, and the rightmost character treated as least significant. Similar to hexadecimal numbers, character strings of length one are padded on the left with zero, while strings of length greater than two are truncated to the rightmost two characters, discarding the leftmost remaining characters.

Note that the enclosing apostrophes are not included in the character string, nor are they included in the character count, with one exception. To include the possibility of writing strings that include apostrophes, a pair of contiguous apostrophes is reduced to a single apostrophe and included in the string as a normal graphic character.

The strings shown to the left below produce the hexadecimal values shown to the right below. (For these examples, note that upper-case ASCII alphabets begin at the encoded hexadecimal value 41, lower-case alphabets begin at 61, a space is hexadecimal 20, and an apostrophe is encoded as hexadecimal 27).

<table>
<thead>
<tr>
<th>INPUT STRING</th>
<th>HEXADECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>'A'</td>
<td>0041</td>
</tr>
<tr>
<td>'AB'</td>
<td>4142</td>
</tr>
<tr>
<td>'ABC'</td>
<td>4243</td>
</tr>
<tr>
<td>'aA'</td>
<td>6141</td>
</tr>
<tr>
<td>'ABC'</td>
<td>0027</td>
</tr>
<tr>
<td>'A'</td>
<td>2727</td>
</tr>
<tr>
<td>' A'</td>
<td>2041</td>
</tr>
<tr>
<td>'A '</td>
<td>4120</td>
</tr>
</tbody>
</table>
2.4 Symbolic References

Given that a symbol table is present during a SID debugging session, you can reference values associated with symbols through the following three forms of a symbol reference:

(a) .s
(b) @s
(c) =s

where s represents a sequence of one to fifteen characters that match a symbol in the table.

Form (a) produces the address value (i.e., the value associated with the symbol in the table) corresponding to the symbol s. Form (b) produces the 16-bit "word" value contained in the two memory locations given by .s, while form (c) results in the 8-bit "byte" value at .s in memory. Suppose, for example, that the input symbol table contains two symbols, and appears as follows:

0100 GAMMA 0102 DELTA

Further, suppose that memory starting at 0100 contains the following byte data values:

0100: 02
0101: 3E
0102: 4D
0103: 22

Based upon this symbol table and these memory values, the symbol references shown to the left below produce the hexadecimal values shown to the right below. Recall that 16-bit 8080 memory values are stored with the least significant byte first, and thus the word values at 0100 and 0102 are 3E02 and 224D, respectively.

<table>
<thead>
<tr>
<th>SYMBOL REFERENCE</th>
<th>HEXADECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>.GAMMA</td>
<td>0100</td>
</tr>
<tr>
<td>.DELTA</td>
<td>0102</td>
</tr>
<tr>
<td>@GAMMA</td>
<td>3E02</td>
</tr>
<tr>
<td>@DELTA</td>
<td>224D</td>
</tr>
<tr>
<td>=GAMMA</td>
<td>0002</td>
</tr>
<tr>
<td>=DELTA</td>
<td>004D</td>
</tr>
</tbody>
</table>

2.5 Qualified Symbols

Note that duplicate symbols can occur in the symbol table due to separately assembled or compiled modules that independently use the same name for differing subroutines or data areas. Further, block structured languages, such as PL/M, allow nested name definitions that are identical, but non-conflicting. Thus, SID allows reference to "qualified symbols" that take the form:

All Information Presented Here is Proprietary to Digital Research
2.5 Qualified Symbols

S1/S2/ ... /Sn

where S1 through Sn represent symbols that are present in the table during a particular session.

SID always searches the symbol table from the first to last symbol, in the order the symbols appear in the symbol file. For a qualified symbol, SID begins by matching the first S1 symbol, then scans for a match with symbol S2, continuing until symbol Sn is matched. If this search and match procedure is not successful, SID prints the "?" response to the console. Suppose, for example, that the symbol table appears as follows:

0100 A 0300 B 0200 A 3E00 C 20F0 A 0102 A

in the symbol file, with memory initialized as shown in the previous section. The unqualified and qualified symbol references shown to the left below produce the hexadecimal values shown to the right below.

<table>
<thead>
<tr>
<th>SYMBOL REFERENCE</th>
<th>HEXADECIMAL VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>.A</td>
<td>0100</td>
</tr>
<tr>
<td>@A</td>
<td>3E02</td>
</tr>
<tr>
<td>.A/A</td>
<td>0200</td>
</tr>
<tr>
<td>.C/A/A</td>
<td>0102</td>
</tr>
<tr>
<td>=C/A/A</td>
<td>004D</td>
</tr>
<tr>
<td>@B/A/A</td>
<td>20F0</td>
</tr>
</tbody>
</table>

2.6 Symbolic Operators

Literal numbers, strings, and symbol references can be combined into symbolic expressions using unary and binary "+" and "-" operators. The entire sequence of numbers, symbols, and operators must be written without embedded blanks. Further, the sequence is evaluated from left to right, producing a four digit hexadecimal value at each step in the evaluation. Overflow and underflow are both ignored as the evaluation proceeds. The final value becomes the command parameter, whose interpretation depends upon the particular command letter that precedes it.

When placed between two operands, the "+" indicates addition of the second operand to the previously accumulated value. The sum becomes the new accumulated value to this point in the evaluation. If the expression begins with a unary "+", then the immediately preceding (completed) symbolic expression is taken as the initial accumulated value (zero is assumed at SID startup). For example, the command:

DFE00+#128,+5

contains the first expression "FE00+#128" which adds FE00 and
(decimal) 128 to produce FE80 as the starting value for this display command. The second expression "+5" begins with a unary "+" which indicates that the previous expression value (FE80) is to be used as the base for this symbolic expression, producing the value FE85 for the end of the display operation. Thus, the command given above is equivalent to:

DFE80,FE85

The "-" symbol causes SID to subtract the literal number or symbol reference from the 16-bit value accumulated thus far in the symbolic expression. If the expression begins with a minus sign, then the initial accumulated value is taken as zero. That is,

-x is computed as 0-x

where x is any valid symbolic expression. For example, the following command:

DFF00-200,-512

is equivalent to the simple command:

DFD00,FE00

A special up-arrow operator, denoted by "^", denotes the top-of-stack in the program under test. In general, a sequence of n up-arrow operators extracts the n-th stacked item in the test program, but does not change the test program stack content or stack pointer. This particular operator is used most often in conjunction with the G (Go) command to set a breakpoint at a return from a subroutine during test, and is described fully under the G command.

2.7 Sample Symbolic Expressions

The formulation of SID symbolic expressions is most often closely related to the program structures in the program under test. Suppose you want to debug a sorting program that contains the data items listed below:

LIST: names the base of a table of byte values to sort, assuming there are no more than 255 elements, denoted by LIST(0), LIST(1), ..., LIST(254).

N: is a byte variable which gives the actual number of items in LIST, where the value of N is less than 256. The items to sort are stored in LIST(0) through LIST(N-1).
I: is the byte subscript which indicates the next item to compare in the sorting process. That is, LIST(I) is the next item to place in sequence, where \( I \) is in the range 0 through \( N-1 \).

Given these data areas, the command

\[
\text{D.LIST,+#254}
\]

displays the entire area reserved for sorting:

\[
\text{LIST(0)}, \text{LIST(1)}, \ldots, \text{LIST(254)}
\]

The command

\[
\text{D.LIST,+=I}
\]

displays the LIST vector up to and including the next item to sort:

\[
\text{LIST(0)}, \text{LIST(1)}, \ldots, \text{LIST(I)}
\]

The command

\[
\text{D.LIST+|=I,+0}
\]

displays only LIST(I). Finally, the command

\[
\text{D.LIST,+=N-1}
\]

displays only the area of LIST that holds active items to sort:

\[
\text{LIST(0)}, \text{LIST(1)}, \ldots, \text{LIST(N-1)}
\]

The exact manner in which SID uses symbolic expressions depends upon the individual command that you issue. The following section details these commands.
Section 3
SID Commands

Enter SID commands at the console following the "#" prompt. The commands direct the debugging process by allowing alteration and display of CPU registers and memory as well as the controlling execution of the program under test.

The following sections describe the commands that SID accepts.

3.1 The Assemble (A) Command

The A command allows you to insert 8080 machine code and operands into the current memory image using standard Intel mnemonics, along with symbolic references to operands. The A command takes the forms:

(a) As
(b) A
(c) -A

where s represents any valid symbolic expression. Form (a) begins inline assembly at the address given by s, where each successive address is displayed until you type a null line (i.e., a single carriage return). Form (b) is equivalent to (a), except the starting address for the assembly is taken from the last assembled, listed, or traced address (see the "L", "T", and "U" commands). The following command sequence, for example, assembles a short program into the Transient Program Area (note that you must terminate each command line with a carriage return):

A100
0100 MVI A,10
0102 DCR A
0103 JNZ 102
0106 RST 7
0107

begin assembly at 0100
load A with hex 10
decrement A register
loop until zero
return to debugger
single carriage return

As each successive address is prompted, you can either enter a mnemonic instruction or return to SID command mode by entering a single carriage return (a single "." is also accepted to terminate inline assembly to be consistent with the "S" command).

Delimiter characters that are acceptable between mnemonic and operand fields include space or tab sequences.

Invalid mnemonics or ill-formed operand fields produce "?" errors. In this case, control returns back to command mode, where you can proceed with another command line, or simply return to assembly mode by typing a single "A", since the assumed starting
address is automatically taken from the last assembled address.

The assembler/disassembler portion of SID is a separate module, and can be removed to increase the available debugging space. Thus, form (c) is entered to remove the module, returning approximately 1 1/2 K bytes. Since the entire SID debugger requires approximately 6 K bytes, this reduces SID requirements to about 4 1/2 K bytes. When the assembler/disassembler module is removed in this manner, the A and L commands are effectively removed. Further, the trace and untrace functions display only the hexadecimal codes, and the traceback utility displays only hexadecimal addresses. Any existing symbol information is also discarded at this point, although such information can be reloaded (see the "I" and "R" commands).

Examples of valid assemble commands are shown below:

A100
A#100
A.CRLF+5
A@GAMMA+@X=-I
A+30

Given that the command A100 has been entered, the following interaction could take place between SID and the operator:

<table>
<thead>
<tr>
<th>SID PROMPT</th>
<th>OPERATOR INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>MVI C,.A-.B</td>
</tr>
<tr>
<td>0102</td>
<td>LXI H,.SOURCE</td>
</tr>
<tr>
<td>0105</td>
<td>LXI D,+100</td>
</tr>
<tr>
<td>0108</td>
<td>MOV A,M</td>
</tr>
<tr>
<td>0109</td>
<td>INX H</td>
</tr>
<tr>
<td>010A</td>
<td>STAX D</td>
</tr>
<tr>
<td>010B</td>
<td>INX D</td>
</tr>
<tr>
<td>010C</td>
<td>DCR C</td>
</tr>
<tr>
<td>010D</td>
<td>JNZ 108</td>
</tr>
<tr>
<td>0110</td>
<td>(&quot;return&quot; only)</td>
</tr>
</tbody>
</table>

A, B, and SOURCE are symbols that appear in the symbol table. In this case, SID computes the address difference between A and B as the operand for the MVI instruction. The LXI H operand becomes the address of SOURCE, while the LXI D instruction receives the operand value .SOURCE+100 because .SOURCE was the immediately preceding symbolic expression value. This particular program segment moves a block of memory determined by the address values of the corresponding symbols.
3.2 The Call (C) Command

The C command performs a call to an absolute location in memory, without disturbing the register state of the program under test. The C Command takes the forms:

(a) Cs
(b) Cs,b
(c) Cs,b,d

Although the C command is designed for use with SID utilities, it can call on test program subroutines to perform program initialization, or to make CP/M BDOS calls that initialize various system parameters before executing the test program.

Form (a) above performs a call on absolute location s, where s is a symbolic expression. In this case, registers BC = 0000 and DE = 0000 in the call. Normal exit from the subroutine is through execution of a RET instruction that returns control to SID, followed by the normal SID prompt.

Form (b) above is equivalent to (a), except that the BC register pair is set to the value of expression b, while DE is set to 0000.

Form (c) is similar to (b); the BC register pair is set to the value b while the DE pair is set to the value of d. Several examples of valid C commands are shown below. Refer also to the SID utility discussion for examples of the C command in utility initialization, data collection, and display functions.

C100
C#4096
C.DISPLAY
C@JMPVEC+=X
C.CRLF,#34
C.CRLF,@X,+=X

3.3 The Display Memory (D) Command

The D command displays selected segments of memory in both byte (8-bit) and word (16-bit) formats. The display appears in both hexadecimal and ASCII form in the output. The D command takes the following forms:

(a) Ds
(b) Ds,f
(c) D
(d) D,f
(e) DWs
(f) DWs,f
(g) DW
(h) DW,f
Forms (a) through (d) display memory in byte format, while forms (e) through (h) display memory in word format. The byte format display appears as:

```
aaaa bb bb bb . . . bb cc . . . cc
```

where aaaa is the base address of the display line and the sequence of (up to) 16 bb pairs represents the hexadecimal values of the data stored starting at address aaaa. The sequence of c's represent the same data area displayed in ASCII format, where possible. A period (.) is displayed as a place holder when the data item does not correspond to a graphic character.

Byte mode displays are "normalized" to address boundaries that are multiples of 16. That is, if the starting address aaaa is not a multiple of 16, then the display line is printed to the next boundary address that is a multiple of 16. Each display line that follows contains 16 data elements until the last display line is encountered.

Command forms (e) through (h) display in word mode which is similar to the byte mode display described above, except that the data elements are printed in a double byte format:

```
aaaa wwww wwww . . . wwww cc . . . cc
```

where aaaa is the starting address for the display line and the sequence of (up to 8) wwww's represent the data items that are stored in memory beginning at aaaa. Similar to the byte mode display, the sequence of c's represent the decoded ASCII characters starting at address aaaa. As in the byte mode display, a period is displayed as a place holder when the character in that position is non-graphic.

Contrary to the byte mode display, address normalization to modulo 16 address boundaries does not occur in the word mode display. Recall that 8080 double words are stored with the least significant byte first, and thus the word mode display reverses each byte pair so that the individual data items are displayed as four digit hexadecimal numbers with the most significant digits in the high-order positions.

Command form (a) displays memory in byte format starting at location s for 1/2 of a standard CRT screen (12 lines). This form of the command is useful when you want to view a segment of memory beginning at a particular position with an indefinite ending address.

Command form (b) is similar to (a), but specifies a particular ending address. In this case, the start address is taken as s with the display continuing through address f. Recall that you can abort excessively long timeouts by depressing any keyboard character, such as a carriage return.
Form (c) is similar to (a) and (b), except the starting address for the display is taken from the last displayed address, or from the value of the memory address registers (HL) in the case that no previous display has occurred since the last breakpoint. It is often convenient, for example, to use form (a) to display a segment of memory, followed by a sequence of D commands of form (c) to continue the display. Each D command displays another 1/2 screen of memory.

