

Norsk Data



SYMBOLIC DEBUGGER **User Guide**

ND-60.158.3 EN



SYMBOLIC DEBUGGER

User Guide

ND-60.158.3 EN

Preface:

THE PRODUCT

This manual describes the SYMBOLIC DEBUGGER product:

SYMBOLIC DEBUGGER	ND-10335D	(ND-500)
	ND-10336D	(ND-100)

THE READER

This manual will be of interest to programmers who are testing programs written in any language whose compiler is able to communicate with the Symbolic Debugger.

PREREQUISITE KNOWLEDGE

The reader should be able to successfully compile and load a program in one of the following languages: ADA, BASIC, COBOL, FORTRAN, PLANC or SIMULA. Some of the commands require more advanced programming experience.

THE MANUAL

This manual describes how to use the Symbolic Debugger. The commands are described in detail. Examples are from both the ND-100 and the ND-500 Debugger.

RELATED MANUALS

Related manuals for the languages with which the Symbolic Debugger can be used are:

ADA-500 User Manual	ND-60.198	(available 1985)
BASIC-500 User Manual	ND-60.197	(available 1985)
COBOL Reference Manual	ND-60.144	
FORTTRAN Ref. Manual	ND-60.145	
PLANC Reference Manual	ND-60.117	

The following manuals are also relevant:

ND Relocating Loader	ND-60.066
ND-500 Loader/Monitor	ND-60.136
BRF Linker User Manual	ND-60.196

TABLE OF CONTENTS

<u>Section</u>	<u>Page</u>
1 INTRODUCTION	1
1.1 Symbolic Debugger Command Summary (HELP OUTPUT)	4
2 USING THE SYMBOLIC DEBUGGER	7
2.1 How to Compile your Programs	9
2.2 How to Load ND-100 Programs	10
2.3 How to Load ND-500 Programs	10
2.4 How to Use the Debugger	11
2.5 An Example using FORTRAN	12
3 COMMANDS - DETAILED DESCRIPTION	17
3.1 ACTIVE-ROUTINES (<maximum number of levels>)	19
3.2 ALIGN-LISTING <program area> <line>	20
3.3 ATTACH-SEGMENT <segment number>	20
3.4 BREAK <routine, label or line> (<count>) (<condition>)	21
3.5 BREAK-ADDRESS <program address> (<count>)	22
3.6 BREAK-RETURN	22
3.7 CHECK-OUT-MODE (<program area>)	24
3.8 COMPARE-DATA <low> <high> (<output file>)	25
3.9 COMPARE-PROGRAM <low> <high> (<output file>)	26
3.10 CONTINUE	26
3.11 DISPLAY (<item or value>)	27
3.12 DUMP-LOG (<output file>)	28
3.13 ENABLED-TRAPS	29
3.14 EXIT	29
3.15 FIND-SCOPE <program address>	29
3.16 FORMATS-DISPLAY <formats (A,D,F,H,O or combinations)>	30
3.17 FORMATS-LOOK-AT <formats (A,D,F,H,I,O or combinations)>	30
3.18 GUARD <item or address> (<(*not*) low (: high)>)	31
3.19 HELP <command name>	32
3.20 INCLUDE-COMMANDS <file name>	33
3.21 INVOKE <routine> (<(parameter,...,parameter)>)	34
3.22 LOCAL-TRAP-DISABLE (<trap conditions>)	35
3.23 LOCAL-TRAP-ENABLE (<trap conditions>)	36
3.24 LOG-CALLS <program area>	37
3.24.1 LOG-CALLS and CHECK-OUT-MODE	37
3.24.2 LOG-CALLS and GUARD	38
3.24.3 LOG-CALLS and STEP	38
3.25 LOG-LINES <program area>	38
3.25.1 LOG-LINES and CHECK-OUT-MODE	39
3.25.2 LOG-LINES and GUARD	39
3.25.3 LOG-LINES and STEP	39

Section	Page
3.26 LOOK-AT-DATA <data address> (<count>) (<output file>) . .	40
3.27 LOOK-AT Subcommands	43
3.28 LOOK-AT-PROGRAM <program address> (<count>) (<output file>) . .	46
3.29 LOOK-AT-REGISTER <register name> (<count>) (<output file>) . .	47
3.30 LOOK-AT-STACK <B register> (<count>) (<output file>) . . .	48
3.31 MACRO <name> <body>	49
3.32 PLACE <file name> (<W>)	51
3.33 PROGRAM-INFORMATION	51
3.34 RESERVE-TERMINAL <logical device number>	52
3.35 RESET-BREAKS (<program area>)	53
3.36 RUN (<program address>)	54
3.37 SCOPE (<module, routine or other item>)	54
3.38 SEGMENT-INFORMATION	55
3.39 SET <variable> (=) <value>	55
3.40 STACK-INSTRUCTIONS (<low>) (<high>)	56
3.41 STEP (<count>)	57
 4 SYMBOLIC DEBUGGER PARAMETERS	 59
4.1 Numeric Constants	61
4.2 Single-Character Constants	64
4.3 String Constants	64
4.4 Expressions	65
4.5 Named Items	67
4.6 Program Area	69
4.7 Program Address	69
4.8 Data Address	70
4.9 Format Specifier	71
4.10 File Name	71
 5 EXAMPLES	 73
5.1 An Example Using FORTRAN-100	75
5.2 A PLANC Example	77
5.3 Another Example in PLANC	79
5.4 Using a File as a Segment	84
5.5 Using a File as a Segment for a COMMON Area	85
 6 ERROR MESSAGES	 87
6.1 Error Messages Common to the ND-100 and the ND-500 Versions	89
6.2 Error Messages Which Apply to the ND-100 Version	92
6.3 Error Messages Which Apply to the ND-500 Version	92
6.4 Note on Error Returns on the ND-100	94
 Index	 95

CHAPTER 1

INTRODUCTION

1 INTRODUCTION

The Symbolic Debugger is an interactive tool for testing programs written in higher-level languages such as FORTRAN, COBOL, and PLANC. The Symbolic Debugger is available on the ND-100 if SINTRAN was generated with at least one Debugger segment. If three segments were generated, that means that three people or less can use the Debugger simultaneously. There are no such limitations on the ND-500.

The Symbolic Debugger contains a powerful set of commands which enable you to control the execution of your program. For example, break or step points can be set to stop the program under certain conditions. You can then inspect or modify program variables, and continue execution until the next break or step point. In this way it is possible to find many program bugs in one run. It is also possible, for instance, to detect which areas of a program have not been executed, and to change the path and frequency of subroutine calls.

The commands available are listed on the following page.

1.1 Symbolic Debugger Command Summary (HELP OUTPUT)

Here is a list of all the commands available in the Symbolic Debugger. The unambiguous abbreviation to the left of each command may be used.

A	ACTIVE-ROUTINES	(<MAXIMUM NUMBER OF LEVELS>)
AL	ALIGN-LISTING	<PROGRAM AREA> <LINE>
B	BREAK	<ROUTINE, LABEL OR LINE> (<COUNT>) (<CONDITION>)
B-A	BREAK-ADDRESS	<PROGRAM ADDRESS> (<COUNT>)
B-R	BREAK-RETURN	
CH	CHECK-OUT-MODE	(<PROGRAM AREA>)
C-D	COMPARE-DATA	<LOW> <HIGH> (<OUTPUT FILE>)
C-P	COMPARE-PROGRAM	<LOW> <HIGH> (<OUTPUT FILE>)
C	CONTINUE	
D	DISPLAY	(<ITEM OR VALUE>)
DU	DUMP-LOG	(<OUTPUT FILE>)
E	EXIT	
FI	FIND-SCOPE	<PROGRAM ADDRESS>
F	FORMATS-DISPLAY	<FORMATS (A,D,F,H,O OR COMBINATIONS)>
F-L	FORMATS-LOOK-AT	<FORMATS (A,D,F,H,I,O OR COMBINATIONS)>
G	GUARD	<ITEM OR ADDRESS> (<(*NOT*) LOW (: HIGH)>)
H	HELP	<COMMAND NAME>
I-C	INCLUDE-COMMANDS	<FILE NAME>
I	INVOKE	<ROUTINE> (<(PARAMETER,...,PARAMETER)>)
L-C	LOG-CALLS	<PROGRAM AREA>
L-L	LOG-LINES	<PROGRAM AREA>
L--D	LOOK-AT-DATA	<DATA ADDRESS> (<COUNT>) (<OUTPUT FILE>)
L	LOOK-AT-PROGRAM	<PROGRAM ADDRESS> (<COUNT>) (<OUTPUT FILE>)
L--R	LOOK-AT-REGISTER	<REGISTER NAME> (<COUNT>) (<OUTPUT FILE>)
L--S	LOOK-AT-STACK	<B REGISTER > (<COUNT>) (<OUTPUT FILE>)
M	MACRO	<NAME> <BODY>
R-T	RESERVE-TERMINAL	<LOGICAL DEVICE NUMBER>
R-B	RESET-BREAKS	(<PROGRAM AREA>)
R	RUN	(<PROGRAM ADDRESS>)
SC	SCOPE	(<MODULE, ROUTINE, OR OTHER ITEM>)
SE	SET	<VARIABLE> (=) <VALUE>
S	STEP	(<COUNT>)

All of these commands are on both the ND-100 and the ND-500. Commands that are not available on both systems are listed on the next page.

