Copyright

Trademark

Disclaimer

TABLE OF CONTENTS

SECTION PAGE

1. SID OPERATION UNDER CP/M

The CP/M symbolic debugger, called SID, expands upon the features of the CP/M standard debugger described in the manual "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" which can be dynamically loaded with SID to provide traceback and histogram facilities. The various functions of SID are given in the sections which follow.

1.1. SID Startup.

The SID program is initiated by typing one of the following commands:

(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 topmost portion of the Transient Program Area (TPA) and overlays the Console Command Processor portion of CP/M (see the "CP/M Interface Guide" and "CP/M Alteration Guide" for a discussion of memory use conventions). Memory organization before SID is loaded is shown in Figure 1, while Figure 2 shows the memory configuration after SID is loaded and relocated. Due to the relocation process, SID is independent of the exact memory size which CP/M manages in a particular computer configuration.

```
                ----------------
(High Memory)  |                |
               |      BDOS      |
               |                |
                ----------------
               |      CCP       |
                ----------------
               |                |
               |                |
               |      TPA       |
               |                |
                ----------------
 (Low Memory)  | JMP BDOS       |
                ----------------
```

Figure 1. Memory Configuration Before SID Loads.

```
 ---------------
|               |
|     BDOS      |
|               |
 ---------------
|     SID       |
|               |
| JMP BDOS      |
 ---------------
|               |
|     TPA       |
|               |
 ---------------
| JMP SID       |
 ---------------
```

Figure 2. Memory Configuration After SID Loads.

After loading and relocating, SID alters the BDOS entry address to reflect the reduced memory size, as shown in Figure 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. This form is often used when the operator wishes 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 program 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 commands C (Call), G (Go), T (Trace), or U (Untrace). It is the programmer's responsibility to ensure 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, the standard "?" error response is issued by SID. If no load error occurs, the SID response is:

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

where nnnn, pppp, and eeee are hexadecimal values which 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 3. Although x.y usually contains machine code, the operator can name an ASCII file, in which case these program addresses are less meaningful.

```
               ----------------
               |              |
               |     BDOS     |
               |              |
               ----------------
               |     SID      |
               ----------------
        eeee:  | (Free Space) |
               |              |
        nnnn:  |              |
               ----------------
               |              |
               |    (Test     |
        pppp:  |   program)   |
               ----------------
               | JMP SID      |
               ----------------
```

Figure 3. Memory Configuration After Test Program Load.

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 program counter is obtained from the last 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 discussed in the ASM and MAC manuals, the program counter value can be given on the "END" statement in the source program. Again, it is the programmer's 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, the "?" response is issued by SID. Otherwise, the principal program locations shown in the previous paragraph are listed at the console.

Command form (d) is used 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 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 the section entitled "SID Utilities."

```
----------------
|     BDOS     |
----------------
|     SID      |
----------------
|     UTL      |
| JMP BDOS     |
----------------
|              |
|     TPA      |
|              |
----------------
| JMP UTL      |
----------------
```

Figure 4. Memory Configuration Following Utility Load.

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 base of SID downward toward the program under test, as shown in Figure 5.

```
----------------
|     BDOS     |
----------------
|     SID      |
----------------
|   (UTL If    |
|   Present)   |
----------------
|              |
|   SYMBOLS    |
|              |
| JMP BDOS     |
----------------
|  Free Space  |
----------------
|              |
| Test Program |
|              |
----------------
| JMP SYMBOLS  |
----------------
```

Figure 5. Memory Configuration Following Symbol Load.

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 which takes on this address value. The symbol consists of up to 16 graphic ASCII characters terminated by one or more tabs (ctl-I) or a carriage return line feed sequence. Note that the operator can optionally create or alter a symbol table using the CP/M editor, as long as this format is followed (see the manual "ED: the CP/M Context Editor" for editing details).

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

SYMBOLS

is printed indicating 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.

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

COMMAND FORM	COMMAND EXAMPLE
(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" which 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.

SID prompts the console for input by typing "#" when ready to accept the next command. 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 command characters. The CP/M line editing functions are:

ctl-C	CP/M system reboot, return to CCP
ctl-E	Physical end-of-line
ctl-P	Print console output (on/off toggle)
ctl-R	Retype current input line
ctl-S	Stop/start console output
ctl-U	Delete current input line
ctl-X	(Same as ctl-U)
ctl-Z	End of console input (not used in SID)
rubout	Delete and echo last character

where the "ctl" function indicates that the control key is held down while the particular function key is depressed. Note further that the ctl-R, ctl-U, and ctl-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, the operator can abort the typeout before it completes by typing any key at the console (a "return" suffices).

The single letter commands which direct the actions of SID are typed at the beginning of the command line. The valid commands are summarized below (lower case command letters are translated to upper case automatically):

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

Although the details of each of the commands are given in later sections, nearly all of the commands accept parameters following the letter which governs the command actions. The parameters may be counters or memory addresses, and may 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 form

```
Dssss,eeee
```

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

```
D100,300
```

which instructs SID to print the hexadecimal and ASCII values contained in memory locations 0100 through 0300.

Although the operator 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.

2. SID SYMBOLIC EXPRESSIONS

An important facility of SID is the ability to reference absolute machine addresses through symbolic expressions. Symbolic expressions may involve names obtained from the program under test which are included in the "SYM" file produced by the CP/M Macro Assembler, or may 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. The purpose of this section is to completely describe symbolic expressions so that they may be incorporated as command parameters in the individual command forms which follow this section.

2.1. Literal Hexadecimal Numbers.

SID normally accepts and displays values in the hexadecimal number base to form 16-bit values from up to four hexadecimal digits. 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. Note that SID translates lower case hexadecimal digits to upper case outside of string apostrophes.

A literal hexadecimal number in SID consists of one or more contiguous hexadecimal digits. If four digits are typed 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 values to the left below produce the hexadecimal and decimal values shown following the "#" to the right below.

INPUT VALUE	HEXADECIMAL	DECIMAL
1	0001	#1
100	0100	#256
fffe	FFFE	#65534
10000	0000	#0
38001	8001	#32769

2.2. Literal Decimal Numbers.

Although SID's normal number base is hexadecimal, the operator can override this base on input by preceding the number by a "#" symbol which indicates that the following number is in the decimal base. In this case, the number which 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.

The input values shown to the left below produce the internal hexadecimal values shown to the right below:

INPUT VALUE	HEXADECIMAL VALUE
#9	0009
#10	000A
#256	0100
#65535	FFFF
#65545	0009

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. Each character is translated internally to its two hexadecimal digit ASCII encoded form. 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. In order to include the possibility of writing strings which include apostrophes, a pair of contiguous apostrophes are 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 alphabetics begin at the encoded hexadecimal value 41, lower case alphabetics begin at 61, a space is hexadecimal 20, and an apostrophe is encoded as hexadecimal 60).

INPUT STRING	HEXADECIMAL VALUE
'A'	0041
'AB'	4142
'ABC'	4243
'aA'	6141
''''	0060
''''''	6060
' A'	2041
'A '	4120

2.4. Symbolic References.

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

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

where s represents a sequence of one to sixteen characters which 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 double precision 16-bit "word" value contained in the two memory locations given by .s, while form (c) results in the single precision 8-bit "byte" value at .s in memory. Suppose, for example, that the input symbol table contains two symbols, and appears as:

```
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.

SYMBOL REFERENCE	HEXADECIMAL VALUE
.GAMMA	0100
.DELTA	0102
@GAMMA	3E02
@DELTA	224D
=GAMMA	0002
=DELTA	004D

2.5. Qualified Symbols.

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

```
S1/S2/ . . . /Sn
```

where S1 through Sn represent symbols which 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 input file. In the case of 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

ØlØØ A ØЗØØ B Ø2ØØ A 3EØØ C 2ØFØ A ØlØ2 A

in the input 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.

SYMBOL REFERENCE	HEXADECIMAL VALUE
.A	ØlØØ
@A	3EØ2
.A/A	Ø2ØØ
.C/A/A	ØlØ2
=C/A/A	ØØ4D
@B/A/A	2ØFØ

2.6. Symbolic Operators.

Literal numbers, strings, and symbol references can be combined into symbolic expressions using unary and binary "+" and "-" delimiters. The entire sequence of numbers, symbols, and operators must be written without imbedded 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 which precedes it.

When placed between two operands, the "+" indicates addition of the previously accumulated value. The value of the following literal or symbolic value is added, and 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:

DFEØØ+#128,+5

contains the first expression "FEØØ+#128" which adds FEØØ and (decimal) 128 to produce FE8Ø as the starting value for this display command. The second expression "+5" begins with a unary "+" which indicates that the previous expression value (FE8Ø) 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:

DFE8Ø,FE85

The "-" symbol causes SID to subtract the literal number or symbol reference from the 16-bit value accumulated thusfar 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 Ø-x

where x is any valid symbolic expression. The command:

DFFØØ-2ØØ,-#512

for example, is equivalent to the simple command

DFDØØ,FEØØ

A special up-arrow operator, denoted by "^", is present in SID to denote the top-of-stack in the program under test. In general, a sequence of n up-arrow operators extracts the nth 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 we wish to debug a sorting program which 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(Ø), LIST(1), ... , LIST(255).

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(Ø) 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 Ø through N-1.

Given these data areas, the command

D.LIST,+255

for example, displays the entire area reserved for sorting:

LIST(Ø), LIST(1), . . . , LIST(255)

The command