Command form (d) is similar to (b) except the starting address is taken automatically as described in form (c) above.

Assume, for example, that decimal values 1 through 255 are stored in memory starting at hexadecimal address 0100. The command:

```
D100,12A
```

produces the expanded form of the display shown below:

```
0100 01 02 03 04 (etc.) 0E 0F 10 .. (etc.) ..
0110 11 12 13 14 (etc.) 1E 1F 20 .. (etc.) ..
0120 21 22 23 24 (etc.) 29 2A 2B !$%&'()++
```

Command forms (e) through (h) parallel the byte display formats given by (a) through (d), except that the display is given in word format. Form (e) displays in word format from location s for 1/2 screen, while form (f) displays from location s through location f. Form (g) displays from the last display location, or from HL if there has been an immediately preceding breakpoint with no intervening display. Form (h) is similar to (g), but displays through location f. The command:

```
DW100,128
```

for example, produces the expanded form of the following output lines:

```
0100 0201 0403 (etc.) 0E0D 100F .. (etc.) ..
0110 1211 1413 (etc.) 1E1D 201F .. (etc.) ..
0120 2221 2423 (etc.) 2928 2B2A !$%&'()++
```

The following are examples of valid D commands:

```
DF3F
D#100,#200
D.GAMMA,.DELTA+#30
D.GAMMA
DW@ALPHA,+#100
```
3.4 The Fill Memory (F) Command

The F command fills memory with a constant byte value, and takes the form:

\[ F s, f, d \]

where \( s \) is the starting address for the fill; \( f \) is the ending (inclusive) address for the fill, and \( d \) is the 8-bit data item to store in locations \( s \) through \( f \). It is your responsibility to not fill memory locations that are occupied by the resident portions of CP/M, including areas reserved for SID. The following are examples of valid F commands:

\[ F100,3FF,FF \]
\[ F.GAMMA,+=#100,#23 \]
\[ F@ALPHA,+=I,=X \]

3.5 The Go (G) Command

The G command passes program control to a program under test. Execution proceeds in real time from the address specified by the G command. That is, the G command releases processor control from SID to the program under test. Execution does not return to SID until a break or pass point is reached (see the "p" command for a discussion of pass points). The operator can force a return to SID, however, by interrupting the processor with a "restart 7" (RST 7) provided by the program under test, or forced by external hardware such as front panel control switches, if available.

The G command takes the following forms:

(a) \( G \)
(b) \( Gp \)
(c) \( G,a \)
(d) \( Gp,a \)
(e) \( G,a,b \)
(f) \( Gp,a,b \)
(g) \( -G \)
(h) \( -Gp \)
(i) \( -G,a \)
(j) \( -Gp,a \)
(k) \( -G,a,b \)
(l) \( -G,p,a,b \)

Forms (a) through (f) start test program execution with symbolic features enabled, while forms (g) through (l) are identical in function, but disable the symbolic features of SID. In particular, form (a) starts test program execution from the program counter (PC) given in the machine state of the program under test (see the "X" command for machine state display). In this case, no breakpoints are set in the test program. Form (b) is similar to
(a), but initializes the test program’s PC to p before starting execution. Again, no breakpoints are set in the test program. Similar to (a), form (c) starts execution from the current value of PC but sets a breakpoint at location a. The test program receives control and runs in real time until the address a is encountered. Note that control returns to SID upon encountering a pass point or RST 7, as described above.

Upon encountering the breakpoint address a, the break address is printed at the console in the form:

*a .s

where s is the first symbol in the table that matches address a, if it exists. Note that the temporary breakpoint at address a is automatically cleared when SID returns to command mode (see the "P" command for permanent breakpoints).

Form (d) combines the functions of (b) and (c): the test program PC is set to the address p and a temporary breakpoint is set at location a. As above, the breakpoint is cleared when control returns to SID. It should be noted that an immediate breakpoint always occurs if p = a. If this is not desired, however, you can use the trace function to single step past the current address, followed by a G command (see the "T" command for actions of the trace facility).

Form (e) extends the breakpoint facility by allowing two temporary break addresses at a and b. Program execution begins at the current PC and continues until either address a or b is encountered. Both temporary break addresses are cleared when SID returns to command mode. Form (f) is similar to (e), except the initial value of PC is set to location p before starting the test program.

Note that the instruction at a breakpoint address is not executed when you use the G command. Suppose, for example, that a subroutine named TYPEOUT is located at address 0302 in a test program, consisting of the machine code:

```
TYPEOUT:
 0302  MOV E,A
 0303  MVI C,2
 0305  JMP 0005
```

Suppose further that you are testing a program that makes calls on the TYPEOUT subroutine where a break address is to be set. Enter the command:

```
G,.TYPEOUT
```

Test program execution proceeds from the current PC value and stops when the TYPEOUT subroutine is reached, with the breakpoint message:

```
*0302 .TYPEOUT
```
indicating that control has returned from the test program to SID. At this point, the program counter of the test program is at location 0302 (i.e., TYPEOUT), and the instruction at this location has not yet been executed. You can execute through the TYPEOUT subroutine using any of the commands G, T, or U. The following is a useful command in this situation:

\[ G,^n \]

This command continues execution from 0302, and sets a breakpoint at the topmost stacked element (given by \("^n\)\). Since the topmost stacked element must be the subroutine return address, this particular G command executes the TYPEOUT subroutine, with a break upon return to the instruction following the original call to TYPEOUT.

Command forms (g) through (l) correspond directly to functions (a) through (f), except that pass points are not displayed until the corresponding pass counters reach 1 (see the "P" command for details of intermediate pass point display).

Note that the essential difference between the G command and the U (Untrace) command is that execution proceeds unmonitored in real time with the G command, while each instruction is executed in single-step mode with the U command. Fully monitored execution under the U command has the advantage that you can regain control at any point in the test program execution. However, execution time of the test program is seriously degraded in Untrace mode since automatic breakpoints are set and cleared following each instruction.

The following are examples of valid G commands:

\[
\begin{align*}
G100 \\
G100,103 \\
G.CRLF,.PRINT,#1024 \\
G@JMPVEC++I,.ENDC,.ERRC \\
G,.ERRSUB \\
G,.ERRSUB,+30 \\
-G100,+10,+10 \\
\end{align*}
\]

3.6 The Hexadecimal Value (H) Command

The H command performs hexadecimal computations including number base conversion operations. The H command takes the following forms:

(a) \( H_a,b \)
(b) \( H_a \)
(c) \( H \)

Form (a) computes the hexadecimal sum and difference using the two operands, resulting in the display:

All Information Presented Here is Proprietary to Digital Research
ssss  dddd

where ssss is the sum a+b, and dddd is the difference a-b, ignoring overflow and underflow conditions.

Form (b) performs number and character conversion, where a is a symbolic expression. The display format in this case is:

    hhhh #ddd  `c` .s

where hhhh is the four digit hexadecimal value of a; #ddd  is the (up to) five digit decimal value of a; c is the ASCII value of a when a is graphic, and s is the first symbol in the table which matches the value a, when such a symbol exists. Assume, for example, that the symbol GAMMA is located at address 0100, as in previous examples. The H commands shown to the left below result in the displays shown to the right below:

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>RESULTING DISPLAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>H01,1</td>
<td>0001  FFFF</td>
</tr>
<tr>
<td>H41</td>
<td>0041  #65 'A'</td>
</tr>
<tr>
<td>H100</td>
<td>0100  #256 .GAMMA</td>
</tr>
<tr>
<td>H=GAMMA</td>
<td>0100  #256 .GAMMA</td>
</tr>
<tr>
<td>H=GAMMA</td>
<td>0001  #1</td>
</tr>
<tr>
<td>H@GAMMA</td>
<td>0201  #513</td>
</tr>
<tr>
<td>HFF+=GAMMA</td>
<td>0100  #256 .GAMMA</td>
</tr>
<tr>
<td>H'A'</td>
<td>0041  #65 'A'</td>
</tr>
<tr>
<td>H'A'+=GAMMA</td>
<td>0042  #66 'B'</td>
</tr>
</tbody>
</table>

Command form (c) prints the complete list of symbols along with their corresponding address values. The list is printed from the first to last symbol loaded, and can be aborted during typeout by depressing any keyboard character.

### 3.7 The Input Line (I) Command

When testing programs that run in the CP/M environment, it is often useful to simulate the command line that the CCP normally prepares upon program load. The I command takes the form:

    Icccccc ... ccc

where the sequence of c's represent ASCII characters that normally follow the test program name in the CCP command line. For example, the CP/M "DUMP" program is normally started in CCP command mode by typing:

    DUMP X.COM

which causes the CCP to search for and load the DUMP.COM file, and
pass the filename "X.COM" as a parameter to the DUMP program. In particular, the CCP initializes two default file control blocks, along with a default command line that contains the characters following the DUMP command.

To trace and debug a program such as DUMP, invoke the SID program with the following command:

```
SID DUMP.COM
```

which loads the command file containing the DUMP machine code. If the symbol table is available, the SID invocation is:

```
SID DUMP.COM DUMP.SYM
```

In either case, SID loads the DUMP program and prompts the console for a command. To simulate the CCP’s command line preparation, type the command:

```
IX.COM
```

where the "I" denotes the Input command, which is followed by the simulated command line. The operator can then commence the debug run with default areas properly setup.

The I command specifically initializes the default file control block in low memory, labelled DFCB1, that is normally located at 005C. The file control block which is initialized by the I command is complete in the sense that the program can simply address DFCB1 and perform and open, make, or delete operation without further initialization. As a convenience, a second filename is initialized at location DFCB2, which is at address DFCB1+0010 (hexadecimal).

It is your responsibility to move the second drive number, filename, and filetype to another region of memory before performing file operations at DFCB1 since the 16-byte region at DFCB2 is immediately overwritten by any file operation. Further, the default buffer, labelled DBUFF, is initialized to contain the entire command line with the first byte of the buffer containing the command line length. In a standard CP/M system, the DBUFF area is assumed to start at 0080 and end at 00FF. Note, however, that the I command restricts the simulated CCP command line to 63 characters since SID’s line buffer is used in the simulation.

Given an I command of the form:

```
I d1:f1.t1 d2:f2.t1
```

where d1: and d2: are (optional) drive identifiers; f1 and f2 are (up to eight character) filenames, and t1 and t2 are (up to three character optional) filetypes, two default file control block names are prepared in the form:
DFCB1: d1' f1' t1' 00 00 00 00
DFCB2: d2' f2' t2' 00 00 00 00
00 (current record field)

If d1: is empty in the original command line, then d1' = 00 (which automatically selects the default drive), otherwise if d1 = A, B, C, or D, then d1' = 01, 02, 03, or 04, respectively, which properly initializes the file control block for automatic disk selection. Field f1' is initialized to the ASCII filename given by f1, padded to an eight character field with ASCII blanks. Similarly, t1' is initialized to the ASCII filetype, padded with blanks in a field of length three.

Lower-case alphabetics in f1 and t1 are translated to upper-case in f1' and t1', respectively. Names that exceed their respective length fields are truncated on the right. Finally, the extent field is zeroed in preparation for a BDOS call to open or make the file.

The second default file control block given by d2, f2, and t2 is prepared in a similar fashion and stored starting at location 006C. Note that the current record field at location 007C is also initialized to 00. If any of the fields f1, t1, f2, and t2 are not included in the command line, their corresponding fields in the default file control blocks are filled with blanks.

Ambiguous references that use the "*" or "?" characters are processed in the same manner as in the CCP: the "*" symbol in a name or type field causes the field to be right-filled with "?" characters. The input lines shown below illustrate the default area initialization which takes place for various unambiguous and ambiguous filenames. The areas shown to the right give the hexadecimal values which begin at the labelled addresses, where ASCII values A, B, C, and D have the hexadecimal values 41, 42, 43, and 44, respectively. Further, the special characters ":", ".", "*", and "?" have the ASCII encoded values 3A, 2E, 2A, and 3F, while an ASCII space has the hexadecimal value 20:

<table>
<thead>
<tr>
<th>COMMAND LINE</th>
<th>DEFAULT DATA AREA INITIALIZATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>DFCB1: 00</td>
</tr>
<tr>
<td></td>
<td>20 20 20 20 20 20 20 20</td>
</tr>
<tr>
<td></td>
<td>20 20 20 00 00 00 00 00</td>
</tr>
<tr>
<td></td>
<td>DFCB2: 00</td>
</tr>
<tr>
<td></td>
<td>20 20 20 20 20 20 20 20</td>
</tr>
<tr>
<td></td>
<td>20 20 20 00 00 00 00 00</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>DBUFF: 00 00</td>
</tr>
</tbody>
</table>
I A.B

DFCB1: 00
41 20 20 20 20 20 20
42 20 20 00 00 00 00

DFCB2: 00
20 20 20 20 20 20 20
20 20 20 00 00 00 00
00
00

DBUFF: 04 20 41 2E 42 00

IA:B.C b:d.e

DFCB1: 01
42 20 20 20 20 20 20
43 20 20 00 00 00 00

DFCB2: 02
44 20 20 20 20 20 20
45 20 20 00 00 00 00
00
00

DBUFF: 0B 41 3A 42 2E 43 20
42 3A 44 2E 45 00

I AA*.B?C D:

DFCB1: 00
41 41 3F 3F 3F 3F 3F
42 3F 43 00 00 00 00

DFCB2: 04
20 20 20 20 20 20 20
20 20 20 00 00 00 00
00
00

DBUFF: 0B 20 41 41 2A 2E 42
3F 43 20 44 3A 00

Note that the I command is also used in conjunction with the R command to read program files and symbol tables after SID has initially loaded. Details of the use of I in this situation are given with the R command that follows.

Additional valid I commands are given below:

I x.dat
Ix.inp y.out
Ia:x.inp b:y.out $-p
ITEST.COM
I TEST.HEX TEST.SYM

All Information Presented Here is Proprietary to Digital Research
3.8 The List Code (L) Command

The L command disassembles machine code in the memory of the machine, with symbolic labels and operands placed in the appropriate fields, where possible. The L command takes the forms:

(a) Ls
(b) Ls,f
(c) L
(d) -Ls
(e) -Ls,f
(f) -L

Form (a) lists disassembled machine code starting at symbolic location s for 1/2 CRT screen (12 lines). Form (b) specifies an exact range for disassembly: s specifies the starting location, and f gives the final disassembly location. Form (c) is similar to (a), but disassembles from the last listed, assembled (see the A command), traced (see the T and U commands), or break address (see the G and P commands). Since form (c) also lists 1/2 CRT screen, it is often used following form (a) to continue the disassembly process through another segment of the program. Forms (d) through (f) parallel (a) through (c), but disable the symbolic features of SID. In particular, the minus prefix prevents any symbol lookup operations during the disassembly.

