DEBUG MODE

The Symbolic Debugging System (SDS) is a conversational facility for testing and debugging programs. This facility was originally provided for 360/370-assembler language programs, but it has been extended to include FORTRAN, PL/I, and PL360 programs. SDS enables the user to initiate the execution of a program and monitor its performance by displaying or modifying instructions and data at strategic points in the program.

SDS may be invoked in an explicit manner with the MTS command

```
$DEBUG FDname
```

where "FDname" is the file or device containing the program to be debugged. The parameters to the $DEBUG command are the same as those for the $RUN command; hence, if the program refers to logical I/O units or has a PAR field, these also may be included on the $DEBUG command, i.e.,

```
$DEBUG program SCARDS=input SPRINT=output PAR=options
```

The user communicates with SDS by entering debug commands from his terminal. SDS prints the prefix character "+" when it is requesting a command. This prefix character also precedes all SDS messages and diagnostics. Initially, debug commands are read from the pseudo-device *SOURCE* and SDS output messages and error diagnostics are written on *SINK*. Commands may be entered in either upper- or lower-case.

When the program has been successfully loaded, SDS prints

```
+READY
+
```

at which point SDS is ready for its first command. Debug commands may be used to execute a portion (or all) of the program being debugged, to establish special conditions that SDS should recognize, or to request that specific storage locations be displayed or modified. For instance, the RUN command initiates execution of the program, and the MAP command lists relevant information about each loaded section. The BREAK command is provided to insert breakpoints at strategic locations in the program. If a program interrupt occurs during execution, or if a breakpoint is encountered, control is returned to SDS which explains why the program has stopped running and requests another command. The DISPLAY command may then be used to selectively examine specific locations in memory or the contents of the general and floating-point registers. Changes to data and instructions may be made with the MODIFY command. The program may be restarted with the CONTINUE command. In this manner, the user may "step" through the program and monitor its performance. A complete description of the debug command language appears at the end of this section.

Logical I/O unit assignments and PAR field information may be specified on the $DEBUG command in the same format as with the $RUN command. If the user has omitted these assignments from the $DEBUG command or desires to change or reset any of the assignments, this may be done by the SET debug command. For example, after issuing the MTS command

```
$DEBUG -LOAD
```

the debug command

```
SET 5=DATA 6=RESULTS PAR=LIST
```

assigns the files DATA and RESULTS to logical I/O units 5 and 6, respectively, and sets the PAR field to the character string LIST. This is the same as if the MTS command

```
$DEBUG -LOAD 5=DATA 6=RESULTS PAR=LIST
```

had been issued originally.

Assigning logical I/O units with the SET command may be used to rewind sequential or line files or to assign new line number ranges for line files but will not affect the status of other types of pseudo-devices (such as *SOURCE* and the position of tapes). The PAR option must be the last parameter specified since the remainder of the input line is taken as the PAR field.

Alternatively, SDS may be invoked implicitly by entering the MTS command

```
$SET DEBUG=ON
```

All programs loaded after this point are processed by SDS. The MTS commands $RUN and $START/$RESTART do not start execution of the program directly, but transfer control to SDS with an implicit debug command of RUN or CONTINUE (respectively). SDS will intercept any error that occurs, print an error comment, and return to the caller (normally MTS command mode). At this point the user may explicitly enter debug command mode with the MTS command

```
$SDS
```

to determine what went wrong with his program. This mode of operation may by disabled by the MTS command

```
$SET DEBUG=OFF
```

The default for the DEBUG option is OFF. When the DEBUG option is ON, all user programs, language translators, and other publicly available programs are processed by SDS; hence, the user should be aware of which programs he does and does not want SDS to process. Normally, when the user is not running one of his own programs, the DEBUG option should be OFF.

The user may specify a local time limit on the $DEBUG command, e.g.,

$DEBUG program T=s

The time limit specified applies only to the execution time of the user's program; time spent in processing SDS commands is not included. When the local time limit is exceeded, execution of the user's is suspended. The user may subsequently resume execution with the CONTINUE command. When execution is resumed, a new local time limit is established which is the time limit specified by the MTS TIME option; the default is no time limit.

BASIC CONCEPTS

The most useful feature of SDS is the ability to selectively refer to locations in the program being debugged. Locations referenced may be either locations within the executable code (e.g., the entry point to a subroutine) or locations that contain data. Storage locations may be referenced symbolically, by relative address, or by absolute (virtual) address. In the symbolic mode, a location is referred to by the symbol assigned to that location at program translation time.

Symbolic referencing is possible only with those language processors which generate a symbol table with the object programs they produce. This symbolic information is contained in SYM records in the object module. Currently, SYM records are generated by the 360/370 Assembler G (*ASMG), the TSS Assembler (*ASMT), the FORTRAN-G and FORTRAN-H compilers (*FTN), the PL/I-F compiler (*PL1), and the PL360 compiler (*PL360). With these processors, the TEST parameter must be specified for the processor to produce symbol table information. The formats are:

```
$RUN *ASMG SCARDS=source SPUNCH=object PAR=TEST

$RUN *ASMT SCARDS=source SPUNCH=object PAR=TEST

$RUN *FTN SCARDS=source SPUNCH=object PAR=TEST

$RUN *FTN SCARDS=source SPUNCH=object PAR=OPT=H,TEST

$RUN *PL1 SCARDS=source SPUNCH=object PAR=TEST

$RUN *PL360 SCARDS=source SPUNCH=object PAR=TEST
```

When referencing by relative or virtual addresses, SDS will process all loaded programs which were produced by a language translator that generates an object module. This excludes interpretive translators such as WATFIV, IF, and SNOBOL4.

Storage Reference Expressions

Since the principle function of SDS is to provide a convenient means of debugging programs, a powerful and flexible method for specifying storage locations must be available. The means for specifying storage locations must have the capability of resolving potential ambiguities that may arise when character strings may be interpreted as symbols or as hexadecimal numbers or when strings of hexadecimal digits may be confused with strings of decimal digits. The constructs that are used to refer to storage locations are call storage reference expressions. Throughout this description of SDS, the terms "location" and "storage reference expression" are used interchangeably.

The general form of as SDS storage reference expression is

S(i)±j

where "S" may be

(1) a symbolic location in the source language program,
(2) a relative address within an object module assigned at assembly or compilation time, or
(3) an absolute (virtual) address corresponding to a location within the loaded program.

"i", an optional index, is, in its simplest form, a signed or unsigned decimal integer, and "j", an optional displacement, is a hexadecimal integer. In determining the location to be referenced, the index specifies the ith element (e.g., data item, instruction) of an array relative to the base "S" and the displacement specifies the number of bytes to be added to (or subtracted from) the location indicated by S(i).[1] A storage reference expression is terminated by a blank. Examples of valid storage reference expressions are:

Expression	Form
ALPHA	S
ALPHA(4)	S(i)
ALPHA(-3)	S(i)
BETA+1C	S+j
BETA(24)+10	S(i)+j

When specifying the "S" component of a storage reference expression, it may be necessary to distinguish explicitly between a symbolic, relative, or absolute address. For example, "A14" could be interpreted as the symbol A14 or the relative or absolute address A14 (hex). By default, an "S" component beginning with a letter is treated as a symbol; if it is not a defined

---

[1]The complete specification for the components of a storage reference expression are more complex than the simple definitions given here. The complete definition is given later in this subsection.

symbol within the program, an attempt is made to treat it as a hexadecimal relative or absolute address. If the "S" component begins with a decimal digit (0-9), the symbol is always treated as a relative or absolute address. Furthermore, an "S" component that does not begin with a letter is treated by default as a relative hexadecimal address if its value is less than 100000 (hex) and is treated as an absolute hexadecimal address if its value is equal to or greater than 100000 (hex). The following table provides several examples of default interpretation.

Expression	Interpretation
A14	symbol or relative address
0A14	relative address
B1234	symbol or relative address
5B1234	absolute address
XYZ	symbol only

When it is necessary to override the default interpretation, one of the modifiers @Q, @R, or @A, may be appended to the expression to force interpretation as a symbol, relative address, or absolute address, respectively. Thus, 0A14 and A14@R both specify a relative hexadecimal address location and B1234@A specifies an absolute hexadecimal address.

As stated above, the simplest forms for indices and displacements are decimal and hexadecimal constants, respectively. When it is more convenient to represent an index using a hexadecimal constant or a displacement using a decimal constant, the modifiers @X and @D may be appended to the constant to force interpretation as a hexadecimal or decimal number, respectively. Thus,

```
BETA(18@X)+16@D
```

is equivalent to

```
BETA(24)+10
```

For FORTRAN or PL/I arrays, the index "i" may be a series of indices separated by commas. Multiple indices may specified, corresponding to the

```
A(6)
B(4,3)
C(0,0,0,-1)
```

To provide further generality in the specification of storage references, both indices "i" and displacements "j" may be storage reference expressions. An index may be given in the form S(i) and a displacement may be given in the form S(i). Displacements may not be used on recursive storage reference expressions. When an index or displacement is expressed as a storage reference rather than as a simple numeric constant, the contents of the location specified is used for the index or displacement. As many levels of recursion as desired may be specified. For example, the expression

```
A(K)
```

specifies that the contents of the location whose symbolic reference is K is to be used as the index of the array A in computing the actual location referenced. Similarly,

```
B(A(K))
```

is a storage reference using two levels of recursion. Indices and displacements are treated as numeric constants where possible. Thus,

```
ALPHA+A
```

refers to the location ALPHA plus ten bytes since displacements are normally treated as hexadecimal constants, but

```
ALPHA+A@Q
```

refers to the location ALPHA plus a displacement specified by the contents of the location whose symbol is A (@Q forces symbolic interpretation) and

```
ALPHA+A@R
```

refers to the location ALPHA plus a displacement specified by the contents of location whose relative address within the program is 00000A (hex). Multiple symbolic indices may be used for FORTRAN and PL/I arrays.

In addition, "i" and "j" may also be arithmetic expressions using the operators +, -, *, and / and any necessary levels of parentheses to achieve the desired order of computation. Elements in these arithmetic expressions may be constants or storage reference expressions. For example,

```
ALPHA(I+2)
ALPHA(I*2+1)
ALPHA((I+10)/2)
ALPHA+7A0-16
ALPHA+(1C+20)*2
```

Numeric constants used in indices are treated as decimal while numeric constants used in displacements are treated as hexadecimal unless overridden by the @X or @D modifiers, respectively.

The following 360/370-assembler language program illustrates the use of storage reference expressions.

```
000000                          PROG    CSECT
000000                                  USING *,15
000000  5810 F014                       L       1,ALPHA
000004  5A10 F018                AD      A       1,C
000008  5010 F014                       ST      1,ALPHA
00000C  58A0 F01C                       L       10,VSYS
000010  07FA                           BR      10
000014  0000000A                ALPHA   DC      F'10'
000018  00000014                C       DC      F'20'
00001C  00000000                VSYS    DC      V(SYSTEM)
000020                          GAMMA   DS      20F
000070  1C022C333C1C022C        DELTA   DC      2P'1,22,333'
                                        END
```

To display the current contents of a location, the DISPLAY command may be used, e.g.,

```
DISPLAY ALPHA
```

converts the contents of ALPHA according to its type and length (as determined by information provided by SYM records) and prints the result. The type refers to the format in which it is displayed (e.g., fixed-point, floating-point, character, hexadecimal, etc.) and the length refers to the number of bytes that are displayed. If ALPHA (a fullword integer) from the above program is displayed (before the program is executed), the result is

```
ALPHA  'F'  +10  (4 bytes)
```

The length is given for types E, F, P, and Z. Similarly,

```
DISPLAY C
```

would produce

```
C  'F'  +20
```

Locations may also be referred to by the relative addresses assigned at translation time. To display the contents of the relative location C, as opposed to the symbolic location C, the command

```
DISPLAY 0C
```

or

```
DISPLAY C@R
```

may be used. This displays the instruction at location 00000C in the form

```
AD(3)  'I'  L   A0F01C
```

The @R modifier is used to indicate a relative address. If no symbol table was provided at translation time, the contents of the location is printed in hexadecimal format as

```
00000C  'X'  58A0F01C
```

The user may display any addressable location in his virtual memory by specifying an absolute (virtual) address. To display the contents of the absolute location 516A28, the command is

```
DISPLAY 516A28
```

If possible, the contents of 516A28 are converted according to type and length and displayed on the user's terminal. If the address given specifies an illegal, an unreferenced, or a protected page, an error message is produced.

A special form of storage reference expression, the block storage reference (or simply the storage block), is provided to specify a range of locations. A storage block is represented by two storage reference expressions separated by exactly three periods (no embedded blanks are allowed). The general form is

```
S(i)±j...S(i)±j
```

For example, to display the linear array GAMMA of dimension 20, the command is

```
DISPLAY GAMMA(1)...GAMMA(20)
```

If the symbol for the upper-bound address is identical to that of the lower-bound address, it may be omitted, e.g.,

```
DISPLAY GAMMA(1)...(20)
```

Blocks may be used to specify any range of addresses as shown in the following examples:

```
DISPLAY ALPHA...GAMMA(2)
DISPLAY 9C0...A28@B
DISPLAY 516000...516A28
```

The upper-bound address must be greater than the lower-bound address.

The user may specify a relocation factor with the SET debug command. This relocation factor is added to every relative address specification that is not overridden by the @C modifier. For example, the command sequence

```
SET RF=517000
DISPLAY 2A0
```

displays the contents of location 5172A0. This make it convenient to display locations using only the relative addresses given in an assembly listing.

Keyword Modifiers

It is often useful to override the implied or assumed type, length, or scale attributes of a referenced location. SDS provides a set of keyword modifiers which temporarily override the attributes by explicitly specifying new attributes. A keyword modifier consists of the modifier character "@" followed by a keyword describing the modifier.

SDS recognizes the following keyword modifiers:

@TYPE=xLiSj sets the type, length, and scale attributes to "x", "i", and "j", respectively. The length and/or scale attributes may be omitted. "x" may be any of the type-codes defined below; "i" and "j" may be any length or scale attribute as defined with the LENGTH and SCALE modifiers below.

x	Type Attribute
A	A-type address constant
B	binary
C	character (EBCDIC)
D	floating-point (long)
E	floating-point (long or short)

F	fixed-point integer
H	fixed-point integer (halfword)
I	instruction
L	logical (FORTRAN)
M	complex (FORTRAN)
P	packed decimal
S	S-type address constant
V	V-type address constant
W	channel command word
X	hexadecimal
Y	Y-type address constant
Z	zoned decimal

@LENGTH=i sets the length attribute to "i", where "i" is an unsigned decimal integer. "i" may range from 1 to 65535. The standard default is 4; this is the value used when there is no symbol information associated with the specified address.

@SCALE=j sets the binary scale factor to "j", where "j" is a signed or unsigned decimal integer. The standard default is 0. Binary scale factors follow the conventions used by the 360/370-assembler language, i.e., the scale factor treats the the constant as if it were multiplied internally by the specified power of two.

@CSECT=cs restricts the storage reference to the control or common section "cs".

@GEN=i restricts the storage reference to generation "i" of the current control section.

@DSECT=ds restricts the storage reference to the dsect "ds".

@PRO=xxx restricts the storage reference to the external procedure "xxx" (PL/I programs only).

@BLOCK=i restricts the storage reference to block level "i" of an external procedure (PL/I programs only).

@INV=i restricts the storage reference to invocation "i" of a recursive procedure or controlled data variable (PL/I programs only).

@#nn specifies the compilation statement number for the data variable (PL/I programs only).

Keyword modifiers may be appended either to commands or to storage reference expressions. If they are appended to a command, they apply to all storage reference expressions appearing in the command; if they are appended to a storage reference expression, they apply only to that expression. A keyword modifier appended to a storage reference expression overrides a keyword appended to a command.

For example, the user may wish to display a symbol in its hexadecimal representation. In the following example, the TYPE keyword modifier is appended to the symbol ALPHA. The code following the "=" is the type code for hexadecimal formatting, as defined below. The command

```
DISPLAY ALPHA@TYPE=X
```

displays the contents of ALPHA (from the previous program) as

```
ALPHA  'X'  0000000A
```

If a length attribute of 8 were also applied, the command would be

```
DISPLAY ALPHA@TYPE=XL8
```

and the symbol would be displayed as

```
ALPHA  'X'  0000000A  00000014
```

Only the first letter of any keyword modifier need be entered; hence, the preceding example could be given as

```
DISPLAY ALPHA@T=XL8
```

If both ALPHA and BETA are to be displayed using a length attribute of 8, the TYPE modifier may be appended to the command, e.g.,

```
$DISPLAY@T=8 ALPHA BETA
```

As many keyword modifiers as needed may be appended to a parameter. For blocks, the TYPE, LENGTH, and SCALE modifiers may be appended to either the upper-bound or lower-bound address, e.g.,

```
DISPLAY 516100@T=FL4...516120
```

or

```
DISPLAY 516100...516120@T=FL4
```

It is often desirable to restrict the use of an SDS parameter to a single control or common section by the use of the CSECT keyword modifier. This is normally done when a symbol or relative address is defined in more than one section. For example,

```
DISPLAY BETA@C=SUBR
```

displays the contents of location BETA in the section SUBR. If the CSECT modifier is used with the lower-bound address of a block parameter, it applies to both addresses (unless overridden on the upper-bound address); if the CSECT modifier is used with the upper-bound address, it applies only to the upper-bound address.