The following three commands are only available on the ND-100:

PL PLACE <FILE NAME> (<W>)
PR PROGRAM-INFORMATION
S-I STACK-INSTRUCTIONS (<LOW>) (<HIGH>)

The following five commands are only available on the ND-500:

A-S ATTACH-SEGMENT <SEGMENT NUMBER>
E ENABLED-TRAPS
L--D LOCAL-TRAP-ENABLE (<TRAP CONDITIONS>)
L--E LOCAL-TRAP-DISABLE (<TRAP CONDITIONS>)
S-I SEGMENT-INFORMATION

CHAPTER 2

USING THE SYMBOLIC DEBUGGER

2 USING THE SYMBOLIC DEBUGGER

2.1 How to Compile your Programs

In general, if you are working on a program that you will be compiling and debugging many times, the easiest way is to create a mode file that you can run each time you want to compile your program.

Here is the general form of that mode file:

```
@DELETE-FILE <object file> <file type>
@<compiler name>
DEBUG-MODE
COMPILE <source file> <list file> "<object file>"
EXIT
```

The default <file type> for the source file and the list file is :SYMB. If you are compiling an ND-100 program, the object file type is :BRF. For ND-500 programs, the object file type is :NRF.

You might want to always use the same name for the program you are currently working on. You could call your current program TEST:SYMB and create a mode file like this if it is a FORTRAN-100 program:

```
@DELETE-FILE TEST:BRF
@FORTRAN-100
DEBUG-MODE
COMPILE TEST TERMINAL "TEST"
EXIT
```

The mode file will work whether the file TEST:BRF exists already or not. If you call the above mode file COMPILE-TEST:SYMB, you could give the following input to SINTRAN:

```
@MODE COMPILE-TEST "TEST:LIST"
```

The file TEST:LIST will contain your program listing, with line numbers that will be useful when you debug your program. Another way to get your listing is to compile like this:

```
@<name of compiler>
DEBUG-MODE
COMPILE TEST "LISTING" "TEST"
EXIT
```

Then the file LISTING:SYMB will contain your program listing.

2.2 How to Load ND-100 Programs

The mode file to load a :BRF file, for instance, TEST:BRF, should look like this:

```
@DELETE-FILE TEST:PROG
@BRF-LINKER
PROG-FILE "TEST"
LOAD TEST
LOAD <any additional modules or libraries you have>
LOAD <library>
ENTRIES-UNDEFINED
EXIT
```

This mode file will work whether the file TEST:PROG already exists or not. The library you load depends on which compiler you used:

Compiler	1- or 2-Bank	Load this :BRF file:
COBOL	1	COBOL-1BANK
COBOL	2	COBOL-2BANK
FORTRAN	1	FORT-1BANK
FORTRAN	2	FORT-2BANK
FTN	1	FTNLIBR
FTN		FTNRTLBR (RT programs)
PASCAL	1	PASCAL-LIB
PASCAL	2	PASCAL-2LIB
PLANC	1	PLANC-1BANK
PLANC	2	PLANC-2BANK

This information is explained in greater detail in the manual for the language you want to load programs in.

2.3 How to Load ND-500 Programs

The mode file to load an :NRF file, for instance, TEST:NRF, should look like this:

```
@ND LINKAGE-LOADER
ABORT-BATCH-ON-ERROR OFF
RELEASE-DOMAIN TEST
DELETE-DOMAIN TEST
SET-DOMAIN "TEST"
LOAD-SEGMENT TEST
LOAD-SEGMENT <library>
EXIT
@CC WRITE @ND-500 TEST to execute TEST or
@CC WRITE @ND-500
@CC DEBUGGER TEST to debug TEST
```

This mode file will work whether the domain TEST exists already or not. It will work for any language if you load the correct library. You will need additional LOAD-SEGMENT statements for any additional modules or libraries you use. The library names are usually the name of the language followed by "-LIB:NRF", for example, COBOL-LIB:NRF is

the library for COBOL.

2.4 How to Use the Debugger

If you have compiled and loaded your file as an ND-100 program, you start the debugger for the program TEST as follows:

```
@DEBUGGER ↵  
ND-100 SYMBOLIC DEBUGGER. VERSION D.  
*PLACE TEST ↵  
*_
```

If the program is an ND-500 program, do the following:

```
@ND-500 ↵  
ND-500 MONITOR VERSION C 82.11.22 / 82.12.16  
N500:DEBUGGER TEST ↵  
ND-500 SYMBOLIC DEBUGGER. VERSION D.  
*_
```

Now you can set a breakpoint at the point in the program where you want to break. You may also set multiple "step points". When program execution reaches those points, you may inspect or modify program information. Then you can continue to execute your program in one of two ways:

- 1) RUN takes you to the next breakpoint
- 2) STEP takes you to the next step point.

2.5 An Example using FORTRAN

Here is a fairly comprehensive example. We shall start by compiling the file FORT-1:SYMB. The program below prints the bit pattern for the integer DEC.

```
@FORTRAN-100 ↵
ND-100/NORD-10 ANSI 77 FORTRAN COMPILER - 203053D
FTN: DEBUG ↵
FTN: CROSS-REFERENCE CROSS-REF:XREF ↵
FTN: COMPILE FORT-1,TERMINAL,FORT-1 ↵
```

Gives useful information
for debugging.

Normally, you will send your
listing to a file or printer.

```
ND-100/NORD-10 ANSI 77 FORTRAN COMPILER - 203053D 12:38 25 OCT 1984
SOURCE FILE: FORT-1:SYMB
```

```
1*      program convert
2*      integer dec, counter, i, value, bits
3*      dimension bits(16)
4*      dec = 35864
5*      counter = 1
6*      do 100 I = 15,0,-1
7*          value = 2 ** I
8*          if (dec .ge. value) then
9*              bits(counter) = 1
10*             dec = dec - value
11*          else
12*              bits(counter) = 0
13*          endif
14*      100 continue
15*      write (1, 1000) (bits(i), i = 1,16)
16*      1000 format (1H , 16(I4))
17*      end
```


----- CROSS REFERENCE -----

The displacement of the data
relative to the B register.

BITS	INTEGER* 2	1-ARRAY	IND(-172)	2	3	9	12	15
CONVERT		PROGRAM			1			
COUNTER	INTEGER* 2	VARIABLE	-170	2	5	9	12	
DEC	INTEGER* 2	VARIABLE	-171	2	4	8	10	10
I	INTEGER* 2	VARIABLE	-167	2	6	7	15	15
VALUE	INTEGER* 2	VARIABLE	-166	2	7	8	10	
\$100	STATEMENT	LABEL	AT 14		6			
\$1000	FORMAT	LABEL				15	16	

- CPU TIME USED: 2.2 SECONDS. 17 LINES COMPILED.

- NO MESSAGES

- PROGRAM SIZE=140 COMMON SIZE=0

FTN: EXIT

@BRF-LINKER ↵

- BRF Linker - JULY 3, 1984

Brl: PROG-FILE FORT-1 ↵

Brl: LOAD FORT-1,F-1BANK ↵

FREE: P 000214-177777 DEBUG 000146

FREE: P 035253-177777 DEBUG 000146

Brl: EXIT ↵

@DEBUGGER ↵

ND-100 SYMBOLIC DEBUGGER. VERSION D.

*PLACE FORT-1 ↵
FORTRAN PROGRAM CONVERT.1*LOG-LINES 8 ↵
*LOG-LINES 13 ↵

*STEP ↵

This is how you can obtain multiple step points. The program will stop at lines 8 and 13 if you use STEP.

Always use STEP to stop at LOG-LINES step points.

CONVERT.8 *DISPLAY ↵

DISPLAY without parameters lists all local variables.

ERRCODE=0 DEC=-29672
VALUE=-32768 BITS(1:16)

COUNTER= 1 I= 15

Notice that DEC and VALUE have incorrect values. The reason will be explained shortly.