The L command output takes the following form:

```
ssssss:
 aaaa opcode operand .ttttt
```

where "ssssss:" represent a symbol which labels the program location given by the hexadecimal address aaaa, when the symbol exists. The "opcode" field gives the 8080 mnemonic for the instruction at location aaaa, and the "operand" field, when present, gives the hexadecimal values which follow the opcode in memory. The symbol ".ttttt" is printed when the instruction references a memory address which matches a symbol in the table.

When the operation code at the list address is not a valid 8080 mnemonic, the output form is:

```
??= hh
```

where hh is the hexadecimal value of the invalid operation code.

Several valid L commands are listed below.

```
L100
L#1024,#1034
L.CRLF
L@ICALL,+30
-L.PRBUFF+=I,+A
```

All Information Presented Here is Proprietary to Digital Research
3.9 The Move Memory (M) Command

The M command allows you to move blocks of data values from one area of memory to another. The M command takes the form:

Ms,h,d

where s is the start address of the move operation; h is the high (last) address of the move, and d is the starting destination address to receive the data. SID transfers one byte at a time from the start address to the destination address. Each time a byte value is moved, the start and destination addresses are incremented by one. The move process terminates when the start address increments past the final f address. The command:

M100,1FF,3000

for example, replicates the entire block of memory from 0100 through 01FF at the destination area from 3000 through 30FF in memory. The data block from 0100 through 01FF remains intact.

Note that data areas may overlap in the move process. The command:

M100,1FF,101

shows an instance where the value at location 0100 is propagated throughout the entire block from 0101 through 0200.

A number of valid M commands are listed below:

M-100,FFD0,100
M.X,+=Z,.Y
M.GAMMA,++FF,.DELTA
M@ALPHA+=X,+#50,+100

3.10 The Pass Counter (P) Command

The P command allows you to set and clear "pass points" and "pass counts" in the program under test. The P command takes the following forms:

(a) Pp
(b) Pp,c
(c) P
(d) -Pp
(e) -P

A "pass point" is a program location to monitor during execution of the test program. Similar to a temporary breakpoint (see the G command), a pass point causes SID to stop execution of the test program each time an active pass point is reached. Unlike a temporary breakpoint, a pass point is not automatically cleared each time it is reached during execution. Further, unlike a
temporary breakpoint, a pass point break occurs after the instruction as the pass address is executed. In this way, you can simply continue the execution of the test program under control of a G command until the next pass point is executed, or until a temporary breakpoint is reached.

Each pass point can have an optional "pass count" which defaults to the value 1. The pass count enhances this facility by allowing several passes through a pass point before the break actually occurs. In particular, a pass count in the range 1-FF (decimal 1 through 255) can be associated with a particular pass point. Each time the instruction at a pass point is executed, its corresponding pass count is decremented. The decrementing process proceeds until the pass count reaches 1, at which time the break address is printed and execution of the test program stops. When a pass count reaches 1, the pass point becomes a permanent break address which halts execution each time the instruction is executed. Note that a pass count does not change once it has reached 1. Up to eight distinct pass points can be actively set at any particular time.

Form (a) sets a pass point at address p with a pass count of 1, causing address p to become a permanent breakpoint. Form (b) is similar, except that the pass count is initialized to c. Form (c) displays these active pass points in the format:

```
cc pppp .sssss
```

where cc is the hexadecimal value of the pass count that is currently associated with the pass address pppp, and sssss is a symbol that matches the address pppp, if such a symbol exists.

Form (d) clears the pass point at address p, while form (e) clears all active pass points. Note that the command:

```
Pp,0
```

is equivalent to form (d).

Each time a pass point is encountered, SID prints the pass information in the format:

```
cc PASS pppp .sssss
```

where cc is the current pass count at pass point pppp (cc is decremented when greater than 1). As above, the symbol sssss corresponding to address pppp is printed when possible.

The special command forms "-G" and "-U" to disable the intermediate pass trace as the counters are decremented down to 1. Suppose, for example, the TYPEOUT subroutine is a part of a program under test, as shown in the G command above. Issue the command:

```
P.TYPEOUT,#30
```
This P command sets a pass point at the location labelled by "TYPEOUT" which is assumed to exist in the symbol table. The pass count is set to decimal 30, which allows the pass point to execute 30 times before a breakpoint is taken. Given that the pass point at TYPEOUT is in effect, the command:

```
G
```

starts execution of the test program with no temporary breakpoint. Each time the pass point is executed, the following pass trace is executed:

```
1E PASS 0302 .TYPEOUT
  (register trace)
1D PASS 0302 .TYPEOUT
  (register trace)
1C PASS 0302 .TYPEOUT
  (register trace)
  ...
01 PASS 0302 .TYPEOUT
  (register trace)
*303
```

The "register trace" shows the state of the CPU registers before the "MOV E,A" at TYPEOUT is executed (see the "X" command for register display format). Note that the final breakpoint address is 0303, which follows the "MOV" instruction at the pass address 0302. Depress any keyboard character during the pass point trace, and SID immediately stops execution following the instruction at the pass point address. If the command

```
-G
```

had been issued, the intermediate pass traces do not appear at the console. In this particular case, only the final trace:

```
01 PASS 0302 .TYPEOUT
  (register trace)
*303
```

is printed. Although the intermediate pass traces are not displayed, you can abort execution by depressing a keyboard character. If an intermediate pass point is encountered with trace disabled, SID aborts execution and returns control to the keyboard.

Temporary breakpoints can also be set while pass points are in effect. That is, commands such as:

Ga,b
Ga,b,c
G,b
G,b,c

can be issued that intermix with the permanent breakpoints that are set with the P command. Note, however, that permanent breakpoints
override the temporary breakpoints that are given by b and c when they occur at the same address. Further, T and U command can trace sections of the test program while permanent breakpoints are in effect. In this case, the pass counts decrement as described above, with a break taken when the count reaches 1.

Valid P commands are shown below:

- P100,FF
- P.BDOS
- P@ICALL+30,#20
- -P.CRLF

3.11 The Read Code/Symbols (R) Command

The R command, in conjunction with the I command, reads program segments, symbol tables, and utility functions into the Transient Program Area. The R command takes the forms:

(a) R
(b) Rd

The I command sets the filenames that will be involved in the read operation. Form (a) reads the program and/or symbol table given by the I command without applying an offset to the load addresses. Form (b) adds the displacement value to each program load address and/or symbol table address. Note that this addition takes place without overflow checks so that negative bias values can be applied. As a simple case, the usual initiation of SID:

A>SID X.COM

can be replaced by the following sequence of commands:

SID
IX.COM
R

Starts SID without a test program
Initialize the input line
Read the test program to memory

The response from SID in this case is exactly the same as the normal initialization, with the "NEXT PC END" message as described in Section 1.

A program and symbol file can be read by preceding the R command with an I command of the form:

I x.y u.v

where x.y is the program to load, and u.v is the symbol table file. Note that y is usually the type "COM"; x is usually the same as u, and v is usually the type "SYM". Thus, the following is a typical command sequence of this form:

IDUMP.COM DUMP.SYM
R
This sequence reads the DUMP.COM program file into the Transient Program Area and loads the symbol table with the information given by DUMP.SYM. Programs with filetype "HEX" load into the locations specified in the Intel formatted hexadecimal records, while programs with filetype "UTL" are assumed to be SID utility functions that load and relocate automatically. All other filetypes are assumed absolute, and load starting at the base of the transient area. Utility functions automatically remove any existing symbol information when they relocate, but in all other cases the symbol load operations are cumulative. In particular, the special input form:

I* u.v
R

skips the program load since there is an asterisk in the program name position, and loads only the symbol table file. Thus, a sequence of commands of the above form can load the symbol tables for selective portions of a large program that was initially developed in small modules.

Suppose, for example, that a report generation program has been developed using MAC, which consists of the following modules:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOMOD.ASM</td>
<td>I/O Module</td>
</tr>
<tr>
<td>SORT.ASM</td>
<td>File Sorting Module</td>
</tr>
<tr>
<td>MERGE.ASM</td>
<td>File Merge Module</td>
</tr>
<tr>
<td>FORMAT.ASM</td>
<td>Report Format Module</td>
</tr>
<tr>
<td>MAIN.ASM</td>
<td>Main Program Module</td>
</tr>
<tr>
<td>DATA.ASM</td>
<td>Common Data Definitions</td>
</tr>
</tbody>
</table>

Suppose further that each module has been separately assembled using MAC, resulting in several "HEX" and "SYM" files corresponding to the individual program segments. The program segments have been brought together using SID to form a memory image by typing the sequence of commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>Start the SID program</td>
</tr>
<tr>
<td>IOMOD.HEX</td>
<td>Initialize IOMOD</td>
</tr>
<tr>
<td>R</td>
<td>Read I/O Module</td>
</tr>
<tr>
<td>SORT.HEX</td>
<td>Initialize SORT</td>
</tr>
<tr>
<td>R</td>
<td>Read Sort Module</td>
</tr>
<tr>
<td>MERGE.HEX</td>
<td>Initialize MERGE</td>
</tr>
<tr>
<td>R</td>
<td>Read Merge Module</td>
</tr>
<tr>
<td>FORMAT.HEX</td>
<td>Initialize FORMAT</td>
</tr>
<tr>
<td>R</td>
<td>Read Format Module</td>
</tr>
<tr>
<td>MAIN.HEX</td>
<td>Initialize MAIN</td>
</tr>
<tr>
<td>R</td>
<td>Read Main Module</td>
</tr>
<tr>
<td>DATA.HEX</td>
<td>Initialize DATA Area</td>
</tr>
<tr>
<td>R</td>
<td>Read Initialized Data</td>
</tr>
</tbody>
</table>

Following this sequence, the Transient Program Area contains the complete memory image of the report generation program. Suppose the information printed following the last R command is:

All Information Presented Here is Proprietary to Digital Research.
NEXT PC END
1B3E 0100 8E00

which indicates that the high memory address is 1B3E. Using the H command:

H1B

you find that 1B (hexadecimal) pages is the same as 27 (decimal) pages. At this point, return to CCP mode by typing either a control-C (warm start), or "G0" command, which leaves the memory image intact. Then issue the command:

SAVE 27 REPORT.COM

to create a memory image file on the diskette. Then re-enter SID using the following command:

SID REPORT.COM

to load the entire module for testing. Individual portions of the report generator can then be symbolically accessed by selectively loading symbol tables from the original modules. For example, the MAIN and SORT modules can be debugged by subsequently loading the corresponding symbol information:

I* MAIN.SYM
R
I* SORT.SYM
R

which prepares the symbol information for subsequent debugging. Individual segments of the report generator are then tested and reassembled. If an error is found in the SORT module, for example, the SORT.ASM file is edited to make necessary changes, and the module is reassembled with MAC, resulting in new "HEX" and "SYM" files for the SORT module only. Given that enough "expansion" area has been provided following the SORT module, SID is reinitiated and the SORT module is included:

SID REPORT.COM
ISORT.HEX SORT.SYM
R

which overlays the changed SORT module in the original report generator memory image. You can then load additional symbol tables by typing I and R commands such as:

I* MAIN.SYM
R
I* DATA.SYM
R

to access symbols in the SORT, MAIN, and DATA modules.

All Information Presented Here is Proprietary to Digital Research
Note that several symbol table files can be concatenated using the PIP program (see the "CP/M Features and Facilities" manual for PIP operation) before SID is invoked. For example, the PIP command:

```plaintext
PIP NOBUGS.SYM=IOMOD.SYM,SORT.SYM,MERGE.SYM,FORMAT.SYM
```

creates a file called NOBUGS.SYM that holds the symbols for IOMOD, SORT, MERGE, and FORMAT. The SID command:

```plaintext
SID REPORT.COM NOBUGS.SYM
```

loads the memory image for the report generator, along with the symbol tables for these particular modules. Additional symbol files can then be selectively loaded using I and R commands. The symbol file for the entire memory image can then be constructed using the PIP command:

```plaintext
PIP REPORT.SYM=NOBUGS.SYM,MAIN.SYM,DATA.SYM
```

which allows you to type:

```plaintext
SID REPORT.COM REPORT.SYM
```

to load the memory image for the report generator, along with the entire symbol table. Recall, however, that the symbol table is always searched in load-order, and thus symbol names which are the same in two modules must be distinguished using qualified symbolic names (see Section 1).

As mentioned above, form (b) allows a displacement value d to be added to each program address and symbol value. The displacement value has no effect, however, when the program is a SID utility (filetype "UTL"). The commands:

```plaintext
IDUMP.HEX DUMP.SYM
R1000
```

for example, cause the DUMP program to be loaded 1000 (hexadecimal) locations above its normal origin, with properly adjusted symbol addresses. Note that the bias value can be any symbolic expression, and thus the command:

```plaintext
R-200
```

first produces a (two's complement) negative number which is added to each address. Since overflow from a 16-bit counter is ignored, this R command loads the program 200 (hexadecimal) locations below the normal load address, with symbol addresses biased by this same amount.

Error reporting during the R command is limited to the standard "?" response, which indicates that either the program or symbol file does not exist, or the program or symbol file is improperly formed. Similar to the SID startup messages, the response:

All Information Presented Here is Proprietary to Digital Research
SYMBOLS

occurs following program load, and appears before the symbol load. Thus, a "?" error before the SYMBOLS response indicates that the error occurred during the program load, while the "?" error after the SYMBOLS message indicates that an error occurred during the symbol file load operation. The exact position of a symbol file error can be found by subsequently using the H command to view the portion of the symbol table that was actually loaded.

3.12 The Set Memory (S) Command

The S command allows you to enter data into main memory. The forms of the S command are:

(a) Ss
(b) SWs

Form (a) allows data to be entered at location s in byte (8-bit) or character string mode, while form (b) stores word (16-bit) mode data items. In either case, the SID program prompts the console with successive addresses, starting at location s, along with the data item presently located at that address. As each line prompt occurs, you can type a single carriage return or a symbolic expression (followed by a carriage return), which is evaluated and becomes the new data item at that location. If you type a single carriage return, then the data element at that location remains unchanged. The S command terminates whenever an invalid data item is detected, or when you type a single "." followed by a carriage return. Form (a) allows single byte data, and produces the standard "?" when a double byte value is entered with a non-zero high-order byte. In addition, form (a) also allows long ASCII string data to be entered in the format:

"cccccc . . .cccccc

where the sequence of c's (terminated with a carriage return) represents graphic ASCII characters to be entered at the prompted location. No translation from lower- to upper-case takes place during entry. Further, the next prompted address is automatically set to the first unfilled location following the input string.