The symbol BLANK may be used for referring to blank-named common sections. For example, in a FORTRAN program, the variable A in a blank-named common section may be displayed by

```
DISPLAY A@C=BLANK
```

If the program already has a control section or common section by the name of BLANK, the SET command may be used alter the blank-named common symbol. The use of BLANK as a blank-named common symbol will not interfere with the use of BLANK for other symbols in the program.

If a control section of a given name appears more than once in the program, the GEN keyword modifier can be used to specify the desired occurrence of that section. The first time a control section of a given name is loaded, it is assigned the generation number 1, the second time it is loaded, it is assigned the generation number 2, etc. The command

```
DISPLAY BETA@C=SUBR@G=2
```

displays the contents of location BETA in the second occurrence of section SUBR.

The DSECT keyword modifier can be used to restrict an SDS parameter to a particular dummy control section, or dsect, when that symbol is defined in both a control section and a dsect.

The modifier character may be changed by the SET MODCH=x command, so that symbols containing an "@" character can be referred to in SDS parameters.

Predefined Symbols

SDS predefines a small number of "useful" symbols. For instance, to display the contents of all of the general registers, the predefined symbol "GRS" is used, e.g.,

```
DISPLAY GRS
```

To refer to the ith general register, the symbol "GRi" is used, where "i" is the register number, ranging from 0-15 (decimal), or 0-9 and A-F (hex).

Similarly, to refer to the floating-point registers, the symbol "FRS" is used. To refer to the jth floating-point register, the symbol "FRj" is used, where "j" is 0, 2, 4, or 6.

The @F=F and @F=E modifiers may be applied to GRS and FRS, respectively, to obtain fixed-point and floating-point conversion of the general and floating-point registers.

To refer to the current program status word (PSW), the symbol "PSW" is used.

To refer to a program-defined location with the same name as an SDS predefined symbol, the @Q modifier must be appended to the symbol. Thus, to display the program-defined symbol GR7, the command is

```
DISPLAY GR7@Q
```

whereas, to display general register 7, the command is

```
DISPLAY GR7
```

Indirection

A level of indirection through any storage reference may be achieved by placing the indirection character "$" at the front of the storage reference. The storage reference is assumed to contain a four-byte address which is used by SDS as the effective address. For example, to display the contents of the location whose address is contained in general register 6, the command is

```
DISPLAY $GR6
```

whereas, to display the contents of general register 6, the command is

```
DISPLAY GR6
```

As many levels of indirection as needed may be achieved by placing the appropriate number of indirection characters at the front of the parameter. For example, to display the location whose address is contained in the location pointed to by general register 2, the command is

```
DISPLAY $$GR2
```

SDS stops at the first illegal address encountered and reports the indirection level. An illegal address is an address which specifies a protected page, an unreferenced page, or a nonexistent page. The chain of indirection need not be contained in SDS-processed control sections.

By convention, the indirection operator applied to the PSW, i.e.,

```
DISPLAY $PSW
```

displays the location addressed by the current PSW.

The indirection operator has higher precedence than "±" used to add a displacement, e.g., the expression $ADDR+4 takes the contents of ADDR and then adds 4 to form a new address, whereas the expression $(ADDR+4) adds 4 to the address of ADDR to form the address used for the indirection. A storage reference may be parenthesized to any desired level to cause the indirection operator to be applied to the desired part. The indirection operator normally applies to a symbol and its index (but not the displacement). The following table shows how parenthesizing may be used.

Parameter	Part Operated on by $
$ADDR+4	ADDR
$(ADDR+4)	ADDR+4
$ADDR(2)	ADDR(2)
$(ADDR)(2)	ADDR(2)
($ADDR)(2)	ADDR
$ADDR(2)+4	ADDR(2)
$(ADDR(2)+4)	ADDR(2)+4

The indirection character may be changed by the SET INDCH=x command, so that symbols beginning with a "$" can be referred to in SDS parameters.

SDS Constants

SDS constants may be specified with the MODIFY and SCAN commands, i.e.,

```
MODIFY S(i)±j constant
SCAN S(i)±j constant
```

In addition, they may be elements in SDS relational and logical expressions (see the subsection "SDS Relational and Logical Expressions").

The format of the SDS constant allows the inclusion of type, length, and scale factor attributes on the constant expression (in a manner similar to the 360/370-assembler language). The prototype is

```
xLiSj'nnnn'
```

where "x" is the type, "Li" is the length, "Sj" is the scale factor attribute, and "nnnn" is the constant; the type, length, and/or scale factor attributes may be omitted. The values for "x", "i", and "j" may be any of the values that are legal with the @T=x, @L=i, and @S=j modifiers. If the type, length, or scale factor attributes are omitted, the standard defaults are assumed. Examples of SDS constants are

```
A'LABEL'
AL2'LABEL2'
C'A  ' or CL3'A'
F'3'
FL2S2'10'
X'4D56'
E'4.6'
```

When used with the MODIFY and SCAN commands, explicit type, length, and scale factor attributes will override the implicit or explicit attributes of the first argument of the MODIFY or SCAN commands.

```
MODIFY ALPHA CL8'ABCDEFGH'
MODIFY ALPHA@T=CL8 'ABCDEFGH'
```

The above two commands are synonymous.

The rules for forming each type of SDS constant are given below.

A-type adcon - If the length of the parameter is three or four bytes, the constant may be any legal virtual address, relative address, or program-defined symbol. If the address is less than 100000 (hex), the current section base and relocation factor are added to produce a virtual address. If the length of the parameter is one or two bytes, the constant is treated as a fixed-point decimal integer of length one or two, respectively. The example below modifies the location ADDR to contain the address of the symbol LOOP.

```
MODIFY ADDR A'LOOP'
```

Binary - A binary constant is a string consisting of the characters 0, 1, and *. The characters 0 and 1 indicate that the corresponding bit positions are taken as 0 and 1, respectively. The character * indicates that the corresponding bit position is to be ignored. For the MODIFY command, * means that the corresponding bit position is left unchanged. For the SCAN command, * means that the corresponding bit position is taken as 0. For an SDS relational expression, * means that the corresponding bit position is not tested in the relation. The maximum length for a binary constant is 64 bits. In the following example, bits 1 and 2 of the variable SWS are modified to 1 and 0, respectively; all other bits are left unchanged. (Bits are numbered left to right starting with bit zero.)

```
MODIFY SWS B'*10*****'
```

Character - Any of the valid EBCDIC characters may be designated in a character constant. Only one character constant may be specified in the second operand to the MODIFY command. Special attention must be given to representing primes as characters; each single prime desired as a character in the constant must be represented as a pair of primes. The maximum length of a character constant is 256 bytes. Two consecutive primes are counted as one character. If the number of characters given is less than the specified length, the constant is padded on the right with blanks.

```
MODIFY CHAR C'ABCDE'
MODIFY CHAR2 CL8'A'
```

CCW - Channel command words are treated as hexadecimal constants of length 8.

Complex - A complex constant consists of a pair of floating-point constants, separated by a comma or a blank. The first constant is the real part and the second constant is the imaginary part. If only one constant is specified, only the real part is used. The length of the entire complex constant is 8 or 16 bytes. Complex constants are used primarily in FORTRAN programs. In the examples below, the location COMP1 is modified to contain 3.0 and 4.0 for the real and imaginary parts of a complex number, respectively, and the location COMP2 is modified to contain 5.0 as the real part with the imaginary part being unchanged. When only the real part is being modified, the E (floating-point) qualifier must be used.

```
MODIFY COMP1 M'3.0,4.0'
MODIFY COMP2 E'5.0'
```

Fixed-point - A fixed-point constant is written as a signed or unsigned decimal number. A fractional part may be included; however, unless a binary scale factor is supplied, the fractional part is deleted. Binary scale factors follow the conventions used by the 360/370-assembler language, i.e., the scale factor multiplies the constant by the specified power of two. A fixed-point constant may vary in length from one to four bytes. Fixed-point constants are right-justified and padded with leading zeros. In the examples below, the location FIX is modified to the constant 2, and the location FIX2 is modified to the constant 32.

```
MODIFY FIX F'2'
MODIFY FIX2 FS3'4'
```

Floating-point - A floating-point constant is written as a decimal number. As an option, a decimal exponent may follow. The number may be an integer, a fraction, or a mixed number. If the length of the constant is 4 bytes or less, the constant is treated as a short-precision constant; otherwise, it is treated as a long-precision constant. The format of the constant is as follows:

(1) The number is written as a signed or unsigned decimal value. The decimal point can be placed before, within, or after the number. If it is omitted, the number is assumed to be an integer. A positive number is assumed if an unsigned constant is specified.
(2) The exponent is optional. If specified, it is written immediately after the number as En, where "n" is an optionally signed decimal value specifying the exponent of the factor 10. The value of the constant may be in the range of .723700515E+76 to .539760535E-78. If an unsigned exponent is speci-

fied, a plus sign is assumed. No binary scale factor may be applied.

In the following examples, the parameter XY is modified to 46.415.

```
MODIFY XY E'46.415'
MODIFY XY E'46415E-3'
MODIFY XY E'+.46415E2'
```

Hexadecimal - A hexadecimal constant consists of one or more of the hexadecimal digits 0-9 and A-F. Only one hexadecimal constant may be specified in the second operand for the MODIFY command. The maximum length of a hexadecimal constant is 256 bytes. Constants that contain an even number of digits are translated as one byte per pair of digits. If an odd number of digits is specified, a leading zero is assumed to provide an even number of digits. If the number of digits given is less than an explicit specified length, the constant is padded on the right with zeros; otherwise, there is no padding. In the example below, the location HEX1 is modified to the hexadecimal constant C1C2C3C4C5.

```
MODIFY HEX1 X'C1C2C3C4C5'
```

Instruction - An "instruction" constant consists of a 360/370-assembler mnemonic and a hexadecimal operand of the appropriate length. The operand is separated from the mnemonic by one or more blanks; blanks may be included within the operand, but not the mnemonic. Extended mnemonics (such as BNM) may be used, in which case the extended mnemonic will include the mask digit of the BC or BCR instruction. The length of the instruction constant need not agree with the length of the instruction being modified. In the first example below, the location INS1 is modified to the instruction "SR 1,2".

```
MODIFY INS1 I'SR 12'
MODIFY INS2 I'BL 0 B098'
```

Logical - Logical constants consist of the constants ".TRUE." and ".FALSE.", which are converted internally to the fixed-point numbers 1 and 0, respectively. If neither ".TRUE." nor ".FALSE." is specified, the constant is treated as a character string. Logical constants are used primarily in FORTRAN programs. In the examples below, the location LOG1 is modified to the constant ".TRUE." and the location LOG2 is modified to the character A.

```
MODIFY LOG1 L'.TRUE.'
MODIFY LOG2 L'A'
```

Packed & zoned - A packed or zoned decimal constant is written as a signed or unsigned decimal value. If the sign is omitted, a plus sign is assumed. The existence of a decimal point in no way affects the conversion of a decimal constant. The decimal point is ignored. Scaling and exponent modifiers may not be used with decimal constants. The maximum length of a decimal constant is 16 bytes.

For packed decimal conversion, each pair of decimal digits is packed into one byte. The rightmost digit and the sign are translated into the rightmost byte. If an even number of packed digits is specified, the leftmost four bits in the leftmost byte are set to zero and the rightmost four bits contain the first digit. If the constant requires fewer bytes than the length of the parameter, the constant is padded on the left with zeros.

For zoned decimal conversion, each decimal digit is translated into one byte. The rightmost byte contains the sign as well as the rightmost digit. If the constant requires fewer bytes than the length of the parameter, the constant is padded on the left with zeros.

```
MODIFY PACK1 P'23'
MODIFY ZONE1 Z'-44'
```

S-type adcon - Not supported.

V-type adcon - V-type adcons are treated as A-type adcons. In the example below, the location VCON is modified to the virtual address 512600.

```
MODIFY VCON V'512600'
```

Y-type adcon - Y-type adcons are treated as fixed-point decimal constants whose length depends on the length of the parameter being modified. In the example below, the location YCON is modified to the integer constant 2.

```
MODIFY YCON Y'2'
```

Output Conversion

All of the conversion types listed with the description of the TYPE keyword modifier are supported by the SDS output-conversion routines. In addition, the symbol "#" is used to specify assembler-generated spaces. The following conventions are used when SDS attempts to display illegal data:

Character - If a hexadecimal code is encountered which has no character equivalent, a question mark is substituted.

Floating-point - If the length of the number exceeds 8 bytes, the number is printed in hexadecimal format.

Fixed-point - If the length of the number exceeds 8 bytes, the number is printed in hexadecimal format.

Instruction - If the opcode field of an instruction does not correspond to a machine operation, the instruction is printed as two hex digits surrounded by asterisks. The instruction is assumed to be 2 bytes in length (RR type).

Packed & zoned - If the length of the number exceeds 16 bytes, the number is printed in hexadecimal format. If an illegal packed or zoned digit is encountered, a question mark is substituted.

SDS Relational and Logical Expressions

SDS relational and logical expressions are designed for use with the IF and WHEN commands to control conditional SDS processing of the user's program.

Relational expressions are formed by combining two SDS parameters or constants with a relational operator. The six relational operators are as follows:

Relational Operator	Definition
.GT. or >	Greater than
.GE. or >=	Greater than or equal to
.LT. or <	Less than
.LE. or <=	Less than or equal to
.EQ. or =	Equal to
.NE. or ¬=	Not equal to

The relational expressions express a condition that can be either true or false. Examples of relational expressions are

```
ALPHA.GT.BETA
ALPHA(2).EQ.F'4'
SWS = B'11**0000'
```

If one side of a relational expression is a constant which has the type attribute omitted, the type attribute from the other side of the expression is assumed, e.g.,

```
ALPHA = '1'
```

is equivalent to

```
ALPHA = F'1'
```

if ALPHA is defined as a fixed-point integer.

The following logical operators may be used to form logical expressions:

Logical Operator	Definition
.AND. or &	Logical and
.OR. or |	Logical or
.XOR.	Logical exclusive or
.NOT. or ¬	Logical not

Each operand for the logical operator must be a relational or logical expression (the negation operator has only a right-hand operand). Examples of logical expressions are

```
¬ALPHA > BETA
ALPHA = F'3' & BETA ¬= F'6'
(ALPHA.EQ.BETA.OR.GAMMA.GT.F'10').AND.DELTA.LE.F'1'
```

Parenthesizing may be used where necessary to achieve the desired operator precedences. Embedded blanks in the expression are legal only around the relational or logical operators.

The precedence hierarchy for SDS operators is as follows (the highest precedence operators are at the top):

$	indirection
+,-	unary operators
*,/	multiplication and division
+,-	addition and subtraction
.xx.	relational operators .GT.,.GE.,.LT.,.LE.,.EQ.,.NE.
.NOT.	negation
.AND.	and
.OR.,.XOR.	or and exclusive or

Current Symbol Character

The current symbol character "*" can be used to represent the last location specified in a debug command. This is most useful in debug command sequences such as

```
DISPLAY ABC
MODIFY * F'1'
```

which displays the contents of location ABC and then modifies that location to the constant '1'.

The current symbol character may have a displacement and/or modifiers appended to it, but not an index. For example,

```
DISPLAY ABC *@T=XL8
```

displays the contents of location ABC according to its type and length, and then displays ABC in hexadecimal format with a length of 8 bytes.

```
DISPLAY ABC(10) *+4
```

displays the locations ABC(10) and ABC(10)+4. (Note: this is not an example of a storage reference block.)

For the case of block parameters, the current symbol will be the lower-bound symbol. For example,

```
DISPLAY ALPHA...BETA *@T=X
```

displays the range from ALPHA to BETA, then the contents of ALPHA in hexadecimal format.

Note: Many debug commands refer to storage locations and, therefore, will change the value of the current symbol character. Hence, the user should be aware of the value that the current symbol character is representing.

PROGRAM RETURNS, DYNAMIC LOADING, AND INTERRUPT PROCESSING

Whenever the user's program returns to the system, or calls the subroutines SYSTEM, MTS, MTSCMD, or ERROR, SDS intercepts the return and returns control to debug command mode. For a return to the system or a call to SYSTEM, the program may not be continued with the CONTINUE command; for the other subroutine calls, the program may be continued. For a call to MTSCMD, the MTS command specified will have been executed.

Whenever the user's program dynamically loads another section via a call to the subroutines LINK, LOAD, or XCTL, SDS may intercept the call and return control to debug command mode. The sections specified in the subroutine calls will have been loaded at this point, but not entered. The user may then set breakpoints before continuing the program. An intercept may also be made on calls to the UNLOAD subroutine and on returns from programs that have been called via LINK and XCTL. This intercept feature may be enabled by the debug command

```
SET XFR=ON
```

Normally, all sections which are dynamically loaded by calls to LINK, LOAD, and XCTL are entered into the symbol table. If the debug command

SET LLX=OFF

is given, these dynamically loaded sections are omitted from the symbol table, thus reducing its size. The setting of the LLX option is independent of the setting of the XFR option.

Whenever a program interrupt, an attention interrupt, a timer interrupt, or an I/O error occurs during the execution of the program, SDS normally intercepts the error condition and returns control to debug command mode.

In the case of a program interrupt, the location of the instruction causing the interrupt is printed. For an imprecise interrupt (protection exception on an IBM Model 360/67 only), an indication that the interrupt was imprecise is also printed. The user's program may be restarted by the CONTINUE command, unless the interrupt was imprecise, in which case, the GOTO or RUN command must be used.

In the case of an attention interrupt, the location at which the interrupt occurred is printed. The user's program may be restarted by the CONTINUE command.