*DISPLAY ADDR DEC ↵
ADDR DEC=000150B
*DISPLAY ADDR VALUE ↵
ADDR VALUE=000153B

We can inspect the data addresses where the values of DEC and VALUE are stored.

*LOOK-AT-DATA ADDR(DEC) ↵

We could also write: *L--D 150B, *L--D 104, but not *L--D DEC.

D 000150B: 106030B -29672
D 000151B: 000001B 1
D 000152B: 000017B 15
D 000153B: 100000B -32768
D 000154B: 000000B 0
D 000155B: 000000B 0
*FORMATS-DISPLAY H O D ↵

Note that DEC and VALUE are each stored in one word (16 bits). Integers greater than 32768 need to be stored in a larger area.

H means hexadecimal, O means octal and D means decimal. Numbers will now appear in all three formats.

*DISPLAY 2**15 ↵
2**15=32768 8000H 100000B*DISPLAY -32768 ↵
-32768=-32768 FFFF8000H 37777700000B
*STEP ↵

DISPLAY accepts expressions. Note that the 8 least significant bits are the same for 32768 and -32768.

```

CONVERT.14      *FORMATS-DISPLAY D ↵
*DISPLAY BITS ↵
BITS=1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
*STEP ↵

```

Sets format display back to only decimal numbers.

```

CONVERT.8      *DISPLAY ↵
ERRCODE=0      DEC=3096
VALUE=16384     BITS(1:16)

```

COUNTER= 1 I= 14

Note that the value of VALUE is correct now that I = 14.

```

CONVERT.14     *DISPLAY BITS ↵
BITS=1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
*STEP ↵

```

```

CONVERT.14     *DISPLAY ↵
ERRCODE=0      DEC=3096
VALUE=8192      BITS(1:16)
*DISPLAY BITS ↵
BITS=0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```

COUNTER= 1 I= 14

BITS(1) is zero again, since the variable COUNTER has not been incremented.

```
*RESET-BREAKS ↵
```

This resets break and step points. The program will execute now without breaks.

```

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PROGRAM TERMINATED AT CONVERT.18

```

Here are the changes that need to be made, and the resulting program.
The old line numbers are given in parentheses:

DEC must be declared as INTEGER*4 because of its size.

VALUE and I need to be declared as INTEGER*4
instead of INTEGER.

This needs to be added after line 7:
counter = counter + 1

```
( 1)      program convert
( 2)      integer    counter, i, bits
           integer*4 dec, i, value
( 3)      dimension bits(16)
( 4)      dec = 35864
( 5)      counter = 1
( 6)      do 100 I = 15,0,-1
( 7)          value = 2 ** I
           counter = counter + 1
( 8)          if (dec .gt. value) then
( 9)              bits(counter) = 1
(10)              dec = dec - value
(11)          else
(12)              bits(counter) = 0
(13)          endif
(14)      100      continue
(15)      write (1, 1000) (i,          i = 15,0,-1)
(16)      write (1, 1000) (bits(i), i = 1,16)
(17)      1000    format (1H ,4 (4(I3),1H ) )
(18)      end
```


CHAPTER 3

COMMANDS - DETAILED DESCRIPTION

3 COMMANDS - DETAILED DESCRIPTION

Following is a list of all available commands with their parameters.

Parameters are enclosed in less than (<) and greater than characters (>). If a parameter is also enclosed in parentheses, it is optional.

<low> <high>	required parameters
(<maximum number of levels>)	optional parameter

If you give commands without parameters, you will only be prompted for the required parameters.

3.1 ACTIVE-ROUTINES (<maximum number of levels>)

This command writes the current routine call hierarchy, starting with the current routine and ending with the main program. The maximum number of levels to be printed may be specified.

```
*ACTIVE-ROUTINES ←|
QUIKSORT.3 CALLED FROM QUIKSORT.44
QUIKSORT.3 CALLED FROM MAIN.23
MAIN.9
*
—
```


3.2 ALIGN-LISTING <program area> <line>

This command is used to adjust the line numbers in the Debugger to correspond with those on a listing which is not up-to-date. Several ALIGN-LISTING commands may be given in order to adjust different parts of the listing. If areas overlap, the command most recently given takes priority over previous ones.

If no program area is specified, the innermost routine in the current scope is assumed.

Let us say that you have added 5 lines of code to the subroutine EVAL and things look like this:

On your listing:	5 . . . 6 7 8 9 10, ... 120
After a new	
compilation:	5 6 7 8 9 10 11 12 13 14, ... 124

In other words, what appears as line 6 on your listing is now line 10. Here is what you can do:

```
*BREAK EVAL.10 ↵
*ALIGN-LISTING EVAL 6 ↵
```

What was line 10 in EVAL will now be 6, 11 will become 7, and so on throughout the rest of EVAL. Lines 1 to 5 will remain unchanged, while the newly compiled lines 6 to 10 will become unnumbered.

You can do this many places. It is best to do it on the first unchanged line following every area where code has been altered. All the unaltered areas will then have the same line numbers as in your listing.

You may align an entire routine:

```
*ALIGN-LISTING PRINT 800 ↵
*BREAK PRINT ↵
*RUN ↵

BREAK AT PRINT.804
*_
```

The first line in the routine PRINT will be numbered 800.

3.3 ATTACH-SEGMENT <segment number>

This command is relevant to the ND-500 only. The current segment is moved to <segment number>. The command SEGMENT-INFORMATION provides a list of all active segments.

3.4 BREAK <routine, label or line> (<count>) (<condition>)

Sets a breakpoint at the specified item, and removes the previous breakpoint set by BREAK. If a routine name is specified, the breakpoint is set at the first line in the routine.

If a positive number K is specified for the count parameter, the program will break every K times the breakpoint is reached.

```
*BREAK SUBCALC.52 10 ↵
*RUN ↵
BREAK AT SUBCALC.52
*
_
```

The program will execute until line 52 is encountered for the 10th time. When the breakpoint is reached, execution terminates and control passes to the Debugger. To continue to the breakpoint again, use RUN. To continue to the nearest step point, use STEP.

If a condition is specified, control passes to the Debugger at the breakpoint only if the condition is true and the variable is local:

```
*BREAK $I0 I > 5 ↵
*RUN ↵
  4          16.00
  5          25.00
  6          36.00
CONDITIONAL BREAK AT SQRS.7
*DISPLAY I ↵
I=6
*
_
```

If I is not local, prefix it with the routine name, for example, CALC.I. Only one breakpoint is allowed, but you may have multiple "step points" by using LOG-LINES. See the examples on pages 14 and 69.

You can also create breakpoints by using GUARD, see page 31.

If you do not know where you can set breakpoints, do the following:

```
*LOG-CALLS... ↵
*CHECK-OUT-MODE ↵
*DUMP-LOG ↵
```

The line numbers where all the routines start will be listed.

3.5 BREAK-ADDRESS <program address> (<count>)

This command is similar to the BREAK command, except that the breakpoint is specified directly as a program address.

Examples:

```
*BREAK-ADDRESS 501 ↵
*RUN ↵
```

Stops at program address 501, not at line 501.

```
*BREAK-ADDRESS 501 10 ↵
*RUN ↵
```

Stops the 10th time that program address 501 is to be executed.

3.6 BREAK-RETURN

Sets a breakpoint at the return address of the current routine, and resumes execution from the current line. If a PLANC routine returns with an error return, the error code is displayed when the breakpoint is reached.

Here is an example with a small PLANC program:

```
1      MODULE EXAMPLE
2      INTEGER ARRAY : stack (0:100)
3      ROUTINE VOID,VOID: PARALLEL
4      INTEGER: x,y
5      3 =: x
6      x =: y
7      6 ERRETURN
8      ENDROUTINE
9      PROGRAM: OUTER
10     INISTACK stack
11     INTEGER : k,m
12     10 =: m
13     PARALLEL
14     ENDROUTINE
15     ENDMODULE
```

We can debug it on the ND-500 as follows:

```
@ND Debugger PLANC-PROG ↵
...
*BREAK PARALLEL ↵
*RUN ↵
PLANC PROGRAM.EXAMPLE.OUTER.9
*LOG-LINES... ↵
*STEP ↵
OUTER.12 * ↵
OUTER.13 * ↵
BREAK AT PARALLEL.5
*BREAK-RETURN ↵
BREAK AT OUTER.13; ERROR RETURN WITH ERRCODE = 6
*
_
```

Each STEP or Carriage Return
advances us one line at a time.

WRITE parameters in PLANC are not updated at BREAK-RETURN.

3.7 CHECK-OUT-MODE (<program area>)

This command removes the step point on each line in the specified area that is executed. You can thus obtain a list of all lines which have never been executed, by using the DUMP-LOG command.

If the command LOG-CALLS is given before the CHECK-OUT-MODE command, DUMP-LOG will list the first line in every routine that was not executed.

If no area is specified, all lines are checked.

See the examples on page 39 and on page 78.

Note:

Since CHECK-OUT-MODE removes step points, you cannot do the following:

```
*LOG-LINES <program area>
*CHECK-OUT-MODE <program area>
*STEP
```

You need to do this instead:

```
*LOG-LINES <program area>
*CHECK-OUT-MODE <program area>
*BREAK <routine, label or line>
*RUN
```

3.8 COMPARE-DATA <low> <high> (<output file>)

The data area specified is compared to the program file contents (:PSEG and :DSEG files on the ND-500). The address of each modified location is displayed, along with the old and new contents.

The default output file is the terminal; the default file type is :LIST.

In the following program, a loop is executed K times. We find the address where K is stored and change K to 20.