A valid input sequence following the command:

S100

is shown below, where the SID prompt is given on the left, and the operator's input lines are shown to the right, where "cr" denotes the carriage return key.
3.12 The Set Memory (S) Command

<table>
<thead>
<tr>
<th>SID PROMPT</th>
<th>OPERATOR INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 C3</td>
<td>34cr</td>
</tr>
<tr>
<td>0101 24</td>
<td>#254cr</td>
</tr>
<tr>
<td>0102 CF</td>
<td>cr</td>
</tr>
<tr>
<td>0103 4B</td>
<td>&quot;ASCII cr</td>
</tr>
<tr>
<td>0108 6E</td>
<td>=X+5 cr</td>
</tr>
<tr>
<td>0109 E2</td>
<td>%cr</td>
</tr>
<tr>
<td>010A D4</td>
<td>.cr</td>
</tr>
</tbody>
</table>

A valid double byte input sequence following the command:

\[ \text{SW.X+\#30} \]

is shown below:

<table>
<thead>
<tr>
<th>SID PROMPT</th>
<th>OPERATOR INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2300 006D</td>
<td>44F cr</td>
</tr>
<tr>
<td>2302 4F32</td>
<td>@GAMMA cr</td>
</tr>
<tr>
<td>2304 33E2</td>
<td>cr</td>
</tr>
<tr>
<td>2306 FFI1</td>
<td>.X+=I-#20 cr</td>
</tr>
<tr>
<td>2308 348F</td>
<td>.cr</td>
</tr>
</tbody>
</table>

3.13 The Trace Mode (T) Command

The T command allows you to single or multiple step a test program while viewing the CPU registers as they change. In addition, you can use the T command with SID utilities to collect test program data for later display (see the section entitled "SID Utilities"). The forms of the T command are:

(a) \text{Tn}  
(b) \text{T}    
(c) \text{Tn,c} 
(d) \text{T,c}  
(e) \text{-T} (with options a - d) 
(f) \text{TW} (with options a - d) 
(g) \text{-TW} (with options a - d)

Form (a) traces program execution from the current value of the program counter PC (see the "X" command for PC value as well as the format of the CPU state display). Form (b) is the trivial case of (a) with an assumed single step count of \( n = 1 \). In either case, the SID program displays the register state, along with the decoded instruction (assuming "-A" is not in effect) before each instruction is executed. For example, the command:

\[ \text{T4} \]

traces four program steps, producing the format:
(register state 1) opcode 1
label:
(register state 2) opcode 2
label:
(register state 3) opcode 3
label:
(register state 4) opcode 4 *babb

showing the register state before each corresponding operation code is executed. Each operation code is written in the same format as the L and X commands, with interspersed symbolic operands decoded wherever possible. In addition, instructions that reference memory, such as INR M, are listed with the memory operand in the form:

opcode M =hh

where "opcode" is the memory referencing instruction, and hh is the hexadecimal value contained in the memory address given by the HL register pair before the operation takes place. The interspersed labels show program addresses when they occur in the flow of execution. The final break address, denoted by "*babb" above, shows the value of the program counter after opcode 4 is executed. You can display the CPU state at this point by typing the single character "X" command.

Forms (c) and (d) are used only with the SID utilities, and automatically perform a CALL c after each instruction executes. The value of c corresponds to a utility entry address for data collection. The following sections detail these forms. Note, however, that form (d) is equivalent to (c) with a single step count of n = 1.

Forms given by (e) parallel (a) through (d), but the preceding minus sign disables the symbolic features of SID. In particular, neither the symbolic operands nor the symbolic labels are decoded in the trace process. This option speeds up the operation of SID slightly in trace mode when large symbol tables are present.

Forms given by (f) parallel (a) through (d), but perform a "trace without call" function. It is often useful, for example, to trace mainline program code, but not trace into the subroutines which are called from the mainline execution. The TW command performs this function by running the test program in real time whenever a subroutine is entered, returning to fully traced mode upon return to the current subroutine level. If a return operation takes place at the current level (i.e., a RET is executed in fully traced mode), then tracing continues at the encompassing subroutine or mainline program level. For example, suppose the mainline and subroutine structure shown below exists in a particular program:

All Information Presented Here is Proprietary to Digital Research
Suppose further that the test program is stopped within subroutine S1 before the call to subroutine S2. The command:

\text{T\#100}

traces from S1 through S2, S3, and so forth until level Sn is encountered. Although this form of the trace could be useful, it is often more enlightening to trace only at a particular subroutine level, and view the effects of the subroutine levels above S1. In this manner, an offending subroutine is often easily discovered without tracing non-essential program flows. If you type the following command while at subroutine level S1, all subsequent levels from S2 and beyond are executed in real time as if a "G" command had been performed at each CALL within S1.

\text{TW\#100}

Upon executing the RET instruction within S1, tracing resumes at the mainline level. Any subroutine calls following CALL S1 at the main level are not subsequently traced.

Forms given by (g) parallel (a) through (d), but disable the symbolic features of SID in the same manner as form (e).

Note that SID allows tracing up to Read Only Memory (ROM) program code. At the point ROM is entered, SID stops the trace operation, and runs the ROM code in real time. An automatic breakpoint is set which intercepts program control when ROM code is exited. The assumption, however, is that ROM code was entered via a subroutine call (CALL or RST n), not via a PCHL or JMP instruction. In any case, the return address following the ROM execution is taken as the topmost address in the test program's stack.

Note further that SID does not trace execution of calls through the BDOS code, since these operations are often quite lengthy, and can occasionally require real time operation to perform various disk functions. Thus, entry to the BDOS is intercepted by SID, and resumed following completion of the BDOS function.

Abort tracing at any time by depressing a keyboard character. Do not use the RST instruction to terminate trace functions.
Valid trace commands are shown below:

T100
T#30,.COLLECT
-TW=I,3E03

3.14 The Untrace Mode (U) Command

The U command is similar to the T command given above, except that the CPU register state is not displayed at each step. Instead, the test program runs fully monitored so that program execution can be aborted at any time, or for the collection of data for a SID utility function. The forms of the U command parallel the T command:

(a) Un
(b) U
(c) Un,c
(d) U,c
(e) -U (with options a - d)
(f) UW (with options a - d)
(g) -UW (with options a - d)

Forms (a) through (d) perform the analogous functions of the "T" command forms (a) through (d), without displaying the register state at each step. Forms given by (e) differ from the T command; however, instead of disabling the symbolic features, the following command forms:

-Un
-U
-Un,c
-U,c

disable the intermediate pass point display (see the "P" command), until the corresponding pass counts reach 1.

Forms given by (f) correspond to the "T" command exactly, except that the trace display is disabled. In this case, the current subroutine level is run fully monitored, but higher subroutine levels run in real time.

Forms given by (g) are similar to (f), but disable the pass point display, as described above.

You can abort execution in untrace mode by depressing any keyboard character. The break address is displayed, and control returns to SID command mode.

Valid U commands are given below:

UFFFF
U#10000,.COLLECT
UW=Gamma,.COLLECT
3.15 The Examine CPU State (X) Command

The X command allows you to examine and alter the CPU state of the program under test. The X command takes the following forms:

(a) X
(b) Xf
(c) Xr

Form (a) displays the entire CPU state in the format:

CZMEI A=aa B=bbbb D=dddd H=hhhh S=ssss P=pppp op sym

where C, Z, M, E, and I represent the true or false conditions of the CPU carry, zero, minus, even parity, and interdigit carry, respectively. If the position contains a "-" then the corresponding flag is false, otherwise the flag letter is printed. The byte value aa is the value of the A register, while the double byte values bbbb, dddd, hhhh, ssss, and pppp, give the 16-bit values of the BC, DE, HL, Stack Pointer, and Program Counter, respectively. The field marked "op" gives the decoded mnemonic instruction at location pppp, unless ":A" is in effect, in which case the hexadecimal value of the operation code is printed. The "sym" field contains a decoded operand, when possible. Refer to the L command for the format of the symbolic instruction decoding. The single letter "X" command might result in a display of the form:

C-M-- A=03 B=34EF D=2000 H=334E S=4323 P=0100 LDA 0223 .Q

which, for example, indicates that the carry and minus flags are true, while the zero, even parity, and interdigit carry flags are false. Further, the A register contains 03, while the B, C, D, E, H, and L registers contain the hexadecimal values 34, EF, 20, 00, 33, and 4E, respectively. The value of the Stack Pointer is 4323, and the Program Counter is at location 0100. The next instruction to execute at location 0100 is an accumulator load (LDA) from location 0233. Further, the first symbol in the table that matches address 0233 is Q.

Form (b) allows you to change the state of the CPU flags. In this case, f must be one of the condition code letters: C, Z, M, E, or I. The present state of the flag is displayed (either the flag letter if true, or a "-" if false). You can either type a single carriage return, which leaves the flag in its present state, or you can type a L to set the flag true, or a 0 to reset the flag to false. Given that the carry flag is true, for example, the command:

XC

produces the SID response:

C

followed by a space, indicating that the carry is currently set, awaiting possible change. Enter a carriage return to leave the flag
set, or a 0 to reset the carry to false. Similarly, if the zero flag is false, the command:

```
XZ
```

produces the SID response:

```
-
```

indicating that the zero flag is false. Enter a carriage return if the state is to remain unchanged, or a 1 to set the zero flag to true.

Form (c) allows alteration of the individual CPU registers, where r is one of the register names A, B, D, H, S, or P. In this case, the current content of the register is displayed, and the console is prompted for input. If you type a single carriage return, the data value remains unchanged. Otherwise, the symbolic expression is evaluated and becomes the new value of the register. Only byte values are acceptable when the "XA" form is used, while double byte values are accepted in the remaining forms. Note that the BC, DE, and HL registers must be altered as a pair. The SID interaction shown below is typical when the A register is altered:

```
XA
A=03 45 cr
```

where you type the "XA"; SID prints the "03" as the value of the A register, and you type "45" as a replacement for A's value. The "cr" represents the carriage return key in this example and in the examples that follow. The following interactions with SID provide additional examples in the format described above:

```
XB
B=34EF cr (data remains unchanged)

XD
D=2000 2300 cr (D changes to 23)

XH
H=334E .GAMMA cr

XS
S=4323 @STKPTR+#100 cr
```
Section 4
SID Utilities

SID utilities are special programs that operate with SID to provide additional debugging facilities. As described in Section 1, you load a SID utility by typing:

SID x.UTL

where x is the name of a utility program, described in the following sections. Upon initiation, the utility program loads, relocates, and prompts the console for any necessary parameters. Then you collect the necessary program test data (using the U or T command), and display the information using a call to the utility display subroutine. The mechanisms for system initialization, data collection, and data display are given in detail below.

4.1 Utility Operation

A particular SID utility loads into memory in much the same manner as a normal test program. The utilities, however, automatically move themselves into high memory, occupying the region directly below the SID program, as described in Section 1. The utility load operation can be accomplished by simply typing the utility name with the SID command as shown above. You can also load a utility during the SID execution, as described in the I and R commands. Recall, however, that all existing symbol information is removed when the utility loads, and must be reinitialized if required for the debugging run.

Normally, a SID utility has three primary entry points: INITIAL for utility (re)initialization, COLLECT for data collection, and DISPLAY for data display. After loading, the utility sets up these symbols in the table, and types the entry point addresses in the format:

```plaintext
.INITIAL = iii
.Collect = cccc
.DISPLAY = dddd
```

where iii, cccc, and dddd are the hexadecimal addresses of the three entry points. Note, however, that the three symbolic names are equivalent to these three addresses.

Following initial sign on, the utility may prompt the console for additional debugging parameters. After the interaction is complete, you can use the I and R commands to load test programs and symbol tables to proceed with the debug session.
During the debug run, data collection takes place by running the test program in monitored mode using the U or T commands. Either of the following commands:

UFFFF,COLLECT
UFFFF,cccc

direct the SID program to run the test program from the current Program Counter for a maximum of 65535 (FFFF hexadecimal) steps, with a call to the data collection entry point of the utility program. Each instruction breakpoint sends information to the utility program, where it is tabulated for later display. Note that in this particular case, you can stop the untrace mode by depressing the return key before the sequence of 65535 steps is completed.

Following a series of data collection operations, enter either of the following commands that call the utility DISPLAY entry point to print the accumulated data:

C.DISPLAY
Cdddd

Then, resume the data collection process, as described above, followed by additional display operations.

At any point, you can reinitialize the utility by typing either of the following commands:

C.INITIAL
Ciii

which causes reinitialization of the utility tables. The utility then prompts for additional parameters to complete the reinitialization process.

Note that loading and executing more than one utility function during a debugging session can produce unpredictable results.

The remaining sections present the functions of the SID utilities.

4.2 The HIST Utility

The HIST utility creates a histogram (bar graph) of the relative frequency of execution in selected program segments of a program under test. The HIST utility allows you to monitor "hot spots" in the test program where the program is executing most frequently.

After initial sign-on, as described in the previous section, the HIST utility prompts the input console:

TYPE HISTOGRAM BOUNDS
You must respond with two symbolic expressions, separated by a comma:

llll, hhhh

where llll is the lowest address to monitor, and hhhh is the highest address. To collect histogram information, you must use one of the following command forms:

Tn,c T,c Twn,c TW,c -Tn,c -T,c -Twn,c -TW,c
Un,c U,c Uwn,c UW,c -Un,c -U,c -Uwn,c -UW,c

where c is either .COLLECT, or the address corresponding to the COLLECT entry point. Although any of these commands may be used, the form:

Un,.COLLECT

is nearly always used since the trace output is disabled, the test program is fully monitored, and data collection takes place at each program step.

Following a series of data collection operations, display the histogram by typing:

C.DISPLAY or Cdddd

The histogram is then printed in the following format:

<table>
<thead>
<tr>
<th>ADDR</th>
<th>RELATIVE FREQUENCY, MAXIMUM VALUE = mmmm</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaa</td>
<td>*****</td>
</tr>
<tr>
<td>bbbb</td>
<td>*******</td>
</tr>
<tr>
<td>cccc</td>
<td>*********</td>
</tr>
<tr>
<td>....</td>
<td>*******</td>
</tr>
</tbody>
</table>
| xxxx | *****************************************
| yyyy | *****************************************
| zzzz | *****                                    |

where addresses aaaa through zzzz span the range from the low to high address range given in the initialization of HIST. The maximum value mmmm is the largest number of instructions accumulated at any of the displayed addresses, and the asterisks represent the bar graph of relative instruction frequencies, scaled according to the maximum value mmmm. The address range is automatically scaled over 64 different address slots (aaaa, bbbb, ..., zzzz, above), with a maximum of 64 asterisks in any particular bar of the graph.