In the event that the user's program has called the system subroutines PGNTTRP or ATTNTRP, SDS may or may not regain control depending on what the user's interrupt routines do. However, the user may disable his own interrupt routines by entering the debug command

SET PGNT=OFF ATTN=OFF

When the interrupt is taken, control is returned immediately to SDS. If the user restarts his program with the CONTINUE command, execution is restarted at the instruction at which the interrupt was taken. The user's interrupt routines are not called. Normal interrupt processing may be resumed by setting the PGNT or ATTN options ON.

In the case of a timer interrupt, the location at which the interrupt occurred is printed. The exceeding of a local time limit is the only type of timer interrupt that returns control to debug mode unless the user-program timer interrupt exit routine returns to the system.

In the case of an I/O interrupt, a system error message (if any) is printed. If the user has suppressed the intercept of the I/O error either by calling the subroutines SETIOERR or SIOERR or by specifying the ERRRTN I/O modifier on the I/O call, control is not returned to debug mode.

If the user's program causes an MTS program interrupt (often caused by passing an illegal parameter to an I/O subroutine), control is returned to debug mode with the message that execution has been prematurely interrupted.

Whenever a program interrupt, an attention interrupt, an I/O interrupt, a timer interrupt, or an intercepted call to LINK, LOAD, XCTL, or UNLOAD occurs, SDS changes its input source to read from *MSOURCE* and its output sink to write on *MSINK*.

If the user's program explicitly calls the subroutine TRACER, or if, on a call to the elementary function library, the traceback program is implicitly called due to the occurrence of an error condition, the traceback program is invoked. An error message may be printed and the prefix character ":" will appear. This prefix character indicates that the traceback program is in control. At this point, the user may enter any of several TRACER commands. Only two are discussed here: the MTS command and the RETURN command. The remainder of the TRACER commands are described in CCMemo 218, "The Traceback Program." If an MTS command is entered, control returns to debug mode. The PSW and registers at this point reflect their status in the traceback program. A CONTINUE debug command returns control to the traceback program instead of resuming execution of the user's program. If a RETURN "name" command is entered, where "name" is the name of the subroutine or library function which caused the traceback program to be invoked, a normal return is made by that subroutine to its calling program. Program execution then proceeds.

BREAKPOINT PROCESSING

SDS provides the user with the facility of setting several different types of breakpoints in strategic locations in his program. When encountered during program execution, these breakpoints cause SDS to assume control of the program and take a certain course of action. Each breakpoint inserted in the user's program is a X'00' code which replaces the opcode of the instruction at which it is inserted and which causes a special program interrupt and returns control to SDS. The opcode of the instruction that is replaced by the breakpoint code is saved for later execution when the program resumes execution. The different types of breakpoints that are currently available are described below.

Global Breakpoints

A global breakpoint returns control to debug command mode. The user may then enter any debug command. Global breakpoints are set by the BREAK command, e.g.,

```
BREAK LOC1
```

sets a global breakpoint at location LOC1. For further details of global breakpoint processing, see the description of the BREAK command below.

Local Breakpoints

A local breakpoint returns control to debug command mode. Local breakpoints are set by the parameters to the RUN and CONTINUE commands, or the second and successive parameters to the GOTO command, e.g.,

```
RUN LOC1
```

sets a local breakpoint at location LOC1 and then initiates execution of the user's program at the entry point. If more than one local breakpoint is set, the first one encountered returns control to debug command mode. Local breakpoints are in effect only for the duration of the current command that set them; they are automatically deleted before the user enters the next command.

At-points

An at-point causes SDS to process a list of prestored debug commands. This list of prestored commands is set up by the AT command (see the description of the AT command).

Breakpoints may not be set at an instruction which is the object of an execute instruction, or at an instruction which will be modified or referenced by the program. For execute instructions, the breakpoint should be set at the execute instruction itself. If the location specified is not halfword-aligned, the breakpoint is not set. A warning is issued if the user sets a breakpoint in a data item.

The BREAK command also may be used to set breakpoints in resident-system storage regions. When the breakpoint is set, a message will be printed indicating that the breakpoint is a "simulator breakpoint." In order for this type of breakpoint to be taken, the program must be simulated at the point of the breakpoint. Simulation of resident-system storage may be achieved by specifying the SIM and FULLSIM options on the SET command, e.g., SIM=ON and FULLSIM=ON. This feature may be used for setting breakpoints in <EFL> (Elementary Function Library), <FIX> (FORTRAN I/O Library), or PL1SYM (Resident-System PL/I Library). If the FULLSIM option is FULL, resident-system subroutines in LCSYMBOL also will be simulated. Warning: Care must be taken in setting breakpoints in other areas of resident-system storage since it cannot be guaranteed that execution of the program may be properly restarted.

Global breakpoints and at-points are restored by the RESTORE or CLEAN commands. The LIST command lists all current breakpoints and at-points. The IGNORE command is used to ignore a breakpoint or an at-point a specified number of times. The command

```
SET BREAK=OFF AT=OFF
```

is used to ignore all breakpoints and at-points without restoring them.

THE SDS SIMULATOR

A program simulator is provided by SDS to allow the user to step through his program one or more instructions at a time. All instructions that are stepped through are simulated by SDS instead of being executed normally.

The simulator is invoked by the STEP command specifying the number of instructions to be stepped, e.g.,

```
STEP 3
```

causes the next three instructions to be simulated. If no step count is given, a step of 1 is assumed. Stepping starts with the current address contained in the PSW. Upon completion of the stepping, control is returned to debug command mode.

If an abnormal condition occurs during stepping, the stepping is terminated at the current instruction and control returns to debug command mode. If the program branches to a legal low-core symbol such as SCARDS or SPRINT, the routine is executed instead of being simulated and stepping resumes upon the return from that routine. The execution of the routine does not count as a step. If the program branches to an address less than 300000 which does not correspond to a legal low-core symbol, stepping is terminated and a warning message is printed. The user may restart the program with either a CONTINUE or GOTO command.

If a program interrupt or attention interrupt occurs and the user has specified an exit routine via the PGNTTRP or ATTNTRP subroutines, these routines are also simulated until the specified stepping count is exhausted. This may be overridden by the PGNT and ATTN options of the SET command.

The cost of simulating a portion of a program compared to executing it normally is approximately 50 to 1. Any local time estimates set by the user or any timer interrupts set by the program will be affected by program simulation since the overhead of simulating the program is taken as a part of the program execution time.

FORTRAN and PL/I users should note that the STEP command specifies machine language instructions in the count. If it is desired to step a specified number of FORTRAN or PL/I instructions, the CONTINUE command should be used specifying local breakpoints.

The SIM option may be used to specify that the program is to be simulated. If the SIM option is ON, any command that starts program execution will invoke the simulator to simulate the program. Resident-system subroutines will not be simulated; instead, they will be executed normally and simulation will resume when the subroutine returns.

The FULLSIM option may be used to specify that resident-system subroutines are to be simulated in addition to the user program. If the FULLSIM option is ON, calls to <EFL> (Elementary Function Library), <FIX> (FORTRAN I/O Library), and PL1SYM (Resident-System PL/I Library) will be simulated. If the FULLSIM option is FULL, calls to the resident-system subroutines in LCSYMBOL will also be simulated. Note: The SIM option must also be ON for the FULLSIM option to be effective.

CONTROL SECTION PROCESSING

When a program consisting of one or more modules[1] is initially loaded by SDS, the control sections, common sections, and dummy control sections of the modules are entered into the SDS map in the order in which they appear in the load module(s). These modules comprise a single map control block within the SDS map. If, during execution, the program calls the loader to load more modules, these also are processed by SDS and entered into the SDS map in a new map control block.

A separate map control block is created for each call on the loader. Hence, the sections that comprise each map control block have a unique storage index number.

The first time a control section name appears in the SDS map, it is assigned a generation number of 1. If a control section is loaded which has the same name as a previously processed control section in an earlier map control block, that section is assigned a new generation number one greater than the last previous generation number used for a section by that name; hence, for each subsequent occurrence of a control section for a given name, there is assigned to it a unique generation number. This generation number can be used via the GEN keyword modifier to distinguish between the different occurrences of control sections with the same name. Dummy control sections (dsects) do not have generation numbers assigned to them.

---

[1] A module is defined as a sequence of load records up to and including an END record. For FORTRAN programs, each main program and subroutine produces a separate object module.

Initially, the entire SDS map is open. This means that the entire map is available for the searching for a symbol. The first section in the map also becomes the current section. This means that the first section is available for the searching for a relative address.

The order of searching through the SDS map for a symbol is as follows:

(1) If the symbol is a predefined symbol, the search terminates immediately.
(2) If only one module of the map is open (see below), only that module is searched. The search covers all control sections, common sections, and defined dsects of that module.
(3) If the entire map is open, all sections may be searched. If a match is found in any section, the remainder of the map is searched for a duplicate symbol. If none is found, the search terminates; otherwise a warning is printed and the first match is used. If a match is found in an undefined dsect, the search continues for a match in a defined section. If such a match is found and no further duplication exists, the search is terminated with no warning message.

The CSECT command provides a means of specifying a new current module. The module containing the section named by the command becomes the new current module and the remaining modules are closed; that is, checking for multiply-defined symbols is suppressed and, if a symbol cannot be located in the current module, no further searching takes place. If the section specified is a control section or dsect, all control sections and dsects in that module are open for searching. For example, the command

```
CSECT SUBR
```

specifies the module containing the section SUBR as the current module. If the command

```
CSECT i
```

is entered, where "i" is an unsigned decimal integer, the module containing the i<u>t</u>h blank-named control section in the SDS map becomes the current module. This is the only way to refer to blank-named (private) control sections in the map. A blank-named common section may be referred to by the blank-name common symbol BLANK.

If the command

```
CSECT *
```

is entered, the first module in the SDS map again becomes the current module and the remaining modules are opened. The open-map character "*" may be changed by the SET OMAPCH=x command.

Object modules composed of multiple assemblies may have name conflicts between control sections, common sections, and dsects. SDS observes the

following conventions when such conflicts occur within a single map control block:

Situation	Action Taken
A csect has the same name as a previously defined csect.	The original csect is retained and the duplicate csect is ignored, unless it is a blank-named csect.
A csect has the same name as a previously defined common section.	The symbols of the csect are merged with the symbols of the common section.
A csect has the same name as a previously defined dsect.	Both the csect and the dsect are retained.
A common section has the same name as a previously defined csect.	The csect is marked as a common section in the SDS map, and the symbols in the common section are merged with the symbols of the csect.
A common section has the same name as a previously defined common section.	The original common section is retained and the symbols of each are merged.
A common section has the same name as a previously defined dsect.	Both the dsect and the common section are retained.
A dsect has the same name as a previously defined csect.	Both the dsect and the csect are retained.
A dsect has the same name as a previously defined common section.	Both the dsect and the common section are retained.
A dsect has the same name as a previously defined dsect in the same module.	The original dsect is retained and the symbols of each are are merged.
A dsect has the same name as a previously defined dsect in a different module (DSECTS=ON).	Both dsects are retained.
A dsect has the same name as a previously defined dsect in a different module (DSECTS=OFF).	The original dsect is retained and the duplicate dsect is ignored.

DEBUGGING ASSEMBLER LANGUAGE PROGRAMS

To generate a 360/370-assembler language object module with SYM records, either the Assembler-G (*ASMG) or the TSS Assembler (*ASMT) should be invoked with the TEST option using a command of the form:

```
$RUN *ASMG SCARDS=source SPUNCH=object PAR=TEST

$RUN *ASMT SCARDS=source SPUNCH=object PAR=TEST
```

The amount of symbolic information produced by each assembler differs. The Assembler-G produces information for the following data types:

- Labeled data variables
- Unlabeled data variables
- Labeled instructions
- Relocatable EQUs
- Assembler-generated spaces (for instruction or data alignment)

The Assembler-G excludes information for the following data types:

- Unlabeled instructions
- Absolute EQUs

The TSS Assembler does not produce as much symbolic information. Information is only produced for the following data types:

- Labeled data variables
- Labeled instructions

The TSS Assembler excludes information for the following data types:

- Unlabeled data variables
- Unlabeled instructions
- Absolute and relocatable EQUs
- Assembler-generated spaces

Since the TSS Assembler does not produce symbolic information to distinguish between unlabeled data variables and unlabeled instructions, SDS assumes that they are both instructions. The user must specify the appropriate type and length modifiers to obtain the correct data conversion. For example,

```
DISPLAY 34@T=FL2
```

displays location 000034 from the example program given below.

When processing indices in SDS parameters for Assembler-G programs, zero duplication factors and assembler-generated spaces are ignored. In the example below, HW(2) refers to the same location as CH; HW(3) and CH(2) both refer to the same location as FW.

```
000020  0009              HW      DC    H'9'
000028                            DS    0D
000028  C1C2C3C4C540      CH      DC    C'ABCDE '
000030  00000005          FW      DC    F'5'
000034  0010                      DC    H'16'
```

Assembler-generated spaces are inserted by the assembler at locations 000022 and 00002E, since the zero duplication factor 0D and the fullword specification for FW require doubleword and fullword alignment, respectively. For TSS Assembler programs, assembler-generated spaces are not ignored; hence HW(2) is not equivalent to CH.

Addressability for dummy control sections (dsects) must be specified by the USING command. The base address specified by the USING command may be static or dynamic. If a location is specified, the base address is static; if a general register is specified, the base address is dynamic, i.e., the base address changes when contents of the register changes. If a general register is to be used as a static base, the indirection operator must be applied to the register. For example,

```
USING DAREA GR1
```

specifies that GR1 contains the base address for the dsect DAREA (dynamic addressability), while

```
USING DAREA $GR1
```

specifies that the current contents of GR1 (when this command is issued) is the base address for DAREA (static addressability). The DROP command can be used to remove the addressability of a dsect. For example,

```
DROP GR1
```

specifies that all dsects addressed by GR1 lose their addressability. See the descriptions of the USING and DROP commands for further details of addressing dsects.

The @D keyword modifier can be used to refer to a location in a particular dsect when that location is defined in both a control section and a dsect. For example,

```
DISPLAY ALPHA@D=DAREA
```

displays the location ALPHA in the dsect DAREA. If a dsect of the same name is defined in several different assemblies, the desired dsect can be specified by appending the @C keyword modifier immediately after the @D modifier. For example,

```
DISPLAY ALPHA@D=DAREA@C=SUBR
```

displays the location ALPHA in the dsect DAREA that is defined in the control section SUBR.

DEBUGGING FORTRAN-G PROGRAMS

To generate a FORTRAN-G object module with SYM records, FORTRAN-G compiler should be invoked with the TEST option using a command of the form:

```
$RUN *FTN SCARDS=source SPUNCH=object PAR=TEST
```

Statement labels may be specified either by using the source-listing statement number (internal format) or the user-defined statement number (external format), if present. The prefix "IS#" must precede a source-listing statement number, e.g.,

```
IS#10
```

is the source-listing statement number 10. The prefix "#" must precede a user-defined statement number, e.g.,

```
#10
```

is the user-defined statement number 10. Only those statement numbers that define executable FORTRAN statements may be used. An executable statement is defined as a statement which is from one of the following categories:

(1) Assignment statements
(2) Control statements
(3) I/O statements

All others, such as those defining DIMENSION, REAL, INTEGER, DATA, COMMON, SUBROUTINE, FUNCTION, ENTRY, EQUIVALENCE, and FORMAT statements are undefined. Both source-listing and user-defined statements must be specified without leading zeros.

The following data type codes are used for variables in FORTRAN programs.

E Real (floating-point)
D Real (floating-point, 8-byte)
F Integer (fixed-point)
H Integer (fixed-point, 2-byte)
L Logical
M Complex
X Hexadecimal

Arguments to FORTRAN subroutines and functions may be one of two types:

(1) reference by value, or
(2) reference by location.

When an argument is passed as a reference by value argument, the actual value of the variable is passed by the calling program to the subprogram. Therefore, there is a copy of that variable in both the calling program and the subprogram. Scalar (undimensioned) arguments are normally passed in this manner. The subprogram uses its own copy of the argument for any

calculations done. Upon return of the subprogram to the calling program, the argument is passed back to the calling program and the calling program's copy is updated. Therefore, when displaying an argument of this type, it is important to keep in mind where the variable is located and when it is displayed.

When an argument is passed as a reference by location argument, only the address of the argument is passed by the calling program to the subprogram. Therefore, only one copy of the argument exists and it is located in the calling program (or a common section). Array arguments are always passed in this manner. The subprogram uses the copy of the argument in the calling program for its calculations. When displaying an argument of this type, either the variable name from the calling program or the variable name from the subprogram argument list may be used. Both refer to the same variable. When using the name from the subprogram argument list, the address passed to the subprogram is used to locate the variable in the calling program. Therefore, the subprogram must have been called at least once for this address to be valid. If the address is invalid, an error comment is produced in the form

xxxxxxx SPECIFIES AN ILLEGAL ADDRESS.

FORTRAN users should note that the STEP command specifies machine language instructions in its count. If it is desired to step a specified number of FORTRAN statements, the CONTINUE command should be used specifying local breakpoints.

The symbol BLANK may be used to refer to blank-named common sections in FORTRAN programs. For example,

```
DISPLAY ALPHA@C=BLANK
```

displays the variable ALPHA from the blank-named common section.