```
D.LIST,+=I
```

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

```
LIST(Ø), LIST(1), . . . , LIST(I)
```

The command:

```
D.LIST+=I,+Ø
```

displays only LIST(I). Finally, the command:

```
D.LIST,+=N-1
```

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

```
LIST(Ø), LIST(1), . . . , LIST(N-1)
```

The exact manner in which symbolic expressions are used within SID is dependent upon the individual command which is issued by the operator. These commands are listed in some detail in the section which follows.

3. SID COMMANDS.

SID commands are entered at the console following the "#" prompt, and direct the debugging process by allowing alteration and display of machine functions as well as controlling execution of the program under test.

The commands which SID accepts are listed and described in alphabetical order in the sections which follow.

3.1. The Assemble (A) Command.

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

(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 a null line (i.e., a single carriage return) is typed by the operator. 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 each command line is terminated by a carriage return):

```
A100                 begin assembly at 0100
0100 MVI A,10        load A with hex 10
0102 DCR A           decrement A register
0103 JNZ 100         loop until zero
0106 RST 7           return to debugger
0107                 single carriage return
```

As each successive address is prompted, the operator may 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 which 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 the operator 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:

```
A1ØØ
A#1ØØ
A.CRLF+5
A@GAMMA+@X-=I
A+3Ø
```

Given that the command A1ØØ has been entered, the following interaction could take place between SID and the operator:

SID PROMPT	OPERATOR INPUT
Ø1ØØ	MVI C,.A-.B
Ø1Ø2	LXI H,.SOURCE
Ø1Ø5	LXI D,+1ØØ
Ø1Ø8	MOV A,M
Ø1Ø9	INX H
Ø1ØA	STAX D
Ø1ØB	INX D
Ø1ØC	DCR C
Ø1ØD	JNZ 1Ø8
Ø11Ø	("return" only)

where A, B, and SOURCE are symbols which are active 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+1ØØ since .SOURCE was the immediately preceeding symbolic expression value. This particular program segment would move 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 forms are:

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

Although the C command is designed for use with SID utilities, it can be used to perform calls on test program subroutines to perform program initialization, or to make CP/M BDOS calls which 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 which returns control to SID, followed by a normal system 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 is used to display selected segments of memory in both byte (8-bit) and word (16-bit) formats. The display appears in both byte and ASCII form in the output. The forms of the D command are:

```
(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 representation 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 which are a multiple 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 which is a multiple of 16. Each display line which 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 which 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 the operator wishes 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 excessively long typeouts can be aborted by depressing any keyboard character, such as a 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 256 are stored in memory starting at hexadecimal address 0100. The command:

```
D100,12A
```

will produce 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 (h), 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 !"#$%&'()*+
```

Examples of valid D commands are:

```
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:

```
Fs,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 the operator's responsibility to not fill memory locations which are occupied by the resident portions of CP/M, including areas reserved for SID. Examples of valid F commands are:

```
F1ØØ,3FF,FF
F.GAMMA,+#1ØØ,#23
F@ALPHA,+=I,=X
```

3.5. The GO (G) Command.

The G command is used to pass 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 several G command forms are:

```
(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 text (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 will return 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 which 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 location a is encountered. It should be noted that an immediate breakpoint will always occur if p = a. If this is desired, however, the operator 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.

It should be noted that the instruction at a breakpoint address is not executed when the G command is used. 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 C,A
    0303    MVI C,2
    0305    JMP 0005
```

Suppose further that the operator is testing a program which makes calls on the TYPEOUT subroutine where a break address is to be set. The command:

G,.TYPEOUT

is entered by the operator. 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 the SID debugger. 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.

The operator can execute through the TYPEOUT subroutine using any of the commands G, T, or U. One useful command in this situation is

G,^

which continues execution from Ø3Ø2, and sets a breakpoint at the topmost stacked element (given by "^"). Since the topmost stacked element must be the subroutine return address, this particular G command has the effect of executing 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 when the U command is used. Fully-monitored execution under the U command has the advantage that the operator 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.

Examples of valid G commands are:

```
G1ØØ
G1ØØ,1Ø3
G.CRLF,.PRINT,#1Ø24
G@JMPVEC+=I,.ENDC,.ERRC
G,.ERRSUB
G,.ERRSUB,+3Ø
-G1ØØ,+1Ø,+1Ø
```

3.6. The Hexadecimal Value (H) Command.

The H command is used to perform hexadecimal computations including number base conversion operations. The forms of the H command are:

(a) Ha,b
(b) Ha
(c) H

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

ssss dddd

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

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

hhhh #ddddd 'c' .s

where hhhh is the four digit hexadecimal value of a, #ddddd is the (up to) six 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 Ø1ØØ, as in previous examples. The H commands shown to the left below result in the displays shown to the right below:

COMMAND	RESULTING DISPLAY
HØ,1	ØØØ1 FFFF
H41	ØØ41 #65 'A'
H1ØØ	Ø1ØØ #256 .GAMMA
H.GAMMA	Ø1ØØ #256 .GAMMA
H=GAMMA	ØØØ1 #1
H@GAMMA	Ø2Ø1 #513
HFF+@GAMMA	Ø1ØØ #256 .GAMMA
H'A'	ØØ41 #65 'A'
H'A'+=GAMMA	ØØ42 #66 'B'

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 which run in the CP/M environment, it is often useful to simulate the command line which is normally prepared by the CCP upon program load. The form of the I command is:

Iccccc ... ccc

where the sequence of c's represent ASCII characters which would 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 file name "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 which contains the characters following the DUMP command.

In order to trace and debug a program such as DUMP, the SID program would normally be invoked by typing:

SID DUMP.COM

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

SID DUMP.COM DUMP.SYM

In either case, SID loads the DUMP program and prompts the console for a command. In order to simulate the CCP's command line preparation, the operator would then type:

IX.COM

where the "I" denotes the Input command, which is followed by the simulated command line. The operator may 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, which is normally located at ØØ5C. 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 file name is initialized at location DFCB2, which is at address DFCB1+ØØ1Ø (hexadecimal). It is the programmer's responsibility to move the second drive number, file name, and file type to another region of memory before performing file operations at DFCB1 since the 16-byte region at DFCB2 will be immediately overwritten by any file operation. Further, the default buffer, labelled DBUFF, is initialized to contain the entire command line with a preceding blank character. In a standard CP/M system, the DBUFF area is assumed to be located start at ØØ8Ø and end at ØØFF. 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) file names, and t1 and t2 are (up to three character optional) file types, two default file control block names are prepared in the form:

```
DFCB1: d1' f1' t1' ØØ ØØ ØØ ØØ
DFCB2: d2' f2' t2' ØØ ØØ ØØ ØØ
       ØØ (current record field)
```

If dl: is empty in the original command line, then dl' = ØØ (which automatically selects the default drive), otherwise if dl = A, B, C, or D, then dl' = Ø1, Ø2, Ø3, or Ø4, respectively, which properly initializes the file control block for automatic disk selection. Field fl' is initialized to the ASCII file name given by fl, padded to an eight character field with ASCII blanks. Similarly, tl' is initialized to the ASCII file type, padded with blanks in a field of length three. Lower case alphabetics in dl, fl, and tl are translated to upper case in dl', fl', and tl', respectively. Names which 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 ØØ6C. Note that the current record field at location ØØ7C is also initialized to ØØ. If any of the fields fl, tl, 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 which use the "*" or "?" character 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 file names. 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 2Ø:

```
     COMMAND LINE      DEFAULT DATA AREA INITIALIZATION

     I                 DFCBl: ØØ
                              2Ø 2Ø 2Ø 2Ø 2EA2Ø 2Ø 2Ø
                              2Ø 2Ø 2Ø ØØ ØØ ØØ ØØ
                       DFCB2: ØØ
                              2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
                              2Ø 2Ø 2Ø ØØ ØØ ØØ ØØ
                              ØØ
                              ØØ

                       DBUFF: ØØ ØØ
```

```
I A.B               DFCB1: 00
                           41 20 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 20 00 00 00 00
                           00
                           00

                    DBUFF: 05 20 20 41 2E 42 00

IA:B.C b:d.e        DFCB1: 01
                           42 20 20 20 20 20 20 20
                           43 20 20 00 00 00 00
                   DFCB2:  02
                           44 20 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 3F
                           42 3F 43 00 00 00 00
                   DFCB2:  04
                           20 20 20 20 20 20 20 20
                           20 20 20 00 00 00 00
                           00
                           00

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

Note that the I command is 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 which 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
```

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 form of the L command are:

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

Form (a) list 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 format of the L command output is:

```
sssss:
aaaa  opcode  operand  .ttttt
```

where "sssss:" 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. Note that instructions may directly reference memory through their operand fields (e.g., CALL, JMP, LDA, LHLD), while other instructions imply a memory address (e.g., STAX B, LDAX D). Instructions which 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.

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'
```

