CRAY RESEARCH, INC.

CRAY X-MP AND CRAY-1® COMPUTER SYSTEMS

SYMBOLIC INTERACTIVE DEBUGGER USER'S GUIDE

SG-0056

RECORD OF REVISION

PUBLICATION NUMBER SG-0056

Each time this manual is revised and reprinted, all changes issued against the previous version in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator, starting with 01 for the first change packet of each revision level.

Every page changed by a reprint or by a change packet has the revision level and change packet number in the lower righthand corner. Changes to part of a page are noted by a change bar along the margin of the page. A change bar in the margin opposite the page number indicates that the entire page is new; a dot in the same place indicates that information has been moved from one page to another, but has not otherwise changed.

Requests for copies of Cray Research, Inc. publications and comments about these publications should be directed to:

CRAY RESEARCH, INC.,
1440 Northland Drive,
Mendota Heights, Minnesota 55120

Revision	Description
	June, 1982 - Original printing.
01	December, 1983 - This change packet brings the manual into agreement with version 1.13 of COS. It adds the TRACEBACK directive, information about use with CFT, examples, and minor additions.
A	January, 1985 - This reprint incorporates change packet 01 with the original printing. No other changes have been made.

PREFACE

This manual describes the operation of the Cray Research, Inc., Symbolic Interactive Debugger (SID), a powerful debugging tool that allows the programmer to debug programs interactively or in batch mode. SID operates on all series and models of Cray Computer Systems. The programmer is assumed to be familiar with the Cray Operating System (COS), and Cray Assembly Language (CAL) or Cray FORTRAN (CFT).

Other publications that the reader may find useful are:

SR-0000	CAL Assembler Version 1 Reference Manual
SR-0009	FORTRAN (CFT) Reference Manual
SR-0011	CRAY-OS Version 1 Reference Manual

CONTENTS

INTRODUCTION 1

The Cray Research, Inc., Symbolic Interactive Debugger (SID) is a programming tool for debugging programs interactively or in batch mode. SID operates under control of the Cray Operating System (COS) on a Cray Computer System. It can be used for programs written in Cray Assembly Language (CAL), Cray FORTRAN (CFT), or a combination. Loading and executing a program with SID requires approximately 100K octal additional words of memory. SID can be used in interactive or batch mode, and can reference variables and statement labels by their names or absolute addresses.

SID is an alternative to the Cray Simulator (CSIM) for testing programs and for inserting snap dumps into the source code. It offers the advantages of speed to simulator users and flexibility to programmers now using snap dumps. The user program does not have to be altered or recompiled, and the user can make decisions about debugging during the session.

With SID the user can:

- Stop a program at any instruction,
- Step through one or more instructions,
- Examine the contents of memory locations (either program or data) in several formats, and
- Patch new values into memory locations.

The user program, which is executed rather than simulated, cannot be stopped when a particular value is written into a memory location.

Because SID is a symbolic debugger, it allows symbols to be used in place of word or parcel addresses. Variables and statement labels can be referenced by name as well as by absolute address. Offsets can be used for both parcel and word addresses, and indirect addressing can be used for word addresses.

WORD ADDRESSES

Word addresses can be absolute addresses or symbolic addresses. A word address is a symbol, a constant, a symbol and constant joined by addition or subtraction, or a subscripted array. Optional parentheses indicate indirect addressing. The address of the symbol must start on the word boundary. Some expressions for a word address are: *symbol*; *number*; *symbol+number*; (*symbol*)-*number*; and (*number*).

Word-address symbols can be user symbols or special symbols.

Examples:

symbol	SUM
number	214 D'137
symbol+number	LIST+3
(symbol)-number	(A5)-5
(number)	(315)

PARCEL ADDRESSES

A parcel address consists of a symbol with parcel attribute, a parcel constant, or a symbol with parcel attribute and a parcel constant joined by addition or subtraction. Indirect addressing cannot be used in parcel address expressions.

Examples:

symbol	S'1000 LOOP N'143 PC
parcel constant	316A
symbol+parcel	LOOP+0C

SUBSCRIPTED VARIABLES

Subscripted variables can be used for array elements in CFT programs. Subscripts must be integer constants or simple integer variables. Indirect addressing and additional offset cannot be used with subscripted variables.

Examples:

```
LIST(3)
TABLE(I,J)
```

SYMBOL DEFINITIONS

Symbols used in SID are either special symbols known to SID or symbols found in a Symbol Table available to the user program. A symbol can be a variable, constant, label, or program segment name available in the Symbol Table, a local or conditional variable, or a register name. Symbols have either word or parcel attributes.

The Symbol Table resides in the symbol dataset. The Symbol Table is available to the CAL user if the program is assembled with the SYM option, or to the CFT user if the program is compiled with the ON=Z option. When the Symbol Table is not available, absolute addresses must be used. For more Symbol Table information, see the COS Tables Descriptions Internal Reference Manual, publication SM-0045.

MODULE

Each symbol in the Symbol Table is associated with a module. The default is the first module named in the symbol dataset. The default can be changed with the MODULEBASE directive.

The module name is the name in the IDENT statement for a CAL program or subprogram, or the name of the program, subroutine, function, or common block in a CFT program. The module name for blank common is //.

A specific module can be associated with a single symbol with 'module.symbol'or 'subroutine.commonblock.symbol'.

If the symbol is not found in a routine, all common blocks for the routine are searched.

PARCEL-ADDRESS SYMBOLS

The symbols for entry points, statement labels, and compiler-generated statement labels are *parcel-address symbols*. Entry points in CFT program segments have the same name as the subprogram.

Statement labels from CAL programs must be used as they appear in the program. CFT statement labels must be preceded by the prefix S' to distinguish them from absolute addresses; for example, the label 10 would be referenced as S'10.

The addresses for some CFT statement labels are not saved. Labels in this category include labels for FORMAT statements, labels that are not referenced, and DO-loop terminal statement labels that are not referenced.

CFT LINE NUMBERS

CFT line numbers can be used also as parcel addresses, but only if the line numbers have been saved in the Symbol Table in $DEBUG. Line numbers are saved for executable statements only if the DEBUG option is specified on the CFT control statement. This option enables recognition of DEBUG and NODEBUG compiler directives and turns on the debug mode as compilation begins. Debug mode prevents vectorization and most other optimization.

Line numbers must be preceded by the prefix N' to be used as parcel addresses in SID, so that SID can distinguish them from absolute addresses. Line numbers are saved only for executable statements, not for FORMAT statements, END statements, or declarations. Line number 1 is the same as the entry point.

COMPILER-GENERATED DO-LOOP LABELS

At the beginning and end of a DO-loop the compiler generates statement labels which are saved in the Symbol Table. These labels consist of the original DO-loop terminal statement label with a suffix. Multiple labels are saved if the same terminal statement is shared by more than one DO-loop. A label with the suffix A, C, E, etc. is at the beginning of the loop, and the label with the suffix B, D, F, etc. is at the end of the loop.

A label in the A family is inside the loop and is encountered before each iteration of the loop. In a vectorized loop this label is encountered only once for each 64 elements processed in the loop. A label in the B family is outside the loop and is encountered only once, when execution of the loop is completed (or if the loop is bypassed because the iteration count is less than 1).

In some cases the compiler does not generate additional labels for DO-loops, so they are not available to SID users. Labels are generated only if they are needed for run-time tests. The B family of labels is generated if there will be a run-time test for zero iteration count. For example,

```
        DO 10 1 = 1,N
```

needs the run-time test

```
        IF (N.LT.1) GO TO 10B
```

before the body of the loop, but no such test is needed when the initial and limit values are constant, as with

```
        DO 10 I = 1,100
```

Labels in the A family are generated if relooping might occur. For example,

```
        DO 10 I = 1,N
```

generates the following type of test at the end of the loop body:

```
        I =I+1
        IF (I.LE.N) GO TO 10A
```

Similar code is generated for

```
        DO 10 I = 1,100
```

The A family of labels is not generated for short vector loops. Short vector loops are loops with constant loop controls that include less than 64 elements and that can be executed using vector registers. For example,

```
        DO 10 I = 1,50
        LIST(I) = 0
 50     CONTINUE
```

See appendix B for further examples of compiler-generated DO-loop labels.

USER SYMBOLS

The user program contains user symbols that refer to variables that have been saved in the Symbol Table. When these variables have the same name as special symbols, the module must also be specified so that the user symbol rather than the special symbol is used.

SPECIAL SYMBOLS

Special symbols recognized by SID include Cray register names, local variable names, and conditional variable names. (For more information on the use of local and conditional variables, see Local and Conditional Variables, this section.)

Register names can be actual register names; for example, A5, B06, S0, T76, V3, V614. The symbolic form can also be used; for example, B.ZA where ZA is a constant in the proper range defined by the = pseudo instruction. The words containing the Vector Mask and Vector Length registers are identified by the symbols VM and VL, respectively. The symbol XP stands for the Exchange Package when used with the DISPLAY directive. PC can be used for the current user address program counter.

LOCAL AND CONDITIONAL VARIABLES

SID uses *local variables* X01 through X15 for each debugging session. Variables X01 through X15 are used as counters or to temporarily store values.

X00 is used as a counter to track breakpoints. During initialization, the SID control statement sets up X00 with an integer value. Each time program execution is interrupted by a breakpoint, X00 decrements by 1. When the count reaches 0, the debug package aborts. For a detailed discussion of breakpoints, see section 3.

If the user program has variables with the same names as local variables, references to the user program variables must include a module name or an address.

Conditional variables are used with a breakpoint package. The 15 conditional variables, C01 through C15, are used with the TESTCONDITION directive (see section 4 of this manual) to determine whether parts of the breakpoint package are executed. The user can check the values of the variables with the DISPLAY directive (also section 4).