DEBUGGING FORTRAN-H PROGRAMS

To generate a FORTRAN-H object module with SYM records, the FORTRAN-H compiler should be invoked with the TEST option using a command of the form:

```
$RUN *FTN SCARDS=source SPUNCH=object PAR=OPT=H,TEST,options
```

Because of the optimizing features of the FORTRAN-H compiler, the symbol table information provided may be of limited use. This is due to several possible transformations which may be performed on the object module by the compiler during optimization. For example, the compiler often moves operations from within a statement to the beginning of the block of statements in which that statement resides if it does affect the logical operation of the program. This makes it quite difficult to follow the exact execution flow of the program using SDS. For instance, a breakpoint may be set at a statement label, and when the breakpoint is reached, the statement

may have already been executed because its text has been moved to the beginning of the block. Furthermore, it is not uncommon for the compiler to move the entire text of a statement, leaving only the label. When this happens to two adjacent labeled statements, both labels reference the same location and SDS does treat them as distinct (since, in fact, they are not). Another difficulty posed by optimization is the fact that variable values are often kept in registers for large ranges of instructions without updating the memory location. This often happens within DO-loops. Since SDS knows only which memory location corresponds to which symbol (and not which register), displaying a variable from SDS may not yield the current value of the variable.

The problems of optimization are eliminated if the program is compiled at optimization level 0. Unfortunately, most of the advantages of using FORTRAN-H are also eliminated at optimization level 0. In general, it is probably better to debug a program using FORTRAN-G, and then recompile it in FORTRAN-H after a working version is obtained. The use of SYM records with FORTRAN-H should be restricted to those cases where this is not possible, or when hidden bugs emerge at a later time.

The FORTRAN-H compiler only produces symbol table information for external (user-defined) statement labels; no information is produced for internal (source-listing) statement labels. As with FORTRAN-G, the label is preceded by "#", e.g., statement label 100 is the symbol #100.

DEBUGGING PL/I PROGRAMS

To generate a PL/I object module with SYM records, the PL/I compiler should be invoked with the TEST option using a command of the form:

```
$RUN *PL1 SCARDS=source SPUNCH=object PAR=TEST,options
```

Organization of a PL/I Program

This section describes the basic organization of a PL/I external procedure. Knowledge of this will aid the user in displaying program data variables and managing his program.

An external procedure has several control sections, the most pertinent of which are described below.

The program control section is the first control section and contains all machine language instructions for the procedure. The name of this section is the first label of the external procedure statement. If the label is longer than seven characters, the first four and last three characters of the label are used to form the section name. This is the control section in which breakpoints and at-points are set by the user.

The static internal control section is the second control section and contains storage for all static internal variables and constants. The section name is that of the program control section, extended on the right with a single letter A and padded on the left with asterisks to eight characters, e.g., for the procedure name PROG, the static internal control section name is ***PROGA.

IHEMAIN is a 4-byte control section which contains the address of the main procedure. IHEMAIN is produced only if there is an external procedure with the option MAIN specified.

IHENTRY is a 12-byte control section which is the entry point to the program. IHENTRY is always produced if there is an external procedure. This section immediately transfers to one of six library routines which initialize the PL/I environment before the start of execution in the main procedure.

Static external variables are control section entries if they are initialized, or common section entries if they are not. All variables which are declared as external by the program are in separate sections, one section allocated for each variable declared.

The program control section is subdivided into units called blocks. Each block is a delimited sequence of statements that constitutes a section of the program. There are two kinds of blocks: procedure blocks and begin blocks.

Blocks within an external procedure are either active or inactive. Each time a block is entered, a dynamic storage area (DSA) is allocated for that block; a block is considered active after its DSA has been allocated and before an exit has been made from the block. The DSA contains the control information and the automatic variable storage for the block. When the block is exited, the DSA is released and the block becomes inactive. At this point the automatic storage for the block is released and variables declared as automatic are no longer available to the program or SDS. The DSAs for all blocks that are active within the procedure are chained together. This chaining of DSAs allows SDS to access all of the program's currently allocated automatic data variables at one time.

The following four SDS modifiers are used for specifying the location and block level of program data variables.

The @P=xxx keyword modifier, where "xxx" is the name of an external procedure, may be used to refer to a variable within a particular external procedure. The scope of "xxx" includes all of the control sections of the procedure, all of the internal procedures and blocks contained in the procedure, and all of the static, automatic, based, and controlled data variables declared within the procedure. External variables which are stored in common sections are not included in the scope of "xxx".

The @B=i keyword modifier, where "i" is a block level number, may be used to refer to a variable within a particular block of an external

procedure. Each block within an external procedure has a block level number associated with it; this number is given in the compilation listing under the level column. In order for a reference to a particular block to be valid, the block must be active, i.e., the block must have been entered and a DSA must be currently allocated for it.

The @#nn keyword modifier, where "nn" is the compilation statement number of the statement in which the variable was declared, may be used to refer to any variable that was explicitly declared in a DECLARE statement. This modifier is necessary in those cases where there are multiple occurrences of automatic variables of the same name at the same block level or where there are multiple occurrences of static variables of the same name in the same external procedure.

The @I=i keyword modifier, where "i" is an invocation number, may be used to refer to separate invocations of recursive procedures or controlled data variables. The use of this modifier is discussed in more detail below.

Data Variable Specification

All PL/I data variables exist in either static, automatic, based, or controlled storage. The conventions for specifying these different data types are given below.

Static variables

Static variables (either external or internal) are always available within the program and may be displayed at any time before, during, or after program execution. The @P modifier may be used to specify a particular procedure for an internal variable; the @C modifier may be used to specify a particular common section for an external variable if the specification would otherwise be ambiguous, e.g.,

```
DISPLAY SDATA@P=FIRST (for internal SDATA)
DISPLAY SDATA@C=SDATA (for external SDATA)
```

Automatic variables

Automatic variables may be displayed only when the blocks declaring them are active. If the block is inactive, the variables are assumed to be unallocated. When the same automatic variable has been declared within several active blocks, the declaration associated with the most recently entered block is assumed unless overridden by the @B modifier, e.g.,

```
DISPLAY ADATA@B=2
```

displays the value of ADATA associated with the second block level. If ADATA was declared in block level 2 at statement 15, the command

```
DISPLAY ADATA@#15
```

could also be used to display its value.

If a block has been entered recursively, the automatic variables associated with the latest entry will be assumed unless overridden by the @I modifier, e.g.,

```
DISPLAY RDATA@I=1
```

displays the value of RDATA associated with the first invocation of the block in which it was declared.

Based variables

Based variables may be displayed only when they are active, i.e., after they are allocated by an ALLOCATE statement and before they are released by a FREE statement in the program. If the variable is not currently allocated, a message is printed to that effect. Each allocation of a based variable has a pointer variable associated with it. If no pointer variable is specified, the pointer variable given with the declaration statement is assumed. This may be overridden by specifying another pointer variable using the standard PL/I "->" notation, e.g.,

```
DISPLAY PTR->BDATA
```

displays the value of BDATA which has PTR as its pointer variable. The pointer variable name may be qualified with the @P, @B, @#, and @I modifiers to obtain the desired base address; the based variable name may be qualified with the @P modifier to obtain the desired base attributes.

Controlled variables

Controlled variables may be displayed only when they are active, i.e., after they are allocated by an ALLOCATE statement and before they are released by a FREE statement in the program. If the variable is not currently allocated, a message is printed to that effect. When the same controlled variable has been allocated several times, the most recent allocation is assumed unless it is overridden by the @I modifier, e.g.,

```
DISPLAY CDATA@I=1
```

displays the value of the first invocation of the controlled storage variable CDATA.

The following data type codes are used for variables in PL/I programs:

```
E   Float Decimal and Binary Real (floating-point)
M   Float Decimal and Binary Complex (floating-point)
P   Fixed Decimal (packed decimal)
```

F Fixed Binary (fixed-point)
C Character string
B Bit string
A Pointer and Label Data
X Area Data and File Data

Special Data Specifications

The following paragraphs describe special considerations that must be followed for certain data variable classifications.

Arrays

Array variables in a PL/I program must be specified with subscripts. An array element specified without a subscript will generate an error message.

Label Variables

Label variables are normally displayed as A-type address constants. If they are displayed in hexadecimal format, they are displayed as 8-byte elements.

Fixed Decimal Variables

Fixed decimal variables are currently displayed in packed decimal format with no scaling performed. A fixed decimal variable declared as

```
DECLARE FDATA FIXED DECIMAL (7,2) INITIAL(6)
```

is displayed in the format

```
FDATA 'P' +0000600
```

Varying Length Character Strings

Varying length character strings are displayed at their current length. This may range from zero to the maximum length declared for the string.

Bit Strings

Bit string variables are displayed as binary constants. An asterisk is used to indicate the offset of the variable within the first byte, e.g.,

```
BITDATA 'B' ****1110
```

indicates a four-bit variable beginning at bit position 4 (bit positions are numbered 0 through 7). Varying length bit strings are displayed at their current length. This may range from zero to the maximum length declared for the string.

Picture Data

Pictured data variables are displayed as character string data using the internal format of the variable. A pictured variable declared as

```
DECLARE PICDATA PICTURE '$ZZ9V.99' INITIAL('12.34')
```

is displayed in the format

```
PICDATA 'C' "$ 12.34"
```

Structures

A structured variable must be specified using its fully qualified name even though a partially qualified name is unique within the program. Currently, the total length of the fully qualified name may not exceed 31 characters; if the name is longer than 31 characters, only the first 31 characters are retained in the symbol table and may be used. For structured array elements, all subscripts must appear at the end of the variable name, e.g.,

```
DISPLAY X.Y.Z(1,2)
```

must be used to display the variable even though X(1).Y.Z(2) might be valid within the program syntax.

Statement Labels

Statement labels may be specified either by using a symbolic statement label name or by using the statement number given in the compilation listing. Only statement labels for executable PL/I statements may be specified. Statements such as DECLARE, FORMAT, PROCEDURE, and ENTRY are not defined. Statement numbers are specified in the form "#nn", where "nn" is the compilation listing statement number, e.g.,

```
BREAK #27
```

sets a breakpoint at statement number 27 in the program.

Area Variables and Offsets

Based area variables may be displayed using either their pointer variables or offsets within the area. When using an offset, the offset must be added to the address of the area variable to form a pointer, i.e.,

```
(area+offset)->variable
```

For example, consider the following sequence of instructions:

```
DECLARE BAREA AREA(256) BASED(APTR),
        1 BAS BASED(BPTR),
          2 OFF OFFSET(BAREA),
          2 VALUE FIXED DECIMAL(6,2),
        QPTR POINTER;

ALLOCATE BAREA;
ALLOCATE BAS IN (BAREA);
ALLOCATE BAS IN (BAREA) SET(QPTR);
BAS.OFF = QPTR;
BPTR -> VALUE = 25;
QPTR -> VALUE = 50;
```

After execution of these instructions, a structured link list is constructed in which the first element has the value 25 and the second element has the value 50. Either of the following commands may be used to display the first element:

```
DISPLAY BAS.VALUE
DISPLAY BPTR->BAS.VALUE
```

Any of the following commands may be used to display the second element:

```
DISPLAY QPTR->BAS.VALUE
DISPLAY (BAREA+BAS.OFF)->BAS.VALUE
DISPLAY (BAREA+(BPTR->BAS.OFF))->BAS.VALUE
```

File Variables

File variables are displayed in hexadecimal format. The region displayed for the file variable is the declare control block (DCLCB) which specifies the attributes of the file. The name of the file is at location 19 (hex) within the DCLCB.

DEBUGGING PL360 PROGRAMS

To generate a PL360 object module with SYM records, the PL360 compiler should be invoked with the TEST option using a command of the form:

```
$RUN *PL360 SCARDS=source SPUNCH=object PAR=TEST,options
```

The following paragraphs describe the basic organization of a PL360 program. Knowledge of this will aid the user in displaying program data variables and managing his program.

A PL360 program is divided into procedures. Each of these procedures is further divided into units called segments. Each segment constitutes a control section, common section, or dummy control section (dsect). There are two types of segments:

A program segment is a control section that contains the machine language instructions and the literal pool for the procedure. The name of the control section is the name given with the PROCEDURE statement. Each global procedure, and segment procedure generates a separate control section; local procedures do not generate separate control sections. Program segments are the control sections in which breakpoints and at-points are set by the user.

A data segment is a control section, common section, or dummy control section that contains storage for the data variables referenced by a program segment. The name of a data segment is the name given with the data segment declaration statement; if no name is supplied, a name is generated by the compiler. The names generated by the compiler are given in the program source listing or may be displayed by the MAP debug command. Each procedure may reference several data segments.

The following data type codes are used for variables in PL360 programs:

E	Real (floating-point)
D	Long Real (floating-point, 8-byte)
F	Integer (fixed-point)
H	Short Integer (fixed-point, 2-byte)
X	Byte (hexadecimal)

Array variables in a PL360 program may be specified with subscripts. The first element of the array has the subscript 1.

Statement labels must be referenced by the label used in the program. If the label is used both as a statement label name and a data variable name, the @C modifier should be used to specify the correct occurrence, e.g.,

```
STMT1@C=MAIN
```

specifies the statement label STMT1 from the segment MAIN.

The compiler generates four special labels which correspond to BEGIN statements, END statements, DSECTS, and the end of the segment. The labels are of the form

```
#Bxxxx
#Exxxx
#Dxxxx
#ENDxxxx
```

where "xxxx" is the source-listing statement number. Leading zeros are required. For example, a BEGIN statement at statement number 4 and an END statement at statement number 8 would have the labels

```
#B0004
#E0008
```

generated by the compiler. These generated labels may be used for setting breakpoints or displaying instructions. Dsect labels may be used with the

USING command to establish addressability for the corresponding dsect. The END label generated for the end of the segment corresponds to the beginning of the literal pool at the end of the segment.

MISCELLANEOUS CONCEPTS

Terse Mode

SDS provides a terse mode of operation which eliminates or shortens many of the confirmation and diagnostic messages printed. Terse mode is enabled by the command

```
SET TERSE=ON
```

and is disabled by the command

```
SET TERSE=OFF
```

If the TERSE option is MTS (the default), the current setting of the MTS TERSE option will be used as the setting for the SDS TERSE option.

The following messages are eliminated in terse mode:

(1) READY. This message is often printed when SDS is requesting a command.
(2) DONE. This message is often printed after SDS has taken some action such as setting or restoring a breakpoint.
(3) Verification by the MODIFY command. The MODIFY command normally verifies the modification by printing both the old and new values of the location.

The following messages are shortened in terse mode:

(1) The breakpoint and at-point interrupt messages give only the address of interruption and do not identify the type of interruption.
(2) The call to SYSTEM, ERROR, MTS, and MTSCMD messages do not give the GR14 return address.

The BRIEF option (which is a synonym for TERSE) or the VERBOSE option (which is an antonym for TERSE) may also be used to control terse mode. The WARNMSG option may be used to control certain warning messages. If the WARNMSG option is ON, all messages concerning addresses outside of control section bounds or subscripts outside of array bounds are suppressed.

Automatic Error-Dumping in Batch

An automatic error-dumping facility similar to that provided by the MTS $ERRORDUMP command is provided for batch users. In the event that an error condition occurs during the execution of the program, a symbolic dump of the program is given. This dump includes the PSW, the general and floating-point registers, and all of the data storage locations in the program. Instructions and other areas not covered by the symbol table are excluded. This facility may be enabled by the command sequence

```
$SET DEBUG=ON
$SDS SET ERRORDUMP=ON
$RUN FDname
```

where "FDname" is the file or device containing the user program to be executed. Note that the MTS $RUN command has been given instead of the $DEBUG command. The error-dump facility may be disabled by the command

```
$SET DEBUG=OFF
```

or

```
$SDS SET ERRORDUMP=OFF
```

Terminal users may obtain a symbolic dump by the DUMP debug command. A sample of the symbolic dump is given with the DUMP command description.

Using SDS Without a Loaded Program

Several of the debug commands may be used successfully even if SDS has not processed the loaded program or if there is no currently loaded program. The user may use the output conversion facilities of SDS to display selected locations of virtual memory. For example, the debug command

```
DISPLAY 516260@T=CL32
```

displays in character format the 32 bytes starting at location 516260, if it is a valid virtual address. The input conversion facilities may be used to modify selected storage, e.g., the debug command

```
MODIFY 516260 CL8'ABCDABCD'
```

modifies the 8 bytes starting at location 516260 to be the character string ABCDABCD.

The above command may be given from MTS command mode in the form of a single $SDS command, e.g.,

```
$SDS MODIFY 516260 CL8'ABCDABCD'
```

After the single command is executed, control is returned to MTS command mode.

Initializing, Resetting, and Terminating SDS Processing

When SDS is initialized, an area of system storage is assigned to SDS and all of the default SDS options are set. SDS is initialized under any of the following conditions:

(1) If DEBUG=OFF is specified (the default), SDS is initialized with the first $DEBUG or $SDS command in the job or after SDS has been previously terminated (see below).
(2) If DEBUG=ON is specified, SDS is initialized with the first $DEBUG, $RUN, $RERUN, $LOAD, $START, $RESTART, or $SDS command in the job or after SDS has been previously terminated.

When SDS is reset, all of the loaded program symbol table information (if any) is released and certain SDS tables are released. The basic SDS work storage remains and the SDS SET options remain in effect (except for the INPUT, OUTPUT, ENTRY, PAR, and logical I/O unit assignments). Input commands are read from *SOURCE* and output is written on *SINK*. SDS is reset under any of the following conditions:

(1) During initialization (see above).
(2) With each $DEBUG, $RUN, $RERUN, and $LOAD command if $SET DEBUG=ON is specified.
(3) With each $DEBUG command if $SET DEBUG=OFF is specified.