```
*LOG-LINES... ↵
*GUARD K ↵
*DISPLAY K ↵
K=0
*RUN ↵
```

By using LOG-LINES, GUARD, and RUN, we break just after K is assigned a value. LOG-LINES is not necessary before GUARD on the ND-500.

```
GUARD VIOLATION AT SQRS.5
*DISPLAY ↵
ERRCODE=0      I=0      K= 5      R= 0.0
*LOOK-AT-DATA ADDR(K)... ↵
D 000122B: 000005B      5      20 ↵
D 000123B: 000000B      0      ↵
*DISPLAY ↵
ERRCODE=0      I=0      K= 20      R= 0.0
*COMPARE-DATA ADDR(K) 200B TERMINAL ↵
D 000122B: 000000B CHANGED TO 000024B
*RESET-BREAKS ↵
```

RESET-BREAKS is necessary, otherwise the program will break in every line where K is not equal to 0.

```
*RUN ↵
```

The default output file is the terminal; the default file type is :LIST.

3.9 COMPARE-PROGRAM <low> <high> (<output file>)

The program area specified by the lower and upper bounds is compared to the program file contents (:PSEG and :DSEG files on the ND-500). Modified locations are displayed with address, old contents and new contents.

The default output file is the terminal; the default file type is :LIST. See also COMPARE-DATA.

Here is an example of changing a MAC instruction:

```
*LOOK-AT-PROGRAM 30B ↵
P 000030B: 030607B 12679 1 STF ,B - 171 153000B ↵
P 000031B: 044021B 18449 H LDA * 21 ↵
*LOOK-AT-PROGRAM 27B 3 ↵
P 000027B: 110612B -28278 FMU ,B - 166
P 000030B: 153000B -10752 V MON
P 000031B: 044021B 18449 H LDA * 21 ↵
*COMPARE-PROGRAM 20B 40B TERMINAL ↵
P 000030B: 030607B CHANGED TO 153000B
*
_
```

3.10 CONTINUE

Execution is resumed from the current line. If you want to specify where you want to resume execution from, use RUN. See page 54. Since CONTINUE is a superfluous command, all examples in this manual use RUN.

Execution will continue until the breakpoint is reached or a GUARD violation occurs. Step points will be skipped.

3.11 DISPLAY (<item or value>)

If you only write DISPLAY, all variables in the innermost routine or module in the current scope are displayed.

```
*DISPLAY ↵
(all variables are listed.)
```

The item(s) and value(s) you specify will be displayed:

```
*DISPLAY I ↵
I=15
*DISPLAY I,J,K ↵
I=15
J=225
K=5
*DISPLAY STRING(1) ↵
STRING(1)=reduced
*_
```

Note that only the name and the bounds of arrays are output unless you specify their names. The same applies to strings.

```
*DISPLAY IND(CURRENT.RIGHT) ↵
IND(CURRENT.RIGHT)=          NAME(1:20)
RESULT=  4.40000000
LEFT=NIL      RIGHT=001054B
*DISPLAY CURRENT.NAME ↵
CURRENT.NAME=DEBUGGER
*DISPLAY CURRENT.RIGHT.NAME(1:4) ↵
CURRENT.RIGHT.NAME(1:4)=else
*_
```

DISPLAY will display according to the formats you specify.

```
*FORMATS-DISPLAY O D H ↵
*DISPLAY 8#101# ↵
8#101#=65 41H 101B
*_
```

You can include several expressions on the same line if you separate them by commas.

You can specify a module or routine name, and all variables in the routine or module are displayed.

Note that DISPLAY can be used for radix conversion when used in conjunction with the FORMATS-DISPLAY command.

```
*FORMATS-DISPLAY O D H ↵
*DISPLAY 2#100_010_110_001# ↵
2#100_010_110_001#=2225 8B1H 4261B
*_
```

3.12 DUMP-LOG (<output file>)

The output of this command depends on the type of log specified.

If LOG-CALLS was specified last, a list of the last 200 routine calls is displayed. See example on page 37.

If LOG-LINES was specified last, a list of the last 200 lines executed is displayed. If a line is the first line in a routine, the routine name is also displayed. See example on page 38.

If CHECK-OUT-MODE was specified last, a list of all the lines or routines (in the area specified in the CHECK-MODE command) that have not been executed is displayed on the terminal. If a line is the first line in a routine, the routine name is also displayed. See the example on page 39.

If you do the following when you start the Debugger, you will list every line in your program that can be logged, even if there are more than 200 lines:

```
*LOG-LINES,... ↵
*CHECK-OUT-MODE ↵
*DUMP-LOG ↵
```

The default output file is the terminal; the default file type is :LIST.

3.13 ENABLED-TRAPS

This command is only on the ND-500 Debugger.

All enabled traps are listed on the terminal.

```
*ENABLED-TRAPS ↵
11 INVALID OPERATION
12 DIVISION BY ZERO
14 FLOATING OVERFLOW
16 ILLEGAL OPERAND VALUE
26 ILLEGAL INDEX
27 STACK OVERFLOW
28 STACK UNDERFLOW
29 PROGRAMMED TRAP
30 DISABLE PROCESS SWITCH TIMEOUT
31 DISABLE PROCESS SWITCH ERROR
32 INDEX SCALING ERROR
33 ILLEGAL INSTRUCTION CODE
34 ILLEGAL OPERAND SPECIFIER
35 INSTRUCTION SEQUENCE ERROR
36 PROTECT VIOLATION
*
_
```

See also the commands LOCAL-TRAP-DISABLE and LOCAL-TRAP-ENABLE.

3.14 EXIT

Returns control to SINTRAN on the ND-100, and to the ND-500 MONITOR on the ND-500.

3.15 FIND-SCOPE <program address>

This command finds the module or routine, and the line number, that correspond to the specified program address. It updates the scope accordingly. The current scope status is displayed.

The difference between FIND-SCOPE and SCOPE (see page 54) is that FIND-SCOPE needs a program address, while SCOPE has a module, routine or line number for its parameter.

3.16 FORMATS-DISPLAY <formats (A,D,F,H,O or combinations)>

Set format(s) for the DISPLAY command. This will not affect the format for the LOOK-AT commands. The default (initial) format setting is D. You obtain it by giving an empty format specification.

Here is what the codes mean:

A = Alphanumeric	H = Hexadecimal
D = Decimal	O = Octal
F = Floating point	

An example is given on page 27.

3.17 FORMATS-LOOK-AT <formats (A,D,F,H,I,O or combinations)>

Set format(s) for the LOOK-AT commands. The default (initial) format setting is obtained by giving an empty format specification.

Here is what the codes mean:

A = Alphanumeric	I = Instruction
D = Decimal	H = Hexadecimal
F = Floating point	O = Octal

An example is given on page 40.

3.18 GUARD <item or address> (<(*not*) low (: high)>)

This command specifies a data item or location to be checked for modifications. If the contents of the item or location are outside the permitted range, a guard violation occurs and control is passed to the Debugger.

USE LOG-LINES or LOG-CALLS before GUARD on the ND-100.

```

*GUARD x 0 : 10 ↵
*RUN ↵
GUARD VIOLATION AT MAIN.55
*DISPLAY X ↵
x=11
*RUN ↵
  
```

0 to 10 is the permitted range.

This will break every time x has a value outside the range 0 to 10.

Any data item which has a single value (PLANC types POINTER, INTEGER, REAL, ENUMERATION, BOOLEAN, and SET) is legal. Array elements (packed and unpacked) and record components (packed and unpacked) may also be specified. Composite items (arrays and records) are illegal.

If an address is given, the location at that address, taken as a single signed integer (ND-100, 16 bits; ND-500, 32 bits), is checked for modifications.

The permitted range is specified by n, where low <= n <= high. If the operator NOT appears, however, the permitted range is n < low or n > high.