CONSTANTS

SID uses the following constant types as values in the BREAKPOINT, PACKAGE, and PATCH directives.

- Integer
- Logical
- String
- Floating-point
- Double-precision
- Complex

Integer constants are also used as counts in the STEP, DISPLAY, FORWARD, and BACKWARD directives and can be used in word-address expressions. The default numeric base for integer constants depends on how the integer constant is used and the default set by the BASE directive. Using the appropriate prefix changes the base to decimal (D'), hexadecimal (X'), or octal (O').

Logical constants are expressed as TRUE or FALSE.

A *string constant* is a 1- to 8-character value used in directives as a constant. It is enclosed in single quotes. If fewer than eight characters are used, the string is left-justified with zero fill.

Parcel constants are used in parcel-address expressions in the BREAKPOINT, PACKAGE, INSTRUCTIONS, PATCH, STEP, and RUN directives. A parcel constant is 1- to 8-octal digits followed by one of the letters A, B, C, or D. If eight digits are used, the first must be either 1 or 0.

NOTE

Only the differences in the *syntax* of constants are described here. For a full discussion of constants, see the FORTRAN (CFT) Reference Manual, CRI publication SR-0009.

SID REPRIEVE

SID has a *reprieve* mechanism to recover from abort conditions or to interrupt an executing interactive program. A SID reprieve is reported to the user as breakpoint 0. The abort code AB*nnn* and the approximate

address where the abort occurred are reported. See the CRAY-OS Version 1 Reference Manual, publication SR-0011, for more information on abort codes.

Because the SID reprieve is set before the user program is initiated, a user reprieve functions correctly but disables the SID reprieve.

INPUT

SID receives input from control statement parameters, an input dataset, the Symbol Table for the user program, and alternate input datasets.

SID reads directives from the input dataset entered at the terminal keyboard in interactive mode, and from alternate input datasets, which are used when the user wants to use saved groups of directives. Input records are read before the user program is started and while execution is halted at breakpoints. (See Breakpoints, section 3.) If SID is running in batch mode and runs out of input records, it issues RUN directives until the user program stops executing. If a valid address is unavailable for the RUN directive, SID issues a QUIT directive to stop execution.

SID reads the Symbol Table in the symbol dataset to find memory locations for variables and statement labels whose symbolic names are used. If no Symbol Table is available for the user program, absolute octal addresses must be used for all memory locations.

OUTPUT

If SID is run in interactive mode, output from SID goes to an output dataset displayed at the terminal, and the echo dataset, specified by the ECH parameter on the SID control statement, receives an echo of the input directives. If SID is run in batch mode, the input directives can be sent to the output dataset using ECH.

The LIST directive sends a copy of the output to a temporary dataset called $LST. Input directives are automatically echoed to $LST when LIST is used. Directive NOLIST stops sending output to $LST.

The echo dataset can store input directives that are going to be reread through the ALTERNATEINPUT directive. If one of the directives caused the program to abort, the alternate input dataset must be edited to delete the directive that caused the problem.

CONVENTIONS

The conventions used in this publication to describe statement and directive syntax are the following.

UPPERCASE	Identifies the command verb or literal parameter
UNDERLINED	Specifies the minimum number of characters required for the verb or parameter to be recognized
Italics	Defines generic terms that represent the words or symbols to be supplied by the user
[] Brackets	Enclose optional portions of a command format
... Ellipsis	Indicates optional use of the preceding item one or more times in succession

CONTROL STATEMENTS 2

SID can be used with programs written in CAL, CFT, or a combination of the two. SID is called into execution when the SID parameter in the LDR control statement is encountered, or if the control statement verb is the name of the dataset containing the absolute binary load module of SID and the user program. The control statements used in setting up a program to be debugged with SID are the CAL or CFT control statement (depending on the language used to write the program), the LDR, the SID, and the user program control statements.

LDR CONTROL STATEMENT

The LDR control statement invokes SID. The parameters shown here for the LDR statement are only those normally used with SID. For more information about the LDR control statement, refer to the CRAY-OS Version 1 Reference Manual, publication SR-0011.

Format:

```
LDR[,DN=dn],SID[='string.'][,AB=adn[,NX]].
```

Parameters:

DN=*dn* Dataset names to load with SID; default is $BLD.

SID Loads SID with the datasets specified in the DN parameter

string Any number of characters within single quotation marks. Equated to the keyword SID and contains the parameters to be passed to SID. The parameters in this string follow the standard format for control statement parameters. If *string* is not used, the default values for the parameters in the SID control statement are used.

AB=*adn* Absolute binary object module. This dataset can be used later to invoke SID without reloading.

NX No execution. Inclusion of this parameter inhibits execution of the loaded program.

SID CONTROL STATEMENT

Control statement parameters for SID either are passed from *string* in the LDR control statement or appear in a control statement in which the verb is the name of the dataset containing the absolute binary load module of the user program and SID.

Format:

```
adn,I=idn,S=sdn,L=ldn,ECH=edn,CNT=n.
```

Parameters:

adn Absolute binary load module name as specified with the AB parameter on the LDR statement

I=*idn* Input dataset name containing the directives to be executed. (Input is entered at the terminal keyboard in interactive mode.) The default is $IN. If the input dataset is not $IN, it must be different from the dataset containing the data for the user program to avoid confusion.

S=*sdn* Symbol dataset name. Uses the same format as the $DEBUG dataset produced by LDR. Default is $DEBUG.

L=*ldn* Listing dataset name used for SID's output listing. Default is $OUT.

ECH=*edn* Echo dataset name; receives an echo of the input directives. Default is no echo dataset; ECH or ECH=$ECHO cause the directives to be echoed to the listing dataset.

CNT=*n* Count; an initial decimal value for local variable X00, which is decremented once each time execution is interrupted for a breakpoint. When the count goes to 0, the debug package aborts. Default is 0 (no abort).

USER PROGRAM CONTROL STATEMENT

The control statement for the user program is read from the control statement file after the first RUN or STEP directive is input. A user program control statement must always be provided. Breakpoints can be set before the control statement is read (for information on breakpoints, see section 3).

In an interactive job, a unique prompt appears when the next control statement is expected. In a batch job, the user program control statement must come immediately after the control statement that invokes SID, which is either the LDR statement, if the NX parameter is not used, or a statement whose verb is the absolute binary load module of the user program and SID.

The verb for the user program control statement does not matter, but the statement must follow the proper syntax for control statements and end with a period.

If the control statement for the user program is to be continued on a second line, the first breakpoint must come *after* the call to GETPARAM or its equivalent, since this control statement is processed through a call to the subroutine (such as GETPARAM); SID only reads the first line of the user program control statement. The user program reads the rest of it.

CAL CONTROL STATEMENT

When using SID to debug a CAL program, the user must include the SYM option in the CAL control statement. SYM tells CAL to generate the Symbol Table used by SID. If the CAL program and SID are loaded together and the absolute binary object module is saved, the user must save the symbol dataset $DEBUG generated by LDR, which contains the Symbol Table. Saving $DEBUG allows symbols to be used for statement labels and variable names. If this procedure is not followed, absolute addresses must be used.

CFT CONTROL STATEMENT

When using SID to debug a CFT program, the user must include ON=Z in the CFT control statement, so that the Symbol Table will be written to $BLD.

Sequence numbers for executable statements are saved if the DEBUG option is used on the CFT control statement. This option implies ON=Z; it enables recognition of DEBUG and NODEBUG compiler directives, and turns

on the DEBUG mode when compilation begins. For code compiled in DEBUG mode, MAXBLOCK=1; this turns off vectorization and most optimization.

If SID and the CFT program are loaded together and saved for later use, the user must also save $DEBUG.

EXAMPLES OF SETTING UP PROGRAMS TO BE DEBUGGED

EXAMPLE 1

This example shows how to set up a user program to be debugged. The user program in this example is stored in a program library. It does not use a control statement, although one must be provided for it. (Refer to the previous heading, User Program Control Statement.)

Set up files of the program to be debugged loaded together with SID, and the $DEBUG file for this program, ahead of time. Then they can be accessed from a later batch or interactive job. To set up and save the two datasets:

```
UPDATE,P=MYPL,Q=MYPROG,I=MODS.
CFT,I=$CPL,ON=Z.
LDR,SID,AB=BOTH,NX.
SAVE,DN=BOTH.
SAVE,DN=$DEBUG,PDN=DBBOTH.
EXIT.
```

For interactive debugging of program MYPROG, # is a prompt for SID input and ! is a prompt for a control statement.

```
! ACCESS,DN=BOTH.
! ACCESS,DN=DBBOTH.
! BOTH,S=DBBOTH.                    This statement invokes SID.
# SID directives
# .
# RUN AT MYPROG.MYPROG
! MYPROG.                           Dummy control statement (required)
# more SID directives
```

To debug the same program in a batch job:

```
ACCESS,DN=BOTH.
ACCESS,DN=$DEBUG,PDN=DBBOTH.
BOTH,ECH.                          This statement invokes SID.
*.                                 Dummy control statement (required)
EXIT.
/EOF
SID directives
```

EXAMPLE 2

This example shows how to compile and debug a CFT program in the same job, which can be done in either an interactive or a batch job. Since the NX parameter is specified on the LDR statement, the absolute binary load module must be used to begin execution of SID.

```
CFT,DEBUG,I=SOURCE.
LDR,AB=ABSBIN,NX,SID.
ABSBIN,ECH.                        This statement invokes SID.
XXX,I=TEST.                        Control statement (required).
```

EXAMPLE 3

This example shows how to assemble a CAL program and invoke SID with the LDR statement.

```
CAL,SYM,I=MYSOURCE.
LDR,SID='ECH.'.                    The LDR statement invokes SID.
MYPROG,I=TEST.                     Control statement (required)
```

BREAKPOINT 3

A *breakpoint* is a position in the user program where execution is to be suspended. When a breakpoint is reached, the user can examine values of variables, patch variables with new values, or set and clear breakpoints. The user program can contain a maximum of 15 breakpoints, numbered from 1 to 15. Breakpoint 0 is reserved for abort conditions reprieved by SID.