When SDS is terminated, all of the loaded program symbol table information (if any) is released and the basic SDS work storage is released. SDS is terminated under any of the following conditions:

(1) The STOP debug command is given.
(2) The $SET DEBUG=OFF command is given.
(3) The $UNLOAD CLS=SDS command is given.
(4) A $RUN, $RERUN, $LOAD, or $UNLOAD command is given (if $SET DEBUG=OFF is specified).

DEBUG COMMAND DEFINITIONS

On the following page is a list of the debug commands available for SDS. Parameters for each command should be separated by blanks. Some of the parameter terms used are:

(1) location - these are storage references of the form S(i)±j.

(2) section - these are control section, common section, or dsect names.

The following notation conventions are used in the prototypes of the commands:

lower-case	- represents a generic type which is to be replaced by an item supplied by the user.
upper-case	- indicates material to be repeated verbatim in the command.
brackets [ ]	- indicates that material within the brackets is optional.
braces {}	- indicates that the material within the braces represents choices, from which exactly one must be selected. The choices are separated by vertical bars.
ellipsis •••	- indicates that the preceding syntactic unit may be repeated.
underlining	- indicates the minimum abbreviation of the command or parameter. Longer abbreviations are accepted.

Note: For descriptions of the ATTNTRP, ERRORDUMP, INPUT, LENGTH, OUTPUT, PGNTTRP, PREFIX, RF, SCALE, TERSE, and TYPE commands, see the appropriate SET command options. These are implemented as individual commands, but are not individually documented.

Summary of Debug Command Prototypes

Commands	Parameters
ACTIVATE	{name|FLOW|CHANGES|REFERENCES} •••
ALTER	location {constant|location}
AT	location ••• [;command]
ATTNTRP	{ON|OFF}
ATTRIBUTE	location •••
BREAK	location •••
CLEAN	[option •••]
COMMENT	[text]
CONTINUE	[location •••]
CSECT	{section|i|*}
DEACTIVATE	{name|FLOW|CHANGES|REFERENCES} •••
DEBUG	filename [I/O units] [limits] [PAR=parameters]
DISPLAY	location[...location] •••
DROP	{section|location} •••
DUMP	[{PSW|GRS|FRS|section} •••]
END	
ERRORDUMP	[{ON|OFF|FULL}]
GOTO	location [location •••]
HEXDISPLAY	location[...location] •••
IF	expression [;command]
IGNORE	breakpoint-label {i|location}
INCLUDE	filename
INPUT	FDname
LENGTH	i
LIST	[option •••]
MAP	[{FULL|DSECT}]
MCMD	[MTS-command]
MODIFY	location {constant|location}
MTS	[MTS-command]
ON	on-condition [;command]
OUTPUT	FDname
PARLIST	location [count]
PGNTTRP	{ON|OFF}
PREFIX	character
QUALIFY	location •••
RESET	[on-condition •••]
RESTORE	[{location|name} •••]
RETURN	[MTS-command]
RF	hexadecimal-constant
RUN	[location •••]
SCALE	i
SCAN	[{section|location1...location2|*} {constant|location}]
SDS	
SET	option •••
STEP	[i [BRANCH]]
STOP	
SYMBOL	location •••
TERSE	{ON|OFF}
TIMETALLY	{ON|OFF|PRINT|RESET|CLEAR} [option •••]

```
TRACE       trace-option [location ...]
TYPE        code
USING       {section|symbol} location
WHEN        {expression|location CHANGES} [;command]
```

ACTIVATE

DEBUG COMMAND DESCRIPTION

Prototype: ACTIVATE {name|FLOW|CHANGES|REFERENCES} ...

Action: If "name" is specified, the corresponding when-condition is activated. This may be used to reactivate a when-condition that has been explicitly deactivated by the DEACTIVATE command or implicitly deactivated after its condition has been satisfied during program execution. See the description of the WHEN command for further details.

If FLOW, CHANGES, or REFERENCES is specified, then flow, change, or reference tracing, respectively, is activated. See the description of the TRACE command for further details of tracing.

Page Revised March 1977 April 1976

ALTER

DEBUG COMMAND DESCRIPTION

Prototype: ALTER location {constant|location}

Action: The ALTER command is identical to the MODIFY command. See the description of the MODIFY command.

AT

DEBUG COMMAND DESCRIPTION

Prototype: (a) AT location ...[;command]

(b) AT location ...
command
command
.
.
END

where "location" specifies the location for an at-point to be inserted in the user's program and "command" is a single debug command or a part of a list of debug commands to be executed in sequence when the at-point is reached in the program. With the single command prototype (a), the command is separated from the at-point symbol list by a semicolon. With the command list prototype (b), SDS enters into command insertion mode (indicated by the "?" prefix character) and reads debug commands until an END command is given.

Action: An at-point is established at each specified location by replacing the opcode of the instruction with X'00'; the original opcode is saved for later execution when execution of the program is resumed or when the at-point is restored or otherwise removed.

With the command list prototype (b), SDS enters into command insertion mode. All commands entered during command mode are saved in a command list for later processing. When an end-of-file or an END command is entered, SDS resumes normal command processing. AT commands (and their respective command lists) may be nested in other IF or WHEN command lists. AT commands may not be nested in other AT command lists; a second AT command will terminate the first AT command list and begin a new command list. If the user enters a null command line during command insertion mode, the previously entered command is deleted from the list of saved commands.

When an at-point is encountered during the execution of the user's program, each of the commands in the command list for that at-point is executed in sequence. When the END command is executed, control returns to the user's program and normal execution is resumed (unless a breakpoint has been encountered during the execution of the command list). If control is to be returned to debug command mode rather than to the program, the SDS command should appear immediately before the END command since the SDS command terminates the processing

of the command list. The processing of the command list is also terminated in the event of an abnormal condition occurring in the program.

Comments: At-points are automatically announced if the command list causes an output message to be generated (e.g., the output from the DISPLAY command), or if there is no command list for the at-point. The COMMENT command may be used for comments or announcements of at-points when no output messages would otherwise be generated.

An at-point should not be set at an instruction which is the object of an execute instruction, or at an instruction which will be modified or referenced by the program. For an execute instruction, the at-point should be set at the execute instruction itself. If the location specified is not halfword-aligned, the at-point is not set. A warning is issued if the user sets an at-point in a data item.

At-points may be removed by either the RESTORE or the CLEAN commands.

Examples:

```
AT LABEL;SDS
```

In the above example, when the at-point at location LABEL is reached, control is returned to debug command mode.

```
AT LABEL1
DISPLAY GRS
SDS
END
```

In the above example, when the at-point at location LABEL1 is reached, the general registers are displayed and control is returned to debug command mode.

```
AT LABEL2
MODIFY ALPHA F'2'
GOTO LABEL3
END
```

In the above example, when the at-point at location is reached, the location ALPHA is modified to the value 2. Execution is then resumed at location LABEL3.

ATTRIBUTE

DEBUG COMMAND DESCRIPTION

Prototype: ATTRIBUTE location ...

Action: The attributes for each parameter in the list are displayed; these include:

the loaded absolute address,
the relative address,
the section name,
the section generation number (if different from 1),
the type,
the length,
the duplication factor (if different from 1),
the scale factor (if different from 0), and
the dimension number (if a FORTRAN or PL/I array).

The one-character codes for the type are defined with the description of the TYPE keyword modifier.

If a symbolic location is specified without the CSECT keyword modifier and all sections in the map are open (a CSECT command has not been specified), all occurrences of the location are displayed.

Example:

```
ATTRIBUTE GAMMA
```

The attributes of GAMMA are displayed as

```
GAMMA:
LA=514020 RA=000020 SECTION=PROG TYPE=F LEN=4 DUP=20
```

Page Revised March 1977 April 1976

BREAK

DEBUG COMMAND DESCRIPTION

Prototype: BREAK location ...

Action: A global breakpoint is established at each location specified by replacing the opcode of the instruction with X'00'; the original opcode is saved for later execution when execution of the program is resumed or when the breakpoint is restored or otherwise removed.

When the program being debugged attempts to execute the instruction at the breakpoint, control is returned to SDS which announces the location of the breakpoint and prompts for its next command. The instruction at the breakpoint has not yet been executed. The status of the program is preserved and may be examined and modified with the appropriate commands. Execution may be resumed with the CONTINUE or STEP commands which execute the instruction at the breakpoint and resume normal sequencing. To restart at some other point, the GOTO command may be used.

Breakpoints may be removed with either the RESTORE or the CLEAN commands.

A breakpoint should not be set at an instruction which is the object of an execute instruction, or at an instruction which will be modified or referenced by the program. For an execute instruction, the breakpoint should be set at the execute instruction itself. If the location specified is not halfword-aligned, the breakpoint is not set. A warning is issued if the user sets a breakpoint in a data item.

For FORTRAN and PL/I programs, the user may set breakpoints by specifying the external (user-defined) or internal (source listing) statement numbers. The "#" must be used to prefix a FORTRAN external statement number or a PL/I internal statement number, e.g.,

```
BREAK #10
```

and "IS#" must be used to prefix a FORTRAN internal statement number, e.g.,

```
BREAK IS#10
```

Note: Only those statement numbers which define executable FORTRAN or PL/I statements may be used. All others, such as those defining DIMENSION, DATA, COMMON, SUBROUTINE, FUNCTION, ENTRY, EQUIVALENCE, FORMAT, or DECLARE statements are undefined.

The BREAK command also may be used to set breakpoints in resident-system storage regions. When the breakpoint is set, a message will be printed indicating that the breakpoint is a "simulator breakpoint." In order for this type of breakpoint to be taken, the program must be simulated at the point of the breakpoint. Simulation of resident-system storage may be achieved by specifying the SIM and FULLSIM options on the SET command, e.g., SIM=ON and FULLSIM=ON. This may be used for setting breakpoints in <EFL> (Elementary Function Library), <FIX> (FORTRAN I/O Library), or PL1SYM (Resident-System PL/I Library). If the FULLSIM option is FULL, resident-system subroutines in LCSYMBOL also will be simulated. Warning: Care must be taken in setting breakpoints in other areas of resident-system storage since it cannot be guaranteed that execution of the program may be properly restarted.

Example:

```
BREAK LOOP 206 503100
```

Global breakpoints are set at the symbolic location LOOP, the relative location 206, and the absolute location 503100.

Page Revised March 1977 April 1976

CLEAN

DEBUG COMMAND DESCRIPTION

Prototype: CLEAN [option ...]

Action: The action specified by "option" is taken. "option" may be any one of the following:

AT	All at-points set by the AT command are deleted.
BREAK	All breakpoints (except simulator breakpoints) set by the BREAK command are deleted.
CALLS	Call tracing is disabled and the call list is deleted.
CHANGES	Change tracing is disabled and the change list is deleted.
COUNT	Count tracing is disabled and the count list is deleted.
FLOW	Flow tracing is disabled.
LABEL	Label tracing is disabled.
ONS	All on-conditions set by the ON command are deleted.
REF	Reference tracing is disabled and the reference list is deleted.
SIM	All simulator breakpoints set by the BREAK command are deleted.
WHENS	All when-conditions set by the WHEN command are deleted.

If no parameter is specified, all breakpoints and at-points are deleted.

Example:

```
CLEAN A
```

All at-points are removed from the program.

COMMENT

DEBUG COMMAND DESCRIPTION

Prototype: COMMENT [text]

Action: If the user has entered the command in command-insertion mode, the comment is printed when the command list is processed.

In debug command mode, the COMMENT command has no effect.

Example:

```
AT LOOP
COMMENT SKIP 2 INSTRUCTIONS AT LOOP
GOTO LOOP(3)
END
```

At the location LOOP, the comment "SKIP 2 INSTRUCTIONS AT LOOP" is printed; then, execution resumes at location LOOP(3).

CONTINUE

DEBUG COMMAND DESCRIPTION

Prototype: CONTINUE [location ...]

Action: Execution of the program is resumed from the point of the last interrupt. If a breakpoint was encountered, execution begins with the instruction at the breakpoint. If an attention interrupt or program interrupt had been taken, execution begins at the location specified by the PSW. In the event of an imprecise program interrupt (IBM Model 360/67 only), a GOTO or a RUN command must be used to restart the user's program.

If a location is specified, a local breakpoint is set at the location specified. This breakpoint code is an X'00' which replaces the opcode of the instruction. When the program encounters a local breakpoint, control is returned to debug command mode. Local breakpoints are in effect only for the duration of the command, and are automatically erased before the user enters his next command.

The CONTINUE command may be used to initiate execution of the program if the initial values of the registers and/or PSW have been modified.

Example:

```
CONTINUE LOC2
```

A local breakpoint is set at location LOC2 and execution of the program is resumed.

CSECT

DEBUG COMMAND DESCRIPTION

Prototype: CSECT {section|i|*}

Action: If "section" is given, the module containing the section named becomes the new current module and the remaining modules are closed; that is, checking for multiply-defined symbols is suppressed, and, if a symbol cannot be located in the current module, no further searching is done. All control sections, common sections, and defined dsects in the module are open for searching.

If "i" is given, the module containing the ith blank-named control section loaded becomes the current module and the remaining modules are closed (see "Control Section Processing" above).

If the open-map character "*" is given, the first module loaded again becomes the current module and the remaining modules are opened.

Comment: To specify a module containing a dsect as the current module, "section" must be specified as

dsect@C=csect

where "csect" is the name of the control section with which the dsect was assembled. The dsect must be previously defined with a USING command.

Examples:

```
CSECT PROG
```

The module containing the section PROG becomes the current module.

```
CSECT DAREA@C=SUBR
```

The module containing the dsect DAREA in the section SUBR becomes the current module.

DEACTIVATE

DEBUG COMMAND DESCRIPTION

Prototype: DEACTIVATE {name|FLOW|CHANGES|REFERENCES} ...

Action: If "name" is specified, the corresponding when-condition is deactivated. See the description of WHEN command for further details.

If FLOW, CHANGES, or REFERENCES is specified, then flow, change, or reference tracing, respectively, is deactivated. See the description of the TRACE command for further details of tracing.

DEBUG

DEBUG COMMAND DESCRIPTION

Prototype: DEBUG filename [I/O units] [limits] [PAR=parameters]

Action: The current program being debugged is unloaded and a new program is loaded from "filename". This command may be used to reload the current program being debugged.

The I/O units, limits, and PAR field specifications are the same as for the MTS $DEBUG command (this command is implemented by passing the entire command line to MTS for processing).

Example:

```
DEBUG -LOAD SCARDS=DATA
```

The program in the file -LOAD is loaded for debugging.

DISPLAY

DEBUG COMMAND DESCRIPTION

Prototype: DISPLAY location[...location] ...

Action: Each location (or element in the storage block) is displayed in the format required by its type and length and printed along with a one-character code which indicates the location's type. The type codes are defined with the description of the TYPE keyword modifier.

If the type and length attributes of a location are unknown, or if a location specifies an address which is incorrectly aligned with respect to its type, the contents of the byte specified by the location, and the contents of the next three bytes, are printed in hexadecimal format.

All global breakpoints and at-points are temporarily restored before each location is displayed.

Example:

```
DISPLAY ALPHA
```

The location ALPHA is displayed according to its type and length attributes.

```
DISPLAY ALPHA...BETA
```

Each location in the range of ALPHA to BETA is displayed according to its type and length.

DROP

DEBUG COMMAND DESCRIPTION

Prototype: DROP {section|location} ...

Action: If "section" is specified, that dsect loses its addressability (see the description of the USING command).

If "location" specifies a general register, all dsects covered by that register are no longer addressable by SDS (i.e., symbols and relative addresses within that dsect cannot be accessed). If "location" specifies a storage address, all dsects based at the virtual address corresponding to that location lose their addressability.

If the parameter may be interpreted both as a dsect name and as a symbol, it is assumed to be a dsect name.

Example:

```
DROP GR1
```

The dsects covered by GR1 are released and are no longer addressable.

Page Revised March 1977 April 1976

DUMP

DEBUG COMMAND DESCRIPTION

Prototype: DUMP [{PSW|GRS|FRS|section} •••]

Action: A symbolic dump of the sections specified is given. The dump includes only the variable storage of the section; instructions and areas not covered by the symbol table are excluded.

For each item dumped, the relative address, the symbolic name, the data type, the converted value, and the hexadecimal value (if the data type is not X) is dumped.

If PSW is specified, the hexadecimal and symbolic address forms are given.

If GRS is specified, the general registers are dumped in hex, fixed-point decimal, and symbolic address format.

If FRS is specified, the floating-point registers are dumped in hexadecimal and floating-point decimal format.

If no parameter is specified, the PSW, the registers, and all sections are dumped.

Comment: Since the output from this command may be extensive, terminal users should set the output device to a file or *PRINT* via the SET OUTPUT=FDname command.

The DUMP option may be used to obtain an abbreviated symbolic dump where all FORTRAN and PL/I arrays are omitted except for the first element in the array. This option may be enabled by setting DUMP=SHORT with the SET command; the default is FULL.

For batch users, an automatic symbolic dump may be obtained in the event of an abnormal program termination via the SET ERRORDUMP=ON debug command. This facility is similar to the $ERRORDUMP command in MTS command mode which produces a hexadecimal dump of a program in the event of an abnormal program termination.

A sample symbolic dump is given on the next page.

Example:

```
SET OUTPUT=*PRINT*
DUMP
```

A symbolic dump of the PSW, the registers, and all loaded sections is produced on *PRINT*.

Page Revised March 1977 April 1976

END

DEBUG COMMAND DESCRIPTION

Prototype: END

Action: If the command is entered in command-insertion mode, the current sequence of commands being entered into command list associated with the AT, IF, or WHEN command is terminated, and control is returned to debug command mode.