3.9. The Move Memory (M) Command.

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

```
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. Data moves 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 01FF.

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 the operator to set and clear "pass points" and "pass counts" in the program under test.

The forms of the P command are:

(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, the operator 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 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.

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. Up to eight distinct pass points can be actively set at any particular time. Form (c) displays these active pass points in the format:

cc pppp .sssss

where cc is the hexadecimal value of the pass count which is currently associated with the pass address pppp, and sssss is a symbol which 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" can be used 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. The command:

```
P.TYPEOUT,#30
```

is issued by the operator. The effect of this particular P command is to set 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 pass trace:

```
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
```

where the "register trace" shows the state of the CPU registers before the "MOV C,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. The operator can depress any keyboard character during the pass point trace, and SID will immediately stop execution following the instruction at the pass point address. If instead, the command

```
-G
```

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

```
Ø1 PASS Ø3Ø2 .TYPEOUT
(register trace)
*3Ø3
```

is printed. Although the intermediate pass traces are not displayed, the operator 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 which intermix with the permanent breakpoints which are set with the P command. Note, however, that permanent breakpoints override the temporary breakpoints which are given by b and c when they occur at the same address. Further, T and U command can be used to 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:

```
P1ØØ,FF
P.BDOS
P@ICALL+3Ø,#2Ø
-P.CRLF
```

3.11. The Read Code/Symbols (R) Command.

The R command is used in conjunction with the I command to read program segments, symbol tables, and utility functions into the transient program area. The forms of the R command are:

(a) R
(b) Rd

The I command is first used to set the file names which 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 d 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:

```
SID X.COM
```

could be replaced by the sequence of commands:

```
SID          Starts SID without a test program
IX.COM       Initialize the input line
R            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, a typical command sequence of this form would be

```
IDUMP.COM DUMP.SYM
R
```

which 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 file type "HEX" load into the locations specified in the Intel formatted hexadecimal records, while programs with file type "UTL" are assumed to be SID utility functions which load and relocate automatically. All other file types 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 the above form could be used to load the symbol tables for selective portions of a large program which was initially developed in small modules.

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

```
IOMOD.ASM           I/O Module
SORT.ASM            File Sorting Module
MERGE.ASM           File Merge Module
FORMAT.ASM          Report Format Module
MAIN.ASM            Main Program Module
DATA.ASM            Common Data Definitions
```

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:

```
SID                 Start the SID program
IIOMOD.HEX          Initialize IOMOD
R                   Read I/O Module
ISORT.HEX           Initialize SORT
R                   Read Sort Module
IMERGE.HEX          Initialize MERGE
R                   Read Merge Module
IFORMAT.HEX         Initialize FORMAT
R                   Read Format Module
IMAIN.HEX           Initialize MAIN
R                   Read Main Module
IDATA.HEX           Initialize DATA Area
R                   Read Initialized Data
```

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:

```
NEXT  PC   END
1B3E  Ø1ØØ 8EØØ
```

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

```
H1B
```

the operator finds that 1B (hexadecimal) pages is the same as 27 (decimal) pages. At this point, the operator returns to CCP mode by typing either a control-C (warm start), or "GØ" command, which leaves the memory image intact. The command:

```
SAVE 27 REPORT.COM
```

is then issued to create a memory image file on the diskette. The operator then re-enters SID using a command of the form:

```
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 could be debugged by subsequently loading the corresponding symbol information:

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

which readies 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. The operator may then load addition symbol tables by typing I and R commands such as:

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

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

Note that several symbol table files can be concatenated using the PIP program (see the "CP/M Features and Facilities" manual for PIP operation) in command mode. For example, the PIP command:

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

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

```
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:

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

which allows the operator to type

SID REPORT.COM REPORT.SYM

in order 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 module 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 (file type "UTL"). The commands

```
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:

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 has the effect of loading 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

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 which was actually loaded.

3.12. The Set Memory (S) Command.

The S command allows the operator 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) is used to store word (16-bit) mode data items. In either case, the SID program proceeds to prompt the console with successive addresses, starting at location s, along with the data item presently located at that address. As each line prompt occurs, the operator has the option of typing a single carriage return or typing a symbolic expression (followed by a carriage return) which is evaluated and becomes the new data item at that location. If a single carriage return is typed, then the data element at that location remains unchanged. The S command terminates whenever an invalid data item is detected, or when the operator types 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:

"ccccc . . .ccccc

where the sequence of c's represent 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:

S1ØØ

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.

```
SID PROMPT        OPERATOR INPUT
 Ø1ØØ  C3           34cr
 Ø1Ø1  24           #254cr
 Ø1Ø2  CF           cr
 Ø1Ø3  4B           "ASCIIcr
 Ø1Ø8  6E           =X+5cr
 Ø1Ø9  E2           '%'cr
 Ø1ØA  D4           .cr
```

A valid double byte input sequence following the command

SW.X+#3Ø

is shown below:

SID PROMPT		OPERATOR INPUT
2900	006D	44Fcr
2302	4F32	@GAMMAcr
2304	33E2	cr
2306	FF11	.X+=I-#20cr
2308	348F	.cr

3.13. The Trace Mode (T) Command.

The T command allows the operator to single or multiple step a test program while viewing the CPU registers as they change. In addition, the T command can be used in conjunction 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) Tn
(b) T
(c) Tn,c
(d) T,c
(e) -T (with options a - d)
(f) TW (with options a - d)
(g) -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:

```
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 *bbbb
```

showing the register state before each corresponding operation code is executed. Each operation code is written in the same format as in the L and X commands, with interspersed symbolic operands decoded wherever possible. The interspersed labels show program addresses when they occur in the flow of execution. The final break address, denoted by "*bbbb" above, shows the value of the program counter after opcode 4 is executed. The CPU state can optionally be displayed at this point by typing the single character "X" command.

Forms (c) and (d) are used only in conjunction 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. Details of the use of these forms are given in sections which follow. Note, however, that form (d) is equivalent to (c) with a single step count of n = 1.

Forms given by (e) parallel (a) through (e), 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 increases 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:

```
MAINLINE     SUBROUTINE 1    SUBROUTINE 2 ... SUBROUTINE n

 . . .       S1: MOV A,C     S2: MOV A,D      Sn: MOV A,L
 CALL S1         . . .           . . .            . . .
 MOV  B,C        CALL S2         . . .            . . .
 MOV  C,D        MOV C,E         CALL S3    ...   MOV C,L
 . . .           MOV D,E         MOV D,H          MOV D,L
 . . .           . . .           . . .            . . .
 JMP 0000        RET             RET              RET
```

Suppose further that the test program is stopped within subroutine S1 before the call to subroutine S2. The command:

T#100

would have the effect of tracing 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 instead, the command:

TW#100

is typed 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. 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).

It should be noted 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), or 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 may occassionally 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.

Tracing can be aborted 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, 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.

Similar to the T command, execution can be aborted 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 the operator to examine and alter the CPU state of the program under test. The forms of the X command are:

(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 register 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 which matches address 0233 is Q.

Form (b) allows the operator 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). The operator can optionally type a single carriage return, which leaves the flag in its present state, or may type a 1 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 by the operator. 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 the operator types a single carriage return, the data value remains unchanged. Otherwise, the symbolic expression typed by the operator 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 03 45cr
```

where the "XA" is typed by the operator, the "03" is printed by SID as the value of the A register, and "45" is typed by the operator as a replacement for A's value. The "cr" represents the carriage return key in this example, and in the examples which follow. The following interactions with SID provide additional examples in the format described above:

```
XB 34EF cr (data remains unchanged)
XD 2000 2300 (D changes to 23)
XH 334E .GAMMA
XS 4323 @STKPTR+#100
```

4. SID UTILITIES.

SID Utilities are special programs which operate with SID to provide additional debugging facilities. As described in Section 1., a SID Utility is loaded by initially typing

SID x.UTL

where x is the name of a utility program, described in the sections which follow. Upon initiation, the utility program loads, relocates, and prompts the console for any necessary parameters. The operator then collects necessary program test data (using the U or T command), and displays 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, or can be loaded 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 therefore be reinitialized if required for the debugging run.

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

```
.INITIAL = iiii
.COLLECT = cccc
.DISPLAY = dddd
```

where iiii, 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, the operator may use the I and R commands to load test programs and symbol tables in order 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 commands

UFFFF,.COLLECT or 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, the operator would most likely stop the untrace mode by depressing the return key before the sequence of 65535 steps completes.

Following a series of data collection operations, the utility DISPLAY entry point can be called to print the accumulated data. Either of the command forms which follow accomplish this function:

C.DISPLAY or Cdddd

The operator may then resume the data collection process, as described above, followed by additional display operations.

At any point, the operator can reinitialize the utility by typing either

C.INITIAL or Ciiii

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

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

The functions of the SID utilities are presented individually in the remaining sections.

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 purpose of the HIST utility is to allow the operator to monitor "hot spots" in the test program where the program is executing most frequently.