A breakpoint is implemented by using a 2-parcel jump instruction. SID places breakpoints at any valid parcel address. The user is responsible for verifying that breakpoints occur at instruction boundaries; that is, in the first parcel, not the second parcel, of a 2-parcel instruction. When a breakpoint is reached, a message is sent to the output dataset identifying the breakpoint, or a user-supplied breakpoint routine can be executed.

Breakpoints can be conditional, and can be associated with a breakpoint package.

CONDITIONAL BREAKPOINT

A *conditional breakpoint* specifies suspension of execution under certain conditions. If the condition is false, execution of the user program is resumed with no notification that the breakpoint was reached. If the condition is true, execution of the user program is suspended, and a message is printed that the breakpoint has been reached.

Conditional breakpoints are useful when a variable picks up the wrong value. Statements assigning a value to that variable can be breakpointed with a conditional clause looking for that value.

BREAKPOINT PACKAGE

A *breakpoint package* allows a group of directives to be carried out automatically when a breakpoint is reached. A breakpoint package can be either conditional or unconditional. If it is conditional, the directives in the package are only used if the conditional expression is true. When the directive NOSTOP is used within the package, execution of the program resumes automatically after the directives in the package

have been executed; otherwise SID waits to receive additional directives. Package size varies, depending on the type of directives the package contains.

The directives in the package are executed in sequential order by input. The exception is NOSTOP, which can occur anywhere in the package. The PACKAGE directive opens a package and ENDPACKAGE closes it. If some directives in a package are only needed for a particular condition, TESTCONDITION is used. (NOSTOP is not affected by TESTCONDITION.) If the local condition or conditions specified in this directive are not set, the package is exited early.

The following directives cannot be included in a package:

- ALTERNATEINPUT
- BACKWARD
- BASE
- BREAKPOINT
- FORWARD
- MODULEBASE
- PACKAGE
- REWIND
- RUN
- STEP
- WHERE
- WIDTH

USER-SUPPLIED BREAKPOINT ROUTINE

A *user-supplied breakpoint routine* is a subroutine the user creates which executes when a particular breakpoint is reached. The calling sequence for a breakpoint routine is:

Format:

```
CALL BPSUBn(breakpoint number, nostop request)
```

Parameters:

n 2-digit breakpoint number in octal

breakpoint number
Decimal integer

nostop request
If 1, SID returns to the user program after finishing the breakpoint routine.
If 0, SID requests a new directive.

The alternate breakpoint routine must be separately compiled and loaded so that it is encountered before the default breakpoint routine. It can be used with conditional breakpoints. Package directives are ignored.

RULES FOR USING BREAKPOINTS

The following rules apply mainly to CAL programs.

- Breakpoints must be placed at instruction boundaries, although SID places breakpoints at any valid parcel address.
- If a breakpoint is placed on a 1-parcel instruction, two instructions are relocated into SID. During the time the breakpoint is installed, a jump to the second instruction causes unpredictable results.
- When a breakpoint is removed, two parcels are written back to the user program. If the breakpoint is placed on the jump-to-B00 instruction that ends a routine, a parcel of data can be restored to an earlier value.
- A warning message is displayed if a new breakpoint moves a return jump instruction. When this breakpoint occurs, the return jump is placed in a temporary location that cannot be used by another breakpoint until after the return from the routine. Exiting the breakpoint with the STEP directive corrects the return address.
- A breakpoint placed in an overlay must be installed between the time the overlay is read into memory and the time execution begins. Remove a breakpoint from an overlay so that the previous instruction is restored in either the correct or a harmless overlay.
- For input and output, SID uses sequential datasets supported by routines from $SYSLIB and $FTLIB, which are not designed to be re-entrant. Therefore, breakpoints cannot be placed on instructions used for sequential datasets. Sequential I/O can be stepped through using CAL calls that avoid FTLIB.

The following rules apply only to CFT programs.

- When a program is stopped at some location other than a referenced statement label (for example, when a breakpoint is set using an

absolute parcel address or when the STEP directive is used), the values of some variables may be stored in registers; if they are, the values stored in their memory locations may not be current. This situation can be avoided by using only statement labels for setting breakpoints, and also by not using the STEP directive with CFT programs.

- The DO-loop variable is kept in a register and cannot be examined by displaying its memory location while the loop is executing.

- When a DO-loop vectorizes, the increment for the loop is 64, except for the first iteration of the loop.

SID DIRECTIVES 4

SID recognizes a set of directives for debugging user programs. Directives are executed in the order in which they are input unless they are in a breakpoint package. For interactive use of SID, directives are entered at an interactive terminal in response to prompts and program output. In batch mode, directives usually reside in a file of the input dataset ($IN) but can be in a separate dataset.

A SID directive consists of a keyword, prepositions, and parameters. A semicolon or the end of a line terminates a directive, so more than one directive can be placed on an input line. When a directive contains an error, all characters up to the next semicolon or end of line are ignored. Extra blanks are ignored.

When SID is ready to accept a directive, it prompts with # (pound sign). After the prompt, the user enters the directive in mixed case followed by a carriage return. # followed by a period indicates a comment line. SID then displays its response in uppercase and lowercase.

In the following directive examples, TEST is the user program.

SYNTAX

A directive has the following format:

$$k\ prep_1\ p_1\ prep_2\ p_2\ prep_n\ p_n$$

Parameters:

k	Keyword
$prep_i$	Prepositions (for example, FOR, AT, IN); not all directives use prepositions, and many prepositions are optional.
p_i	Parameters dependent on directive. The order of input for the parameters cannot be changed.

The underlined portions of the directives specify the minimum number of characters required for a keyword or parameter to be recognized.

ACTIVE DIRECTIVE

The ACTIVE directive displays a list of all current breakpoints and breakpoint packages.

Format:

```
ACTIVE
```

Example:

```
# ACTIVE
    Breakpoint 1 at 120 in TEST
    Breakpoint 2 at 220 in ADD
         if SUM in TEST > 12
    Package 4 at 140 in TEST
    Breakpoint 13 at 310 in SUBTRACT
         if VAL1 in SUBTRACT = 0.15600000000000E+02
```

ALTERNATEINPUT DIRECTIVE

The ALTERNATEINPUT directive allows groups of directives to be read from a dataset other than the input dataset. The ALTERNATEINPUT directive specifies an input dataset from which to read directives after the current record has been processed. The new dataset is read until an EOF is reached, when the input is again read from the original input dataset. An alternate input dataset cannot contain another ALTERNATEINPUT directive.

Format:

```
ALTERNATEINPUT dn [NOREWIND]
```

Parameters:

dn Name of the alternate input dataset

NOREWIND Specifies that the alternate input dataset should not be rewound before it is read

BACKWARD DIRECTIVE

The BACKWARD directive scrolls the display backward from the last display. The format of the previous display is used, and the default for the display count is the count for the previous display.

Format:

```
BACKWARD [[FOR] count]
```

Parameter:

count Number of words to be displayed; default is the count for the previous DISPLAY, FORWARD, or BACKWARD directive.

Examples:

```
# DISPLAY LIST(7) FOR 3
    Memory display in decimal at LIST(7) in TEST
00000225                          -4
00000226                          94
00000227                           5
# .
# BACKWARD
    Memory display in decimal at LIST(4) in TEST
00000222                          93
00000223                           0
00000224                         -45
# .
# BACKWARD 2
    Memory display in decimal at LIST(2) in TEST
00000220                         -12
00000221                          56
```

BASE DIRECTIVE

The BASE directive changes the default numeric base used for converting numbers input to SID. The default base is mixed.

Format:

```
BASE newbase
```

Parameter:

newbase New default numeric base. Permitted values are OCTAL, DECIMAL, or MIXED.

Constant prefixes, O', D', and X', override the default base.

In mixed mode, the values for octal and decimal are assigned as follows:

Octal	Decimal
Word addresses	Word address offsets
Parcel addresses	Counts for display directives and the STEP directive
Parcel address offsets	Bit positions and counts
Parcel values	Word values for breakpoint tests or word patches
Bit field values	Breakpoint numbers
	Local and conditional variable numbers

BREAKPOINT DIRECTIVE

The BREAKPOINT directive installs or removes a breakpoint in the user program.

Format to install a breakpoint:

```
BREAKPOINT n [[AT] parcel address [IF word address rel value]]
```

Format to remove a breakpoint:

```
BREAKPOINT n
```

Parameters:

n A breakpoint number between 1 and 15

parcel address
Location of the instruction where execution is interrupted

word address
Location of a value to be tested; can be a register or a local variable.

rel Relation tested for is one of the following:

=
<
>
<=
>=
<>

NOTE

These symbols must be entered in the format shown to be recognized.

value Constant to be compared to the value in *word address*; can be signed integer, floating-point, logical, double precision, complex, or a string. The base is decimal in mixed mode.

Examples:

```
# BREAKPOINT 1 AT S'120
    Breakpoint 01 installed
# BR 2 ADD.S'220 IF SUM >12
    Breakpoint 02 installed
# BREA 15 SUBTRACT.S'310 IF SUBTRACT.VAL1=15.6
    Breakpoint 15 installed
# BR 5 S'140
    Breakpoint 05 installed
# BR 5
    Breakpoint 05 removed
# .
# BR 2 at CHECK12+0C
    **** WARNING:  existing breakpoint replaced
    Breakpoint 02 installed
```

COMMENT DIRECTIVE

A comment directive is copied to the echo dataset (see section 1, Output). It can be at the beginning of an input record or following a semicolon.