In debug command mode, the END command has no effect.

Example:

```
AT LOOP
DISPLAY GR1
END
```

The command list for the at-point at location LOOP is terminated by the END command and control is returned to debug commard mode.

GOTO

DEBUG COMMAND DESCRIPTION

Prototype: GOTO location [location ...]

Action: Execution of the program is resumed at the specified location. The location must specify a halfword-aligned address.

If more than one location is specified, local breakpoints are set at the second, third, etc., locations. This breakpoint code is an X'00' which replaces the opcode of the instruction. When the program encounters a local breakpoint, control is returned to debug command mode. Local breakpoints are in effect only for the duration of the command, and are automatically erased before the user enters the next command.

The GOTO command may be used to initiate the execution of the program if a different entry point is desired.

Example:

```
GOTO LOC1 LOC2
```

A local breakpoint is set at location LOC2 and execution of the program is resumed at location LOC1.

HEXDISPLAY

DEBUG COMMAND DESCRIPTION

Prototype: HEXDISPLAY location[...location] ...

Action: Each location is displayed in the hexadecimal format used by the SDUMP subroutine (see the description of the SDUMP subroutine in MTS Volume 3). For storage blocks, the entire block is displayed.

Global breakpoints and at-points are not restored before those locations are displayed.

Example:

```
HEXDISPLAY 5002A0...5002EC
```

The storage block starting at location 5002A0 is dumped in hexadecimal format.

IF

DEBUG COMMAND DESCRIPTION

Prototype: (a) IF expression; command

(b)
```
IF expression
command
command
   .
   .
END
```

where "expression" is a relational or logical expression and "command" is a single debug command or a part of a list of debug commands to be executed in sequence.

With prototype (a), the single debug command is executed if the value of the expression is true; otherwise, the command is not executed. The command is separated from the expression with a semicolon. With the command list prototype (b), if the value of the expression is true, SDS enters into command insertion mode (indicated by the "?" prefix character) and reads debug commands until an END command is given; when the END command is given, the entire sequence of commands is then executed. If the value of the expression is false, command insertion mode is not entered.

IF commands (and their respective command lists) may be nested in other IF, AT, or WHEN command lists. Each nested command list must be individually terminated by an END command (unless a single command form is used). If an IF command and its associated command list is nested in another command list, and if the value of the expression is false, the associated command list is skipped.

Verification that an expression is true will be given only if the IF command is executed from another IF or AT command list. This verification is suppressed in terse mode.

The IF command is designed to test a static condition in the user's program (as opposed to a dynamic condition which is tested by the WHEN command). Since this condition is tested when the IF command is executed, the most common use for the IF command is in conjunction with the AT command for patching programs or testing program conditions.

Examples:
```
IF ALPHA > F'4'; GOTO LABEL2
```

In the above example, execution of the user's program transfers to location LABEL2 if ALPHA has a value greater than 4.

```
IF ALPHA = F'6'
MODIFY BETA F'0'
MODIFY GAMMA F'100'
END
```

In the above example, BETA and GAMMA are modified if ALPHA has the value 6.

```
AT LABEL
IF ALPHA ¬= F'0'
MODIFY BETA F'0'
GOTO LABEL4
END
END
```

In the above example, when the at-point LABEL is reached in the user's program, the AT command list is executed. If ALPHA is not equal to the value 0, BETA is modified and the program resumes execution at location LABEL4. Note that there must be an END command for both the IF command list and the AT command list.

```
AT LABEL3
IF ALPHA < F'0' & BETA > F'0'
MODIFY ALPHA F'0'
IF BETA > F'10'; MODIFY BETA F'10'
END
DISPLAY GRS
END
```

In the above example, when the at-point LABEL3 is reached in the user's program, the AT command list is executed. If ALPHA is less than 0 and BETA is greater than 0, ALPHA is modified. In addition if BETA is greater than 10, BETA is modified. After all modifications are made (if any), the general registers are displayed and the program resumes execution. Note that the first IF command (of prototype (b)) has a second IF command (of prototype (a)) nested in its command list.

IGNORE

DEBUG COMMAND DESCRIPTION

Prototype: IGNORE breakpoint-label {i|location}

Action: Ordinarily, when control reaches a global breakpoint or an at-point in the program, execution is interrupted and control returns to SDS. The IGNORE command provides a means of suppressing this interruption each time control reaches the breakpoint, for a total of "i" times, where "i" is either a decimal integer or the contents of the location specified. The i+1st time control reaches the breakpoint, the interruption is taken as usual.

If a location is specified, the type of the storage location may be fixed-point (fullword or halfword), floating-point (long or short), or hexadecimal (4 bytes or less). A location that is in the form of a relative address must have the @R modifier appended.

The ignore count must be positive and may not exceed 65535.

Example:

```
IGNORE  LOOP 10
```

The breakpoint at the location LOOP will be ignored 10 times.

INCLUDE

DEBUG COMMAND DESCRIPTION

Prototype: INCLUDE filename

Action: The symbols that are contained in the object file "filename" are included in the SDS symbol table. If a symbol table does not currently exist, one is constructed. Only SYM, ESD, and END records are processed; all other types of loader records are ignored. The loaded addresses of external symbols are obtained from the system loader map; hence, the MTS SYMTAB option must be ON.

Examples:

```
$RUN filename                    initiate program execution
   .
   .
USER PROGRAM INTERRUPT.  PSW = xxxxxxxx xxxxxxxx
$SDS                             enter debug mode
INCLUDE filename                 build SDS symbol table
SYM $PSW                         display interrupt address
   .                             begin debugging session
   .
```

The above sequence of commands illustrates how the INCLUDE command can be used for initiating a debugging session after an unexpected error is detected in a program.

```
INCLUDE filename                 include dsect symbols
USING dsect address              set dsect base address
DISPLAY symbol
   .
   .
```

The above sequence of commands illustrates how the INCLUDE command can be used for displaying selected areas of dynamically allocated storage for programs that were not processed by SDS at load time. In this case, "filename" contains SYM records that were generated for the dsect.

LIST

DEBUG COMMAND DESCRIPTION

Prototype: LIST [option ...]

Action: The action specified by "option" is taken. "option" may be any one of the following:

AT	All at-points set by the AT command are listed.
BREAK	All breakpoints (except simulator breakpoints) set by the BREAK command are listed.
CALLS	The status of call tracing is given.
CHANGES	The status of change tracing along with the change list is given.
COUNT	The current counts recorded by count tracing are listed.
FLOW	The status of flow tracing is given.
LABEL	All label points specified for label tracing are listed.
MODS	All program modifications made by the MODIFY command are listed (except modifications made in an at-command list).
ONS	The status of all on-conditions set by the ON command are listed.
REF	The status of reference tracing along with the reference list is given.
SIM	All simulator breakpoints set by the BREAK command are listed.
WHENS	The status of all when-conditions set by the WHEN command are listed.

If no parameter is specified, all breakpoints and at-points are listed.

Example:

```
LIST A
```

A listing of all the currently set at-points is produced.

MAP

DEBUG COMMAND DESCRIPTION

Prototype: MAP [{FULL|DSECT}]

Action: A map is produced, listing each control section and common section in the user's program. The map includes the section name, section type, the section length, the loaded address, the relocation factor, and the storage index number. If the storage index number is omitted, then it is the same as the storage index number of the previous section.

Blank-named (private) control sections are specified by unsigned decimal integers assigned according to their order in the SDS map. This integer is the only way to refer to a blank-named control section. A blank-named common section is specified in the map as a blank symbol. The blank-name common symbol (initially BLANK) is used to refer to a blank-named common section.

The pseudo-register name symbol (initially PRAREA) is used to refer to the pseudo-register area.

If the FULL parameter is specified, the map also includes all dsects and library-loaded sections. If the dsect is undefined, the address field is blank; if the dsect is defined, the address field contains the current address definition for that dsect. Addresses 000001 through 00000F are used to indicate that the dsect is defined by the current contents of GR1 through GRF (GR15), respectively.

If the DSECT parameter is specified, the map only includes the currently defined dsects; control sections, common sections, and undefined dsects are omitted.

The symbols used for the map type are:

CS	control section definition
CM	common section definition
DS	dsect definition
PR	pseudo-register area
LCS	library control section definition
LCM	library common section definition
LDS	library dsect definition

Example:

```
MAP FULL
```

A full map is printed as follows:

NAME	TYPE	LENGTH	ADDRESS	RELOC	SI#
MAIN	CS	000268	5034C8	5034C8	0080
COM1	CM	000070	503730	503730	
DSEC1	DS	0000F4			
COM2	CM	000020	5037A0	5037A0	
SUBR	CS	000228	5037C0	5037C0	
DSEC2	DS	00002C	000009		
SQRT	LCS	000032	5039F0	5039F0	

MCMD

DEBUG COMMAND DESCRIPTION

Prototype: MCMD MTS-command

Action: Control returns to MTS command mode where the specified MTS command is executed. After the command is executed, control returns immediately to debug command mode.

As an alternative, any input line beginning with a dollar sign "$" is also executed as an MTS command.

Example:

```
MCMD $DUMP ON *PRINT*
```

In the above example, control returns to MTS command mode where a hexadecimal dump of the user's program is taken.

```
MCMD EDIT PROGRAM.SOU
edit commands
   .
   .
MTS
```

In the above example, the MTS file editor is invoked directly from debug command mode. After the file PROGRAM.SOU has been edited, control returns to debug command mode. In this manner, the user may update his program source file while debugging his program object file.

Page Revised March 1977 April 1976

MODIFY

DEBUG COMMAND DESCRIPTION

Prototype: MODIFY location {constant|location}

Action: The first parameter specifies the location that is to be modified and the second parameter specifies the value to be used for the modification.

constant specifies a list of one or more constants delimited by blanks or commas. The entire list is enclosed in primes. The constant or list of constants is placed in the specified location(s). If no attributes are specified for the constants, they are converted according to the type and length attributes of the first parameter.

If the second parameter specifies a storage location, then the hexadecimal contents of that location is used for the modification. The length used is the length attribute of the first parameter.

If the second parameter is a hexadecimal constant, the length of the constant, rather than the length of the parameter being modified, is used when SDS makes the modification.

Verification of the modification is given by printing both the old value and the new value of the location modified. Verification may be suppressed by entering terse mode (see the TERSE option in the SET command description). A listing of all modifications made to the program (except modifications made in an at-command list) may be obtained by using the LIST command, e.g.,

```
LIST MODS
```

Global breakpoints and at-points are temporarily restored before the modification if the breakpoint address is equal to the modification address.

Examples:

```
MODIFY BETA(1) F'10,20,30,40'
```

The first four locations starting with BETA(1) are modified to the constants 10, 20, 30, and 40, respectively.

```
MODIFY DELTA X'0000003E'
```

DELTA is modified in hexadecimal format to the constant 0000003E.

MODIFY GR1 GR3

The contents of general register 3 are copied to general register 1. GR3 is unchanged.

Page Revised March 1977 April 1976

MTS

DEBUG COMMAND DESCRIPTION

Prototype: MTS [MTS-command]

Action: Control returns to the caller (normally MTS command mode). The MTS command $SDS may be used to return control to debug command mode, from which the user can then resume debugging his program. An optional MTS command may be specified, which is executed before control is returned to the caller.

The effect of this command is identical to that of the RETURN command.

Example:

```
MTS $DUMP ON *PRINT*
```

In the above example, control returns to MTS command mode where a hexadecimal dump of the user's program is produced.

ON

DEBUG COMMAND DESCRIPTION

Prototype: (a) ON on-condition; command

(b) ON on-condition
command
command
.
.
END

where PGNT, ATTN, IOERR, LOCAL, or XFR specifies a program on-condition and "command" is a single debug command or a part of a list of debug commands to be executed in sequence.

With the single command prototype (a), a single debug command is specified, separated from the on-condition by a semicolon. If the condition occurs during program execution, the debug command is executed.

With the command list prototype (b), SDS enters into command insertion mode (indicated by the "?" prefix character) and reads debug commands until an END command is given. If the condition occurs during program execution, the debug command list is executed.

The following on-conditions may be specified:

PGNT	Program interrupt
ATTN	Attention interrupt
IOERR	I/O error
LOCAL	Local time estimate exceeded
TIMN	Timer interrupt
XFR	Call to suboutine LINK, LOAD, XCTL, or UNLOAD
MTS	Call to subroutine MTS
MCMD	Call to subroutine MTSCMD
SYSTEM	Call to subroutine SYSTEM
ERROR	Call to subroutine ERROR
RETURN	Program return to system

If PGNT is specified and a program interrupt occurs during program execution, the single debug command or command list is executed. If the program has called the subroutine PGNTTRP, the command list is executed before the PGNTTRP exit routine is called.

If ATTN is specified and an attention interrupt occurs during program execution, the single debug command or command list

is executed. If the program has called the subroutine ATTNTRP, the command list is executed before the ATTNTRP exit routine is called.

If IOERR is specified and an I/O interrupt occurs during program execution, the single debug command or command list is executed. If the program has called the subroutines SETIOERR or SIOERR, the command list is executed before the exit routine is called.

If LOCAL is specified and a local time estimate is exceeded during program execution, the single debug command or command list is executed.

If TIMN is specified and a timer interrupt occurs during program execution, the single debug command or command list is executed.

If XFR is specified and the program calls the system subroutines LINK, LOAD, XCTL, or UNLOAD, the single debug command or command list is executed. For calls to LINK, LOAD, and XCTL, the sections specified will have been loaded, but not entered, at the time the command list is executed. For calls to UNLOAD, the sections specified will have been unloaded at the time the command list is executed. The command list is also executed when sections that have been loaded by calls to LINK and XCTL are unloaded.

If MTS, MCMD, SYSTEM, or ERROR is specified and a call is made to one of the subroutines MTS, MTSCMD, SYSTEM, or ERROR, the single debug command or command list is executed.

If RETURN is specified and the program returns to the system (via the instruction BR 14), the single debug command or command list is executed.

The RESET and CLEAN commands may be used to delete on-conditions. The prototype for the RESET command is

RESET on-condition ...

The prototype for the CLEAN command is

CLEAN ONS

The LIST command may be used to list all of the current on-conditions. The prototype for the LIST command is

LIST ONS

Examples:

```
ON PGNT; GOTO LABEL2
```

In the above example, execution of the user's program transfers to location LABEL2 if a program interrupt occurs.

```
ON PGNT
IF GR15.EQ.F'0'
COMMENT SKIP SUBROUTINE SUBRX
GOTO $GR14
END
END
```

The above example illustrates how the ON command can be used to bypass the execution of an unloaded or unavailable external subroutine. For example, assume that the following two instructions are used to call the subroutine SUBRX:

```
L    15,=V(SUBRX)
BALR 14,15
```

If, after the calling program is loaded, the symbol SUBRX remains undefined, the literal V(SUBRX) will have a value of zero and a program interrupt will occur when a call to SUBRX is attempted. The above ON command list will test for such a program interrupt condition. If this type of program interrupt occurs, the comment "SKIP SUBROUTINE SUBRX" will be printed and execution will be resumed at the next instruction following the BALR instruction. If general register 15 is nonzero, normal program interrupt processing will occur. Note that there must be an END command for both the IF command list and the ON command list.

Page Revised March 1977 April 1976

PARLIST

DEBUG COMMAND DESCRIPTION

Prototype: PARLIST location [count]

Action: The PARLIST command may be used to display the contents of a parameter list in symbolic form with each parameter converted according to its data type. The address of the parameter list is specified by "location". The number of parameters to be displayed in the parameter list is specified by "count"; if omitted, parameters are displayed until the parameter list is exhausted or until an illegal parameter list adcon is encountered.

For each parameter in the parameter list, the following information is displayed:

the address of the parameter,
the symbolic name of the parameter, and
the value of the parameter.

If "location" contains the address of the system parameter list, the system parameter list is displayed in a special character format.

Example:

```
PARLIST $GR1
```

The parameter list pointed to by general register 1 is displayed.

April 1976 Page Revised March 1977

QUALIFY

DEBUG COMMAND DESCRIPTION

Prototype: QUALIFY location ...

Action: The length, type, and scale attributes of a symbol are changed according to the type, length, and scale modifiers appended to that symbol.

If the length attribute of a symbol is changed, any nonzero duplication factor associated with the symbol is automatically set to one; e.g., if a symbol is assembled with a type and length 256C is qualified with a length of 256, it in effect becomes C1256.

Predefined symbols or PL/I symbols may not be qualified.

Example:

```
QUALIFY ABC@T=X
```

The type attribute of ABC is changed to X (hexadecimal).

Page Revised March 1977 April 1976

RESET

DEBUG COMMAND DESCRIPTION

Prototype: RESET [on-condition ...]

Action: If no parameter is specified, all SDS SET options are reset to their default values. A list of these defaults, and the corresponding SET command options to change their values, is presented below.