```

*GUARD K NOT 50:70 ↵
*RUN ↵

GUARD VIOLATION AT LOOPS.9
*DISPLAY K ↵
K=60
*GUARD ↵
*RUN ↵
  
```

This removes GUARD.

If only low is specified, then high is set equal to low. If no range is specified, the permitted range becomes the single value of the current contents of the specified address. Permitted range, low:high, cannot be specified for PLANC SETs.

To continue, use RUN or STEP. If you want to remove GUARD, use it without parameters.

On the ND-100, the amount of checking is determined by using the LOG-CALLS or the LOG-LINES command. LOG-CALLS specifies that checking is to be performed at the entry to the routines. LOG-LINES means that checking is to be performed on every logged line. If a program area is specified, checking is performed only in the specified program area.

On the ND-500, checking is done by the hardware throughout the entire program.

3.19 HELP <command name>

The HELP command lists available commands on the terminal. Only those commands that have <command name> as a subset are listed. If <command name> is null, then all available commands are listed. Each command is followed by a parameter list, if it has any. Required parameters are enclosed in angular brackets: < >. Optional parameters are enclosed in parentheses and angular brackets: (< >).

3.20 INCLUDE-COMMANDS <file name>

This will include the commands from the file. For example, you might want to create a file called MACROS:SYMB with the following contents:

```
macro std
formats-display
macro dho
formats-display d h o
macro x
display;run
macro y
x;x;x;
display
```

Then you can do the following to include your macros and ensure that they have been defined properly:

```
*INCLUDE-COMMANDS MACROS ↵
*MACRO STD
BODY: *MACRO DHO
BODY: *MACRO X
BODY: *MACRO Y
BODY: *DISPLAY
ERRCODE=0          STRING          I= 0
K=0                X= 0.0           IMAX= 0
*MACRO ↵
NAME: ↵
Y      X;X;X;
X      DISPLAY;RUN
DHO    FORMATS-DISPLAY D H O
STD    FORMATS-DISPLAY
*
_
```

All the macros you have defined on the file MACROS:SYMB are now available to you.

3.21 INVOKE <routine> (< (parameter,...,parameter) >)

This command is used to call routines. Parameters will not be checked. You must ensure that you call the routine with the correct number of parameters, and that the actual and formal parameters are compatible.

If the routine is a FORTRAN subroutine or a PLANC standard routine, all items that have a defined address (when the INVOKE command is executed) are legal. Constants are allowed. If the routine is a normal PLANC routine, simple variables (ENUMERATION, BOOLEAN, POINTER and INTEGER) and records are legal.

3.22 LOCAL-TRAP-DISABLE (<trap conditions>)

This command is only on the ND-500 Debugger. Several traps can be specified on the same line, separated by spaces or commas. Always use hyphens between words in trap names! Abbreviations are accepted.

Example:

```
*LOCAL-TRAP-DISABLE A-T-F A-T-R FL-UND ↵
```

ADDRESS-TRAP-FETCH, ADDRESS-TRAP-READ, and FLOATING-UNDERFLOW are disabled.

If (<trap conditions>) is empty, all traps are disabled. If (<trap conditions>) is HELP, all available trap conditions are listed on the terminal.

In the following example, the program LOOPS divides by zero. By disabling trap 12, "Division by zero", control will not go to the Debugger when a number is divided by zero in the program.

```
*RUN ↵
```

```
DIVISION BY ZERO AT LOOPS.4
```

```
*LOCAL-TRAP-DISABLE HELP ↵
```

```
9  OVERFLOW
11 * INVALID OPERATION
12 * DIVISION BY ZERO
13  FLOATING UNDERFLOW
14 * FLOATING OVERFLOW
15  BCD OVERFLOW
16 * ILLEGAL OPERAND VALUE
17  SINGLE INSTRUCTION TRAP
18  BRANCH TRAP
19  CALL TRAP
20  BREAKPOINT INSTRUCTION TRAP
21  ADDRESS TRAP FETCH
22  ADDRESS TRAP READ
23  ADDRESS TRAP WRITE
24  ADDRESS ZERO ACCESS
25  DESCRIPTOR RANGE
26 * ILLEGAL INDEX
27 * STACK OVERFLOW
28 * STACK UNDERFLOW
29 * PROGRAMMED TRAP
```

The trap conditions with an asterisk (*) are disabled. The others are enabled.

```
30 * DISABLE PROCESS SWITCH TIMEOUT
31 * DISABLE PROCESS SWITCH ERROR
32 * INDEX SCALING ERROR
33 * ILLEGAL INSTRUCTION CODE
34 * ILLEGAL OPERAND SPECIFIER
35 * INSTRUCTION SEQUENCE ERROR
36 * PROTECT VIOLATION
*LOCAL-TRAP-DISABLE DIVISION-BY-ZERO
*RUN ↵
```

We could have written *L-T-D DIV since DIV is an unambiguous abbreviation of DIVISION-BY-ZERO.

3.23 LOCAL-TRAP-ENABLE (<trap conditions>)

This command is for the ND-500 only. Several traps can be specified on the same line separated by spaces or commas. Always use hyphens between words in trap names!

If (<trap conditions>) is empty, all default traps are enabled. If (<trap conditions>) is HELP, all available trap conditions are listed on the terminal.

Example:

```
*LOCAL-TRAP-ENABLE PROT-VIOL. I-I-C ↵
```

The PROTECT-VIOLATION and ILLEGAL-INSTRUCTION-CODE traps are enabled.

3.24 LOG-CALLS <program area>

This command logs all routine calls in a cyclic buffer. This buffer can be inspected by means of the DUMP-LOG command (see page 28). The buffer can hold a maximum of 200 entries.

```
*LOG-CALLS,... ↵
*BREAK,PRINT 5 ↵
*RUN ↵
BREAK AT PRINT.21
*DUMP-LOG ↵
LOOPS PRINT PRINT REDUCE REDUCE PRINT
REDUCE REDUCE PRINT REDUCE REDUCE PRINT
*EXIT ↵
```

If a module or routine is specified, all routines that are called in the specified module or routine are logged.

This command is normally used in conjunction with other commands. The next sections show some examples:

3.24.1 LOG-CALLS and CHECK-OUT-MODE

This is how you can log all the routines in your program that are not called:

```
*LOG-CALLS,... ↵
*CHECK-OUT-MODE ↵
... (BREAK and RUN)
*DUMP-LOG ↵
```

You can also specify an area:

```
*LOG-CALLS,... ↵
*CHECK-OUT-MODE MAIN.20:MAIN.40 ↵
... (BREAK and RUN)
*DUMP-LOG ↵
```

Any routine not called in the area MAIN.20 to MAIN.40 will be logged.

You can list all routines by using DUMP-LOG immediately after LOG-CALLS and CHECK-OUT-MODE:

```
*LOG-CALLS,... ↵
*CHECK-OUT-MODE ↵
*DUMP-LOG ↵
LOOPS.6 PRINT.21 REDUCE.34
*_
```

That may be useful when you start debugging your program.

3.24.2 LOG-CALLS and GUARD

You only need to use LOG-CALLS or LOG-LINES before GUARD on the ND-100.

```
*LOG-CALLS,... ↵
*GUARD CEVAL ↵
*RUN ↵
```

Every time a routine is called, the Debugger will check to see if the value of CEVAL has changed.

3.24.3 LOG-CALLS and STEP

```
*LOG-CALLS MAIN.50 : MAIN.70 ↵
*STEP ↵
```

Each CR (Carriage Return) will bring you to the next routine call in the area MAIN.50 to MAIN.70, and each routine call will be logged.

3.25 LOG-LINES <program area>

This commands logs all executed line numbers in a cyclic buffer. This buffer can be inspected by means of the DUMP-LOG command (see page 28). The buffer can hold a maximum of 200 entries.

```
*LOG-LINES,... ↵
*BREAK PRINT 5 ↵
*RUN ↵
BREAK AT PRINT.21
*DUMP-LOG ↵
LOOPS.6 7 8 9 10 11 14 12 13
PRINT.21 22 23 26 23 26 23 26 23
26 27 28 29 LOOPS.14 12 13
PRINT.21 22 23 24 REDUCE.34 35 36 37
(etc.)
*
_
```

If a module or routine is specified, only the lines executed in the specified module or routine are logged.

LOG-LINES is normally used in conjunction with other commands. Here are some examples:

3.25.1 LOG-LINES and CHECK-OUT-MODE

```
*LOG-LINES PRINT ↵  
*CHECK-OUT-MODE PRINT ↵  
*BREAK 14 ↵  
*RUN ↵  
BREAK AT LOOPS.14  
*DUMP-LOG ↵  
PRINT.24  
*  
—
```

The only line in PRINT that was not executed was line 24.

3.25.2 LOG-LINES and GUARD

You only need to use LOG-CALLS or LOG-LINES before GUARD on the ND-100.