Format:

```
. [comment]
```

Parameter:

comment Optional documentation

DECREMENTVARIABLE DIRECTIVE

The DECREMENTVARIABLE directive subtracts 1 from one or more local variables. C_i is set or becomes true when X_i is decremented to 0.

Format:

```
DECREMENTVARIABLE Xn1 [Xn2 ... Xnn]
```

Parameter:

Xn_i Local variables where n is a 2-digit integer

DISPLAY DIRECTIVE

The DISPLAY directive displays the contents of one or more memory locations, registers, or local variables, in any of several formats. DISPLAY C00 displays the conditional variables (C00-C15), numbered from left to right, where 0 is off and 1 is on. XP stands for Exchange Package when used with this directive.

Format for a full word display:

```
DISPLAY [AT] word address [[IN] format [AND format]] [[FOR] count]
```

Parameters:

format Display format; can be OCTAL, DECIMAL, HEXADECIMAL, FLOATING, DOUBLE, COMPLEX, ASCII, PARCEL, ADDRESS, and LOGICAL. ADDRESS displays the value as a parcel address. The default is OCTAL for CAL programs or the type of the variable being displayed for CFT programs.

Using AND with *format* causes a dual display. For example, a word can be displayed in octal with its ASCII equivalent next to it.

word address
First location to be displayed; can be XP for the Exchange Package (A and S registers plus the P, B00, VM and VL registers), a register name, or a constant. Using an * in this field causes the directive to use the word address of the last DISPLAY or PATCH directive.

count Number of words to be displayed. Default is 1, except for vector registers where the default is the vector length.

Format for a masked display:

```
DISPLAY [AT] word address BIT firstbit COUNT bitcount [[IN] format]
```

Parameters:

word address
: Location to be displayed. Using an * in this field causes the directive to use the word address of the last DISPLAY or PATCH directive.

firstbit
: Position of the first bit in the field to be displayed; the default base is decimal.

bitcount
: Decimal size of the field to be displayed

format
: Display format; can be OCTAL, DECIMAL, or SIGNEDDECIMAL; with signed decimal, the first bit of the field is used as a sign bit. The default is octal.

Examples:

```
# DISPLAY CVAL; DISPLAY CVAL IN OCTAL FOR 2
    Memory display in complex at CVAL in TEST
00000235     0.45000000000000E+01,0.34000000000000E+08
    Memory display in octal at CVAL in TEST
00000235     0400034400000000000000
00000236     0400324033144000000000
# .
# DISPLAY X
    Memory display in floating pt at X in TEST
00000232     0.45700000000000E+02
# .
# DISPLAY LVAL
    Memory display in logical at LVAL in TEST
00000241     TRUE
# .
# DISPLAY K
    Memory display in decimal at K in TEST
00000216                          41
# .
# DISPLAY K IN OCTAL
    Memory display in octal at K in TEST
00000216     000000000000000000051
# .
```

Example (continued):

```
# DISPLAY LIST FOR 10
    Memory display in decimal at LIST(1) in TEST
00000217                        234
00000220                        -12
00000221                         56
00000222                         93
00000223                          0
00000224                        -45
00000225                         -4
00000226                         94
00000227                          5
00000230                         63
# .
# DISPLAY LIST
    Memory display in decimal at LIST(1) in TEST
00000217                        234
# D *+2
    Memory display in decimal at LIST(3) in TEST
00000221                         56
# D * O
    Memory display in octal at LIST(3) in TEST
00000221     0000000000000000000070
# .
# DISPLAY XP
    Exchange package display
     P register:              00000267c     Vector length:  00000100
     Return addr (B00):       00000267a     Vector mask:    0000000000000
     A0 = 00000342   S0 = 0400065554631463146315              @.6LLLLM
     A1 = 00000216   S1 = 00000000000000000000               ........
     A2 = 00000215   S2 = 00000000000000000042               .......B
     A3 = 00000214   S3 = 0400076103737166621320              @.D>y["P
     A4 = 00000215   S4 = 00000000000000000051               .......)
     A5 = 00000214   S5 = 0400007610000000000000             @.D.....
     A6 = 00000000   S6 = 0377757676355442641626             ?)(gl.C.
     A7 = 00000210   S7 = 00000000000000000007              ........
# .
# DISPLAY B00 ADDRESSS
    B - registers display as parcel address at B00
00000000                        00000267a
# .
# DISPLAY A7
    A - registers display at A7
00000007   00000210
# DISPLAY (A7)
    Memory (AX) display at (A7)
00000210   0000000000000000000025
# DISPLAY (A7)+2
    Memory (AX) display at (A7)+2
00000212   0000000000000000000015
```

Example (continued):

```
# DISPLAY A7
    A - registers display at A7
00000007   00000210
# DISPLAY (*)+2
    Memory (AX) display at (A7)+2
00000212   00000000000000000015
```

ENDPACKAGE DIRECTIVE

The ENDPACKAGE directive closes a breakpoint package.

Format:

```
ENDPACKAGE
```

FORWARD DIRECTIVE

The FORWARD directive scrolls the display forward from the last address displayed for the number of words specified. The default count is the last count used. The format of the previous display is used.

Format:

```
FORWARD [[FOR] count]
```

Parameter:

count Number of words to be displayed. The default is the count for the previous DISPLAY, FORWARD, or BACKWARD directive.

Examples:

```
# DISPLAY LIST(6)
    Memory display in decimal at LIST(6) in TEST
00000224                    -45
```

Example (continued):

```
# FORWARD 3
    Memory display in decimal at LIST(7) in TEST
00000225                          -4
00000226                          94
00000227                           5
# .
# F 1
    Memory display in decimal at LIST(10) in TEST
00000230                          63
```

HELP DIRECTIVE

The HELP directive provides information about SID while the program is in use.

Format:

```
HELP [topic]
```

Parameter:

topic Name of the directive or term for which information is desired. Specifying HELP without a topic name causes a menu of HELP topics to be displayed.

HELP information is not echoed to $LST.

INCREMENTVARIABLE DIRECTIVE

The INCREMENTVARIABLE directive adds 1 to one or more local variables.

Format:

```
INCREMENTVARIABLE Xn1[Xn2...Xnn]
```

Parameter:

Xn_i	Local variable where n_i is a 2-digit integer

INSTRUCTIONS DIRECTIVE

The INSTRUCTIONS directive displays CAL instructions.

Format:

```
INSTRUCTIONS [AT] parcel address [FOR count]
```

Parameters:

parcel address	Location of the first CAL instruction to be displayed
count	Number of instructions to be displayed

Examples:

```
# INSTRUCTIONS AT TEST FOR 5
    Instruction display at TEST in TEST
 00000200a   A1  210,0
 00000200c   A2  211,0
 00000201a   A3  212,0
 00000201c   A0  A2-A1
 00000201d   JAP  203a
# .
# . PC is the symbol for the user program P register, or program counter
# .
# I PC
    Instruction display at 201c
 00000201c   A0  A2-A1
```

LIST DIRECTIVE

The LIST directive begins sending an echo of the output from SID to $LST as well as to the output dataset.

Format:

```
LIST
```

LOAD DIRECTIVE

The LOAD directive loads a local variable with a value from a register or memory. Both LOAD and STORE are useful for keeping values of user-program variables. For example, to see what happens if variable Z has a value of -584.01859 instead of -584.01858, load the old value, -584.01859, into a local variable so it can be stored again later.

Format:

```
LOAD Xn FROM word address
```

Xn — A local variable where *n* is a 2-digit integer

word address — Memory location to be copied to Xn; can be a register.

Example:

```
# DISPLAY X01;DISPLAY Y OCTAL
    X-variables display at X01
00000001      0000000000000000000000
    Memory display in octal at Y in TEST
00000233      0400046263146314631463
# LOAD X01 FROM Y
# DISPLAY X01;DISPLAY Y OCTAL
    X-variables display at X01
00000001      0400046263146314631463
    Memory display in octal at Y in TEST
00000233      0400046263146314631463
```

MODULEBASE DIRECTIVE

The MODULEBASE directive specifies a default module for looking up symbols.

Format:

```
MODULEBASE modulename
```

Parameter:

modulename
Name of the new default module which is the name of a program, subprogram, or common block in a CFT program, or the name in the IDENT statement in a CAL routine. A particular occurrence of a common block can be specified with 'subprogram.commonblock'.

Examples:

```
# DISPLAY I
    Memory display in decimal at I in TEST
00000225                          36
# MODULE BLOCK1
    Module BLOCK1 begins at word 200
# DISPLAY M
    Memory display in decimal at M in BLOCK1
00000201                           6
# MODULE ADD.BLOCK1
    Module ADD begins at word 260
    Module BLOCK1 begins at word 200
# DISPLAY Y
    Memory display in decimal at Y in BLOCK1
00000201                           6
```

NOLIST DIRECTIVE

The NOLIST directive stops output from going to $LST.

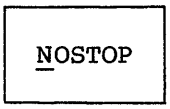
Format:

```
NOLIST
```

NOSTOP DIRECTIVE

A breakpoint package uses the directive NOSTOP to have SID continue with the program being tested after the directives in the package have been executed. NOSTOP can be placed anywhere in a package and is not affected by a TESTCONDITION directive.

Format:



Example:

```
# PACKAGE 6 AT S'120
    Breakpoint 06 installed
# DISPLAY SUM
# NOSTOP
# ENDPACKAGE
# .
# BREAKPOINT 7 AT S'220
    Breakpoint 07 installed
# .
# RUN AT TEST
#   Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                          0
    Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                         20
    Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                         25
    Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                         25
    Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                         31
    Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                         31
    Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                         40
    Breakpoint 6 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207                         52
    Breakpoint 7 at 220 in TEST
```

PACKAGE DIRECTIVE

The PACKAGE directive opens a breakpoint package. Packages can be conditional.