Parameter	Command	Default
At-point processing	AT	ON
Command-insertion prefix character	ATPREFIX	?
Attention interrupt processing	ATTN	ON
Blank-named common symbol	BLANK	BLANK
Breakpoint processing	BREAK	ON
Duplicate common processing	COMMON	OFF
Duplicate dsect processing	DSECTS	OFF
Dump command processing	DUMP	FULL
SDS command source echoing	ECHO	MTS
Errordump facility	ERRORDUMP	OFF
Full scan option	FULLSCAN	OFF
Low-core simulation	FULLSIM	OFF
Indirection character	INDCH	$
Indexing	INDEX	ON
SDS command source	INPUT	*MSOURCE*[1]
Simulator instruction set	IS	70
Default length	LEN	4
LINK,LOAD,XCTL,UNLOAD processing	LLX	ON
Modifier character	MODCH	@
Open-map character	OMAPCH	*
SDS output sink	OUTPUT	*MSINK*[1]
Program interrupt processing	PGNT	ON
Pseudo-register area name	PRAREA	PRAREA
SDS prefix character	PREFIX	+
Default relocation factor	RF	0
Default scale factor	SCALE	0
Automatic simulation	SIM	OFF
Terse mode	TERSE	MTS
Timer interrupts	TIMN	OFF
Default type	TYPE	X (hex)
Warning messages	WARNMSG	ON
LINK,LOAD,XCTL,UNLOAD intercept	XFR	ON

[1]Initially, the command source and the output sink are *SOURCE* and *SINK*, respectively.

If PGNT, ATTN, IOERR, LOCAL, TIMN, XFR, MTS, MCMD, SYSTEM, ERROR, or RETURN is specified, the corresponding on-condition is deleted. The other SDS SET options are not reset.

April 1976 Page Revised March 1977

RESTORE

DEBUG COMMAND DESCRIPTION

Prototype: RESTORE [{location|name} ...]

Action: If "location" is specified, either

(1) the breakpoint or at-point is deleted and the original instruction is restored if it was specified by the BREAK or AT command, or
(2) the location is removed from the change or reference trace list if it was specified by the TRACE command.

If "name" is specified, the corresponding when-condition is deleted.

If no parameter is specified, the most recently entered global breakpoint or at-point set by the BREAK or AT command is deleted and the original instruction is restored.

"location" must be specified to restore a simulator breakpoint.

Example:

```
RESTORE LOC2
```

The breakpoint or at-point at location LOC2 is removed from the program.

RETURN

DEBUG COMMAND DESCRIPTION

Prototype: RETURN [MTS-command]

Action: Control returns to the caller (normally MTS command mode). An optional MTS command may be specified, which is executed before control is returned to the caller. The $SDS command returns control to debug command mode.

The effect of this command is identical to that of the MTS command.

Page Revised March 1977 April 1976

RUN

DEBUG COMMAND DESCRIPTION

Prototype: RUN [location ...]

Action: Control is transferred to the entry point of the program and execution is started. General registers 1, 13, 14, and 15 are set to the following values:

GR1 points to the PAR field character string.
GR13 points to a system save area.
GR14 contains the return address.
GR15 contains the entry point address.

The other registers and the floating-point registers are set to zero. The program mask and condition code are set to zero. The entry point and the PAR field may be changed by the ENTRY and PAR options of the SET command.

If a location is specified, a local breakpoint is established at the specified location by replacing the opcode of the instruction with X'00'. When the program encounters a local breakpoint, control will be returned to debug command mode. Local breakpoints are in effect only for the duration of the command, and are automatically deleted before the user enters the next command.

For programs that are serially reusable, i.e., capable of being rerun several times without being reloaded, the RUN command may be used to restart the program at its entry point. Programs that are serially reusable are either reentrant or they do not modify their constant areas. User program storage that is dynamically acquired is not automatically released by SDS when the program is rerun.

The CONTINUE, GOTO, and STEP commands also may be used to initiate execution of the program without altering the general or floating-point registers.

Example:

```
RUN LOC2
```

A local breakpoint is set at location LOC2 and execution of the user's program is initiated.

SCAN

DEBUG COMMAND DESCRIPTION

Prototype: SCAN [{section|location1...location2|*} {constant|location}]

Action: If "section" is specified, the named section is searched for the value specified. If "location1...location2" is specified, the area bounded by "location1" and "location2" is searched. If "*" is specified, all loaded control sections, common sections, and defined dsects are searched. In all cases, all unreferenced or protected pages are skipped without comment.

If a constant is specified, the value of the constant is used as the argument for the scan; if a location is specified, the contents of that location is used as the argument.

If the type specified is hexadecimal, character, or packed or zoned decimal, the length of the constant specified is taken as the length of the value to scan for; otherwise, the default type and length attributes are used for the scan if no TYPE and LENGTH modifiers are specified on the first parameter.

If no parameter is specified, then the search resumes starting at the first location beyond the previous match from the previous SCAN command.

The search for the specified value is performed with respect to the appropriate boundary alignment of the value specified, i.e., instructions are scanned for on halfword boundaries, character constants on byte boundaries, etc.

The address parameter may contain unreferenced or protected pages within its range; e.g.,

```
SCAN 600000...6FFFFF C'ABCDEF'
```

scans all of the user's referenced virtual memory in segment 6 for the character string ABCDEF; all unreferenced or protected pages within this range are ignored without comment.

The FULLSCAN option may be used to scan the address range for all occurrences of the argument instead of stopping after the first occurrence. This may be enabled by setting FULLSCAN=ON with the SET command; the default is OFF.

Examples:

```
SCAN MAIN D'3.33762'
```

The section MAIN is scanned for the double-precision floating-point constant 3.33762.

```
SCAN 616100...6162DB C'OUTPUT'
SCAN
```

The region 616100...6162DB is scanned for the character constant OUTPUT. Then the remainder of the same region is scanned for a second occurrence of OUTPUT. If the first SCAN command is terminated without encountering a successful match, the second SCAN command has no effect.

April 1976 Page Revised March 1977

SDS

DEBUG COMMAND DESCRIPTION

Prototype: SDS

Action: If an AT, IF, or WHEN command list is being processed, the command list processing is terminated and control is returned to debug command mode. The SDS command is normally given as the command preceding the END command in the command list.

In debug command mode, the SDS command has no effect.

This command is not the same as the MTS $SDS command which is used to enter debug command mode from MTS command mode.

Example:

```
AT LOC5
DISPLAY A(1)...(9)
DISPLAY FRS
SDS
END
```

After the AT command list for the at-point at location LOC5 is processed, control is returned to debug command mode.

SET

DEBUG COMMAND DESCRIPTION

Prototype: SET option ...

Action: The SET command is used to alter the status of various SDS options, default attributes, or default characters. The valid keyword parameters are as follows:

AT={ON|OFF} If the option is OFF, all current at-points are ignored and program execution are automatically continued. The at-points are not restored; if an IGNORE command is in effect for the at-point, its count is decremented as usual each time it is encountered. The default is ON.

ATPREFIX=char The command-insertion mode prefix character becomes the character specified by char. The default is "?".

ATTN={ON|OFF} If the option is OFF, user attention interrupt exit routines set up by calls to the subroutine ATTNTRP are disabled; SDS will process all attention interrupts. The option may be set to OFF before or after the user's program has called the subroutine ATTNTRP. The option may be set to ON to restore normal attention interrupt processing by the program. The default is ON.

BLANK=name The blank-named common symbol used to refer to blank-named common sections is set to name. The symbol may not begin with the current indirection character "$", and may not contain any of the following characters: ()+-,.='*/ or the current modifier character "@". The default value is BLANK.

BREAK={ON|OFF} If the option is OFF, all current global breakpoints are ignored and program execution are automatically continued. The breakpoints are not restored; if an IGNORE command is in effect for the breakpoint, its count is decremented as usual each time it is encountered. The default is ON.

BRIEF={ON|OFF|MTS} If the option is ON, SDS enters terse mode which eliminates many confirmation and diagnostic messages. If the option is OFF, complete message processing occurs. BRIEF is a synonym for TERSE and an antonym for VERBOSE. The default is the setting of the corresponding MTS BRIEF option.

COMMON={ON|OFF} If this option is OFF, the symbols of all multiple occurrences of common sections of the same name with the same storage index number are merged, thus reducing the size of the symbol table. If this option is ON, the symbols are kept separate; this is useful for checking whether each occurrence of a common section of the same name is identical in length and symbol order. The default is OFF.

DSECTS={ON|OFF} If the option is OFF, all multiple occurrences of dsects of the same name with the same storage index number are ignored, thus reducing the size of the symbol table. This option must be set to OFF before the program is loaded. The default is OFF.

DUMP={SHORT|FULL} If this option is SHORT, an abbreviated symbolic dump is produced by the DUMP command; this dump omits all FORTRAN and PL/I arrays except for the first element in the array; all other data variables are included. If this option is FULL, the symbolic dump contains all data variables. The default is FULL.

ECHO={ON|OFF|MTS} If the option is ON, input commands to SDS are echoed on the current output file or device if it is different from the input file or device. The default is the setting of the corresponding MTS ECHO option.

ENTRY=location The entry point to the user's program is set to the location specified; it may be a symbolic, relative, or virtual address.

ERRORDUMP={ON|OFF|FULL}
If the option is ON and the user is running in batch mode, a symbolic dump is automatically given in the event of an abnormal program termination; library-

loaded sections are excluded from the dump. If this option is FULL, library-loaded sections are included in the dump. The default is OFF.

FULLSCAN={ON|OFF} If this option is ON, the SCAN command scans for all occurrences of the argument in the specified address range instead of stopping after the first occurrence. The default is OFF.

FULLSIM={ON|OFF|FULL}
If this option is ON, subroutines that reside in the low-core symbol tables <EFL>, PL1SYM, and <FIX> are simulated. If this option is FULL, all low-core subroutines are simulated. If this option is OFF, these subroutines are executed instead of simulated when called by the SDS simulator as is always the case for subroutines residing in LCSYMBOL. The default is OFF.

INDCH=char The indirection character becomes the character specified by "char". The default is "$".

INDEX={ON|OFF} If the option is OFF, only displacements from the nearest symbolic location are used to display an address. If this option is ON, both indexes and displacements are used. The default is ON.

INPUT=FDname SDS reads subsequent commands from the file or device specified by "FDname". If an end-of-file is detected from the new command stream, or if an attention interrupt, a program interrupt, or a breakpoint (not an at-point) is encountered, SDS returns to *MSOURCE* for its commands.

IS={67|70} If this option is 67, the IBM Model 360/67 instruction set is used to simulate the user program. If this option is 70, the 370 instruction set is used to simulate the program. The default is 70.

For programs running on a IBM Model 360/67, the following extended-precision (16-byte), floating-point 370 instructions are simulated approximately by

their equivalent long-precision (8-byte), floating-point instructions:

Extended-Precision	Long-Precision
LRER	LER
LDDR	LDR
AXR	ADR
SXR	SDR
MXR	MDR
MXDR	MDR
MXD	MD

The MC (Monitor Call) instruction is treated as a NOP instruction. All other nonprivileged 370 instructions are simulated exactly.

For programs running on a 370, the following IBM Model 360/67 RPQ instructions are simulated approximately by their nearest 370 equivalent instructions:

Model 360/67 RPQ	370 Equivalent
MDDR	MXDR
MDD	MXD

All other IBM Model 360/67 RPQ instructions (ADDR, ADD, SDDR, SDD, AX, SX, MX, DX, LX, SLT, SWPR, BAS, and BASR) are simulated exactly.

LEN=i

The default length attribute is set to "i", where "i" is an unsigned decimal integer between 1 and 65535. Initially, the default length attribute is 4.

LLX={ON|OFF}

If the option is OFF, all sections which are dynamically loaded by calls to the subroutines LINK, LOAD, or XCTL are not entered into the symbol table, thus reducing the size of the symbol table. The default is ON. Note that this option is independent of the setting of the XFR option which controls the intercepting of calls to the LINK, LOAD, XCTL, or UNLOAD subroutines.

LIOunit=FDname

Logical I/O unit assignments for the user's program may be given for both input and output units. This may be used to rewind sequential or line files but

will not affect the status of other types of pseudo-devices (such as *SOURCE* and the position of tapes).

MODCH=char — The modifier character becomes the character specified by "char". The default is "@".

OMAPCH=char — The "open-map" character becomes the character specified by "char". The default is "*".

OUTPUT=FDname — SDS writes subsequent output lines to the file or device specified by "FDname". If an attention interrupt, a program interrupt, or a breakpoint (not an at-point) is encountered, SDS switches its output to *MSINK*.

PAR=text — The PAR field is replaced by the character string specified by "text". Since "text" includes the remainder of the input line, the PAR option must be the last option specified on the SET command.

PGNT={ON|OFF} — If the option is OFF, user program interrupt exit routines set up by calls to the subroutine PGNTTRP are disabled; SDS processes all program interrupts. The option may be set to OFF before or after the user's program has called the subroutine PGNTTRP. The option may be set to ON to restore normal program interrupt processing by the program. The default is ON.

PRAREA=name — The pseudo-register area symbol used to refer to the pseudo-register area is set to "name". The symbol may not begin with the current indirection character "$", and may not contain any of the following characters: ()+-,.="*/ or the current modifier character "@". The default value is PRAREA.

PREFIX=char — The SDS prefix character becomes the character specified by "char". The default is "+".

RF=constant — The unsigned hexadecimal "constant" defines the default relocation factor. This constant is added to every relative address which is not qualified by the @C

modifier. The default value for the relocation factor is 0.

SCALE=i — The default binary scale factor is set to "i", where "i" is a signed or unsigned decimal integer. Initially, the default scale factor is 0.

SIM={ON|OFF} — If this option is ON, any command that starts program execution calls the SDS simulator to simulate the program. The default is OFF. By using the SIM option in conjunction with the IS=67 option, programs using the IBM Model 360/67 instruction set may be simulated on the 370.

TERSE={ON|OFF|MTS} — If the option is ON, SDS enters terse mode which eliminates many confirmation and diagnostic messages. If the option is OFF, complete message processing occurs. TERSE is a synonym for BRIEF and an antonym for VERBOSE. The default is the setting of the corresponding MTS TERSE option.

TIMN={ON|OFF} — If the option is OFF, timer interrupts set by the user program are ignored and program execution continues. The default is ON.

TYPE=x — The default type attribute is set to "x", where "x" is any of the SDS type codes described with the description of the TYPE keyword modifier. Initially, the default type attribute is X (hexadecimal).

VERBOSE={ON|OFF|MTS} — If the option is OFF, SDS enters terse mode which eliminates many confirmation and diagnostic messages. If the option is ON, complete message processing occurs. VERBOSE is an antonym for BRIEF and TERSE. The default is the setting of the corresponding MTS VERBOSE option.

WARNMSG={ON|OFF} — If the option is OFF, all warning messages concerning addresses outside of csect bounds and subscripts outside of array bounds are suppressed. The default is ON.

XFR={ON|OFF} If the option is ON, SDS intercepts all calls to the subroutines LINK, LOAD, XCTL, and UNLOAD, and LINK-returns and return to debug command mode. The modules specified in subroutine calls to LINK, LOAD, and XCTL are loaded and the registers are set up for the execution of the loaded modules (in the case of LINK and XCTL). For XCTL, the calling program is unloaded and its symbols are purged from the SDS map. For UNLOAD and LINK-return, the module is unloaded and its symbols are purged from the SDS map. If the option is OFF, SDS does not intercept the subroutine calls. The default is OFF. Note that this option is independent of the setting of the LLX option which controls the symbol table processing of modules dynamically loaded by calls to the LINK, LOAD, or XCTL subroutines.

Examples:

```
SET LEN=8 TYPE=C TERSE=ON
```

This command sets the default length attribute to 8 bytes, the default type attribute to C (character), and sets terse mode ON.

```
SET SCARDS=INPUT SPRINT=OUTPUT PAR=EXEC
```

This command sets SCARDS and SPRINT to the files INPUT and OUTPUT, respectively, and replaces the current PAR field with the character string EXEC.

STEP

DEBUG COMMAND DESCRIPTION

Prototype: STEP [i [BRANCH]]

Action: The next "i" machine language instructions in the user's program are simulated before control returns to SDS. If "i" is not specified, only the next instruction is simulated.

If BRANCH is specified, the user's program is simulated until either "i" instructions are executed or a successful branch is executed. If simulation is terminated due to a successful branch, the number of unexecuted instructions remaining in the step count "i" is printed in the form

REMAINING STEP COUNT = x

Comments: If the user attempts to STEP past a branch instruction, the branch is taken as usual unless the program is transferring to a legal low-core symbol such as SCARDS or SPRINT. In this case, the routine is executed, not simulated, and stepping resumes at the return address. The instructions executed in the routine are not counted in the stepping count.

If the branch address is in resident-system storage and does not correspond to the entry point of a legal low-core symbol, simulation is terminated and a warning message is printed. The user must restart his program with either a CONTINUE or GOTO command.

If STEP is used instead of RUN to initiate program execution, registers 1, 13, 14 and 15 are loaded with the appropriate values (see the RUN debug command description).

FORTRAN and PL/I users should note that the STEP command specifies machine language instructions in the count. If it is desired to step a specified number of FORTRAN or PL/I instructions, the CONTINUE command should be used specifying local breakpoints.

Example:

```
STEP 10
```

The next 10 instructions in the program are simulated.

Page Revised March 1977 April 1976

STOP

DEBUG COMMAND DESCRIPTION

Prototype: STOP

Action: SDS processing is terminated and control is returned to the caller (normally MTS command mode).

All loaded-program symbol table information and SDS work storage are released.

If the user's program was loaded via the $DEBUG command, the program is unloaded; if the program was loaded via the $RUN, $RERUN, or $LOAD commands, the program is not unloaded.

If STOP is encountered in an AT, IF, or WHEN command list, it is interpreted as an SDS command and control is returned to debug command mode.

SYMBOL

DEBUG COMMAND DESCRIPTION

Prototype: SYMBOL location ...

Action: The symbolic name for the specified location is printed. If no symbol table is present, the relative address and section name are printed. If this cannot be done, the corresponding virtual address is printed.