Given the above display, the "hot spot" is around the address range xxxx to zzzz. In this case, type either of the following commands to reinitialize the HIST utility:

C.INITIAL
Ciiii
Then the HIST initialization prompt and response follow, as shown below:

TYPE HISTOGRAM BOUNDS xxxx, zzzz

You can then rerun the test program using the command:

UFFFF,.COLLECT

After leaving enough time for the test program to reach "steady state," interrupt program execution by typing a return during the monitored execution. The display function is then reinvoked to expand the region between xxxx and zzzz, resulting in a more refined view of the frequently executed region.

The L command can subsequently determine the exact instructions that are most frequently executed. If possible, the sequence of instructions can be somewhat improved, with an overall improvement in program performance.

4.3 The TRACE Utility

The TRACE utility obtains a backtrace of the instructions that led to a particular break address in a program under test. For example, a program might have an error condition that arises from a sequence of instructions that are difficult to find under normal testing. In this case, TRACE can collect program addresses as the test program executes, and display these addresses and instructions in most recent to least recent order when you request. To invoke SID with the TRACE utility, enter the following command:

SID TRACE.UTL

The utility responds as follows:

INITIAL = iiii
COLLECT = cccc
DISPLAY = dddd

In this case, the TRACE utility also prints the message:

READY FOR SYMBOLIC BACKTRACE

which indicates that the assembler/disassembler portion of SID is present, and will disassemble instructions when the backtrace is requested.

You can then proceed to load a test program with optional symbol table. For example, you can load the DUMP program, by typing the command:

IDUMP.COM DUMP.SYM
R
The usual response:

"NEXT PC END"

indicates that the test program is loaded. At this point, the SID
debugger is executing in high memory, along with the TRACE utility
and the test program symbols. The test program is present in low
memory, ready for execution.

To obtain the simplest backtrace, type one of the U or T
command forms shown with the HIST utility. In particular, a U
command of the form:

U#500,.COLLECT

executes 500 (decimal) program steps, and then automatically stops
program execution. Type the following command to obtain a backtrace
to the stop address:

C.DISPLAY

This command causes TRACE to display the label, address, and
mnemonic information in the form:

label-255:
  addr-255  opcode-255  sym-255
label-254:
  addr-254  opcode-254  sym-254
label-253:
  addr-253  opcode-253  sym-253
  ...
label-000:
  addr-000  opcode-000  sym-000

where label-255 down through label-000 represent the decoded
symbolic labels corresponding to addresses given by addr-255 down
through addr-000, when the symbolic labels exist. Opcode-255 down
through opcode-000 represent the mnemonic operation codes
corresponding to the backtraced addresses, and sym-255 down through
sym-000 denote the symbolic operands corresponding to the operation
codes, when the symbols exist. The operation codes are displayed in
the same format as the list command. Note that in this display, the
most recently executed instruction is at location addr-255, while
the least recently executed instruction is at location addr-000. 
TRACE accounts for up to 256 instructions, which accumulate in T or
U mode. The accumulated instructions are not affected by the
DISPLAY function, but are cleared by the following call to
reinitialize:

C.INITIAL

Full benefit of the TRACE utility requires concurrent use of
TRACE with pass points (see the "P" command). In particular, pass
points are first set at program locations that are of interest in
the backtrace. The program is then run to an intermediate location
where the test begins. At this intermediate test point, use the U command to execute the test program in fully monitored mode, with data collection at the COLLECT entry point of TRACE. Upon encountering one of the pass points in U mode, program execution breaks, and you regain control in SID command mode. The DISPLAY function of TRACE is then invoked to obtain the required backtrace information.

As an example of this process, suppose the DUMP program is in memory with the TRACE utility, as shown above. Suppose further that you want to view the actions of the DUMP program on the first call to BDOS (i.e., the first call from DUMP to the CP/M Basic Disk Operating System, through location 0005). Assuming the symbol table is loaded, type the command:

```
P.BDOS
```

which sets a pass point at the BDOS entry, with corresponding pass count = 1. Then execute DUMP in monitored mode, collecting data at each instruction:

```
UFFFF,.COLLECT
```

The untrace count of FFFF (65535) instructions is, of course, too many in this case, but the assumption is that the DUMP program stops at the BDOS call before the instruction count is exceeded (if it does not, depress any keyboard character to force a program break). In this case, the DUMP program executes only a few instructions before the BDOS call, resulting in the break information:

```
01 PASS 0005 .BDOS
-ZEI A=80 B=0014 D=005C H=0000 S=0249 P=0005 JMP CCDF
*CCDF
```

showing the pass count 1, pass address 0005, symbolic location BDOS, register state, and break address. Since execution to this point was monitored and data was collected, invoke the TRACE function:

```
C.DISPLAY
```

which results in the display:

```
BDOS:
0005  JMP  CCDF
01CA  CALL 0005 .BDOS
01C8  MVI  C,OF
01C5  LXI  D,005C .FCB
01C2  STA  007C .FCBCR
SETUP:
01C1  XRA  A
010A  CALL 01C1 .SETUP
0107  LXI  SP,0257 .STKTOP
0104  SHLD  0215 .OLDSP
0103  DAD  SP
0100  LXI  H,0000
```
Note that in this particular case, only 11 instructions were executed before the BDOS call, and thus the full 256 instruction capacity had not been exceeded. In fact, the backtrace shown above gives the complete history of the DUMP execution, from the first instruction at address 0100. You can then proceed to execute the DUMP program further by simply typing:

`UFFFF,.COLLECT`

with a break at the following call on BDOS. Given that the program execution is to stop on the 20th call on BDOS, type the pass command:

`P.BDOS,#20`

to set the pass count at 20 (decimal). Enter the command:

`UFFFF,.COLLECT`

if intermediate passes are to be traced. Alternatively, type the command:

`-UFFFF,.COLLECT`

to disable intermediate traces. In either case, execution stops at the 20th BDOS call, and you can enter the display command:

`C.DISPLAY`

to view the trace to this particular BDOS call.

Abort long typeouts by typing any keyboard character during the display. The ctl-S key freezes the display during output. Finally, recall that you can issue "C.DISPLAY" any number of times to reproduce the backtrace since the command does not clear the TRACE buffer.

You can also use the TRACE utility when the disassembler module is not present. In this case, the instruction addresses are listed in the trace, while the mnemonics are not included. For example, the sequence of commands shown below loads the TRACE utility without the disassembler module, followed by the DUMP program without its symbol table:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>Load the SID Program</td>
</tr>
<tr>
<td>-A</td>
<td>Remove the Disassembler</td>
</tr>
<tr>
<td>ITRACE.UTL</td>
<td>Ready the TRACE Utility</td>
</tr>
<tr>
<td>R</td>
<td>Read the TRACE Utility</td>
</tr>
<tr>
<td>IDUMP.COM</td>
<td>Load the DUMP Program</td>
</tr>
</tbody>
</table>

In this case, the TRACE utility prints the following sign-on message:

"-A" IN EFFECT, ADDRESS BACKTRACE

All Information Presented Here is Proprietary to Digital Research
The backtrace information is subsequently displayed in the format:

\[
\text{addr-255} \text{ addr-254} \text{ addr-253} \ldots \text{ addr-248} \\
\text{addr-247} \text{ addr-246} \text{ addr-245} \ldots \text{ addr-240} \\
\ldots \\
\text{addr-007} \text{ addr-006} \text{ addr-005} \ldots \text{ addr-000}
\]
Section 5
SID Sample Debugging Sessions

This section contains several examples of SID debugging sessions. The examples are based upon a "bubble sort" of a byte value list. The bubble sort program is first listed in its undebugged form. A series of test, edit, and reassembly processes are shown which result in a final debugged program. In each case, the operator interaction with CP/M, ED, MAC, or SID is shown in normal type, while comments on each of the processes are given alongside in italics.

The dialogue that follows contains the following sequence of operations:

(1) TYPE SORT.PRN
    Lists initial SORT program.
(2) TYPE SORT.SYM
    Shows the SORT symbol table.
(3) TYPE SORT.HEX
    Shows the SORT HEX file.
(4) SID SORT.HEX SORT.SYM
    1st debugging session.
(5) ED SORT.ASM
    1st re-edit of SORT program.
(6) MAC SORT
    1st reassembly of SORT.
(7) TYPE SORT.SYM
    Shows new symbol table.
(8) SID SORT.HEX SORT.SYM
    2nd debugging session.
(9) ED SORT.ASM
    2nd re-edit of SORT program.
(10) MAC SORT
    2nd reassembly of SORT.
(11) SID SORT.HEX SORT.SYM
    3rd debugging session.
(12) ED SORT.ASM
    3rd re-edit of SORT.
(13) MAC SORT
    3rd reassembly of SORT.
(14) LOAD SORT
    Create a COM file for SORT.
(15) SID SORT.COM SORT.SYM
    4th debugging session.
(16) SID SORT.COM SORT.SYM
    Re-entry to SID for debugging.
(17) SID SORT.COM SORT.SYM
    Re-entry to SID for debugging.
(18) SID SORT.COM SORT.SYM
    Re-entry to SID for debugging.
(19) ED SORT.ASM
    4th re-edit of SORT.
(20) MAC SORT
    4th reassembly of SORT.
(21) SID SORT.HEX SORT.SYM
    5th debugging session.
(22) ED SORT.ASM
    5th re-edit of SORT.
(23) MAC SORT
    5th reassembly of SORT.
(24) SID SORT.HEX SORT.SYM
    6th debugging session.
(25) ED SORT.ASM
    6th (last) re-edit of SORT.
(26) MAC SORT $+$
    6th (last) reassembly.

Following the debugging sessions, the final corrected SORT program is given in its debugged form.
Three separate debugging sessions are then shown that use the HIST and TRACE utilities to monitor the execution of the tested SORT program. The operations shown here include:

(27) SID HIST.UTL Load the HIST Utility.
(28) SID TRACE.UTL Load the TRACE Utility.
(29) SID Load SID, TRACE follows.

As a final example, a simple program that calls the BDOS is listed, followed by a single debugging session. This particular example shows the actions of SID when subroutines are traced, followed by calls on the CP/M BDOS. The operations in this case are:

(30) TYPE IO.PRN List the IO program
(31) SID IO.HEX IO.SYM Enter SID for debugging
TYPE SORT.PRN

; SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
; ELEMENTS OF 'LIST' ARE PLACED INTO
; DESCENDING ORDER USING BUBBLE SORT

0100 ORG 100H ;BEGINNING OF TPA
0000 = REBOOT EQU 000OH ;CP/M REBOOT LOCATION

0100  213801    SORT: LXI H,SW
0103  3601     MVI M,1 ;SW = 1
0105  213901    LXI H,I ;INDEX TO SORT LIST
0108  3600     MVI M,0 ;I = 0

; COMPARE I WITH ARRAY SIZE
COMP: ;HL ADDRESS INDEX I
010A  3A6201    LDA N ;LENGTH OF VECTOR
010D  8E     CMP M ;CHECK FOR N+I
010E  C21901    JNZ CONT ;CONTINUE IF UNEQUAL

; END OF ONE PASS THROUGH LIST
0111  213801    LXI H,SW ;NO SWITCHES?
0114  7E     MOV A,M ;FILL A WITH SW
0115  87     ORA A ;SET FLAGS
0116  C30000    STOP: JMP REBOOT ;RESTART CCP

; CONTINUE THIS PASS
CONT: ;ADDRESSING I, SO LOAD LIST(I)
0119  5F     MOV E,A ;LOW(I) TO E REGISTER
011A  1600    MVI D,O ;HIGH(I) = 0
011C  215A01    LXI H,LIST ;BASE OF LIST
011F  19     DAD D ;ADDR LIST(I)
0120  7E     MOV A,M ;LIST(I) IN A REGISTER
0121  23     INX H ;ADDR OF LIST(I+1)
0122  8E     CMP M ;LIST(I):LIST(I+1)
0123  DA3101    JC INCI ;SKIP IF PROPER ORDER

; CHECK FOR LIST(I) = LIST(I+1)
0126  CA3101    JZ INCI ;SKIP IF EQUAL

; ITEMS ARE OUT OF ORDER, SWITCH
0129  4E     MOV C,M ;OLD LIST(I+1) TO C
012A  77     MOV M,A ;NEW LIST(I+1) TO M
012B  28     DCX H ;ADDR LIST(I)
012C  71     MOV M,C ;NEW LIST(I) TO M
012D  213801    LXI H,SW ;SWITCH COUNT IS SW
0130  34     INR M ;SW = SW + 1

; INC: ;INCREMENT INDEX I
0131  213901    LXI H,I
0134  34     INR M ;I = I + 1
0135  C30A01    JMP COMP ;TO COMPARE I WITH N-1

; DATA AREAS
0138  DW: DS 1 ;SWITCH COUNT
0139  I: DS 1 ;INDEX
013A  DS 32 ;16 LEVEL STACK

STACK:
015A  060340A0BLIST: DB 5,3,4,10,8,130,10,4
0162  08     M: DB $-LIST ;LENGTH OF LIST
0163  END

All Information Presented Here is Proprietary to Digital Research
SID User's Guide

5 SID Sample Debugging Sessions

2 TYPE SORT.SYM
010A COMP 0119 CONT 0139 I 0131 INCI 015A LIST
0152 N 0000 REBOOT 0100 SORT 015A STACK 0116 STOP
0138 SW

3 TYPE SORT.HEX
1001000021380136012139013603A6021BEC21997
10011000012138017B7C30005F1600215A0119B2
100120007E23E0A3101CA31014E772B713801AD
01013000341390134C30A0136
00915A0005030A0A8020A0408E6
0000000000

4 SID SORT.HEX SORT.SYM Start SID with HEX and SYM files
SID VERS 1.4
SYMBOLS
NEXT PC END
0163 0100 55B7 Next free address is 183, Program Counter is 100
#0.LIST,**N=1 and end of TPA is 55B7
015A: 05 03 04 0A 08 82 ...... Display initial list of items to sort
0160: 0A 04 ...... #6, STOP Execute test program until "STOP" symbol address encountered
#0116 .STOP Now at the STOP address, examine data list:
#0.DLIST,**N=1
015A: 05 03 04 0A 08 82 ...... Hasn't changed!
0160: 0A 04 ...... #XP where is the program counter?
reset PC back to beginning and try again with trace on:
#T10
----- A=01 B=0000 D=0008 H=0138 S=0100 P=0100 LXI H,0138 .SW
----- A=01 B=0000 D=0008 H=0138 S=0100 P=0103 MVI M,01 .SW
----- A=01 B=0000 D=0008 H=0138 S=0100 P=0105 LXI H,0139 .I
----- A=01 B=0000 D=0008 H=0139 S=0100 P=0108 MVI M,00 .I
COMP:
----- A=01 B=0000 D=0008 H=0139 S=0100 P=010A LDA 0162 .N
----- A=08 B=0000 D=0008 H=0139 S=0100 P=010D CMP M=00 .I
----- A=08 B=0000 D=0008 H=0139 S=0100 P=010E JNZ 0119 .CONT
CONT:
----- A=08 B=0000 D=0008 H=0139 S=0100 P=0119 MOV E,A LIST(1), LIST(+1)
----- A=08 B=0000 D=0008 H=0139 S=0100 P=011A MVI D,00
----- A=08 B=0000 D=0008 H=0139 S=0100 P=011C LXI H,015A .LIST
----- A=08 B=0000 D=0008 H=0139 S=0100 P=011F DAD D
----- A=08 B=0000 D=0008 H=0162 S=0100 P=0120 MOV A,M .N
----- A=08 B=0000 D=0008 H=0162 S=0100 P=0121 INX H
----- A=08 B=0000 D=0008 H=0163 S=0100 P=0122 CMP M=58
----- A=08 B=0000 D=0008 H=0163 S=0100 P=0123 JC 0131 .INCI
INCI:
----- A=08 B=0000 D=0008 H=0163 S=0100 P=0131 LXI H,0139 .I
*0134 Looks like we've discovered a bug! We have entered at "CONT"
#60