Format:

```
PACKAGE n [AT] parcel address [IF word address rel value]
```

Parameters:

n Breakpoint number between 1 and 15

parcel address
Location of the instruction where execution is interrupted

word address
Location of a value to be tested; can be a register or a local variable.

rel Relation tested for is one of the following:

=
<
>
<=
>=
<>

NOTE

These symbols must be entered in the format shown in order to be recognized.

value Constant to be compared to the value in *word address*; can be signed integer, floating-point, logical, double precision, complex, or string; default base is decimal when in mixed-base mode.

Example:

```
# PACKAGE 4 at S'140
    Breakpoint 04 installed
# DISPLAY LIST FOR 6
# DISPLAY DVAL; DISP DVAL IN OCTAL FOR 2
# ENDPACKAGE
# RUN AT TEST
    Breakpoint 4 at 140 in TEST
    Memory display in decimal at LIST in TEST
00000217                        234
00000220                        -12
00000221                         56
00000222                         93
00000223                          0
00000223                        -45
    Memory display in double prec. at DVAL in TEST
00000237      0.99999999999999999999999999995D-01
    Memory display in octal at DVAL in TEST
00000237      0377756314631463146314
00000240      0000006314631463146314
```

PATCH DIRECTIVE

The PATCH directive enters new values into memory, registers, or local variables. It can patch values into a full word, a bit field, or a parcel.

Format for a full-word patch:

PATCH [AT] *word address* WITH $value_1$ [$value_2$...$value_n$]

Parameters:

word address
: First location to receive a new value. Using an * in this field causes the directive to use the word address of the last DISPLAY or PATCH directive.

$value_i$
: Values to be stored. Values are integer, floating-point, double precision, complex, logical, or strings. Default base is decimal when in mixed mode, but integer values can be given in octal or hexadecimal. The values are stored in successive memory locations.

Format for a bit-field patch:

```
PATCH [AT] word address BIT firstbit COUNT bitcount WITH value
```

Parameters:

word address
: Location of the word to be patched. Using an * in this field causes the directive to use the word address of the last DISPLAY or PATCH directive.

firstbit
: Position of the first bit in the field to be patched; default base is decimal when in mixed mode.

bitcount
: Decimal size of the field to be patched

value
: Value to be stored. The default is octal and must be an unsigned integer. This value can be changed to decimal or hexadecimal by using the prefixes D' or X', respectively.

Format for a parcel patch:

```
PATCH PARCEL [AT] parcel address WITH value1 [value2...valuen]
```

Parameters:

parcel address
: Location of the first parcel to receive a new value

$value_n$
: Value to be stored. The default is octal and must be an unsigned integer. This value can be changed to decimal or hexadecimal by using the prefixes D' or X', respectively.

Example of a full-word patch with a floating-point value:

```
# DISPLAY Y
    Memory display in floating pt at Y in TEST
00000233      0.52000000000000E+01
# PATCH Y WITH 12.7
# DISPLAY Y
    Memory display in floating pt at Y in TEST
00000233      0.12700000000000E+02
```

Example of a word patch with negative integer value using the special symbol *:

```
# DISPLAY LIST(4)
    Memory display in decimal at LIST(4) in TEST
00000222                          93
# PATCH * WITH -924
# D *
    Memory display in decimal at LIST(4) in TEST
00000222                         -924
```

Example of a bit-field patch of an X-variable:

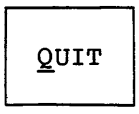
```
# .
# DISPLAY X03
    X-variables display at X03
00000003      00000000000000000000
# PATCH X03 BIT 10 COUNT 6 WITH 12
# DISPLAY X03
    X-variables display at X03
00000003      00001200000000000000
```

Example of a parcel patch:

```
# WHERE TEST
    Symbol TEST in TEST is at parcel 242a
# D 242 IN PARCEL
    Memory display in parcels
00000242      110700 000301 020000 000303
# PATCH PARCEL AT TEST+0B WITH 000302
# D 242 PARCEL
    Memory display in parcels
00000242      110700 000302 020000 000303
```

QUIT DIRECTIVE

The QUIT directive terminates a debugging session without running the entire user program.



REPEAT DIRECTIVE

The REPEAT directive repeats directives that are in the previous input record. A blank line followed by a carriage return is the format for this directive. If the last record from the input dataset was an ALTERNATEINPUT directive, the repeated record is the last record from the alternate input dataset. A record can be repeated more than once. For example, to repeat a line three times, enter three blank lines.

RESETCONDITION DIRECTIVE

The RESETCONDITION directive deactivates conditional variables.

Format:

RESETCONDITION Cn_1 [$Cn_2 \ldots Cn_n$]

Parameter:

Cn_i	Conditional variable to be deactivated, where n_i is a 2-digit integer

REWIND DIRECTIVE

The REWIND directive is used to rewind one or more local datasets. The specified datasets are positioned at their initial points. If a dataset is already at its initial point, this directive has no effect on that dataset.

Format:

REWIND dn_1 [$dn_2 \ldots dn_n$]

Parameter:

dn_i	Name of dataset to be rewound

RUN DIRECTIVE

The RUN directive executes the user program to the next breakpoint or to the end of the program if no breakpoint is encountered.

Format:

```
RUN [[AT] parcel address]
```

Parameter:

parcel address
: Address where execution begins; normally present only for the first RUN or STEP directive to specify the start of the program. If no parcel address is given, execution resumes at the current location.

Example:

```
# BREAKPOINT 1 AT S'110
    Breakpoint 01 installed
# BREAKPOINT 2 AT S'120
    Breakpoint 02 installed
# RUN AT TEST
    Breakpoint 1 at 110 in TEST
# RUN
    Breakpoint 2 at 120 in TEST
```

SETCONDITION DIRECTIVE

The SETCONDITION directive activates conditional variables.

Format:

```
SETCONDITION Cn1 [Cn2...Cnn]
```

Parameter:

Cn_i
: Conditional variables to be activated, where n_i is a 2-digit integer

Conditions are also affected by DECREMENTVARIABLE. If variable Xn_i becomes 0 as a result of the decrement directive, condition Cn_i is activated.

STEP DIRECTIVE

The STEP directive causes a specified number of CAL instructions in the user program to be executed.

Format:

```
STEP [AT parcel address] [[FOR] count]
```

Parameters:

parcel address
: Address where execution begins; normally present only for the first RUN or STEP directive to specify the start of the program. If no parcel address is used, program execution resumes at the current location.

count
: Number of CAL instructions to execute. Default is 1.

****

CAUTION

Avoid using STEP while executing $FTLIB. Sequential I/O can be stepped through using CAL calls that avoid FTLIB. Using STEP in $SYSLIB can tie the traceback information in a loop.

****

Examples:

```
# STEP AT TEST
    STEP command ended at P=00000200c
# STEP FOR 5;DISPLAY A1 IN DECIMAL FOR 3
    STEP command ended at P=00000203b
```

Example (continued):

```
    A-registers display in decimal at A1
00000001                        21
00000002                        32
00000003                        13

# STEP 5;D * 3
    STEP command ended at P=00000201c
    A - registers display in decimal at A1
00000001                        21
00000002                        13
00000003                        32
# STEP;D * 3
    STEP command ended at P=00000201d
    A - registers display in decimal at A1

00000001                        21
00000002                        13
00000003                        32
```

STORE DIRECTIVE

The STORE directive stores the value from a local variable in a specified memory location or register. Both LOAD and STORE are useful for keeping values of user-program variables. For example, to see what happens if variable Z has a value of -584.01859 instead of -584.01858, load the old value, -584.01859, into a local variable so it can be stored again later.

Format:

STORE X*n* AT *word address*

Parameters:

X*n* — A local variable where *n* is a 2-digit integer

word address — Memory location into which the value from X*n* is to be stored

Example:

```
# DISPLAY X01;DISPLAY Z OCTAL
    X-variables display at X01
00000001 040004414631463146
    Memory display in octal at Z in TEST
00000234 000000000000000000000
# STORE X01 AT Z
# DISPLAY X01;DISPLAY Z OCTAL
    X-variables display at X01
00000001 04000441463146314631463146
    Memory display in octal at Z in TEST
00000234 04000441463146314631463146
```

TESTCONDITION DIRECTIVE

The TESTCONDITION directive is used to exit a breakpoint package if one of the conditional variables being tested is activated.

Format:

TESTCONDITION Cn_1 [Cn_2...Cn_n]

Parameter:

Cn_i	Conditional variable to be tested, where n_i is a 2-digit integer

Example:

To display the variables SUM and I every fourth time a loop is executed, use the following package (S'210 is a label in the loop):

```
# PATCH X03 WITH 4; RESET C03
# PACKAGE 4 AT S'210
    DECREMENT X03
    NOSTOP
    TESTCONDITION C03
    DISPLAY SUM
    DISPLAY I
```

Example (continued):

```
    PATCH X03 WITH 4
    RESET C03
    ENDPACKAGE
# .
# BREAKPOINT 5 AT S'220
# .
# RUN AT TEST
    Breakpoint 4 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207       25
    Memory display in decimal at I in TEST
00000205        4
    Breakpoint 4 at 210 in TEST
    Memory display in decimal at SUM in TEST
00000207       52
    Memory display in decimal at I in TEST
00000205        8
    Breakpoint 5 at 220 in TEST
```

TRACEBACK DIRECTIVE

The TRACEBACK directive reports the calling tree from the current program counter for the user program back to the user's main program.