```
*LOG-LINES CALC ↵  
*GUARD CEVAL ↵  
*RUN ↵
```

The Debugger will tell you if the value of CEVAL changes anywhere in the routine CALC.

3.25.3 LOG-LINES and STEP

```
*LOG-LINES MAIN.50 : MAIN.70 ↵  
*STEP ↵
```

Each CR (Carriage Return) will bring you to the next line in the area MAIN.50 to MAIN.70 and each line number will be logged.

Note:

We advise you NOT to use LOG-LINES on your entire program if you have a large program. Specify part of your program instead. Otherwise you will slow down program execution considerably.

3.26 LOOK-AT-DATA <data address> (<count>) (<output file>)

This command and the related commands LOOK-AT-PROGRAM, LOOK-AT-REGISTER and LOOK-AT-STACK enable data locations, program locations and registers to be inspected and modified.

```
*LOOK-AT-DATA 320 10 ↵
```

The data in the addresses 320 to 332 (octal!) will be printed. If you do not specify count, one location will be output.

If you are employing an alternative page table from a 1-bank program, addresses within the alternative page table can be accessed by specifying addresses in the range 200,000B to 377,777B. (ND-100 only.)

```
*LOOK-AT-DATA 320 1000 "DATA:LIST" ↵
```

In the above example, control returns to the Debugger when the 1000 locations have been output. If you send the output to your terminal, control remains within the LOOK-AT command, and you may use the subcommands described below.

CR (Carriage Return) causes an advance to the next item without changing the contents of the current item. All subcommands are terminated by CR. Printing a dot (.), a semicolon (;), or EXIT returns you to the Debugger:

```
*FORMATS-LOOK-AT O H ↵
*LOOK-AT-DATA ADDR(CURRENT.NAME) ↵
D 001010B: 000142B 0062H
D 001011B: 067542B 6F62H EXIT ↵
*
_
```

Note that the contents of each location is printed in the format(s) specified by the FORMATS-LOOK-AT command.

Below you will find the special notation that is available when you have given a LOOK-AT command. Subcommands are listed in the next section.

HELP <command name> Lists available LOOK-AT subcommands on the terminal.

EXIT or ;
 or . Returns control to the Debugger's command processor.

m Deposits the value of the expression m (which can also be a string constant) in the current location and advances to the next location.

m,n/ This prints n locations, starting with the contents of location m. See the example on page 46.

Here are some examples that illustrate the notation:

```

*DISPLAY ↵
ERRCODE=0      I=0      K= 5
KTAL= 0
*LOOK-AT-DATA ADDR(I)... ↵
D 000057B: 000000B      0      EXIT ↵
*LOOK-AT-DATA 125B... ↵
D 000125B: 000011B      9      ↵
*LOOK-AT-DATA ADDR(K)... ↵
D 000060B: 000005B      5      20 ↵
D 000061B: 000000B      0      ↵
*LOOK-AT-PROGRAM 30B ↵
P 000030B: 120606B -24186 ! MPY ,B - 172 153000B ↵
P 000031B: 004610B 2440 STA ,B - 170 ↵
*LOOK-AT-PROGRAM 30B ↵
P 000030B: 153000B -10752 y MON ↵
*LOOK-AT-DATA ADDR(K)... ↵
D 000060B: 000024B      20      ↵

```

In the above example, three ways of exiting were shown (., ↵, and EXIT), and the value 20 was stored in data address 60B. The value 153000B replaced 120606B in program address 30B.

Here is the special notation to be used with the slash (/) command:

- m/ Take the value of m as the next address and display this location.

- / Take the contents of the current location as the next address and display this location (indirection).

- // (Restricted for the moment to the ND-100.) When in program mode only, the second slash will cause the current word to be interpreted as an instruction. The operand of the instruction is taken as the next location.

- m,/ Take the value of m as the next address and display n locations, where n is the last count entered.

- ,n/ Take the contents of the current location as the next address and display n locations.

- ,/ Take the contents of the current location as the next address and display n locations, where n is the last count entered.

Here are some examples:

```

*SET K = 11B ↵
*LOOK-AT-PROGRAM K ↵
P 000011B: 171400B -3328 s SAX 0 11B+100B/ ↵
P 000111B: 000102B 66 B STZ * 102 / ↵
P 000102B: 000064B 52 4 STZ * 64 / ↵
P 000064B: 024130B 10328 (X LDD * 130 234B/ ↵
P 000234B: 134345B -18203 8e JPL * - 33 // ↵
P 000201B: 146147B -13209 Lg COPY SL DX ; ↵

```

3.27 LOOK-AT Subcommands

The following subcommands apply to LOOK-AT-DATA, LOOK-AT-PROGRAM, LOOK-AT-REGISTER, and LOOK-AT-STACK.

Here is how you list the subcommands:

```
*FORMATS-LOOK-AT 0 ↵
*LOOK-AT-DATA ↵
DATA ADDRESS: ADDR(I) ↵
D 01000000030B: 12016007106B HELP ↵
COMMAND NAME: ↵
  BYTE (ND-500 only)
  CODE <INSTRUCTION>
+DATA
  DOUBLE-FLOATING
  DOUBLE-WORD (ND-100 only)
+EXIT
  EXTRA-FORMATS <FORMATS A, D, F, H, I OR O>
  FLOATING
  FORMATS <FORMATS A, D, F, H, I OR O>
  HALF-WORD (ND-500 only)
+HELP <COMMAND NAME>
NEXT
PREVIOUS
+PROGRAM
REGISTER
WORD
```

DATA, PROGRAM, REGISTER, and STACK

Within a LOOK-AT command one can go directly to one of the other LOOK-AT commands by using one of these subcommands.

Example:

```
*FORMATS-LOOK-AT 0 ↵
*LOOK-AT-DATA 41B ↵
D 01000000041B: 000000B PROGRAM ↵
P 01000000041B: 000B REGISTER ↵
P: 01000000004B STACK ↵
PREVIOUS B: 00000000000B
RETURN ADDRESS: 33402000000B
NEXT B: 00463600000B
AUX: 00035147001B
NO. OF PARAMETERS: 22406407130B
D 00000000024B 24B: 03200253775B DATA ↵
*
_
```

NEXT and PREVIOUS

Within the LOOK-AT-STACK command these subcommands can be used to move between the stack frames.

See the example on page 48.

WORD, FLOATING, and DOUBLE-FLOATING (ND-100 and ND-500)

DOUBLE-WORD (ND-100 only)

BYTE and HALF-WORD (ND-500 only)

With the LOOK-AT commands one can display values in units of several different sizes. These subcommands specify the desired size.

Here are some examples from an ND-500 program:

```
*FORMATS-LOOK-AT H ↵
*LOOK-AT-DATA ADDR(I) ↵
D 01000000030B: 50380E46H BYTE ↵
D 01000000030B: 50H ↵
D 01000000031B: 38H ↵
D 01000000032B: 0EH ↵
D 01000000033B: 46H 30B/ ↵
D 01000000030B: 50H WORD ↵
D 01000000030B: 50380E46H ↵
D 01000000034B: 00000000H 30B/ ↵
D 01000000030B: 50380E46H HALF-WORD ↵
D 01000000030B: 5038H ↵
D 01000000032B: 0E46H i ↵
*
```

In the following example, X is declared as real in a PLANC-500 program; Y8 is declared as REAL8:

```
*DISPLAY ↵
X= 1.25600      Y8= 1.2560000000000000      EXP= 3.14000
NAME(1:60)
*FORMATS-LOOK-AT H D ↵
*DISPLAY ADDR(Y8) ↵
ADDR(Y8)=01000000034B
*LOOK-AT-DATA ADDR(X) ↵
D 01000000030B: 4050624DH 1079009869 ↵
D 01000000034B: 4050624DH 1079009869 30B/ ↵
D 01000000030B: 4050624DH 1079009869 FLOATING ↵
D 01000000030B: 4050624DH 1079009869 1.25600
D 01000000034B: 4050624DH 1079009869 1.25600 34B/ ↵
D 01000000034B: 4050624DH 1079009869 1.25600 FORMATS H ↵
D 01000000034B: 4050624DH DOUBLE-FLOATING ↵
D 01000000034B: 4050624DH,D2F1A9FCH 1.2560000000000000 i ↵
*
```

FLOATING is useful for inspecting the values of real numbers. DOUBLE-FLOATING is only helpful for real numbers stored in 2 words (32 bits).

FORMATS <formats A, D, F, H, I or O>

EXTRA-FORMATS <formats A, D, F, H, I or O>

In FORMATS and EXTRA-FORMATS, the abbreviations have the following meaning:

A = Alphanumeric	I = Instruction
D = Decimal	H = Hexadecimal
F = Floating point	O = Octal

The formats set by means of the FORMATS-LOOK-AT command may be temporarily changed with these subcommands. The FORMATS subcommand is similar to the FORMATS-LOOK-AT, except that the formats are valid only until exit from LOOK-AT. The EXTRA-FORMATS command is similar to the FORMATS command, except that the specified formats are added to those already set.

3.28 LOOK-AT-PROGRAM <program address> (<count>) (<output file>)

Inspect and modify program locations. This command is similar to the LOOK-AT-DATA command, except that I format (symbolic instructions) is enabled as default. Decimal addresses are default, so remember to write B after octal addresses!

In the following example, the program is changed so that the number 0 will be printed on your terminal:

```
*LOOK-AT-REGISTER P... ↵
P: 000011B 9 ↵
*LOOK-AT-PROGRAM 11B ↵
P 000011B: 171400B -3328 s SAX 0 CODE SAT 1 ↵
P 000012B: 135032B -17894 : JPL I * 32 CODE SAA 60 ↵
P 000013B: 000004B 4 STZ * 4 CODE MON 2 ↵
P 000014B: 000051B 41 ) STZ * 51 CODE MON 65 ↵
P 000015B: 000012B 10 STZ * 12 CODE MON 0 ↵
P 000016B: 000004B 4 STZ * 4 11B,5/ ↵
P 000011B: 171001B -3583 r SAT 1
P 000012B: 170460B -3792 q0 SAA 60
P 000013B: 153002B -10750 V MON OUTBT
P 000014B: 153065B -10699 V5 MON QERMS
P 000015B: 153000B -10752 V MON ↵
*RUN ↵
0
@_
```

Here is a very short example from a ND-500 program:

```
*LOOK-AT-PROGRAM ↵
PROGRAM ADDRESS: 120B ↵
P 01'120B: W LOOPI B.024B:S,B.030B:S,-060B-->01'40B CODE ↵
INSTRUCTION: W LOOPI B.030B:S,B.024B:S,40B ↵
P 01'124B: RET 120B/ ↵
P 01'120B: W LOOPI B.030B:S,B.024B:S,-060B-->01'40B ↵
*_
```

Note that we abbreviated a few addresses with an apostrophe to save space. 01000000120B and 01'120B both mean segment number 1, address 120B.

Some examples of LOOK-AT-PROGRAM are also given in the previous section on page 41 and 42.

3.29 LOOK-AT-REGISTER <register name> (<count>) (<output file>)

Inspect and modify CPU registers. This command is similar to the LOOK-AT-DATA command.

```
*LOOK-AT-REGISTER P 9 ↵
P:      003216B      1678
X:      000030B        24
T:      002734B     1500 \
A:      000001B         1
D:      000024B        20
L:      000764B      500 t
S:      000140B        96 `
B:      000216B      142
W:      000002B         2  EXIT ↵
*
_
```

On the ND-100, W is the current alternative page table. Note that its value is 2 above. Its value must be 2 or 3.

3.30 LOOK-AT-STACK <B register> (<count>) (<output file>)

Inspect and modify locations in the stack. This command is similar to the LOOK-AT-DATA command, except that both absolute and relative addresses are displayed. Locations in the stack header are given by name rather than by address.

Addresses entered with the slash (/) command are taken as relative to the B register of the current stack frame that is being examined.

In the following example, a FORTRAN program calls the subroutine print which in turn calls the subroutine reduce. Print has 3 parameters, reduce has 2.

```

*BREAK REDUCE ↵
*RUN ↵
subroutine print
      5 reduced          4 times
      .5500000E+01      5
subroutine print
BREAK AT REDUCE.34
*FORMATS-LOOK-AT 0 ↵
*LOOK-AT-STACK.. ↵
PREVIOUS B:           01000000314B
RETURN ADDRESS:       01000000222B
NEXT B:               01000000224B
AUX:                  00000000000B
NO. OF PARAMETERS:   00000000002B
D 01000000604B        24B: 01000000074B PREV ↵
PREVIOUS B:           01000000024B
RETURN ADDRESS:       01000000114B
NEXT B:               01000000224B
AUX:                  00000000000B
NO. OF PARAMETERS:   00000000003B
D 01000000340B        24B: 01000000060B PREV ↵
PREVIOUS B:           00000000000B
RETURN ADDRESS:       00000000000B
NEXT B:               01000000224B
AUX:                  00000000000B
NO. OF PARAMETERS:   00000000000B
D 01000000050B        24B: 00000000010B EXIT ↵
*
_

```

3.31 MACRO <name> <body>

This builds macro commands composed of one or more basic commands and other macro commands. The macro name can be any character string and is terminated by a space or a comma. Only the first eight characters are significant. The rest of the line following the macro name is taken as the macro body. The macro body is not terminated by semicolon, thus several commands can be included in the same macro body.

If the macro body is empty, the corresponding macro is erased.

If the macro name is empty, all the currently defined macros are displayed on the terminal:

```
*MACRO ↵
NAME: X ↵
BODY: DISPLAY; RUN ↵
*MACRO ↵
NAME: Y ↵
BODY: X;X;X;X ↵
*MACRO, ↵
Y      X;X;X;X
X      DISPLAY; RUN
*
_
```

No name and no body will
list the macros you have
defined.

A macro parameter is referenced in the macro body as "n", where n is a one-digit number (1 - 9). See the example below.

A macro name is used in the same way as a command name. It can be abbreviated in the same way, too. However, macro parameters are not asked for if omitted, but taken to be empty strings when the macro is expanded. A macro name can also be used as a LOOK-AT subcommand.

Examples of MACRO:

```
*MACRO DX,DISPLAY P.NAME(0),P.NAME(1);SET P,P.LINK ↵
*SET P,ELEM ↵
*DX ↵
P.NAME(0)= 101B 65
P.NAME(1)= 102B 66
*
_
```

The first parameter
you give will be
inserted here.

```
*MACRO DY,DISPLAY P.NAME("1") ↵
*DY 5 ↵
P.NAME(5)= 106B 70
*
_
```

Here is a useful macro to define:

```
*MACRO ↵
NAME: VIEW ↵
BODY: LOG-CALLS,...;CHECK-OUT-MODE:DUMP-LOG ↵
```

Try it when you start the Debugger. You will get a good overview of your program.

Macros are useful in programs with records and pointers:

```
*MACRO ↵
NAME: SHOW ↵
BODY: DISPLAY "1".NAME,"1".LEFT,"1".RIGHT ↵
*SHOW CURRENT ↵
CURRENT.NAME=bob
CURRENT.LEFT=NIL
CURRENT.RIGHT=001032B
*SHOW CURRENT.RIGHT ↵
CURRENT.RIGHT.NAME=else
CURRENT.RIGHT.LEFT=NIL
CURRENT.RIGHT.RIGHT=001054B
*
_
```

3.32 PLACE <file name> (<W>)

This command exists in the ND-100 Debugger only. It reads a program from a program file (:PROG) into the user's memory (background segment). The program counter is set to the start address, the status register to zero, and the alternative page table to 2. The current alternative page table may be examined by LOOK-AT-REGISTER W. The scope is set according to the start address.

If you use the optional parameter W, you get write access to your :PROG file. Each update you do with LOOK-AT-DATA or LOOK-AT-PROGRAM will be performed on your :PROG file at the same time. Use W with care!