5 ED SORT.ASM
"A Bring all the text into memory
"V Enter Verify mode for line numbers, then find the place to change
#1 *FADDRESSING
28: *OLT
28: ; ADDRESSING I, SO LOAD LIST(I) Delete the line
28: *KT
28: MOV E,A ;LOW(I) TO E REGISTER
28: *I
28: LDA I ;LOAD I TO A REGISTER Insert the change
28: "E Terminate the editing session

All Information Presented Here is Proprietary to Digital Research
MAC SORT
CP/M MACRO ASSEM 2.0
0166
001H USE FACTOR
END OF ASSEMBLY

Re-assemble the SORT program

TYPE SORTSYM
010A COMP 0119 CONT 013C I 0134 INCI 0150 LIST
0165 N 0000 REBOOT 0100 SORT 0150 STACK 0116 STOP
0138 SW

Here's the symbol table:

SID SORT.HEX SORT.SYM
SID VERS 1.4
SYMBOLS
NEXT PC END
0166 0100 5587
#P.STOP
#G
01 PASS 0116 .STOP
------ A=7C B=0008 D=0081 H=0138 S=0100 P=0116 .MP 0000 .REBOOT
*0000 .REBOOT
#H=N
0082 #130
#O.LIST,+7
0150: 03 04 05 ...
0160: 08 0A 0A 04 08 ....
#ISORT.HEX
#R
NEXT PC END
0166 0100 5587
#P
#G
01 0116 .STOP
#P.SORT,FF
#G

Let's try again, load the HEX and SYM files

Set a "pass point" at STOP to prevent reboot
Start (unmonitored) execution

We made it to the STOP label, check values

A=7C B=0008 D=0081 H=0138 S=0100 P=0116 .MP 0000 .REBOOT

What's the value of the byte variable N?
130? Very strange! How did that happen?
Oh well, let's look at the data values:

They are almost sorted, looks like we have
some trouble near the end of the vector,
let's reload the machine code and try
again:

Program counter remains at 0100, what
are the active pass points?
The one at STOP remains set, let's also
monitor the SORT loop point, but not
break right away.

Here's the first time through SORT

It stopped immediately! It doesn't look good!

We know there should have been several loops
through the SORT label, since the data is
unordered. Let's try again — reload the code
(note that the reload is necessary here, since
the data is initialized in the code area).

What active pass points exist?
Wait a minute — referring back to the
original listing, it appears that the code
preceding the STOP label is incomplete:
there should be a conditional jump back to
the SORT label - maybe that's why the program
never makes it back!
ED SORT.ASM

Oh well, back to the editor for a quick fix. Append all text (A), and enter Verify mode (V). Then find STOP.

1: *FSTOP:
24: *OLT
24: STOP: JMP REBOOT ;RESTART CCP
24: *.
23: ; END OF SORT PROCESS, REBOOT
23: *I and enter insert mode (I)
23: JNZ CONT ;CONTINUE IF NOT EQUAL
24: .ct-Z, and "return"
25: E
26: wait, I forgot the ct-Z, now I've got the E command in
26: *. my input buffer. Type the ct-Z, go back up one line,
25: E delete the E, then end the edit
25: *XT
25: ; END OF SORT PROCESS, REBOOT
25: *E OK, we made the change, now re-assemble

MAC SORT
CP/M MACRO ASSEM 2.0
0169
001H USE FACTOR
END OF ASSEMBLY

SID SORT.HEX SORT.SYM
SID VERS 1.4
SYMBOLS
NEXT PC END
0169 0100 5587
P.SORT,FF
P.STOP
#P
FF 0100 .SORT
01 0119 .STOP
#G
FF PASS 0100 .SORT
----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI H,013E .SW
01 PASS 0119 .STOP
-Z- E= A=00 B=006A D=0007 H=013E S=0100 P=0119 JMP 0000 .REBOOT
*0000 .REBOOT
N=N
0008 #B
#D,LIST,+N
0160: 01 01 03 04 04 05 07 08 08 ........ These values look a bit strange? Try again:
#I.SORT.HEX
#R
NEXT PC END
0169 0100 5587
#D,LIST,+N-1
0160: 05 03 04 0A 0A 82 0A 04 ........
#L,CONT
CONT:
011C LDA 013F .I
011F MOV E,A
0120 MKI D,00
0122 LXI H,0160 .LIST
0125 DAD D
0126 MOV A,M
0127 INX H
0128 CMP M
0129 JC 0137 .INCI
012C JZ 0137 .INCI
012F MOV C,M
#P12F,FF
Here's where the switch occurs, let's set a pass point here and watch the data addresses:
#P
FF 0100 .SORT
01 0119 .STOP
FF 012F

Invoke the macro assembler with SORT as input.
Set a pass point at sort, with a high count.
Execute the test program
First time through SORT label:
Stopped again! Arghgh!
Let's look at some values:
N=8, looks better than last time
Machine code reloaded, display initial values:
Let's take a look at the process of switching two data items - the code appears down below the "CONT" label, so we'll disassemble a portion of the program.
SID User's Guide

5 SID Sample Debugging Sessions

#6

FE PASS 0100 .SORT

J-Z: A=00 B=006A D=0007 H=013E S=0100 P=0100 LXI .H,013F .SW

Switching at address 161, looks OK.

FF PASS 012F

---I A=05 B=006A D=0000 H=0161 S=0100 P=012F MOV C,M

FE PASS 012F

Switching at 162, looks good.

FD PASS 012F

164 is the next to switch, looks good.

FD PASS 012F

166 is probably the next one.

*0130

So what's wrong? This section of code seems to work.

#-P

#1SORT.HEX

#-

NEXT PC END 0169 0100 55B7

#L_CONT=5

0121 NOP

0122 LXI H,0160 .LIST

0125 DAD D

0126 MOV A,M

0127 INX H

Here's the code where the element switching occurs, let's watch the program switch the first element:

0128 CMP M

0129 JC 0137 .INCI

012C JZ 0137 .INCI

012F MOV C,M

0130 MOV M,A

0131 DCX H

*0129

OK, here we are, ready to test and switch, if necessary.

*110

---I A=05 B=0000 D=0000 H=0161 S=0100 P=0129 JC 0137 .INCI

---I A=05 B=0000 D=0000 H=0161 S=0100 P=012C JZ 0137 .INCI

---I A=05 B=0000 D=0000 H=0161 S=0100 P=012F MOV C,M

---I A=05 B=0000 D=0000 H=0161 S=0100 P=0130 MOV M,A

---I A=05 B=0000 D=0000 H=0161 S=0100 P=0131 DCX H

---I A=05 B=0000 D=0000 H=0160 S=0100 P=0132 MOV M,C LIST

---I A=05 B=0000 D=0000 H=0160 S=0100 P=0133 LXI H,013E .SW

---I A=05 B=0000 D=0000 H=013E S=0100 P=0134 INS M=01 .SW

*0137 .INCI

Well, that went nicely - elements switched, SW=1

*0137 .INCI

0160: 03 05 04 0A 08 82 0A 04 ........

*0160: 03 05 04 0A 08 82 0A 04 ........

#H=1

0000 .REBOOT #0

0000 .REBOOT #0

#0, .INCI

Proceed to the INCI label

*0137 .INCI

Here we are, let's look at the data:

#0.LIST,+7

0160: 03 05 04 0A 08 82 0A 04 ........

0000 .REBOOT #0

Looks good, trace past the label and break

*0133 .INCI

0160: 03 04 ...

Looks good, proceed past INCI

*0137 .INCI

Here we are (again), how's the data?

*0137 .INCI

0160: 03 04 ...

Looks good, proceed past INCI

*0137 .INCI

Here we are (again), how's the data?

*0137 .INCI

0160: 03 04 ...

*0137 .INCI

0160: 03 04 ...

*0137 .INCI

0160: 03 04 ...

*0137 .INCI

0160: 03 04 ...

*0137 .INCI

0160: 03 04 ...

*0137 .INCI

0160: 03 04 ...

*0137 .INCI

0160: 03 04 ...

Stop

*0119 .STOP

Egad! Here we are at the the STOP label. Why aren't we making it back to SORT?

0160: 01 01 03 04 04 05 07 08 08 ........

Tsk! Tsk! The data's messed up again.

All Information Presented Here is Proprietary to Digital Research
Let's reload and try again.

Here's where the switch count is incremented:

INCI:
0137  LXI  H,013F .1
013A

*0136
#0.DLIST,+1
0160: 03  .
#U
Use V to move past break address.

*0137 .INCI
It's actually easier to use the pass point feature
if we want to view the action of the INR M,
since the P command stops execution after the
pass point is executed.

01 PASS 0136
----1 A=05 B=0003 D=0000 H=013E S=0100 P=0136 INR M=01 .SW
*0137 .INCI
SW = 2, looks good.
#0.DLIST,+1
0160: 03 04  ..
#N
Data values look good.
#0160 08 4
0169 0A .
#G
Let's change N to a smaller value so the program
doesn't loop so many times: 4 is a good number.
End input with "GO" at pass point

01 PASS 0136
----1 A=04 B=0008 D=0003 H=013E S=0100 P=0136 INR M=03 .SW
*0137 .INCI
Stopped at next instruction.
#0.DLIST,+1
0160: 03 04 05 08  ..
Data values so far.
#H=SW
0004 #4
SW value at this point is 4.
#FFFF
Let's watch it run for a few steps:

------ A=0A B=0008 D=0003 H=013E S=0100 P=0137 LXI H,013F .1
------ A=0A B=0008 D=0003 H=013F S=0100 P=013A INR M=03 .1
------ A=0A B=0008 D=0003 H=013F S=0100 P=013B JMP 010A .COMP
COMP:
------ A=0A B=0008 D=0003 H=013F S=0100 P=010A LDA 0168 .N
------ A=0B B=0008 D=0003 H=013F S=0100 P=0100 CMP M=04 .I
-Z-E1 A=04 B=0008 D=0003 H=013F S=0100 P=010E JNZ 011C .CONT
-Z-E1 A=04 B=0008 D=0003 H=013F S=0100 P=0111 LXI H,013E .SW
-Z-E1 A=04 B=0008 D=0003 H=013F S=0100 P=0114 MOV A,M .SW
-Z-E1 A=04 B=0008 D=0003 H=013E S=0100 P=0115 ORA A
------ A=04 B=0008 D=0003 H=013E S=0100 P=0116 JNZ 011C .CONT
CONT:
------ A=04 B=0008 D=0003 H=013E S=0100 P=011C LDA 013F .1
*011F
#GO
Very interesting! We seem to be
going back to "CONT" rather than "SORT."
Let's go back to the editor an fix it up.

This is a simple change: append all text, enter line
verify mode, find "ORA" and make the change:

22: *VOLT
22: ORA A ;SET FLAGS
22: */ return" to move down one line
23: JNZ CONT ;CONTINUE IF NOT EQUAL
23: *CONT1?SORT!VOLT ;Substitute SORT for CONT
23: JNZ SORT ;CONTINUE IF NOT EQUAL
23: */ return" to move down another line
24: ;
24: */ return" again.
25: ; END OF SORT PROCESS, REBOOT
25: *E End the edit

All Information Presented Here is Proprietary to Digital Research
5 SID Sample Debugging Sessions

13 MAC SORT
CP/M MACRO ASSEM 2.0
0169
001H USE FACTOR
END OF ASSEMBLY

Call out MAC for another assembly.

14 LOAD SORT
FIRST ADDRESS 0100
LAST ADDRESS 0168
BYTES READ 0047
RECORDS WRITTEN 01

Just for a little variation, we'll create a SORT.COM file for testing under SID.

15 SID SORT.COM SORT.SYM
SID VERS 1.4
Back to SID, using the COM and SYM files
SYMBOLS
NEXT PC END
0180 0100 5587
#P.STOP
#D.LIST,+N-1
Here's the original data:
0160: 05 03 04 0A 08 82 0A 04 ........
#G
Unmonitored GO
Oops! We didn't get control back, there must
be an infinite loop - we can get control back by
forcing a front panel RST 7 (interrupt 7),
or simply bail-out with a cold start.

63K CP/M VERS 1.3

16 SID SORT.COM SORT.SYM
SID VERS 1.4
Let's start again, but be a little more selective
in setting breakpoints.
SYMBOLS
NEXT PC END
0180 0100 5587
#P.STOP
#P.SORT,FF
#G
Set a pass point at STOP, as before
and one at SORT with a pass count of 255.
GO with pass trace disabled.

01 PASS 0100
Stopped with 255 passes through SORT - too many!
---- A=01 B=006A D=00FF H=013E S=0100 P=0100 LXI H,013E
*0103
#D.LIST,+N-1
How's the data?
0160: 03
#H=N
Hmmm... looks like N was destroyed.
0000 .REBOOT #0
#H=I
0000 .REBOOT #0
#G,.COMP
There's a good possibility that we're running off
the end of the LIST vector into the variable N,
let's stop at the COMP label and watch the end test.
*010A .COMP
#T5
---- A=01 B=006A D=00FF H=013F S=0100 P=010A LDA 0168 .N
---- A=00 B=006A D=00FF H=013F S=0100 P=0100 CMP M=00 .L
-Z-E1 A=00 B=006A D=00FF H=013F S=0100 P=010E JNZ 011C .CONT
-Z-E1 A=00 B=006A D=00FF H=013F S=0100 P=0111 LXI H,013E .SW
-Z-E1 A=00 B=006A D=00FF H=013E S=0100 P=0114 MOV A,M .SW
*0115 Hey, this isn't going to work! We'll be comparing
#GO LIST(N-1) with LIST(N), but the last LIST element is
at LIST(N-1). Let's try a quick fix.
SID SORT.COM SORT.SYM

SID VERS 1.4

SYMBOLS

NEXT PC END

O180 0100 5587

#L: COMP

#A10A

O10A LDA 0158 .N

O100 CMP M

O10E JNZ 011C .CONT

O111 LXI H,013E .SW

O114 MOV A,M

Here's the reference to N - let's change this to N-1 with a "hot patch" in memory, to see if it works, then we'll go back to the original source program and make the necessary changes. We're not using the area of memory starting at 0200, so patch a jump over the LDA instruction, and fix-up some patch code.