After initial signon, as described in the previous section, the HIST utility prompts the input console with

TYPE HISTOGRAM BOUNDS

The operator 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. In order to collect histogram information, the operator must use one of the 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 all of these commands are optional, the single 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, the histogram is displayed by typing

C.DISPLAY or Cdddd

and the histogram is printed in the format:

```
HISTOGRAM:
   ADDR        RELATIVE FREQUENCY, MAXIMUM VALUE = mmmm
   aaaa        *****
   bbbb        *******
   cccc        *********
   ....
   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 difference address slots (aaaa, bbbb, ... ,zzzz, above), with a maximum of 64 asterisks in any particular bar of the graph.

Given the above display, for example, the "hot spot" is around the address range xxxx to zzzz. In this case, it would be worthwhile reinitializing the HIST utility by typing

C.INITIAL or Ciiii

The HIST initialization prompt and response should then be

TYPE HISTOGRAM BOUNDS xxxx,zzzz

The operator may then rerun the test program using the command

UFFFF,.COLLECT

After leaving enough time for the test program to reach "steady state," the operator then interrupts 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 frquently executed region.

The L command can subsequently be used to determine the exact instructions which 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 is used to obtain a backtrace of the instructions which lead to a particular break address in a program under test. A program may have an error condition, for example, which arises from a sequence of instructions which are difficult to find under normal testing. In this case, TRACE can be used to collect program addresses as the test program executes, and display these addresses and instructions in most recent to least recent order when requested by the operator. Normal invocation of SID with the TRACE utility is:

SID TRACE.UTL

with the normal utility response:

```
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 be used to disassemble instructions when the backtrace is requested.

The operator may then proceed to load a test program with optional symbol table. The DUMP program, for example, could be loaded by subsequently typing:

```
IDUMP.COM DUMP.SYM
R
```

with the usual "NEXT PC END" response indicating that the test program is loaded. At this point, the SID debugger is executing in high memory, along with the TRACE utility. The test program is present in low memory, ready for execution.

The simplest backtrace is obtained by typing 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. The operator may then obtain a backtrace to the stop address by typing:

```
C.DISPLAY
```

which 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 will account 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 a 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 which are of interest in the backtrace. The program is then run to an intermediate location where the test begins. At this intermediate test point, the U command is used 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 the operator regains 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 the operator wishes 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, the operator first types:

```
P.BDOS
```

which sets a pass point at the BDOS entry, with corresponding pass count = 1. The operator then executes 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 will stop at the BDOS call before the instruction count is exceeded (if it does not, the operator can depress any keyboard character to force a program stop). 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, the TRACE function can be invoked:

```
C.DISPLAY
```

which results in the display:

```
BDOS:
  0005  JMP   CCDF
  01CA  CALL  0005   .BDOS
  01C8  MVI   C,0F
  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. The operator may 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, the operator can type the pass command:

P.BDOS,#20

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

UFFFF,.COLLECT

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

-UFFFF,.COLLECT

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

C.DISPLAY

to view the trace to this particular BDOS call.

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

The TRACE utility can also be used 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:

```
SID            Load the SID Program
-A             Remove the Disassembler
ITRACE.UTL     Ready the TRACE Utility
R              Read the TRACE Utility
IDUMP.COM      Load the DUMP Program
```

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

```
"-A" IN EFFECT, ADDRESS BACKTRACE
```

The backtrace information is subsequently displayed in the format:

```
addr-255 addr-254 addr-253 . . . addr-248
addr-247 addr-246 addr-245 . . . addr-240
                  . . .
addr-007 addr-006 addr-005 . . . addr-000
```

5. SID SAMPLE DEBUGGING SESSIONS.

This section contains several examples of SID debugging sessions. The examples are based upon a "bubble sort" of a list of byte values. The bubble sort program is first listed in its first 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 which 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 $+S	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 which 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 which calls the

BDOS is listed, followed by a single debugging session. The purpose of this particular example is to show 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

```
(1)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     0000H   ;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 BE                     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 B7                     ORA     A       ;SET FLAGS
                    ;       END OF SORT PROCESS, REBOOT
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,0     ;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 BE                     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 2B                     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
                    ;
                    INCI:   ;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                SW:     DS      1       ;SWITCH COUNT
0139                I:      DS      1       ;INDEX
013A                        DS      32      ;16 LEVEL STACK
                    STACK:
                    ;
015A 0503040A08LIST:        DB      5,3,4,10, 8,130,10,4
0162 08             N:      DB      $-LIST  ;LENGTH OF LIST
0163                        END
```

```
(2) TYPE SORT.SYM
010A COMP      0119 CONT      0139 I        0131 INCI     015A LIST
0162 N         0000 REBOOT    0100 SORT     015A STACK    0116 STOP
0138 SW

(3) TYPE SORT.HEX
:10010000213801360121390136003A6201BEC21997
:10011000012138017EB7C300005F1600215A011982
:100120007E23BEDA3101CA31014E772B71213801AD
:080130003421390134C30A0136
:09015A000503040A08820A0408E6
: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 163, Program Counter is 100
#D.LIST,+=N-1                             and end of TPA is 55B7
015A: 05 03 04 0A 08 82 ......
0160: 0A 04 ..                  Display initial list of items to sort
#G,.STOP    Execute test program until "STOP" symbol address encountered

*0116 .STOP      Now at the STOP address, examine data list:
#D.LIST,+=N-1
015A: 05 03 04 0A 08 82 ......      Hasn't changed!
0160: 0A 04 ..
#XP              where is the program counter?
P=0116 100        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     SW=1
 ----- A=01 B=0000 D=0008 H=0138 S=0100 P=0105 LXI  H,0139 .I    I=0
 ----- 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       N=I?
 ----- A=08 B=0000 D=0008 H=0139 S=0100 P=010D CMP  M=00 .I
 ----I A=08 B=0000 D=0008 H=0139 S=0100 P=010E JNZ  0119 .CONT
CONT:                                                   No, so compare
 ----I A=08 B=0000 D=0008 H=0139 S=0100 P=0119 MOV  E,A    LIST(i), LIST(i+1)
 ----I A=08 B=0000 D=0008 H=0139 S=0100 P=011A MVI  D,00
 ----I A=08 B=0000 D=0008 H=0139 S=0100 P=011C LXI  H,015A .LIST
 ----I A=08 B=0000 D=0008 H=015A S=0100 P=011F DAD  D
 ----I A=08 B=0000 D=0008 H=0162 S=0100 P=0120 MOV  A,M .N  What's this?
 ----I A=08 B=0000 D=0008 H=0162 S=0100 P=0121 INX  H       Why did we
 ----I A=08 B=0000 D=0008 H=0163 S=0100 P=0122 CMP  M=58    fetch N?
 C-M-I A=08 B=0000 D=0008 H=0163 S=0100 P=0123 JC   0131 .INCI
INCI:
 C-M-I 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"
#G0         with N in the accumulator, rather than I, which is expected!


(5) ED SORT.ASM            Back to the editor to make the  changes
*#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
  29:  ctl-Z                                          change
  29: *E    Terminate the editing session
```

(6)

```
MAC SORT
CP/M MACRO ASSEM 2.0
0166
001H USE FACTOR
END OF ASSEMBLY
```

Re-assemble the SORT program

(7)

Here's the symbol table:

```
TYPE SORT.SYM
010A COMP       0119 CONT       013C I          0134 INCI       015D LIST
0165 N          0000 REBOOT     0100 SORT       015D STACK      0116 STOP
013B SW
```

(8)

```
SID SORT.HEX SORT.SYM
SID VERS 1.4
SYMBOLS
NEXT  PC  END
0166 0100 55B7
#P.STOP
#G
```

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

Set a "pass point" at STOP to prevent reboot

Start (unmonitored) execution

```
01 PASS 0116 .STOP
 ----- A=7C B=0008 D=0081 H=013B S=0100 P=0116 JMP  0000 .REBOOT
*0000 .REBOOT
```

We made it to the STOP label, check values

```
#H=N
0082 #130
#D.LIST,+7
015D: 03 04 05 ...
0160: 08 0A 0A 04 08 .....
#ISORT.HEX
#R
NEXT  PC  END
0166 0100 55B7
```

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:

```
#XP
P=0100
#P
01 0116 .STOP
#P.SORT,FF
#G
```

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.

```
FF PASS 0100 .SORT
 ----- A=7C B=0008 D=0081 H=013B S=0100 P=0100 LXI  H,013B .SW
01 PASS 0116 .STOP
 ----- A=79 B=0008 D=0081 H=013B S=0100 P=0116 JMP  0000 .REBOOT
*0000 .REBOOT
#ISORT.HEX
#R
NEXT  PC  END
0166 0100 55B7
```

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).

```
#P
01 0116 .STOP
FE 0100 .SORT
#GO
```

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!

```
(9) ED SORT.ASM                 Oh well, back to the editor for a
   *#AV                          quick fix.  Append all text (#A), and
     1: *FSTOP:                  enter Verify mode (V).  Then find STOP.
    24: *OLT
    24:  STOP:    JMP      REBOOT   ;RESTART CCP
    24: *-                       Go up one line (-)
    23:  ;        END OF SORT PROCESS, REBOOT
    23: *I                       and enter insert mode (I)
    23:           JNZ      CONT     ;CONTINUE IF NOT EQUAL
    24:  ; ctl-Z, and "return"
    25:  E
    26:   wait, I forgot the ctl-Z, now I've got the E command in
    26: *-  my input buffer.  Type the ctl-Z, go back up one line,
    25:  E  delete the E, then end the edit
    25: *KT
    25:  ;        END OF SORT PROCESS, REBOOT
    25: *E  OK, we made the change, now re-assemble

(10) MAC SORT               Invoke the macro assembler with SORT as input.
   CP/M MACRO ASSEM 2.0
   0169
   001H USE FACTOR
   END OF ASSEMBLY

(11) SID SORT.HEX SORT.SYM  Here we go again, I sure hope this is the
  SID VERS 1.4              last time (but it probably isn't).
  SYMBOLS
  NEXT  PC  END
  0169 0100 55B7
  #P.SORT,FF                Set a pass point at sort, with a high count.