Format:

```
TRACEBACK
```

Example:

```
# TRACEBACK
    Routine SUB3 stopped at P=346a
    SUB3 was called by SUB2 at line 6, P=317c
    SUB2 was called by SUB1 at line 7, P=265c
    SUB1 was called by TEST at line 12, P=220a
```

WHERE DIRECTIVE

The WHERE directive returns the address for a variable or label or information about why its value cannot be displayed. It also returns the symbol associated with an address or the closest symbol whose address is less than the address given.

Format:

```
WHERE loc1 [loc2...locn]
```

Parameter:

loc$_i$ Variable name, label, or line number used in the program or word or parcel address. Names or labels can be preceded by module names; for example, modulename.symbol.

Example:

```
# WHERE TEST.S'120 ADD.VAL1
    Module TEST begins at word 200
    Symbol 120 in TEST is at parcel 263b
    Symbol ADD in ADD is at parcel 326a
    Symbol VAL1 in ADD is a dummy argument whose current actual argument
    is at word 214
# WHERE 220
    Symbol LIST in TEST is at word 217
```

WIDTH DIRECTIVE

The WIDTH directive changes the display width. The default width is one column except for the Exchange Package. Other possible widths are 80 characters or 132 characters.

Format:

```
WIDTH newwidth
```

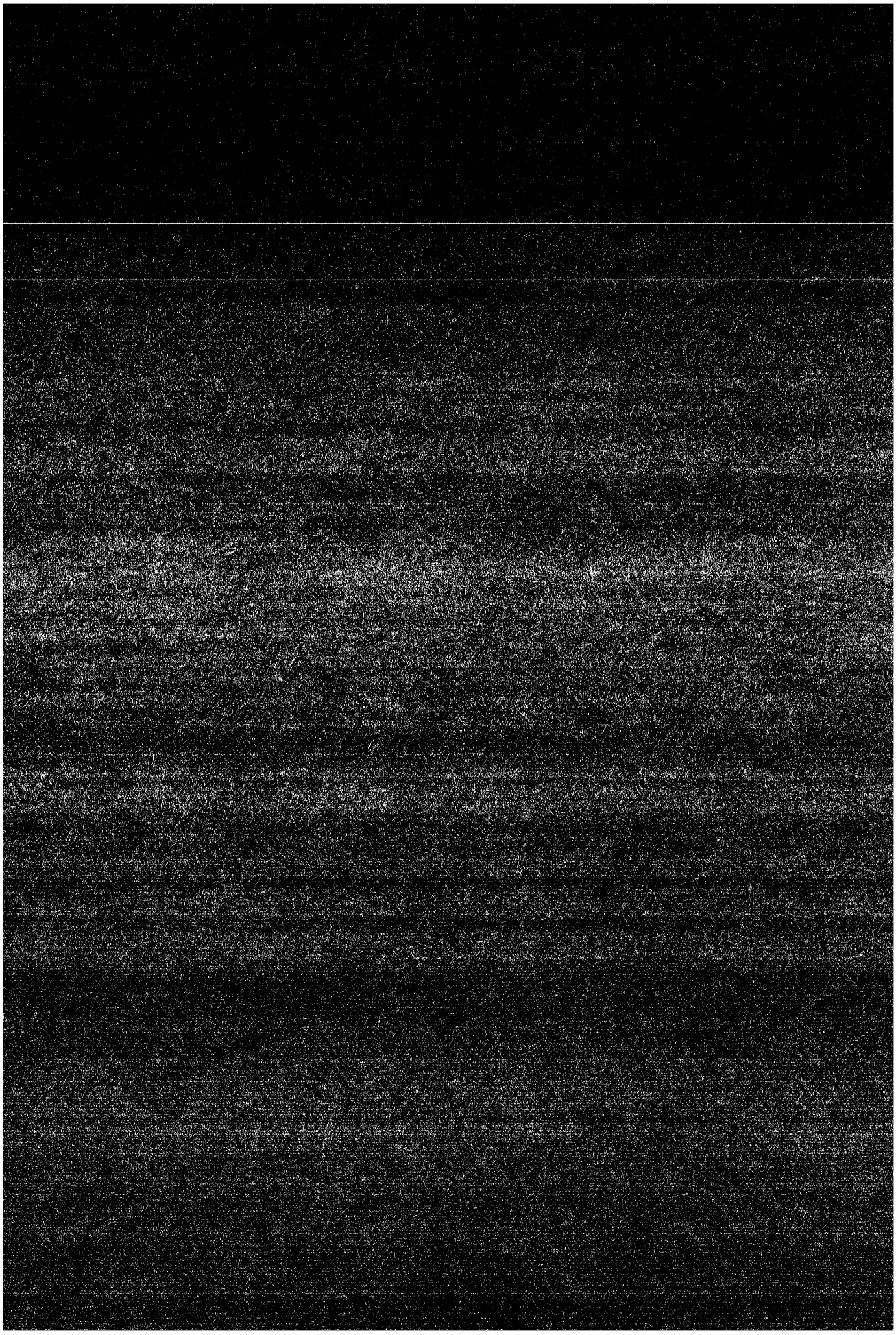
Parameter:

newwidth New width for the display; can be COLUMN, SCREEN, or PAGE.

Example:

```
# DISPLAY 200 IN PARCEL FOR 6
    Memory display in parcels
00000200  100100 000210 100200 000211
00000201  100300 000212 006000 002024
00000202  001014 030701 030102 030207
00000203  031032 012000 001024 030702
00000204  030203 030307 006000 001006
00000205  110100 000213 110200 000214
# DISPLAY X FOR 3
    Memory display at X in TEST
00000210  000000000000000000025
00000211  000000000000000000040
00000212  000000000000000000015
# DISPLAY ANS IN ASCII FOR 6
    Memory display in ASCII at ANS in TEST
00000213 ONE.....
00000214 TWO.....
00000215 THREE...
0000216  FOUR....
00000217 FIVE....
00000220 SIX.....
# .
# WIDTH SCREEN
# DISPLAY 200 IN PARCEL FOR 6
    Memory display in parcels
00000200  100100 000210 100200 000211 100300 000212 006000 002024
00000202  030203 030307 006000 001006 110100 000213 110200 000214
# DISPLAY X FOR 3
    Memory display at X in TEST
00000210  000000000000000000025 000000000000000000040 000000000000000000015
# DISPLAY ANS IN ASCII FOR 6
    Memory display in ASCII at ANS in TEST
00000213  ONE..... TWO..... THREE... FOUR.... FIVE.... SIX.....
```

APPENDIX SECTION



SID MESSAGES A

SID produces uncoded and coded messages. Uncoded messages either inform the user of an error or warn the user of problems if the present action continues. Uncoded messages point to the character that is causing the error. A coded message beginning with DB, means SID has aborted and is not reprievable. Coded messages go to the logfile and uncoded messages go to the output dataset.

UNCODED MESSAGES

**** ERROR: a file name is required
The directive ALTERNATEINPUT was used without a file name.

**** ERROR: a parcel address is required
An INSTRUCTION directive did not contain a parcel address or the special symbol PC.

**** ERROR: a period must be followed by a symbol
A symbol was followed by a period and another character, causing SID to treat the first symbol as a module name and to expect another symbol following the period. A period followed by a space always terminates the directive.

**** ERROR: a relation is required in an IF clause
The word address in the conditional clause of a BREAKPOINT or PACKAGE directive was not followed by a legal test relation (one of =, <>. <=, >, or >=). To be recognized, a test relation must be entered exactly as it is shown in this manual.

**** ERROR: a symbol or address is required
The directive name WHERE was followed by something other than a symbol, a word address, or a parcel address.

**** ERROR: address is out of range
A word address is outside of the user address space (larger than JCFL).

**** ERROR: address is too large
An address was more than 24 bits.

**** ERROR: alternate input file is not available
The dataset used in an ALTERNATEINPUT directive is not local to the job.

**** ERROR: bad syntax in complex number
A complex number used the wrong format.

**** ERROR: bad syntax in number
An illegal character was encountered while parsing a number.

**** ERROR: base must be OCTAL, DECIMAL, or MIXED
Something other than OCTAL, DECIMAL, or MIXED or their abbreviations followed the keyword BASE.

**** ERROR: bit count is too large
The bit count in a masked PATCH directive did not fit into the remainder of the word because the starting bit number plus the bit count was greater than 64.

**** ERROR: bit number is too large
The starting bit for a masked PATCH or masked DISPLAY directive is greater than 63 decimal.

**** ERROR: breakpoint address refused
A SID logic problem caused the breakpoint address to be rejected.

**** ERROR: breakpoint number needed
A BREAKPOINT or PACKAGE directive was missing a breakpoint number. This number, between 1 and 15, must follow the directive keyword.

**** ERROR: can't be used in package and one is open
A directive that cannot be used in a breakpoint package was entered while a package was still open.

**** ERROR: common block table is missing
The symbol table record for a common block is missing, probably because the subroutine was not compiled or assembled with the necessary option.

**** ERROR: condition refused
A SID logic failure caused the breakpoint condition to be refused.

**** ERROR: count must be numeric
One of the directives FORWARD or BACKWARD was followed by a non-numeric input token when a count was expected.

**** ERROR: count must be unsigned integer
The bit count for a masked PATCH or masked DISPLAY directive was not an unsigned integer constant as expected.

**** ERROR: count not allowed for masked display
A masked display directive included a display count, but only one value at a time can be shown with a masked display.

**** ERROR: directive must begin with keyword
The first token of an input record or the first token following a semicolon was not a SID keyword; the directive name could have been misspelled.

**** ERROR: directive must end with ; or end of line
An extra token was found at the end of a directive.

**** ERROR: dual format not allowed for masked display
The keyword AND followed the display format in a masked DISPLAY directive, but masked displays use only one display format.

**** ERROR: empty parentheses
Empty parentheses were used in a word-address expression.

**** ERROR: exponent overflow
The exponent for a floating-point value was too large.

**** ERROR: exponent underflow
The exponent for a floating-point value was too small.

**** ERROR: fewer subscripts than declared
An array reference contained fewer subscripts than dimensions declared in the array.