#A200

O200 LDA .N

O203 DCR A

O204 CMP M

O205 JNZ .CONT

O208 JMP 111

O20B #P205,FF

#P,STOP

#P111,FF

#S,N

O158 08 4

O169 00 0

Everything is ready, let's go...

FF PASS 0205

First pass through the patch code:

---E1 A=03 B=0000 D=0000 H=013F S=0100 P=0205 JNZ 011C .CONT

FE PASS 0205

Went to CONT that time, second pass:

---A A=03 B=0000 D=0000 H=013F S=0100 P=0205 JNZ 011C .CONT

FD PASS 0205

Went to CONT again, next pass:

---A A=03 B=0004 D=0001 H=013F S=0100 P=0205 JNZ 011C .CONT

FC PASS 0205

And so-forth:

-2-EI A=03 B=0002 D=0013 F=0100 P=0205 JNZ 011C .CONT

FF PASS 0111

Must be the end of one cycle:

-2-EI A=03 B=0002 D=0013 F=0100 P=0111 LXI H,013E .SW

FB PASS 0205

Now back through the patch code:

---E1 A=03 B=0004 D=0002 H=013F S=0100 P=0205 JNZ 011C .CONT

FA PASS 0205

---A A=03 B=0000 D=0000 H=013F S=0100 P=0205 JNZ 011C .CONT

FF PASS 0205

---A A=03 B=0004 D=0001 H=013F S=0100 P=0205 JNZ 011C .CONT

FB PASS 0205

---E2-EI A=03 B=0002 D=0013 F=0100 P=0205 JNZ 011C .CONT

FE PASS 0111

---E2-EI A=03 B=0004 D=0013 F=0100 P=0111 LXI H,013E .SW

0314

This is getting monotonous again, so push the "return" key to stop the action.

0160: 03 04 05 0A .... Data looks good, run in monitor mode:

-UFFF

-2-EI A=03 B=0004 D=0002 H=013E S=0100 P=0114 MOV A,M

0138

Push the "return" key to abort early.

H

Value of N is still 4 (that's nice!)

0004 #4

Value of I is currently 2. This program should have stopped, but didn't for some reason.
SID User's Guide

SID VERS 1.4

Let's try another approach. Suppose we construct a really trivial case, and we'll set
LIST(0) = 0, LIST(1) = 1

\#5.N
0168 08 2
0169 00 .
\#5.LIST
0160 05 0
0161 03 1
0162 04

Things are ready to go, run completely traced:

P.STOP
#TFFFF

\[\begin{align*}
A &= 00 & B &= 0000 & D &= 0000 & H &= 0000 & S &= 0100 & P &= 0100 & LXI & H,013E & SW \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013E & S &= 0100 & P &= 0103 & MVI & M,01 & SW \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013E & S &= 0100 & P &= 0105 & LXI & H,013F & .I \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 0108 & MVI & M,00 & .I
\end{align*}\]

COMP:

\[\begin{align*}
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010A & LDA & 0168 & .N \\
----- & A &= 02 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010D & CMP & M=00 & .I \\
----- & A &= 02 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010E & JNZ & 011C & .CONT
\end{align*}\]

CONT:

\[\begin{align*}
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 011F & MOV & E,A \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 0120 & MVI & D,00 \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 0122 & LXI & H,0160 & .LIST \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 0160 & S &= 0100 & P &= 0125 & DAD & D \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 0160 & S &= 0100 & P &= 0126 & MOV & A,M & .LIST \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 0161 & S &= 0100 & P &= 0127 & INX & H \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 0161 & S &= 0100 & P &= 0128 & CMP & M=01 \\
C-ME: & A &= 00 & B &= 0000 & D &= 0000 & H &= 0161 & S &= 0100 & P &= 0129 & JC & 0137 & .INCI
\end{align*}\]

INCI:

Not switched!

C-ME: & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 013A & INR & M=00 & .I \\
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 013B & JMP & 010A & .COMP
\]

COMP:

\[\begin{align*}
----- & A &= 00 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010A & LDA & 0168 & .N \\
----- & A &= 02 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010D & CMP & M=01 & .I \\
----- & A &= 02 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010E & JNZ & 011C & .CONT
\end{align*}\]

CONT:

\[\begin{align*}
----- & A &= 01 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 011F & MOV & E,A \\
----- & A &= 01 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 0120 & MVI & D,00 \\
----- & A &= 01 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 0122 & LXI & H,0160 & .LIST \\
----- & A &= 01 & B &= 0000 & D &= 0000 & H &= 0160 & S &= 0100 & P &= 0125 & DAD & D \\
----- & A &= 01 & B &= 0000 & D &= 0000 & H &= 0161 & S &= 0100 & P &= 0126 & MOV & A,M \\
----- & A &= 01 & B &= 0000 & D &= 0000 & H &= 0161 & S &= 0100 & P &= 0127 & INX & H \\
----- & A &= 01 & B &= 0000 & D &= 0000 & H &= 0162 & S &= 0100 & P &= 0128 & CMP & M=04 \\
\end{align*}\]

INCI:

Not switched (again!)

\]

COMP:

\[\begin{align*}
----- & A &= 02 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010D & CMP & M=02 & .I \\
----- & A &= 02 & B &= 0000 & D &= 0000 & H &= 013F & S &= 0100 & P &= 010E & JNZ & 011C & .CONT
\end{align*}\]

\]


*0103

All Information Presented Here is Proprietary to Digital Research
SID User's Guide

5 SID Sample Debugging Sessions

19
ED SORT.ASM
*#AVFSORT:ZOLT
8: SORT: LXI H, SW
8: *-
7: ;
7: *2
9: MVI M, 1 ; SW = 1
9: *2SLIZ0/IZOLT
9: MVI M, 0 ; SW = 0
9: *-
8: SORT: LXI H, SW
8: *1
8: MVI A, 1
9: STA SW ; SW = 1 FIRST TIME THRU
10:
10: *

20
MAC SORT
CP/M MACRO ASSEM 2.0
016E
001H USE FACTOR
END OF ASSEMBLY

21
SID SORT.HEX SORT.SYM
SID VERS 1.4
SYMBOLS
NEXT PC END
016E 0100 5587
#0.LIST, ++N
Here's the initial data:
0165: 05 03 04 0A 08 08 04 08 ...........
#6, STOP

*011E .STOP
#0.LIST, ++N
We made it, here's the data:
0165: 03 04 04 05 08 08 0A 0A 08 7B 82 .........
0170: E5 .
#ISORT.HEX
#R
NEXT PC END
016E 0100 5587
#P.SORT
#G

01 PASS 0105 .SORT Here's the first pass through SORT:
-2-E- A=01 B=0004 D=000A H=0143 S=0100 P=0105 LXI H,0143 .SW
*0108 Break at 0108, check value of N:
#H=N
0008 #8
#G
OK initially, continue the execution with G.

01 PASS 0105 .SORT We have passed through the data once:
-------- A=75 B=002A D=007A H=0143 S=0100 P=0105 LXI H,0143 .SW
*0108 Clear all pass points.
#P
#G..INCI

0078 #123 .'.
#ISORT.HEX
#R
NEXT PC END
016E 0100 5587
#G..INCI

01 PASS 0105 .SORT Oops! The initial pass point is still set.
-------- A=01 B=002A D=007A H=0143 S=0100 P=0105 LXI H,0143 .SW
*0108 Now, try again:
#P
#G..INCI

*013C .INCI
#H=N
0008 #8
#G..CONT
Go to the CONT label, then stop at INCI.

*0121 .CONT
#G..INCI

Back to the editor—change the entry code to initialize SW

Re-assembly, again

We've fixed the SW initialization problem, which should halt the program at the proper time, but we may still have a problem with the end of LIST test (remember that "hot patch").
Here's the initial data: N is altered during execution.
Let's reload and make sure we know what the problem is—Set a pass point at SORT, check N
Back at INCI now. Check value of N

Remains at 8. If we keep this up, we'll be typing break addresses all day. We can run the next few passes through INCI automatically by setting a pass count (use 6 in this case), then run with -G to disable intermediate traces. We now stop 6 iterations later:

```assembly
01 PASS 013C
---E- A=82 8=0004 D=0006 H=0143 S=0100 P=013C LXI H,014
```

Check N: remains at 8, then check I to compare passes: I=0,1,2,3,4,5,6 has been executed. We are now about to set I = 7, but the test at COMP is "JNZ" which allows execution one too many times (which we already know about).

```
22 ED SORT.ASM
*#AV
1:  *FLDA
17:  *OLT
17:  LDA  N ;LENGTH OF VECTOR
17:  *  ;"return" to go to next line
18:  CMP  M ;CHECK FOR N=I
18:  *I  ;Insert the instruction before the "CMP" opcode.
18:  DCR  A ;N-1 IN A REGISTER
19:  ; (NOTE THAT N MUST BE 2 OR LARGER)
20:  ctt-Z
20:  *F*I
49:  *D*T
49:  MOV  M,A ;NEW LIST*I-C-DI(IZOLT
49:  MOV  M,A ;NEW LIST(I+1) TO M Looks better now.
49:  *F32
64:  *OLT
64:  DS 32 ;16 LEVEL STACK
64:  *2K
64:  ;
64:  *E
```

Back to the editor, change the end of LIST test to compare I with N-I rather than N.

Now a little clean-up work - there is a typo in a comment line at address 012A in the listing:

```
23 MAC SORT
CP/M MACRO ASSEM 2.0
014F
001H USE FACTOR
END OF ASSEMBLY
```

Re-assemble the source program.

```
24 SID SORT.HEX SORT.SYM
SID VERS 1.4
SYMBOLS
NEXT PC END
014F 0100 55BF
#D.LIST,**N
0146: 05 03 04 0A 08 82 0A 04 08 ...........
#6,STOP
?
#6,..STOP
```

Back to SID - this should be the last time!

Initial data:

```
0146: 05 03 04 0A 08 82 0A 04 08 ...........
```

Ok, ok. Let's try it with an "address reference" to the label STOP:

```
011F .STOP
```

That's better, now look at the data:

```
hooray! It's finally sorted.
```

```
0146: 03 04 04 05 08 0A 0A 82 08 ...........
```

Back at INCI now. Check value of N

Remains at 8. If we keep this up, we'll be typing break addresses all day. We can run the next few passes through INCI automatically by setting a pass count (use 6 in this case), then run with -G to disable intermediate traces. We now stop 6 iterations later:

```assembly
01 PASS 013C
---E- A=82 8=0004 D=0006 H=0143 S=0100 P=013C LXI H,014
```

Check N: remains at 8, then check I to compare passes: I=0,1,2,3,4,5,6 has been executed. We are now about to set I = 7, but the test at COMP is "JNZ" which allows execution one too many times (which we already know about).

```
```
At this point, we have checked-out this particular SORT program using this particular set of data items. This does not, of course, mean that the program is fully debugged. There could be cases which are not tested properly since we have not included all boundary conditions (the data items 00 and FF, for example, should be included). Further, there are program segments which could be incorrect, but which have no negative effects on the program. The initialization of SW to the value 1 before the label SORT, for example, does not affect the program, but is superfluous. We now have a program which appears to work, but must undergo further tests before it is considered a production program.
SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
ELEMENTS OF 'LIST' ARE PLACED INTO
ASCENDING ORDER USING BUBBLE SORT

0100 = ORG 100H  ;BEGINNING OF TPA
0000 = REBOOT EQU 000H  ;CP/M REBOOT LOCATION

0100 3E01
0102 324401
0105 214401
0108 3600
010A 214501
010D 3600

; COMPARE I WITH ARRAY SIZE
COMP: ;HL ADDRESS INDEX I
010F 3A4E01
0112 3D
0113 BE
0114 C22201

; END OF ONE PASS THROUGH LIST
0117 214401
011A 7E
011B 37
011C 20501
011F C30000

STOP: ;JMP REBOOT ;RESTART CCP

; CONTINUE THIS PASS
CONT: 
0122 3A4501
0125 5F
0126 1600
0128 214601
012B 19
012C 7E
012E BE
012F 0A3D01

; CHECK FOR LIST(I) = LIST(I+1)
0132 CA3D01
0135 4E
0136 77
0137 2B
0138 71
0139 214401
013C 34

; INCI: ;INCREMENT INDEX I
013D 214501
0140 34
0141 C30F01

; DATA AREAS
0144 SW: DS 1  ;SWITCH COUNT
0145 I: DS 1  ;INDEX

0146 503040A08LIST: DB $, 9, 3, 4, 10, 8, 130, 10, 4
014E 08

END

010F COMP
014E N
0000 REBOOT
0105 SORT
011F STOP
0144 SW
SID HIST.UTL
SID VERS 1.4
TYPE HISTOGRAM BOUNDS 100,200
.INITIAL = 5221
.COLLECT = 5224
.DISPLAY = 5227
#ISORT.HEX SORT.SYM
#R
SYMBOLS
NEXT PC END
0600 0100 5187
#P.STOP
#P.SORT,3
#-G

01 PASS 0100
----- A=02 B=0004 D=0006 H=013F S=0100 P=0100 LXI H,013F
*0103
#-P.SORT
#OFF.Collect
----- A=02 B=0004 D=0006 H=013F S=0100 P=0103 MV1 M,01 .SW
*0127
#C.DISPLAY
HISTOGRAM:
ADDR  RELATIVE FREQUENCY, LARGEST VALUE = 0309
0100  ****
0104  **
0108  ******************  most frequently executed address:
010C  ********************************
0110  **
0114  ****

...
011C  **********************
0120  *********************************************
0124  *****************************************
0128  **********************
012C  ****
0130
0134
0138  **********************
013C  **********************

...
0200 *

#L10C
010C LXI B,8930
010F JNZ 011D .CONT
0112 LXI H,013F .SW
0115 MOV A,M
0116ORA A
0117JNZ 0100 .SORT
STOP:
011A JMP 0000 .REBOOT
#L11C
011C NOP
CONT:
011D LOA 0140 .I
0120 MOV E,A
0121 MVI 0,00
0123 LXI H,0161 .LIST
0126 DAD D
0127 MOV A,M
0128 INX H
0129 CMP M
012A JNC 0138 .INCI
012D JZ 0138 .INCI

Start SID with the HIST utility
Monitor 0100 through 0200.
Enter point addresses in HIST.
Load the SORT program with symbols.
Program loaded, now loading symbols.
Permanent break at STOP address.
Execute to "steady state" conditions by passing the SORT label three times before break.
"-G" prevents intermediate pass traces.
We're now at the third pass through SORT.
Remove the pass point at SORT, run monitored from this point for OFF steps, collect data.
Stopped after OFFSteps, display collected data:

What's happening around the most frequently executed address?
This is where the end of LIST test takes place, so it is reasonable that this segment of code would be executed heavily. We could improve performance by reducing the length of this segment. The value of N-1 could, for example, be maintained in register C throughout the computations, while the value of i could be kept in register E, with 00 in D.
There is also heavy execution around location 011C.