  P.STOP                    also set a pass point at STOP with count 1, this
  #P                        will stop the first time through
  FF 0100 .SORT
  01 0119 .STOP
  #G                        Execute the test program

  FF PASS 0100 .SORT        First time through SORT label:
   ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100 LXI  H,013E .SW
  01 PASS 0119 .STOP        Stopped again! Arrggh!
   -Z-E- A=00 B=006A D=0007 H=013E S=0100 P=0119 JMP  0000 .REBOOT
  *0000 .REBOOT
                            Let's look at some values:
  H=N
  0008 #8                   N=8, looks better than last time
  #D.LIST,+=N
  0160: 01 01 03 04 04 05 07 08 08 ......... These values look a bit
  #ISORT.HEX                                  strange?!  Try again:
  #R
  NEXT  PC  END
  0169 0100 55B7
  #D.LIST,+=N-1             Machine code reloaded, display initial values:
  0160: 05 03 04 0A 08 82 0A 04 ........
  #L.CONT
  CONT:                     Let's take a look at the process of switching
    011C  LDA  013F .I      two data items - the code appears down below
    011F  MOV  E,A          the "CONT" label, so we'll disassemble a
    0120  MVI  D,00         portion of the program.
    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          Here's where the switch occurs, let's set a pass
  #P12F,FF                  point here and watch the data addresses:
  #P
  FE 0100 .SORT
  01 0119 .STOP
  FF 012F
```

```
#G

FE PASS 0100 .SORT         Here's the first pass through SORT
 -Z-E- A=00 B=006A D=0007 H=013E S=0100 P=0100 LXI  H,013E .SW
FF PASS 012F               Switching at address 161, looks OK!
 ----I A=05 B=006A D=0000 H=0161 S=0100 P=012F MOV  C,M
FE PASS 012F               Switching at 162, looks good.
 ----I A=05 B=0003 D=0001 H=0162 S=0100 P=012F MOV  C,M
FD PASS 012F               164 is the next to switch, looks good.
 ----I A=0A B=0004 D=0003 H=0164 S=0100 P=012F MOV  C,M
FC PASS 012F               166 is probably the next one.
 ---E- A=82 B=0008 D=0005 H=0166 S=0100 P=012F MOV  C,M
*0130                      So what's wrong?  This section of
#                          code seems to work.

#-P                        Clear all the pass points, and reload
#ISORT.HEX                 the machine code for another test.
#R
NEXT  PC  END
0169 0100 55B7
#L.CONT+5
  0121  NOP
  0122  LXI  H,0160 .LIST
  0125  DAD  D
  0126  MOV  A,M        Here's the code where the element
  0127  INX  H          switching occurs, let's watch the
  0128  CMP  M          program switch the first element:
  0129  JC   0137 .INCI
  012C  JZ   0137 .INCI
  012F  MOV  C,M
  0130  MOV  M,A
  0131  DCX  H
#G,129

*0129                   OK, here we are, ready to test and
#T10                    switch, if necessary.
 ----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=0003 D=0000 H=0161 S=0100 P=0130 MOV  M,A
 ----I A=05 B=0003 D=0000 H=0161 S=0100 P=0131 DCX  H
 ----I A=05 B=0003 D=0000 H=0160 S=0100 P=0132 MOV  M,C .LIST
 ----I A=05 B=0003 D=0000 H=0160 S=0100 P=0133 LXI  H,013E .SW
 ----I A=05 B=0003 D=0000 H=013E S=0100 P=0136 INR  M=01 .SW
*0137 .INCI             Well, that went nicely - elements switched, SW=1
#D.LIST,+7
0160: 03 05 04 0A 08 82 0A 04 ........
#H=I                    The data looks good at this point.
0000 .REBOOT #0
#G,.INCI                Proceed to the INCI label

*0137 .INCI             Here we are, let's look at the data:
#D.LIST,+7
0160: 03 05 04 0A 08 82 0A 04 ........
#H=I
0000 .REBOOT #0         Looks good, trace past the label and break
#T
 ----- A=05 B=0003 D=0000 H=013E S=0100 P=0137 LXI  H,013F .I
*013A
#G,.INCI                Go  to the INCI label again.

*0137 .INCI             Here we are (again), how's the data?
#D.LIST,+=I
0160: 03 04 ..          Looks good, proceed past INCI
#T
 ---E- A=05 B=0004 D=0001 H=013E S=0100 P=0137 LXI  H,013F .I
*013A                   And loop again . . .
#G,.INCI

*0137 .INCI             Here we are (again), how's the data?
#D.LIST,+=I
0160: 03 04 05 ...      Looks good, this is getting monotonous, let's
#G,.SORT,.STOP          go for it!  Stop at either SORT or STOP

*0119 .STOP             Egad! Here we at the the STOP label.  Why
#D.LIST,+=I             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.
```

```
#ISORT.HEX         Let's reload and try again.
#R
NEXT  PC  END
0169 0100 55B7
#L136,+3
  0136  INR  M     Here's where the switch count is incremented
INCI:
  0137  LXI  H,013F .I
  013A
#G,136             Execute the program and break
                   at SW = SW + 1
*0136
#D.LIST,+=I        Look at data values:
0160: 03
#U                 Use U to move past break address
 ----I A=05 B=0003 D=0000 H=013E S=0100 P=0136 INR  M=01 .SW
*0137 .INCI        It's actually easier to use the pass point feature
#P136              if we want to view the action of the INR M,
#G                 since the P command stops execution after the
                   pass point is executed.
01 PASS 0136
 ----I A=05 B=0004 D=0001 H=013E S=0100 P=0136 INR  M=02 .SW
*0137 .INCI        SW = 2, looks good.
#D.LIST,+=I
0160: 03 04 ..     Data values look good.
#S.N               Let's change N to a smaller value so the program
0168 08 4          doesn't loop so many times:  4 is a good number.
0169 0A .          End input with ".".
#G                 "GO" to pass point

01 PASS 0136       Here we are, switch value is incremented:
 ----I A=0A B=0008 D=0003 H=013E S=0100 P=0136 INR  M=03 .SW
*0137 .INCI        Stopped at next instruction.
#D.LIST,+=I
0160: 03 04 05 08 ....   Data values so far.
#H=SW
0004 #4            SW value at this point is 4.
#TFFFF             Let's watch it run for a few steps:
 ----- A=0A B=0008 D=0003 H=013E S=0100 P=0137 LXI  H,013F .I
 ----- A=0A B=0008 D=0003 H=013F S=0100 P=013A INR  M=03 .I
 ----- 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=04 B=0008 D=0003 H=013F S=0100 P=010D CMP  M=04 .I
 -Z-EI A=04 B=0008 D=0003 H=013F S=0100 P=010E JNZ  011C .CONT
 -Z-EI A=04 B=0008 D=0003 H=013F S=0100 P=0111 LXI  H,013E .SW
 oZ-EI A=04 B=0008 D=0003 H=013E S=0100 P=0114 MOV  A,M .SW
 -Z-EI 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 .I
*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.
```

(12)

```
ED SORT.ASM
*#AVFORA           This is a simple change:  append all text, enter line
  22: *OLT         verify mode, find "ORA" and make the change:
  22:          ORA     A        ;SET FLAGS
  22: *               "return" to move down one line
  23:          JNZ     CONT     ;CONTINUE IF NOT EQUAL
  23: *SCONT!ZSORT!ZOLT         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
```

```
(13) MAC SORT
CP/M MACRO ASSEM 2.0
0169                         Call out MAC for another assembly.
001H USE FACTOR
END OF ASSEMBLY

(14) LOAD SORT
                             Just for a little variation, we'll create a
FIRST ADDRESS 0100           SORT.COM file for testing under SID.
LAST  ADDRESS 0168
BYTES READ    0047
RECORDS WRITTEN 01

(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 55B7
#P.STOP             Set a pass point at STOP to prevent reboot
#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
63K CP/M VERS 1.3         forcing a front panel RST 7 (interrupt 7),
                    or simply bail-out with a cold start.
(16) SID SORT.COM SORT.SYM
SID VERS 1.4        Let's start again, but be a little more selective
SYMBOLS             in setting breakpoints.
NEXT  PC  END
0180 0100 55B7
#P.STOP             Set a pass point at STOP, as before
#P.SORT,FF          and one at SORT with a pass count of 255.
#-G                 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,
*010A .COMP         let's stop at the COMP label and watch the end test.
#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=010D CMP  M=00 .I
 -Z-EI A=00 B=006A D=00FF H=013F S=0100 P=010E JNZ  011C .CONT
 -Z-EI A=00 B=006A D=00FF H=013F S=0100 P=0111 LXI  H,013E .SW
 -Z-EI 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.
```

(17)