**** ERROR: first RUN requires a parcel address
A starting address must be given the first time the user code is entered with a RUN or STEP directive. The starting address is usually the name of the program.

**** ERROR: first STEP requires a parcel address
A starting address must be given the first time the user code is entered with a RUN or STEP directive. The starting address is usually the name of the program.

**** ERROR: help file is not available
The file $DBHELP, which contains the HELP text, is not local to the job and is not available on the system directory.

**** ERROR: HELP topic tables are too small
The tables that hold the directory for the help files are not large enough to hold information about all of the HELP topics.

**** ERROR: indirect address is too large
An indirect address is outside the user's address area.

**** ERROR: input token is not a symbol
A SID logic error caused a routine for looking up symbols in the symbol dataset to be called with a nonsymbol argument.

**** ERROR: invalid hexadecimal digit
A value preceded by the prefix X' contained an invalid hexadecimal digit and a hexadecimal value was expected.

**** ERROR: invalid octal digit
An invalid character was detected in an octal number.

**** ERROR: invalid parcel specifier
A character other than A, B, C, or D followed an octal address when a parcel specifier was expected.

**** ERROR: invalid relation
The IF clause in a BREAKPOINT or PACKAGE directive used a relation other than =, <>, <, <=, >, or >=. To be recognized, a test relation must be entered exactly as it is shown in this manual.

**** ERROR: keyword AND requires a format
The keyword AND in a DISPLAY directive was followed by something other than a valid format specifier.

**** ERROR: keyword AT requires a parcel address
The keyword AT in a RUN or STEP directive was not followed by a valid parcel-address expression.

**** ERROR: keyword FOR requires a count
The keyword FOR in a DISPLAY, FORWARD, BACKWARD or INSTRUCTION directive was not followed by an unsigned integer.

**** ERROR: keyword FOR requires a step count
The keyword FOR in a STEP directive was not followed by an unsigned integer.

**** ERROR: keyword IN requires a format
The keyword IN in a DISPLAY directive was not followed by a recognized format name.

**** ERROR: label not saved; see HELP LABELS
The address for a statement label was not saved. LABELS, a HELP topic, lists several possible reasons for labels not being saved.

**** ERROR: LOAD requires an X-variable
The keyword LOAD was not followed by an X-variable name.

**** ERROR: maximum string length is 8 characters
A literal string used as a value had more than eight characters.

**** ERROR: memory address is missing
No address was given when a word-address expression was expected.

**** ERROR: module name must be a symbol
The keyword MODULEBASE was not followed by a symbol.

**** ERROR: module not found
The module name preceding a variable name or label was not found in the Debug Map Table in $DEBUG.

**** ERROR: module with this address not found
An address used in a WHERE directive was outside the range of subroutines and common blocks listed in the Debug Map Table in $DEBUG.

**** ERROR: must be used in package and none is open
A directive that can only be used in a breakpoint package was entered when no package was open.

**** ERROR: need constant to specify a register
A register was specified with a symbol other than a constant, for example, 'B.ZA' where ZA was not defined with the '=' pseudo-op.

**** ERROR: nesting of ALTERNATEINPUT is not implemented
An ALTERNATEINPUT directive was used in an alternate input file.

**** ERROR: no module has been specified
There is no current default module; this message should occur only following a MODULEBASE directive with an error.

**** ERROR: not a normal symbol
Only symbols classified as constant or normal in the Symbol Table can be processed by SID currently.

**** ERROR: null field
A value decoded by $NICV had a null field; probably from a logic problem in SID.

**** ERROR: number is too large
A number in a SID directive takes more than 64 bits.

**** ERROR: number must be between 00 and 15 decimal
The number for a local X or C variable was not in the correct range. For mixed-base mode (the default), this number is decimal and must be between 00 and 15.

**** ERROR: number must be between 00 and 17 octal
The number for a local X or C variable was not in the correct range. In octal-base mode this number is octal and must be between 0 and 17.

**** ERROR: number must be between 1 and 15 decimal
A breakpoint or package number was not in the correct range. For mixed-base mode (the default) or decimal mode, this number is decimal and must be between 1 and 15.

**** ERROR: number must be between 1 and 17 octal
A breakpoint or package number was not in the correct range. For octal-base mode, this number is octal and must be between 1 and 17.

**** ERROR: package contains undecodeable command
A SID logic failure placed a command that SID cannot decode in a breakpoint package.

**** ERROR: package overflow
The last directive entered is ignored and the package is closed.

**** ERROR: parcel address is missing
No parcel address was given when a parcel-address expression was expected.

**** ERROR: parcel address is out of range
A parcel-address expression is out of the user's address area.

**** ERROR: parcel value must fit in 16 bits
The value given in a PATCH PARCEL directive was too large to fit into 16 bits.

**** ERROR: PATCH list is too long
The list of values used in a PATCH directive is longer than the current limit.

**** ERROR: problem with format of $DEBUG
The Symbol Table in $DEBUG does not have the expected format.

**** ERROR: requested format is not implemented
The format requested for a DISPLAY directive is not implemented.

**** ERROR: REWIND requires a file name
The keyword REWIND was followed by something other than a valid file name.

**** ERROR: routine GETNONBL failed
Routine GETNONBL in subroutine DBGTOKEN failed due to a SID logic error.

**** ERROR: search for dimension descriptor is lost
A Symbol Table search was lost due to a SID logic failure or $DEBUG format problem.

**** ERROR: search for dimension symbol entry is lost
A Symbol Table search was lost due to a SID logic failure or $DEBUG format problem.

**** ERROR: STORE requires an X-variable
The keyword STORE was not followed by a local variable name.

**** ERROR: subroutine table is missing
The Symbol Table record for a subroutine is missing, probably because the subroutine was not compiled or assembled with the necessary option.

**** ERROR: subroutine table is missing for *modulename*
The subroutine table for the module named in the message is missing. The message is issued in response to the WHERE directive using a word or parcel address, and usually gives the name of a library routine containing the address.

**** ERROR: subscript must be an integer
An array subscript was something other than an integer constant or an integer variable.

**** ERROR: subscript must be followed by , or)
An array subscript was followed by something other than a comma or closing parenthesis.

**** ERROR: symbol * not permitted in an IF clause
The symbol * was used for the word address in the IF clause of a BREAKPOINT or PACKAGE directive.

**** ERROR: symbol file is not available
A symbolic address was used in a directive but the dataset containing the Symbol Table is not available. The dataset can be missing for two reasons.

- The program was not compiled or assembled with the necessary options.
- The absolute binary file was saved for later use but the symbol dataset was not saved or not accessed.

**** ERROR: symbol is external
The address is meaningless for a symbol that is external to the module in which it is accessed.

**** ERROR: symbol is not a parcel
A symbol used in a parcel-address expression was not a label or entry point.

**** ERROR: symbol is not an array
A symbol followed by subscripts is not defined as an array.

**** ERROR: symbol is not word aligned
The symbol used in a word-address expression has a bit offset and cannot be used.

**** ERROR: symbol is too long
A symbol in a directive was longer than the longest valid symbol that could be expected in that position.

**** ERROR: symbol search is lost
A SID logic error caused the wrong Symbol Table record to be searched.

**** ERROR: symbol was not found
A variable or label was not found in the Symbol Table record for the subprogram or common block last set up as the default with the MODULEBASE directive, or for the module whose name preceded the variable or label.

**** ERROR: the keyword AT is required
The X variable in a STORE directive was not followed by the required keyword AT.

**** ERROR: the keyword COUNT is required
A masked PATCH or masked DISPLAY directive did not include the keyword COUNT between the number of the first bit and the bit count.

**** ERROR: the keyword FOR is required
The display count in an INSTRUCTIONS directive was not preceded by the keyword FOR.

**** ERROR: the keyword FROM is required
The X variable in a LOAD directive was not followed by the required keyword FROM.

**** ERROR: the keyword IF is required
The parcel address in a BREAKPOINT or PACKAGE directive was not followed by the keyword IF.

**** ERROR: the keyword WITH is required
A PATCH directive was entered without the keyword WITH between the first address to be patched and the list of patch values.

**** ERROR: there is no address to scroll from
A FORWARD or BACKWARD directive was used when the last display did not use an address that could be reused, or before any DISPLAY directive was used.

**** ERROR: this address overlaps another breakpoint
Installing more than one breakpoint on a parcel can cause the wrong code to be replaced when breakpoints are removed. The latest BREAKPOINT or PACKAGE directive is ignored.

**** ERROR: this format not allowed for masked display
An invalid display format was used with a masked DISPLAY directive.

**** ERROR: this value cannot be tested
The value to be tested against in the IF clause of a BREAKPOINT or PACKAGE directive is an unrecognized type or a type that cannot be tested.

**** ERROR: too many subscripts in array reference
An array reference contained more subscripts than dimensions declared in the array.

**** ERROR: two-word formats not allowed in dual display
One of the formats DOUBLE or COMPLEX was used in a DISPLAY directive with another format. Only 1-word formats are allowed in dual-format displays.

**** ERROR: unbalanced parentheses
Unbalanced parentheses were used in a word-address expression.

**** ERROR: unrecognized base, octal used
A prefix other than O', D', and X' was used with a number, so octal was used as the base for converting the number.

**** ERROR: unrecognized directive
The first token in an input record or following a semicolon was not recognized as a directive name or a valid abbreviation of a directive name.

**** ERROR: value is too large for bit field
The value given in a masked PATCH directive does not fit into the specified bit field.

**** ERROR: value must be string or number
The value used in a PATCH directive was not a string or a recognized type of number.

**** ERROR: value must be unsigned integer
The value for a masked or parcel patch was not an unsigned integer. Negative values must be entered using their octal representation.