This is where we go on each element comparison whether we switch elements or not.
SID TRACED.UTL
SID VERS 1.4
INITIAL = 5321
COLLECT = 5324
DISPLAY = 5327
READY FOR SYMBOLIC BACKTRACE
#ISORT.HEX SORT.SYM
R
SYMBOLS
NEXT PC END
0600 0100 5287
#P.STOP
#P.CONT.3
#UFFF..COLLECT
----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI H,013F .SW
03 PASS 011D .CONT
-i A=07 B=0000 D=0000 H=0140 S=0100 P=011D LDA 0140 .I
02 PASS 011D .CONT
----i A=07 B=0003 D=0000 H=0140 S=0100 P=011D LDA 0140 .I
01 PASS 011D .CONT
----i A=07 B=0004 D=0000 H=0140 S=0100 P=011D LDA 0140 .I
*0120
#C.DISPLAY
BACKTRACE:
CONT:
011D LDA 0140 .I
010F JNZ 011D .CONT
010E CMP M
010D DCR A
COMP:
010A LDA 0169 .N
013C JMP 010A .COMP
013B INR M
INCI:
0138 LXI H,0140 .I
0137 INR M
0134 LXI H,013F .SW
0133 MOV M,C
0132 DCR H
0131 MOV M,A
0130 MOV C,M
0120 JZ 0138 .INCI
012A JC 0138 .INCI
0129 CMP M
0128 INX H
0127 MOV A,M
0126 DAD D
0123 LXI H,0161 .LIST
0121 MVI D,00
0120 MOV E,A
CONT:
011D LDA 0140 .I
010F JNZ 011D .CONT
010E CMP M
010D DCR A
COMP:
010A LDA 0169 .N
#60

Load the TRACE utility with SID.

TRACE entry points.

Indicates that assembler/disassembler is present.

Ready the SORT program and symbol table.

Load program and symbols to memory.

Permanent break at the STOP label.

Pass through CONT three times before stopping.

Untrace mode, print intermediate pass points.

Stopped on the third pass.

Display the backtrace from CONT.

Most recently executed instruction.

(abort with "return")
Start SID without loading any programs.

Remove assembler/disassembler package.

Ready the TRACE utility.

Read the TRACE package to memory.

TRACE entry point addresses.

No assembler/disassembler present.

Read the SORT program

Read to memory.

Permanent break at STOP address,
pass point at CONT with pass count 3
Run monitored, collect data, no intermediate

Stopped on third pass through CONT

0100 315401
0103 C31501
JMP START ;START EXECUTION

0106 0E02
MVI C,CONOUT;CONSOLE OUTPUT #
0108 5F
MOV E,A ;CHARACTER TO E
0109 C30500
JMP BOOS ;RET THROUGH BOOS

WRMSG: ;WRITE MESSAGE STARTING AT HL 'TTL 00
MOV A,M ;NEXT CHARACTER
010C 7E
ORA A ;DO?
010E 8B
RR Z ;RETURN IF SO
010F C06001
CALL WRCHAR ;OTHERWISE WRITE IT
0112 C3C001
JMP WRMSG ;FOR ANOTHER CHARACTER

START: ;BEGINNING OF MAIN PROGRAM
0115 212A01
LXI H,WALLAMSG ;PART 1 OF MESSAGE
0118 C0C001
CALL WRMSG ;WRITE IT
011B 212A01
LXI H,WALLAMSG ;PART 2 OF MESSAGE
011C C0C001
CALL WRMSG ;WRITE IT
0121 213001
LXI H,WASHMSG ;PART 3 OF MESSAGE
0124 C0C001
CALL WRMSG
0127 C00000
STOP: ;JMP REBOOT ;STOP THE PROGRAM

DATA AREAS
WALLAMSG:
012A 57414C4C41
DB 'WALLA'
WASHMSG:
0130 57415348
DB 'WASH'
0134 DS 32 ;16 LEVEL STACK
0154 STACK: END
SID User's Guide  5  SID Sample Debugging Sessions

SID 10.HEX 10.SYM
SID VERS 1.4  Load the test program using the HEX and SYM files.
SYMBOLS
NEXT  PC  END
0134  0100  55A9
#6, .WRMSG  GO from 0100 to the first call on WRMSG

*010C .WRMSG  Now trace from the WRMSG subroutine:

*💬*: #1000
----- A=57  B=0000  D=0000  H=012A  S=0152  P=010C  MOV  A,M .WALLAMSG
----- A=57  B=0000  D=0000  H=012A  S=0152  P=010D  ORA  A
----- A=57  B=0000  D=0000  H=012A  S=0152  P=010E  RZ
----- A=57  B=0000  D=0000  H=012A  S=0152  P=010F  CALL 0106 .WCHAR  First
WCHAR:
----- 57  B=0000  D=0000  H=012A  S=0150  P=0106  MOV  C,02  with 57 ("W")
----- A=57  B=0002  D=0000  H=012A  S=0150  P=0108  MOV  E,A
----- A=57  B=0002  D=0057  H=012A  S=0150  P=0109  JMP  0005 .BDOS

BDOS:
----- A=57  B=0002  D=0057  H=012A  S=0150  P=0005  JMP  55AA  Function # 2,
----- A=57  B=0002  D=0057  H=012A  S=0150  P=55AA  JMP  5CA4  Character "W"
----- A=57  B=0002  D=0057  H=012A  S=0150  P=5CA4  XTHL
----- A=57  B=0002  D=0057  H=0112  S=0150  P=5CA5  SHLD  60B2  (SID code to
----- A=57  B=0002  D=0057  H=0112  S=0150  P=5CA8  XTHL  intercept call)
----- A=57  B=0002  D=0057  H=012A  S=0150  P=5CA9  JMP  60E0  = first character
-Z,E- A=00  B=0000  D=0200  H=793B  S=0152  P=0112  JMP  010C .WRMSG  now we're
WCHAR:
----- 57  B=0000  D=0200  H=793B  S=0152  P=0112  MOV  A,M .WALLAMSG  back to our
-Z,E- A=00  B=0000  D=0200  H=793B  S=0152  P=0100  ORA  A  program, with
-Z,E- A=00  B=0000  D=0200  H=793B  S=0152  P=010E  RZ  another CALL.
-Z,E- A=00  B=0000  D=0200  H=793B  S=0152  P=010E  RZ
-Z,E- A=00  B=0000  D=0200  H=793B  S=0154  P=011B  LX1  H,012A .WRMSG
-Z,E- A=00  B=0000  D=0200  H=012A  S=0154  P=011E  CALL 010C .WRMSG

WRMSG:
-Z,E- A=00  B=0000  D=0200  H=012A  S=0152  P=010C  MOV  A,M .WALLAMSG
-Z,E- A=57  B=0000  D=0200  H=012A  S=0152  P=0100  ORA  A
----- A=57  B=0000  D=0200  H=012A  S=0152  P=010E  RZ
----- A=57  B=0000  D=0200  H=012A  S=0152  P=010F  CALL 0106 .WCHAR

WCHAR:
----- A=57  B=0000  D=0200  H=012A  S=0150  P=0106  MOV  C,02
----- A=57  B=0002  D=0200  H=012A  S=0150  P=0108  MOV  E,A  abort with "return"

*0109
#6, .WRMSG  GO, skip traces
W Should be ALLA ..., what happened?
*010C .WRMSG

#0100  Trace without call:
-Z,E- A=00  B=0000  D=0200  H=793B  S=0152  P=010C  MOV  A,M
-Z,E- A=00  B=0000  D=0200  H=793B  S=0152  P=010D  ORA  A
-Z,E- A=00  B=0000  D=0200  H=793B  S=0152  P=010E  RZ
-Z,E- A=00  B=0000  D=0200  H=793B  S=0154  P=0121  LX1  H,0130 .WASHMSG
-Z,E- A=00  B=0000  D=0200  H=0130  S=0154  P=0124  CALL 010C .WRMSG
STOP:  Called WRMSG, printed another "W" and stopped!
-Z,E- A=00  B=0000  D=0200  H=793B  S=0154  P=0127  JMP  0000 .REBOOT
REBOOT:  abort with "return" so we can restart.
-Z,E- A=00  B=0000  D=0200  H=793B  S=0154  P=0000  JMP  7A03

# It appears that the WRMSG subroutine is not saving the HL register pair, nor is HL being incremented on each loop.

All Information Presented Here is Proprietary to Digital Research
#A10F
010F JMP 200  We'll put a "hot patch" at the end of the WRMSG
0112 subroutine to save the HL pair, call the WRCHAR
#A200 subroutine, restore the HL pair, then increment HL.
#200 PUSH H   We're not using the region above 200, so place patch
0201 CALL .WRCHAR in this region.
0204 POP H
0205 INX H
0206 JMP .WRMSG
0209
#G100, .WRMSG Ok, now restart the program and stop at the first call
to WRMSG.
*G10C .WRMSG Here we are. HL addresses the message to print, which
is the default display address following a breakpoint:
012A: 57 41 4C 4C 41 20 WALLA = message to print.
0130: 57 41 53 48 56 55 52 53 20 31 2E 34 31 00 02 WASHVERS 1.4$1

#TH100 Trace without calls: shows only the activity in WRMSG.
----- A=00 B=0000 D=0000 H=012A S=0152 P=010C MOV A,M .WALLMSG
----- A=57 B=0000 D=0000 H=012A S=0152 P=010D ORA A
     first character
----- A=57 B=0000 D=0000 H=012A S=0152 P=010E RZ
     Is $57 = "W"?
----- A=57 B=0000 D=0000 H=012A S=0152 P=010F JMP 0200
     Now in patch
     area.
----- A=57 B=0000 D=0000 H=012A S=0152 P=0120 PUS H 0004 POP H
----- A=00 B=0000 D=0200 H=793B S=0150 P=0204 POP H
----- A=00 B=0000 D=0200 H=012A S=0152 P=0205 INX H
----- A=00 B=0000 D=0200 H=0128 S=0152 P=0206 JMP 010C .WRMSG
     Move to next
----- A=00 B=0000 D=0200 H=0128 S=0152 P=010C .WRMSG Looping back.

WRMSG:  
-Z-E A=00 B=0000 D=0200 H=0128 S=0152 P=010C MOV A,M
-Z-E A=41 B=0000 D=0200 H=0128 S=0152 P=010D ORA A
-Z-E A=41 B=0000 D=0200 H=0128 S=0152 P=010E RZ
-Z-E A=41 B=0000 D=0200 H=0128 S=0152 P=010F JMP 0200
-Z-E A=00 B=0000 D=0200 H=0128 S=0152 P=0200 PUS H 0004 POP H
-Z-E A=00 B=0000 D=0200 H=0128 S=0152 P=0205 INX H
-Z-E A=00 B=0000 D=0200 H=0128 S=0152 P=0206 JMP 010C .WRMSG

WRMSG:  
-Z-E A=00 B=0000 D=0200 H=012C S=0152 P=010C MOV A,M
-Z-E A=01 B=0000 D=0200 H=012C S=0152 P=010C MOV A,M
*0100
*0100

*P, STOP Set a permanent break at STOP, then GO from
#G100 the beginning of the program.
#G100 WASHVERS 1.4$1: WASHVERS 1.4$1: WASHVERS 1.4$1
01 PASS 0127 .STOP Things look better, but "00" byte missing on messages.
-Z-E A=00 B=0000 D=0200 H=013E S=0154 P=0127 JMP 0000 .REBOOT
*0000 .REBOOT
#S, WALLMSG+4 Place a 00 byte at the end of each message.
012E 41 (leave this value, 41 = "A" in WALLA)
012F 20 0 (changed to 30 from blank)
0130 57
#S, WASHMSG+4 Place 00 byte at the end of the second message.
0134 56 0
0135 45 .
#G100 Break at STOP remains set, GO from the beginning.
WALL, WASHVERS Looks good, we now have enough information to
01 PASS 0127 .STOP go back and change the source program using ED.
-Z-E A=00 B=0000 D=0200 H=0134 S=0154 P=0127 JMP 0000 .REBOOT
*0000 .REBOOT
#G0

All Information Presented Here is Proprietary to Digital Research
Index

A
address, 5
ambiguous references, 25
Assemble (A) command, 15
assembler/disassembler module, 16

B
backtrace, 46, 47
binary delimiters, 12
bubble sort, 51
byte format, 18

C
Call (C) command, 17
COLLECT, 43
COLLECT entry point, 48
command letters, 7
command line, 5
command parameters, 7
commands, 5
CPU flags, 40

data collection, 44
default file control block, 24, 25
DISPLAY, 43, 47, 48
Display Memory (D) command, 17
DUMP, 48

E
entry points, 43
error messages, 2, 5, 16, 34
Examine CPU State (X) command, 40

F
Fill Memory (F) command, 20

G
Go (G) command, 20

H
Hexadecimal Value (H) command, 22
HIST Utility, 44
histogram, 44

I
INITIAL, 43
Input Line (I) command, 23

L
L command, 46
L command output, 27
line editing controls, 6
List Code (L) command, 27
literal character values, 10
literal decimal numbers, 9
literal hexadecimal numbers, 9

M
memory organization, 1
Move Memory (M) command, 28

O
operation code, 37

P
Pass Counter (P) command, 28
pass counts, 28, 29
pass points, 28, 47, 48
PIP, 34
program load, 35
program segments, 32

Q
qualified symbols, 12
R

Read Code/Symbols (R) command, 31
register trace, 30
reinitialization, 44
relocation, 1, 2
report generator, 33
ROM code, 38

S

search and match procedure, 12
Set Memory (S) Command, 35
symbol, 5
symbol file error, 35
symbol load, 5, 35
symbol table, 5
symbol table files, 34
symbolic expressions, 7, 9, 12
symbolic references, 11

T

Trace Mode (T) command, 36
trace process, 37
TRACE Utility, 46
trace without call function, 37

U

unary delimiters, 12
Untrace Mode (U) command, 39
up-arrow operator, 13
utilities, 43
utility functions, 3, 32
utility load operation, 43

W

word format, 18