```
SID SORT.COM SORT.SYM
SID VERS 1.4              Let's re-enter SID with a clean memory
SYMBOLS                   image, and look at the machine code
NEXT  PC  END             below the "COMP" label.
0180 0100 55B7
#L.COMP
COMP:
  010A  LDA  0168 .N      Here's the reference to N - let's change this
  010D  CMP  M            to N-1 with a "hot patch" in memory, to see
  010E  JNZ  011C .CONT       if it works, then we'll go back to the
  0111  LXI  H,013E .SW       original source program and make the
  0114  MOV  A,M          necessary changes.  We're not using the area
#A10A                     of memory starting at 0200, so patch a jump
010A  JMP 200             over the LDA instruction, and fix-up some
010D                      patch code.
#A200
0200  LDA .N              Replace the LDA instruction which now has JMP 200.
0203  DCR A               N-1 in accumulator (N better be 2 or larger!)
0204  CMP M               and compare with memory (HL addresses I),
0205  JNZ .CONT           jump to CONT if continuing, otherwise
0208  JMP 111             jump back to the next instruction in sequence
020B                      after the patch.
#P205,FF                  Set a pass point to watch the JNZ take place
#P.STOP                   and catch any returns to the CCP.
#P111,FF                  Set a pass point at the patch return address.
#S.N                      Reduce the size of N for this test to 4.
0168 08 4
0169 00 .
#G                        Everything is ready, let's go...

FF PASS 0205              First pass through the patch code:
 ---EI 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:
 ----I A=03 B=0003 D=0000 H=013F S=0100 P=0205 JNZ  011C .CONT
FD PASS 0205              Went to CONT again, next pass:
 ----I A=03 B=0004 D=0001 H=013F S=0100 P=0205 JNZ  011C .CONT
FC PASS 0205              And so-forth:
 -Z-EI A=03 B=0004 D=0002 H=013F S=0100 P=0205 JNZ  011C .CONT
FF PASS 0111              Must be the end of one cycle:
 -Z-EI A=03 B=0004 D=0002 H=013F S=0100 P=0111 LXI  H,013E .SW
FB PASS 0205              Now back through the patch code:
 ---EI A=03 B=0004 D=0002 H=013F S=0100 P=0205 JNZ  011C .CONT
FA PASS 0205
 ----I A=03 B=0004 D=0000 H=013F S=0100 P=0205 JNZ  011C .CONT
F9 PASS 0205
 ----I A=03 B=0004 D=0001 H=013F S=0100 P=0205 JNZ  011C .CONT
F8 PASS 0205
 -Z-EI A=03 B=0004 D=0002 H=013F S=0100 P=0205 JNZ  011C .CONT
FE PASS 0111
 -Z-EI A=03 B=0004 D=0002 H=013F S=0100 P=0111 LXI  H,013E .SW
*0114                     This is getting monontonous again, so
#D.LIST,+=N-1             push the "return" key to stop the action.
0160: 03 04 05 0A ....    Data looks good, run in monitored mode:

-UFFFF
 -Z-EI A=03 B=0004 D=0002 H=013E S=0100 P=0114 MOV  A,M
*013B                     Push the "return" key to abort early.
#H=N                      Value of N is still 4 (that's nice!)
0004 #4                   Value of I is currently 2.  This program
#H=I                      should have stopped, but didn't for some
0002 #2                   reason.
```

```
(18) SID SORT.COM SORT.SYM
SID VERS 1.4          Let's try another approach.  Suppose we
SYMBOLS               construct a really trivial case:  we'll set
NEXT  PC  END         N = 2 (two items to sort), and
0180 0100 55B7              LIST(0) = 0, LIST(1) = 1
#S.N
0168 08 2
0169 00 .
#S.LIST
0160 05 0
0161 03 1
0162 04 .
                      Things are ready to go, run completely traced:
P.STOP
#TFFFF
 ----- 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
COMP:
 ----- 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
 ----I A=02 B=0000 D=0000 H=013F S=0100 P=010E JNZ  011C .CONT
CONT:
 ----I A=02 B=0000 D=0000 H=013F S=0100 P=011C LDA  013F .I
 ----I A=00 B=0000 D=0000 H=013F S=0100 P=011F MOV  E,A
 ----I A=00 B=0000 D=0000 H=013F S=0100 P=0120 MVI  D,00
 ----I A=00 B=0000 D=0000 H=013F S=0100 P=0122 LXI  H,0160 .LIST
 ----I A=00 B=0000 D=0000 H=0160 S=0100 P=0125 DAD  D
 ----I A=00 B=0000 D=0000 H=0160 S=0100 P=0126 MOV  A,M .LIST
 ----I A=00 B=0000 D=0000 H=0160 S=0100 P=0127 INX  H
 ----I 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
INCI:                  Not switched!
 C-ME- A=00 B=0000 D=0000 H=0161 S=0100 P=0137 LXI  H,013F .I
 C-ME- A=00 B=0000 D=0000 H=013F S=0100 P=013A INR  M=00 .I
 C---- A=00 B=0000 D=0000 H=013F S=0100 P=013B JMP  010A .COMP
COMP:
 C---- A=00 B=0000 D=0000 H=013F S=0100 P=010A LDA  0168 .N
 C---- A=02 B=0000 D=0000 H=013F S=0100 P=010D CMP  M=01 .I
 ----I A=02 B=0000 D=0000 H=013F S=0100 P=010E JNZ  011C .CONT
CONT:
 ----I A=02 B=0000 D=0000 H=013F S=0100 P=011C LDA  013F .I
 ----I A=01 B=0000 D=0000 H=013F S=0100 P=011F MOV  E,A
 ----I A=01 B=0000 D=0001 H=013F S=0100 P=0120 MVI  D,00
 ----I A=01 B=0000 D=0001 H=013F S=0100 P=0122 LXI  H,0160 .LIST
 ----I A=01 B=0000 D=0001 H=0160 S=0100 P=0125 DAD  D
 ----I A=01 B=0000 D=0001 H=0161 S=0100 P=0126 MOV  A,M
 ----I A=01 B=0000 D=0001 H=0161 S=0100 P=0127 INX  H
 ----I A=01 B=0000 D=0001 H=0162 S=0100 P=0128 CMP  M=04
 C-M-- A=01 B=0000 D=0001 H=0162 S=0100 P=0129 JC   0137 .INCI
INCI:                  Not switched (again)!
 C-M-- A=01 B=0000 D=0001 H=0162 S=0100 P=0137 LXI  H,013F .I
 C-M-- A=01 B=0000 D=0001 H=013F S=0100 P=013A INR  M=01 .I
 C---- A=01 B=0000 D=0001 H=013F S=0100 P=013B JMP  010A .COMP
COMP:
 C---- A=01 B=0000 D=0001 H=013F S=0100 P=010A LDA  0168 .N
 C---- A=02 B=0000 D=0001 H=013F S=0100 P=010D CMP  M=02 .I
 -Z-EI A=02 B=0000 D=0001 H=013F S=0100 P=010E JNZ  011C .CONT
 -Z-EI A=02 B=0000 D=0001 H=013F S=0100 P=0111 LXI  H,013E .SW
 -Z-EI A=02 B=0000 D=0001 H=013E S=0100 P=0114 MOV  A,M .SW
 -Z-EI A=01 B=0000 D=0001 H=013E S=0100 P=0115 ORA  A
 ----- A=01 B=0000 D=0001 H=013E S=0100 P=0116 JNZ  0100 .SORT
SORT:                  No items were switched - SW not set to 0!
 ----- A=01 B=0000 D=0001 H=013E S=0100 P=0100 LXI  H,013E .SW
*0103
```

19

```
ED SORT.ASM
*#AVFSORT:!ZOLT
    8:  SORT:   LXI     H,SW
    8: *-
    7:  ;
    7: *2
    9:          MVI     M,1     ;SW = 1
    9: *2S1!ZO!ZOLT
    9:          MVI     M,0     ;SW = 0
    9: *-
    8:  SORT:   LXI     H,SW
    8: *I
    8:          MVI     A,1
    9:          STA     SW      ;SW = 1 FIRST TIME THRU
   10:
   10: *E
```

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

20

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

Re-assemble, again

21

```
SID SORT.HEX SORT.SYM
SID VERS 1.4            We've fixed the SW initialization problem, which
SYMBOLS                 should halt the program at the proper time, but
NEXT  PC  END           we may still have a problem with the end of
016E 0100 55B7          LIST test (remember that "hot patch"?).
#D.LIST,+=N             Here's the initial data:
0165: 05 03 04 0A 08 82 0A 04 08 .........
#G,.STOP
                        GO, unmonitored to the STOP (how's that for
*011E .STOP             confidence?).
#D.LIST,+=N             We made it, here's the data:
0165: 03 04 04 05 08 08 0A 0A 0B 7B 82 ...........
0170: E6 .              Data is sorted in ascending order, but there's too
#ISORT.HEX              much of it!  We still have the problem that N is
#R                      altered during execution.
NEXT  PC  END           Let's reload and make sure we know what the
016E 0100 55B7          problem is-
#P.SORT                 Set a pass point at SORT, check N
#G

01 PASS 0105 .SORT Here's the first pass through SORT:
 -Z-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                 OK initially, continue the execution with G.
#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
#H=N
007B #123 '.'           N has been altered, which we expected, since we
#ISORT.HEX              are testing LIST(N-1) against LIST(N) and performing
#R                      a switch if unordered.
NEXT  PC  END
016E 0100 55B7          Let's reload and scope in on the problem:
#G,.INCI                Stop at the point where I becomes I + 1:

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                   Clear all pass points.
#-P
#G,.INCI                Now, try again:

*013C .INCI             Stopped at first entry to INCI, check value of N:
#H=N                    N is still 8, looks good.
0008 #8
#G,.CONT                Go to the CONT label, then stop at INCI.