If a location has more than one symbolic name, all of its names are printed.

Examples:

```
SYMBOL ALPHA+4 $GR14 516020
```

This example displays the above locations in the following form:

```
ALPHA+4 = VSYS IN SECTION SUBR
$GR14 = RETRN(8) IN SECTION SUBR
516020 = BETA IN SECTION MAINPROG
516020 = BMARK IN SECTION MAINPROG
```

In this example, the location 516020 has two symbolic names.

TIMETALLY

DEBUG COMMAND DESCRIPTION

Prototype: TIMETALLY {ON|OFF|PRINT|RESET|CLEAR} [option ...]

Action: The TIMETALLY command may be used to monitor the execution of a program and to produce a histogram depicting the distribution of CPU or real time within the program, i.e., to show where the program spends its time.

The TIMETALLY algorithm is a technique of random sampling of program execution to determine the distribution of CPU activity.

When TIMETALLY is invoked, an internal data structure is constructed representing the various control sections and entry points in the program. Each of the control sections is divided into partitions of a fixed size and a counter is established for each partition. A timer interrupt is set up to occur after a specified number of milliseconds of user program execution. When the timer interrupt occurs, the data structure is scanned to determine in which partition the interrupt occurred, and its counter is incremented by one. Another timer interrupt is set up for the same interval and program execution is resumed. Each time a timer interrupt occurs, the appropriate counter is incremented and a new timer interrupt is set up. This pattern is continued throughout program execution. When the TIMETALLY results are printed, a determination of the total number of interrupts taken is made on a partition-by-partition basis and a histogram depicting the partition percentages is printed.

TIMETALLY intercepts calls to MTS resident-system subroutines (those defined in LCSYMBOL, <EFL>, <FIX>, and PL1SYM) and saves, for each call, the name of the subroutine and the point from which the subroutine was called. This enables the determination of the time spent within system subroutines as a function of the points from which they were called.

A special partition ?SYSTEM is established for counting the number of timer interrupts taken in MTS and the supervisor for which the cause is not apparent. One common cause of this problem is terminal output. Since terminal output may be overlapped with program execution, the asynchronous handling of the completion of an output operation represents system CPU activity for which the cause is not apparent.

A special partition ?USER is established for counting the number of timer interrupts taken at addresses outside of MTS

and the supervisor, but which are not contained in any of the partitions defined for the user program. A program which dynamically acquires a region of storage, moves instructions into it, and proceeds to execute them, may produce such interrupts.

The following options may be specified on the TIMETALLY command. More than one option may be specified on the command. The first five options control the enabling and disabling of the TIMETALLY facility.

ON — This option invokes TIMETALLY and enables the collection of TIMETALLY data during program execution. Execution of the program may be started by the RUN debug command.

OFF — This option disables the collection of data. This is needed only if the user desires to collect data in a smaller portion of the program.

PRINT — This option prints the accumulated TIMETALLY data collected during program execution.

RESET — This option resets the accumulated data collected to zero. This is needed only if the user desires to issue a subsequent RUN command without reloading the program.

CLEAR — This option terminates the TIMETALLY facility and releases all TIMETALLY data and storage. This storage is also automatically released when the program is unloaded.

The remaining options control the behavior of the TIMETALLY facility.

OUTPUT=FDname
This option specifies the file or device on which the data results are to be printed. If this is omitted, the results are printed on the current output file or device.

OBL=n — This option specifies the maximum line length (output record length) of the histogram. If this option is omitted, a value of 132 or the maximum output record length of the file or device specified by the OUTPUT option is assumed, whichever is smaller. At least one line of the histogram will be of maximum length. "n" must be in the range $33 \leq n \leq 132$.

PSIZE=n This option sets the size (in bytes) of the partitions into which the control sections are divided. If this option is omitted, a partition size of 256 bytes is assumed.

DELTAT=dd
DTDEV=ee These options set the interval to be used for scheduling timer interrrupts during execution of the program. Timer interrupts are scheduled with an interval time that is uniformly distributed over the interval "dd±ee" milliseconds. The purpose of the uniform random distribution is to avoid unwanted correlation between sampling frequencies and cyclic program execution. If DELTAT is not specified, a value of 1.5 is assumed. If DTDEV is not specified, a value of DELTAT/2 is assumed. The values specified must be such that dd-ee > 0.4.

{CPUTIME|REALTIME}
This option specifies whether the timer interrupts are to be based on CPU time or real time. If CPUTIME is specified, the timer interrupts are based on CPU time using the value specified by DELTAT. If REALTIME is specified, the timer interrupts are based on real time using a DELTAT value of 200 milliseconds. The default is CPUTIME.

SYS={NONE|NORMAL|FULL}
This option specifies the manner in which timer interrupts that occur in system storage (segments 0-5) are to be handled. The default is NORMAL.

If NONE is specified, timer interrupts occurring in system storage are ignored; this enables estimates of "nonsystem" timing only.

If NORMAL is specified, information is recorded indicating the name of the resident-system subroutine called and the point from which the subroutine is called.

If FULL is specified, in addition to recording the information specified by NORMAL, segments 0 through 5 are treated as large control sections and information is recorded giving the location where the timer interrupt occurred. Since the timing statistics produced by the FULL option are in terms of virtual addresses, the user must have an MTS system load map in order to interpret this output. In addition, knowledge of the

internal structure of MTS is needed to meaningfully use this option.

The TIMETALLY facility may be used while the program is being actively debugged. However, data is not collected if the program is being simulated, e.g., when the STEP or WHEN commands are being used.

Example: The following example illustrates the use of the TIMETALLY facility in SDS. Input from the user is in uppercase; output from SDS is in lowercase.

```
#DEBUG PROGRAM 5=INPUT 6=OUTPUT
+ready
+TIMETALLY ON
+done
+RUN

     program execution

+user program return.
+TIMETALLY PRINT

     TIMETALLY results

+done
+STOP
#
```

TRACE

DEBUG COMMAND DESCRIPTION

Prototype:

(a) TRACE FLOW

(b) TRACE LABEL location •••

(c) TRACE CALLS [location •••]

(d) TRACE COUNT location •••

(e) TRACE {LINKAGE|SAVEAREA}

(f) TRACE {CHANGES|REFERENCES} location •••

Action:

Prototype (a) is used to specify flow tracing of the program. With flow tracing, each time the program makes a branch, both the new and old instruction locations and the current condition code are printed in the form:

newlocation FROM oldlocation (CC=x)

Prototype (b) is used to specify label tracing of the program. With label tracing, each time the program passes through one of the specified locations, a message is printed in the form:

*** AT LABEL location

Prototype (c) is used to specify subroutine call and return tracing. A special call intercept is established at all control section bases and entry points and any optionally specified locations in the program. When a call intercept is encountered, a message is printed in the form

subroutine CALLED FROM address

and a return intercept is established using the address contained in GR14. When the return intercept is encountered, a message is printed in the form

subroutine RETURNS TO address

and the return intercept is removed. Warning: Call and return tracing will only work properly for programs that call subroutines using the standard MTS subroutine linkage conventions.

Prototype (d) is used to count the number of times an instruction at a specified location is executed. No confirmation of instruction execution is printed during program execution. The LIST command may be used to list the accumulated counts for each location, i.e.,

LIST COUNTS

Prototype (e) is used to display the current savearea chain or subroutine linkage chain. If SAVEAREA specified, the contents of the current savearea chain is displayed starting from the current savearea and going back to the entry point of the program. If LINKAGE is specified, the contents of the registers at each subroutine call is printed starting from the current subroutine call and going back to the entry point of the program.

Prototype (f) is used to specify location change and reference tracing. With CHANGE tracing, each time the value of one of the locations specified is changed to a different value, the new value is printed along with the location at which it was changed in the form:

symbol value location

With REFERENCE tracing, each time one of the locations specified is referenced, the value is printed along with the reference type and the location at which the reference was made in the form:

symbol value type location

The reference types are F (fetch) and S (storage).

FLOW, CHANGE, or REFERENCE tracing is either active or inactive. An active trace condition forces any command that starts program execution to call the SDS simulator to simulate the program. An inactive trace condition does not force program simulation. Trace conditions may be activated or deactivated by the ACTIVATE and DEACTIVATE commands. LABEL, CALL, COUNT, SAVEAREA, and LINKAGE tracing do not force program simulation.

A trace condition is initially active; for flow tracing, each branch instruction is tested for a successful branch. When a branch is made, the appropriate trace message is printed. For change and reference tracing, each location specified is tested after every instruction in the user's program for changes or references. When a change or reference is made, the appropriate trace messages are printed.

Several trace conditions may be active at any one time, although the more conditions and symbols being tested after

every instruction step, the more expensive the use of the command will be.

The ACTIVATE and DEACTIVATE commands may be used to control the range of tracing. The prototypes for these commands are:

```
ACTIVATE {FLOW|CHANGES|REFERENCES} ...
DEACTIVATE {FLOW|CHANGES|REFERENCES} ...
```

An example of the use of these commands is given below.

The CLEAN command is used to delete trace conditions (and their corresponding location lists, if any) from the SDS tables. The prototype for the CLEAN command is

```
CLEAN {FLOW|LABEL|CALLS|COUNT|CHANGES|REFERENCES} ...
```

The RESTORE command is used to delete specified locations for change and reference trace conditions. The prototype for the RESTORE command is

```
RESTORE location ...
```

The LIST command may be used to list all of the current trace and their corresponding location lists. The list includes the trace condition name, location list, and current status. The prototype for the LIST command is

```
LIST {FLOW|LABEL|CALLS|COUNT|CHANGES|REFERENCES} ...
```

An example of the LIST command is:

```
LIST FLOW CHANGES
```

The listing output is in the following form:

```
FLOW TRACING (ACTIVE)
CHANGE TRACING (ACTIVE)
  ALPHA BETA
```

Examples:

```
TRACE FLOW
```

The above example produces output in the following form:

```
MARK1 IN SECTION MAIN FROM MARK0 (CC=0)
MARK4 FROM MARK3 (CC=2)
MARK6 FROM MARK5 (CC=2)
SUBR IN SECTION SUBR FROM MARK8 (CC=0)
```

As seen in the above set of trace messages, the name of the section is printed only when it changes from the previous section.

```
TRACE CHANGES ALPHA BETA
```

The above example produces output in the following form:

```
ALPHA  'F'  +2  MARK2 IN SECTION MAIN
ALPHA  'F'  +3  MARK4
BETA   'E'  1.  MARK6
```

```
TRACE REFERENCES ALPHA BETA
```

The above example produces output in the following form:

```
ALPHA  'F'  +1  (F)  MARK1 IN SECTION MAIN
ALPHA  'F'  +2  (S)  MARK2
ALPHA  'F'  +3  (S)  MARK4
BETA   'E'  1.  (S)  MARK6
BETA   'E'  1.  (F)  MARK7
```

```
TRACE FLOW
DEACTIVATE FLOW
AT MARK4; ACTIVATE FLOW
AT MARK10; DEACTIVATE FLOW
```

In the above example, the ACTIVATE and DEACTIVATE commands are used in conjunction with the AT command to restrict the range over which flow tracing is active. Flow tracing is active and the program is simulated in the range of MARK4 to MARK10; otherwise, flow tracing is inactive and the program is not simulated.

USING

DEBUG COMMAND DESCRIPTION

Prototype: USING {section|symbol} location

Action: The section (normally a dsect) named by "section" may be assigned an address in two ways:

(1) "location" is a storage location which is used as a static base address for the section.
(2) "location" is a general register, the contents of which is used as a dynamic base address for the section (i.e., the base address varies with the contents of the register). If the address is given in the form of $GRx, the current contents of the register is used as a static base for the section.

A section may be redefined by subsequent USING commands. The section length may be redefined by specifying "section@L=i" or "location@L=i". If the section is not a dsect, a warning message is printed when it is redefined.

If the section is a dsect and occurs in more than one assembly or if there is a csect and dsect of the same name, the @D and @C keyword modifiers may be used to specify the desired dsect.

"symbol" may be specified to assign an address for a dsect at an offset from the dsect base. Only symbols contained in dsects are valid.

The DROP command may be used to undefine a dsect.

Examples:

```
USING DSECT1 GR1
```

The section DSECT1 becomes dynamically addressable by the contents of general register 1.

```
USING DSECT2 516200
```

The section DSECT2 becomes statically addressable by the base address 516200.

```
USING WORKAREA@C=SUBA $WADDR
```

The dsect WORKAREA from the assembly which contains the section SUBA becomes statically addressable by the current contents of the location WADDR.

WHEN

DEBUG COMMAND DESCRIPTION

Prototype: (a) WHEN [name] {expression|location CHANGES}; command

(b) WHEN [name] {expression|location CHANGES}
command
command
.
.
END

where "name" is an optional name to be assigned to the when-condition (for subsequent referencing of the condition), "expression" is a relational or logical expression, "location" is a storage reference location, and "command" is a single debug command or a part of a list of debug commands to be executed in sequence.

With the single command prototype (a), a single debug command is specified, separated from the expression by a semicolon. When the expression is satisfied during program execution, the debug command is executed.

With the command list prototype (b), SDS enters into command insertion mode (indicated by the "?" prefix character) and reads debug commands until an END command is given. When the expression is satisfied during program execution, the debug command list is executed.

The expression that must be satisfied during program execution may be either a relational or logical expression or a single location in the form

location CHANGES

If a single location is specified, the expression is satisfied when the location specified changes in value.

When-conditions are either active or inactive. An active when-condition forces any command which starts program execution to call the SDS simulator to simulate the program. An inactive when-condition does not force program simulation.

The when-condition is initially active; its expression is tested before the first instruction and subsequently after every instruction in the user's program is executed until it is satisfied. When the expression is satisfied, the single command or list of commands specified is executed automati-

cally and the when-condition is deactivated. When a when-condition is satisfied, confirmation is given by printing the expression and the address of the next instruction to be executed in the program. This confirmation is suppressed in terse mode.

Several when-conditions may be active at any one time, although, the more expressions that are being tested after every instruction step, the more expensive the use of the command will be.

If a "name" is specified with the command, it must be given in the form of "/xxx", where "xxx" is an arbitrary string of one to three characters. If the name is omitted, SDS assigns a name to the when-condition. This name is used for subsequent activating, deactivating, or deleting the condition from the SDS tables.

WHEN commands (and their respective command lists) may be nested in other WHEN, IF, or AT command lists. Each nested command list must be individually terminated by an END command (unless a single command form is used).

The ACTIVATE and DEACTIVATE commands may be used to control the range of condition testing. The prototypes for these commands are:

ACTIVATE name •••
DEACTIVATE name •••

where "name" is the name of the when-condition to be activated or deactivated. The ACTIVATE command may be used to reactivate a when-condition that has been explicitly deactivated by the DEACTIVATE command or implicitly deactivated after its expression is satisfied during program execution.

The RESTORE and CLEAN commands may be used to delete when-conditions from the SDS tables.

The prototype for the RESTORE command is

RESTORE name •••

where "name" is the name of the when-condition to be deleted.

The prototype for the CLEAN command is

CLEAN WHENS

The LIST command is used to list all of the current when-conditions. The list includes the name, expression, and

activity status for each when-condition. The prototype for the LIST command is

LIST WHENS

Users should be aware that the use of the WHEN command is rather expensive in terms of processing time. This is due to the fact that while when-conditions are active, the program is being simulated and considerable testing of the program is being performed to determine when the expressions specified are satisfied. Hence, users should use the WHEN command with restraint and only for those parts of the program that are being debugged. The ACTIVATE and DEACTIVATE commands will aid in restricting the use of the WHEN command to reasonably small portions of the program.

The cost of simulating a portion of a program compared to executing it normally is approximately 50 to 1. This ratio becomes larger as when-conditions are activated. The upper limit of this ratio depends on the following:

(1) The number and complexity of when-conditions that are active at any one time.
(2) The number of times the expressions must be tested during the simulation of the program. Expressions involving the general registers are tested each time the specified register changes value. Expressions involving floating-point registers are tested each time any floating-point register changes value. Expressions involving program locations (variables) are tested each time a program location within the lower and upper bounds of all program locations being tested changes value; that is, if the expression being tested is

ALPHA = BETA & DELTA = GAMMA

where ALPHA, BETA, DELTA, and GAMMA are locations in the program in order of increasing address, then each time any location between ALPHA and GAMMA changes value, the expression is tested.

Examples:

WHEN ALPHA = F'6'; SDS

In the above example, execution of the user's program halts when the value of ALPHA becomes 6; control is returned to debug command mode.

WHEN ALPHA CHANGES; SDS

In the above example, execution of the user's program halts when the value of ALPHA changes to a new value; control is returned to debug command mode.

```
WHEN GR5 <= X'0000FFFE'
MODIFY GR5 X'0000FFFF'
GOTO LABEL2
END
```

In the above example, when the value of general register 5 becomes less than or equal to 0000FFFE, the register is modified to 0000FFFF and program execution is resumed at location LABEL2.

```
WHEN /ABC ALPHA = F'0'; SDS
DEACTIVATE /ABC
AT LOC3; ACTIVATE /ABC
AT LOC4; DEACTIVATE /ABC
```

In the above example, the ACTIVATE and DEACTIVATE commands are used in conjunction with the AT command to restrict the range over which the when-condition /ABC is active. The when-condition is active and the program is simulated in the range of LOC3 to LOC4, and the expression is tested during this time; otherwise, the condition is inactive and the program is executed rather than simulated.

Page Revised March 1977 April 1976

11/23/79 21:35:52 LIST CC