**** ERROR: width must be COLUMN, SCREEN, or PAGE
Something other than COLUMN, SCREEN, or PAGE followed the keyword WIDTH.

**** ERROR: X or C variable required
One of the keywords INCREMENTVARIABLE or DECREMENTVARIABLE was not followed by an X variable; or, SETCONDITION, RESETCONDITION, or TESTCONDITION was not followed by a condition variable.

**** WARNING: existing breakpoint replaced
When the last breakpoint was installed, an earlier breakpoint with the same number was replaced.

**** WARNING: subscript out of bounds
The subscript for an array is not in the range declared for the array. The subscript is still used, which could cause the wrong location to be used.

**** WARNING: the first parcel has been moved at label *label*
When a breakpoint is installed, two parcels of the user code are replaced by a jump to SID. If the second parcel is referenced in a jump instruction, the second half of the jump instruction is executed instead of the instruction that was replaced.

**** WARNING: return jump instruction moved
A breakpoint was placed on a return jump instruction, so the return address for the subroutine could be wrong. There is no problem, however, if only one breakpoint is set or if the breakpoint is exited through the STEP directive.

**** WARNING: stepped into a wild jump from P=address
The address for the user program's P register is outside the user's address area (>JCHLM). This could happen when data rather than code was executed with the STEP directive, or if an invalid address was patched into a jump instruction.

CODED MESSAGES

DB001 - SYMBOL FILE IS NOT AVAILABLE
The dataset used with the S parameter is not local to the job.

DB002 - INPUT FILE IS NOT AVAILABLE
The dataset used with the I parameter is not local to the job.

DB003 - CNT VALUE IS NOT A POSITIVE INTEGER
The value given for the CNT parameter was negative or contained non-numeric characters.

DB004 - OVERLAPPING BREAKPOINTS
A breakpoint was placed inside the SID code or a library routine used by the breakpoint routines was stepped into.

DB005 - COUNT DECREMENTED TO ZERO

The variable initialized by the CNT parameter on the SID control statement was decremented to 0. This variable is decremented once each time a breakpoint is reached.

DB006 - EMPTY CONTROL STATEMENT FILE

The control statement file was empty when SID, running in batch mode, tried to read the control statement for the user program.

EXAMPLES OF COMPILER-GENERATED DO-LOOP LABELS

B

The following examples show generated statement labels for DO-loops.

EXAMPLE 1 - SIMPLE DO-LOOP

Original DO-loop:

```
        DO 10 I = 1,N
        CALL X(I)
  10    CONTINUE
```

Same loop with compiler-generated labels:

```
        I = 1
        IF (N.LT.1) GO TO 10B
 10A    CONTINUE
        CALL X(I)
        I = I+1
        IF (I.LE.N) GO TO 10A
 10B    CONTINUE
```

EXAMPLE 2 - LOOP FOR WHICH LABEL B IS NOT SAVED

Original DO-loop:

```
        DO 30 I = 1,200
        CALL Z(I)
  30    CONTINUE
```

Same loop with compiler-generated labels:

```
        I = 1
 30A    CONTINUE
        CALL Z(I)
        I = I+1
        IF (I.LE.200) GO TO 30A
```

EXAMPLE 3 - NESTED DO-LOOPS WITH SHARED TERMINAL STATEMENT

Original DO-loop:

```
        DO 20 I = 1,N
        DO 20 J = 1,M
        CALL Y(I,J)
 20     CONTINUE
```

Same loop with compiler-generated labels:

```
        I = 1
        IF (N.LT.1) GO TO 20B
20A     CONTINUE
        J = 1
        IF (M.LT.1) GO TO 20D
20C     CONTINUE
        CALL Y(I,J)
        J = J+1
        IF (J.LE.M) GO TO 20C
20D     CONTINUE
        I = I+1
        IF (I.LE.N) GO TO 20A
20B     CONTINUE
```

EXAMPLE 4 - LOOP FOR WHICH THE ORIGINAL LABEL IS SAVED

Original DO-loop:

```
        DO 40 I = 1,N
        IF (LIST(I).EQ.0) GO TO 40
        CALL A(LIST(I))
 40     CONTINUE
```

Same loop with compiler-generated labels:

```
        I = 1
        IF (N.LT.1) GO TO 40B
40A     CONTINUE
        IF (LIST(I).EQ.0) GO TO 40
        CALL A(LIST(I))
 40     CONTINUE
        I = I+1
        IF (I.LE.N) GO TO 40A
40B     CONTINUE
```

EXAMPLE 5 - SHORT VECTOR LOOP

Original DO-loop:

```
        DO 50 I = 1,50
        A(I) = 0
50      CONTINUE
```

Same loop as set up by the CFT compiler:

```
        A(1:50) = 0
        I = 51
```

A(1:50) here means elements A(1) through A(50)

EXAMPLE 6 - VECTORIZED DO-LOOP

Original DO-loop:

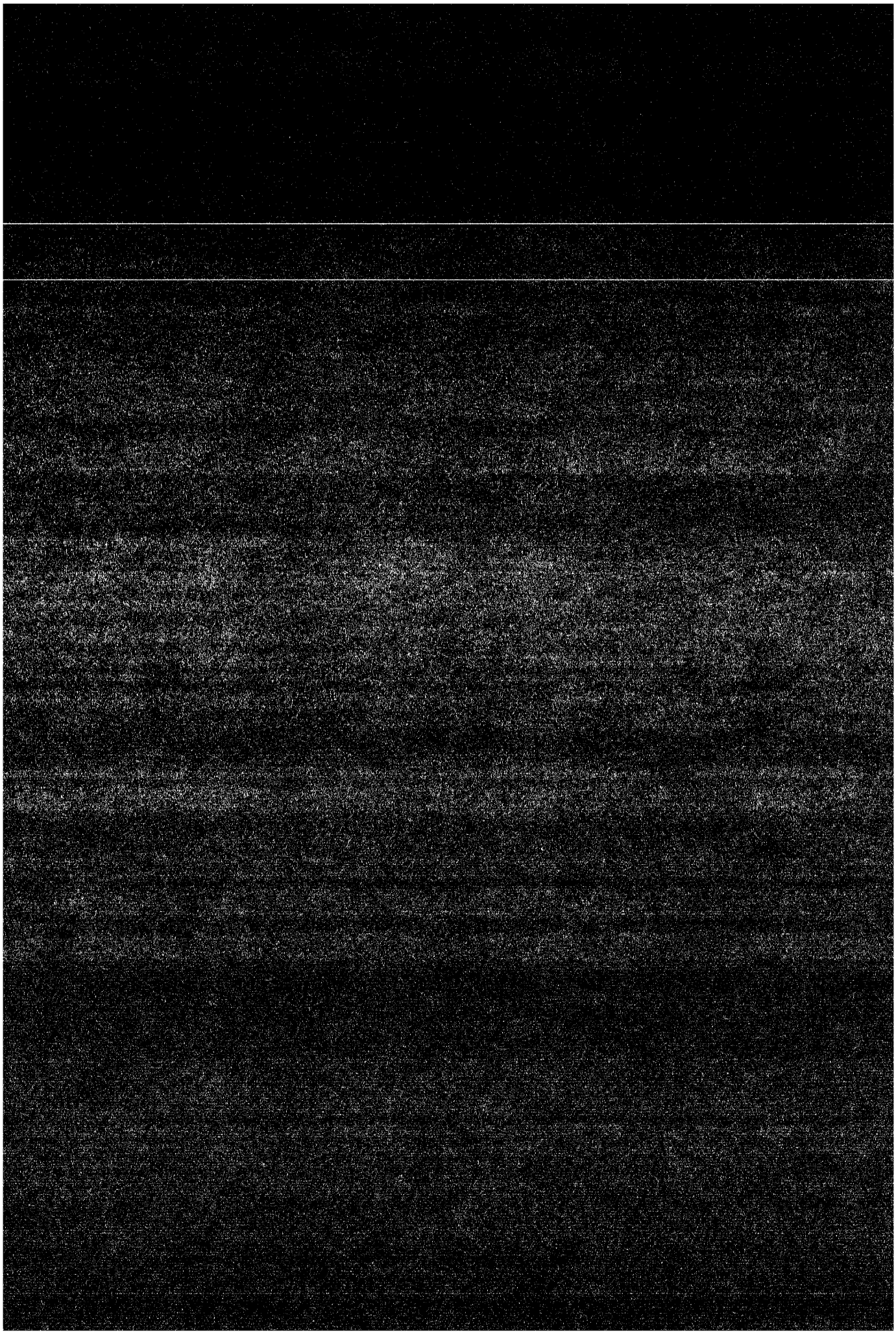
```
        DO 60 I = 1,N
        A(I) = 0
60      CONTINUE
```

Same loop with compiler-generated labels:

```
        I = 1
        REMAIN = MOD(N,64)
        IF (N.LT.1) GO TO 60B
60A     CONTINUE
        A(I:I+REMAIN-1) = 0
        I = I+REMAIN
        IF (I.LE.N) GO TO 60A
        REMAIN = 64
60B     CONTINUE
```

A(I:I+REMAIN-1) here means elements A(I) through A(I+REMAIN+1). The compiler-generated variable REMAIN serves to cycle I in steps of 64. Note that the first loop iteration will process the leftover, mod 64, segment.

INDEX



INDEX

READERS COMMENT FORM

Symbolic Interactive Debugger (SID) User's Guide SG-0056 A

Your comments help us to improve the quality and usefulness of our publications. Please use the space providec below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.

READERS COMMENT FORM

Symbolic Interactive Debugger (SID) User's Guide SG-0056 A

Your comments help us to improve the quality and usefulness of our publications. Please use the space providec below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

Attention:
PUBLICATIONS

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.