*0121 .CONT
#G,.INCI
```

```
*013C .INCI           Back at INCI now.  Check value of N
#H=N
0008 #8               Remains at 8.  If we keep this up, we'll be typing
#P.INCI,6             break addresses all day.  We can run the next few passes
#-G                   through INCI automatically by setting a pass count (use 6
                      in this case), then run with -G to disable intermediate
01 PASS 013C          traces.  We now stop 6 iterations later:
 ---E- A=82 B=0004 D=0006 H=0143 S=0100 P=013C LXI  H,0144
*013F
#H=N                  Check N:  remains at 8, then
0008 #8               check I to compare passes:  I=0,1,2,3,4,5,6 has been
#H=I                  executed.  We are now about to set I = 7, but the test
0006 #6               at COMP is "JNZ" which allows execution one too many
                      times (which we already know about).
```

22

```
ED SORT.ASM
*#AV                        Back to the editor, change the end of LIST test
   1: *FLDA                 to compare I with N-1 rather than N.
  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:   ctl-Z
  20: *F*I              Now a little clean-up work - there is a typo in
  49: *OT               a comment line at address 012A in the listing:
  49:            MOV    M,A      ;NEW LIST*I*-C-DI(!ZOLT
  49:            MOV    M,A      ;NEW LIST(I+1) TO M   Looks better now.
  49: *F32              We are not using the 8080 stack, so get rid of it.
  64: *OLT
  64:            DS     32       ;16 LEVEL STACK
  64: *2KT
  64:  ;
  64: *E                Complete the edit.
```

23

```
MAC SORT
CP/M MACRO ASSEM 2.0
014F                        Re-assemble the source program.
001H USE FACTOR
END OF ASSEMBLY
```

24

```
SID SORT.HEX SORT.SYM
SID VERS 1.4
SYMBOLS                     Back to SID - this should be the last time!
NEXT  PC  END
014F 0100 55BF
#D.LIST,+=N                 Initial data:
0146: 05 03 04 0A 08 82 0A 04 08 .........
#G,STOP

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

*011F .STOP                 That's better, now look at the data:
#D.LIST,+=N                 hooray!  It's finally sorted.
0146: 03 04 04 05 08 0A 0A 82 08 .........
#H=N
0008 #8                     Is N ok?     Yes, it's still 8.
#G0                         Hold it!  The data is in ascending order, but it is
                            supposed to be in descending order!  This will
                            be an easy fix.
```

```
(25) ED SORT.ASM
*#A
*T
;       SORT PROGRAM IN CP/M ASSEMBLY LANGUAGE
*
;       ELEMENTS OF 'LIST' ARE PLACED INTO
*
;       DESCENDING ORDER USING BUBBLE SORT
*SDES!ZASC!ZOLT
;       ASCCENDING ORDER USING BUBBLE SORT
*SCC!ZC!ZOLT
;       ASCENDING ORDER USING BUBBLE SORT
*E                         Took care of that problem.

(26) MAC SORT $+S
CP/M MACRO ASSEM 2.0
014F                       Re-assemble with the symbol table option.
001H USE FACTOR
END OF ASSEMBLY
```

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     0000H   ;CP/M REBOOT LOCATION
                   ;
0100 3E01                  MVI     A,1
0102 324401                STA     SW      ;SW = 1 FIRST TIME THRU
0105 214401        SORT:   LXI     H,SW
0108 3600                  MVI     M,0     ;SW = 0
010A 214501                LXI     H,I     ;INDEX TO SORT LIST
010D 3600                  MVI     M,0     ;I = 0
                   ;
                   ;       COMPARE I WITH ARRAY SIZE
                   COMP:   ;HL ADDRESS INDEX I
010F 3A4E01                LDA     N       ;LENGTH OF VECTOR
0112 3D                    DCR     A       ;N-1 IN A REGISTER
                   ;       (NOTE THAT N MUST BE 2 OR LARGER)
0113 BE                    CMP     M       ;CHECK FOR N=I
0114 C22201                JNZ     CONT    ;CONTINUE IF UNEQUAL
                   ;
                   ;       END OF ONE PASS THROUGH LIST
0117 214401                LXI     H,SW    ;NO SWITCHES?
011A 7E                    MOV     A,M     ;FILL A WITH SW
011B B7                    ORA     A       ;SET FLAGS
011C C20501                JNZ     SORT    ;CONTINUE IF NOT EQUAL
                   ;
                   ;       END OF SORT PROCESS, REBOOT
011F C30000        STOP:   JMP     REBOOT  ;RESTART CCP
                   ;
                   ;       CONTINUE THIS PASS
                   CONT:
0122 3A4501                LDA     I       ;LOAD I TO A REGISTER
0125 5F                    MOV     E,A     ;LOW(I) TO E REGISTER
0126 1600                  MVI     D,0     ;HIGH(I) = 0
0128 214601                LXI     H,LIST  ;BASE OF LIST
012B 19                    DAD     D       ;ADDR LIST(I)
012C 7E                    MOV     A,M     ;LIST(I) IN A REGISTER
012D 23                    INX     H       ;ADDR OF LIST(I+1)
012E BE                    CMP     M       ;LIST(I):LIST(I+1)
012F DA3D01                JC      INCI    ;SKIP IF PROPER ORDER
                   ;
                   ;       CHECK FOR LIST(I) = LIST(I+1)
0132 CA3D01                JZ      INCI    ;SKIP IF EQUAL
                   ;
                   ;       ITEMS ARE OUT OF ORDER, SWITCH
0135 4E                    MOV     C,M     ;OLD LIST(I+1) TO C
0136 77                    MOV     M,A     ;NEW LIST(I+1) TO M
0137 2B                    DCX     H       ;ADDR LIST(I)
0138 71                    MOV     M,C     ;NEW LIST(I) TO M
                   ;
0139 214401                LXI     H,SW    ;SWITCH COUNT IS SW
013C 34                    INR     M       ;SW = SW + 1
                   ;
                   INCI:   ;INCREMENT INDEX I
013D 214501                LXI     H,I
0140 34                    INR     M       ;I = I + 1
0141 C30F01                JMP     COMP    ;TO COMPARE I WITH N-1
                   ;
                   ;       DATA AREAS
0144               SW:     DS      1       ;SWITCH COUNT
0145               I:      DS      1       ;INDEX
                   ;
0146 0503040A08LIST:       DB      5,3,4,10,  8,130,10,4
014E 08            N:      DB      $-LIST  ;LENGTH OF LIST
014F                       END

010F COMP     0122 CONT     0145 I       013D INCI     0146 LIST
014E N        0000 REBOOT   0105 SORT    011F STOP     0144 SW
```

(27)

```
SID HIST.UTL                     Start SID with the HIST utility
SID VERS 1.4
TYPE HISTOGRAM BOUNDS 100,200    Monitor 0100 through 0200.
.INITIAL = 5221
.COLLECT = 5224                  Entry point addresses in HIST.
.DISPLAY = 5227
#ISORT.HEX SORT.SYM              Load the SORT program with symbols.
#R
SYMBOLS                          Program loaded, now loading symbols.
NEXT  PC  END
0600 0100 51B7
#P.STOP                          Permanent break at STOP address.
#P.SORT,3                        Execute to "steady state" conditions by
#-G                              passing the SORT label three times before break.
                                 "-G" prevents intermediate pass traces.
01 PASS 0100
 ----- A=02 B=0004 D=0006 H=013F S=0100 P=0100 LXI  H,013F
*0103                            We're now at the third pass through SORT.
#-P.SORT                         Remove the pass point at SORT, run monitored
#UFFF,.COLLECT                   from this point for 0FFF steps, collect data.
 ----- A=02 B=0004 D=0006 H=013F S=0100 P=0103 MVI  M,01 .SW
*0127                            Stopped after 0FFFsteps, display collected data:
#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        What's happening around the most frequently executed address?
  010C  LXI  B,BE3D
  010F  JNZ  011D .CONT   This is where the  end of LIST test takes place,
  0112  LXI  H,013F .SW   so it is reasonable that this segment of code would
  0115  MOV  A,M          be executed heavily.  We could improve performance
  0116  ORA  A            by reducing the length of this segment.  The value
  0117  JNZ  0100 .SORT   of N-1 could, for example, be maintained in register
STOP:                     C throughout the computations, while the value of
  011A  JMP  0000 .REBOOT I could be kept in register E, with 00 in D.
#L11C
  011C  NOP               There is also heavy execution around location 011C.
CONT:
  011D  LDA  0140 .I      This is where we go on each element comparison
  0120  MOV  E,A          whether we switch elements or not.
  0121  MVI  D,00
  0123  LXI  H,0161 .LIST
  0126  DAD  D
  0127  MOV  A,M
  0128  INX  H
  0129  CMP  M
  012A  JC   0138 .INCI
  012D  JZ   0138 .INCI
#GO
```

```
(28) SID TRACE.UTL              Load the TRACE utility with SID.
SID VERS 1.4
INITIAL = 5321
COLLECT = 5324                 TRACE entry points.
DISPLAY = 5327
READY FOR SYMBOLIC BACKTRACE     Indicates that assembler/disassembler is present.
#ISORT.HEX SORT.SYM            Ready the SORT program and symbol table.
#R                             Load program and symbols to memory.
SYMBOLS
NEXT  PC  END
0600 0100 52B7
#P.STOP                        Permanent break at the STOP label.
#P.CONT,3                      Pass through CONT three times before stopping.
#UFFFF,.COLLECT                Untrace mode, print intermediate pass points.
 ----- 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
 ---EI A=07 B=0003 D=0000 H=0140 S=0100 P=011D LDA  0140 .I
01 PASS 011D .CONT
 ---EI A=07 B=0004 D=0001 H=0140 S=0100 P=011D LDA  0140 .I
*0120                          Stopped on the third pass.
#C.DISPLAY                     Display the backtrace from CONT.
BACKTRACE:
CONT:                          Most recently executed instruction.
  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  DCX  H
  0131  MOV  M,A
  0130  MOV  C,M
  012D  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:                          Least recently executed instruction.
  010A  LDA  0169 .N           (aborted with "return")
#GO
```

29
```
SID                                  Start SID without loading any programs.
SID VERS 1.4
#-A                                  Remove assembler/disassembler package.
#ITRACE.UTL                          Ready the TRACE utility.
#R                                   Read the TRACE package to memory.
INITIAL = 5921
COLLECT = 5924                       TRACE entry point addresses.
DISPLAY = 5927
"-A" IN EFFECT, ADDRESS BACKTRACE    No assembler/disassembler present.
#ISORT.HEX SORT.SYM                  Ready the SORT program
#R                                   Read to memory.
SYMBOLS
NEXT  PC  END
0600 0100 58B7
#P.STOP                              Permanent break at STOP address,
#P.CONT,3                            pass point at CONT with pass count 3
#-UFFFF,.COLLECT                     Run monitored, collect data, no intermediate
 ----- A=00 B=0000 D=0000 H=0000 S=0100 P=0100 21 013F      pass information.
01 PASS 011D
 ---EI A=07 B=0004 D=0001 H=0140 S=0100 P=011D 3A 0140
*0120                                Stopped on third pass through CONT
#C.DISPLAY
BACKTRACE:  most recent addresses
011D 010F 010E 010D 010A 013C 013B 0138
0137 0134 0133 0132 0131 0130 012D 012A
0129 0128 0127 0126 0123 0121 0120 011D
010F 010E 010D 010A 013C 013B 0138 0137
0134 0133 0132 0131 0130 012D 012A 0129
0128 0127 0126 0123 0121 0120 011D 010F
010E 010D 010A 0108 0105 0103 0100 least recent address.
#GO
```

30
```
TYPE IO.PRN

                        ;       SIMPLE BDOS OUTPUT PROGRAM
 0100                           ORG     100H    ;BEGINNING OF TPA
 0000 =                 REBOOT  EQU     0000H   ;REBOOT ENTRY POINT
 0005 =                 BDOS    EQU     0005H   ;BDOS ENTRY POINT
 0002 =                 CONOUT  EQU     2       ;CONSOLE OUTPUT #
                        ;
 0100 315401                    LXI     SP,STACK;LOCAL STACK
 0103 C31501                    JMP     START   ;START EXECUTION
                        ;
                        WRCHAR: ;WRITE CHARACTER FROM REGISTER A
 0106 0E02                      MVI     C,CONOUT;CONSOLE OUTPUT #
 0108 5F                        MOV     E,A     ;CHARACTER TO E
 0109 C30500                    JMP     BDOS    ;RET THROUGH BDOS
                        ;
                        WRMSG:  ;WRITE MESSAGE STARTING AT HL 'TIL 00
 010C 7E                        MOV     A,M     ;NEXT CHARACTER
 010D B7                        ORA     A       ;00?
 010E C8                        RZ              ;RETURN IF SO
 010F CD0601                    CALL    WRCHAR  ;OTHERWISE WRITE IT
 0112 C30C01                    JMP     WRMSG   ;FOR ANOTHER CHARACTER
                        ;
                        START:  ;BEGINNING OF MAIN PROGRAM
 0115 212A01                    LXI     H,WALLAMSG      ;PART 1 OF MESSAGE
 0118 CD0C01                    CALL    WRMSG           ;WRITE IT
 011B 212A01                    LXI     H,WALLAMSG      ;PART 2 OF MESSAGE
 011E CD0C01                    CALL    WRMSG           ;WRITE IT
 0121 213001                    LXI     H,WASHMSG       ;PART 3 OF MESSAGE
 0124 CD0C01                    CALL    WRMSG
 0127 C30000            STOP:   JMP     REBOOT          ;STOP THE PROGRAM
                        ;
                        ;       DATA AREAS
                        WALLAMSG:
 012A 57414C4C41                DB      'WALLA '
                        WASHMSG:
 0130 57415348                  DB      'WASH'
 0134                           DS      32      ;16 LEVEL STACK
                        STACK:
 0154                           END
```

(31)

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

*010C .WRMSG          Now trace from the WRMSG subroutine:
#T100
 ----- A=00 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 .WRCHAR    First
WRCHAR:                                                   call to WRCHAR
 ----- A=57 B=0000 D=0000 H=012A S=0150 P=0106 MVI  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:                                                     Call to 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 6D52      (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  6E06W = first character
 -Z-E- A=00 B=0000 D=0200 H=793B S=0152 P=0112 JMP  010C .WRMSG    now we're
WRMSG:                                                    back to our
 -Z-E- A=00 B=0000 D=0200 H=793B S=0152 P=010C MOV  A,M       program, with
 -Z-E- A=00 B=0000 D=0200 H=793B S=0152 P=010D ORA  A         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 LXI  H,012A .WALLAMSG
 -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=010D 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 .WRCHAR
WRCHAR:
 ----- A=57 B=0000 D=0200 H=012A S=0150 P=0106 MVI  C,02
 ----- A=57 B=0002 D=0200 H=012A S=0150 P=0108 MOV  E,A      abort with "return"
*0109
#G,.WRMSG    GO, skip traces
W            Should be ALLA ..., what happened?
*010C .WRMSG
#TW100       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 LXI  H,0130 .WASHMSG
 -Z-E- A=00 B=0000 D=0200 H=0130 S=0154 P=0124 CALL 010C .WRMSGW
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
*7A03
#                It appears that the WRMSG subroutine is not saving the HL
                 register pair, nor is HL being incremented on each loop.
```

```
#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.
0200  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.
*010C .WRMSG         Here we are.  HL addresses the message to print, which
#D                   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 45 52 53 20 31 2E 34 24 31 00 02 WASHVERS 1.4$1..

#TW100               Trace without calls:  shows only the activity in WRMSG.
 ----- A=00 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          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
 ----- A=57 B=0000 D=0000 H=012A S=0152 P=0200 PUSH H          area.
 ----- A=57 B=0000 D=0000 H=012A S=0150 P=0201 CALL 0106 .WRCHARW = character
 -Z-E- A=00 B=0000 D=0200 H=793B S=0150 P=0204 POP  H
 -Z-E- A=00 B=0000 D=0200 H=012A S=0152 P=0205 INX  H          Move to next
 -Z-E- A=00 B=0000 D=0200 H=012B S=0152 P=0206 JMP  010C .WRMSG  character
WRMSG:                                                         Looping back.
 -Z-E- A=00 B=0000 D=0200 H=012B S=0152 P=010C MOV  A,M
 -Z-E- A=41 B=0000 D=0200 H=012B S=0152 P=010D ORA  A
 ---E- A=41 B=0000 D=0200 H=012B S=0152 P=010E RZ
 ---E- A=41 B=0000 D=0200 H=012B S=0152 P=010F JMP  0200
 ---E- A=41 B=0000 D=0200 H=012B S=0152 P=0200 PUSH H          Here's the next
 ---E- A=41 B=0000 D=0200 H=012B S=0150 P=0201 CALL 0106 .WRCHARA character
 -Z-E- A=00 B=0000 D=0200 H=793B S=0150 P=0204 POP  H                (="A")
 -Z-E- A=00 B=0000 D=0200 H=012B S=0152 P=0205 INX  H
 -Z-E- A=00 B=0000 D=0200 H=012C 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
*010D                                                          Abort with "return"
#P.STOP              Set a permanent break at STOP, then GO from
#G100                the beginning of the program:
WALLA WASHVERS 1.4$1WALLA WASHVERS 1.4$1WASHVERS 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.WALLAMSG+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 00 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.
WALLAWALLAWASH       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
```