```
*PLACE TEST W ↵
FORTRAN PROGRAM.  SQRS.1
*
_
```

See an example of this on page 56.

3.33 PROGRAM-INFORMATION

The command is relevant to the ND-100 only. It lists the following information from the program file:

```
start address
restart address
lower and upper bounds for the program and data
lower and upper bounds for debug information
```

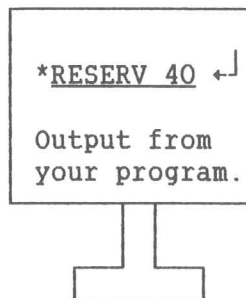
Example:

```
*PLACE TEST ↵
FORTRAN PROGRAM.  SQRS.1
*PROGRAM-INFORMATION ↵
START, RESTART:    000011B,  000011B
PROGRAM, DATA:    000000B - 035065B
DEBUG-INFORMATION: 000000B - 000063B
*
_
```

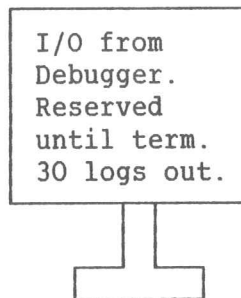
3.34 RESERVE-TERMINAL <logical device number>

People who debug screen-handling programs may prefer to use two terminals while they debug. By giving the `RESERVE-TERMINAL` command, your program output will go to your terminal. At the same time, you can give and get input and output from the Debugger from the terminal you reserve. To free the reserved terminal, you must log out from the other terminal. Here is a picture to illustrate the situation:

Your terminal
(number 30)



Nearby terminal
(number 40)



You start the Debugger from terminal 30, reserve 40 and move there. All input and output to/from the Debugger will be on terminal 40.

When you are finished, you return to terminal 30 and log out to free terminal 40.

3.35 RESET-BREAKS (<program area>)

If no program area is specified, all breakpoints and step points set with BREAK, CHECK-OUT-MODE, LOG-CALLS or LOG-LINES are reset. A breakpoint set by means of the BREAK-ADDRESS command is reset only if it is the first instruction in a line.

If a program area is specified, the breakpoint at that address is removed.

Here is how you remove all breakpoints and execute your program:

```
*RESET-BREAKS ↵
*RUN ↵
```

Here is how you normally remove a breakpoint:

```
*LOG-LINES LOOPS ↵
*GUARD I ↵
*RUN ↵

GUARD VIOLATION AT LOOPS.12
*DISPLAY ↵
ERRCODE=0      STRING      I= 5
K=60           X= 1.10000000 IMAX= 10
*RESET-BREAKS ↵
*BREAK PRINT ↵
*RUN ↵
```

You may remove specific step points by specifying a program area:

```
*LOG-LINES 8 ↵
*LOG-LINES 22 ↵
*LOG-LINES CALC ↵
*STEP ↵
...
*STEP ↵
BREAK AT CALC.8
*RESET-BREAKS 8 ↵
*RESET-BREAKS CALC.1:CALC.100 ↵
*STEP ↵
```

3.36 RUN (<program address>)

If no program address is specified, execution is resumed from the current line. RUN works exactly as CONTINUE. If you specify a program address, control is transferred directly to that address.

If you want to start execution from line 15 in XYZ, do this:

```
*RUN ADDR(XYZ.15) ↵
```

Execution will continue until the breakpoint is reached or a GUARD violation occurs. Step points will be skipped.

3.37 SCOPE (<module, routine or other item>)

This command finds the specified module or routine and updates the scope accordingly. The current scope status is displayed. If no module or routine is specified, the current scope is not affected, but it is displayed.

```
*ACTIVE-ROUTINES ↵
PRINT.17 CALLED FROM LOOPS.14
LOOPS.1
*DISPLAY ↵
ERRCODE=0      I=5      X= 5.50000000
STRING
K=0      INTX=0
*SCOPE LOOPS ↵
LOOPS.1
*DISPLAY ↵
ERRCODE=0      STRING      I= 5
J= 20
K=60      X= 5.50000000      IMAX= 10
*
_
```

The difference between FIND-SCOPE and SCOPE is that FIND-SCOPE needs a program address, while SCOPE has a module, routine or line number for its parameter.

3.38 SEGMENT-INFORMATION

This command is relevant to the ND-500 only. Information about the currently active segments is displayed on the terminal in the form of a table as in the example below:

```
*SEGMENT-INFO ↵
SEGMENT  FILE C1 C2 NAME
PSEG  1 1777B  2  0 (PACK-TWO:DEBUG)SEGMENT-D001-S01
DSEG  1 1776B
LINK  1 1775B  3
PSEG 26      OB      (SYM-DEB)DEBUGGER
DSEG 26      OB
LINK 26      OB
PSEG 30      OB      (PACK-REM:DOMAINS)FORTRAN-LIB-H00
DSEG 30      OB
LINK 30      OB
```

When the Debugger starts, a monitor call to the 500 Monitor produces a list of all active segments. The list may contain a FORTRAN library segment. SEGMENT-INFORMATION can be used to obtain segment numbers for use in the ATTACH-SEGMENT command.

3.39 SET <variable> (=) <value>

This command is used to set program variables. Any variable reference which has a defined address can be set.

For example:

```
*SET XX 10 ↵
*SET KK.LL(37)=KK.LL(37) + 2 ↵
*SET STRING(3)='TEXT' ↵
```

It is possible to set an array equal to an array, for instance, a PLANC array equal to a FORTRAN array. The truncation is as for PLANC if the dimensions differ. A real array can be set equal to an integer array, a packed array can be set equal to an unpacked array, and vice versa. An array may also be set to a constant; if the array is real or integer, then the constant will take the form of the array, as in:

```
*SET INTEGER ARRAY = 3.142 ↵
```

Here the constant is truncated to 3 before assignment. The rules for arrays also apply to subarrays.

In addition to constants, values may be strings, bytes, or FORTRAN characters. For example, an element of a bytes array can be set equal to a string.

3.40 STACK-INSTRUCTIONS (<low>) (<high>)

This command will increase the speed at which a :PROG file executes by up to 20%. In order to make those changes permanent, write W after you write PLACE and your file name. You must have an ND-100 CX computer.

Here is an example of how a chess program was made faster:

```
@FILE-STAT CHESS:PROG,... ↵
FILE 5 : (PACK-TWO:DEBUG)CHESS:PROG;1
...
OPENED 33 TIMES
CREATED 09.19.24 AUGUST 23, 1984
OPENED FOR READ 10.34.07 NOVEMBER 22, 1984
OPENED FOR WRITE 10.34.07 NOVEMBER 22, 1984
66 PAGES , 280576 BYTES IN FILE

@DEBUGGER-BD ↵
ND-100 SYMBOLIC DEBUGGER. VERSION D.
*PLACE CHESS W ↵
*STACK-INSTRUCTIONS ↵
1202 MICROINSTRUCTIONS SUBSTITUTED
*EXIT ↵

@FILE-STAT CHESS:PROG,... ↵
FILE 5 : (PACK-TWO:DEBUG)CHESS:PROG;1
...
OPENED 34 TIMES
CREATED 09.19.24 AUGUST 23, 1984
OPENED FOR READ 10.38.23 NOVEMBER 22, 1984
OPENED FOR WRITE 10.38.23 NOVEMBER 22, 1984
66 PAGES , 280576 BYTES IN FILE
```

The instructions will be adapted to the ND-100 microinstruction set. This program was found to execute 8% faster after the above operation was performed.

3.41 STEP (<count>)

You will continue to the next step point. Step points can be defined by LOG-LINES or LOG-CALLS. The count parameter specifies the number of steps to take.

When you reach a step point, the Debugger stops and outputs the current routine and line. If you then type Carriage Return, you will continue the number of steps specified in the count parameter. Otherwise, you may give some commands and then use STEP to go to the next step point.

Example:

```
*LOG-LINES MAIN.110 ↵
*STEP 10 ↵
MAIN.110
*
_
```

You may trace by writing:

```
*LOG-LINES... ↵
*STEP 0 ↵
```

Your program will execute until it is finished, and every line executed will be listed.

If you want to step instruction by instruction, use STEP -1:

```
*LOG-LINES... ↵
*STEP -1 ↵
SQRS.000012B JPL I * 36 * ↵
034501B JXZ * 4 * ↵
034505B LDA I * - 24 * ↵
034506B SAT 3 * ↵
034507B SKP DA UEQ ST * ↵
034510B JMP * 6 * ↵
034516B BSET ZRO SSPT *
_
```

Each Carriage Return will advance you to the next instruction.

CHAPTER 4

SYMBOLIC DEBUGGER PARAMETERS

4 SYMBOLIC DEBUGGER PARAMETERS

This chapter explains the arguments that can be used in command parameters. Here is a list that contains most of the possibilities:

- Numeric constants can be expressed as decimal, octal, hexadecimal, binary and real numbers.
- Single-character constants.
- String constants.
- Expressions involving the above types and the operators +, -, SHIFT, *, /, **, .(dot), IND and ADDR. In conditional expressions, >, >=, <, <=, =, and <> are also available.

Note: Array indexing and subarray specification are also available.

- Named items, such as modules, routines, labels, lines, etc.
- Program area.
- Program address.
- Data address.
- Format specifier.
- File name.

Each of the above categories will be explained on the following pages.

4.1 Numeric Constants

Constants are used in the DISPLAY and SET command, the LOOK-AT commands, as well as in other commands. There are many ways of expressing numeric constants. Here are 12 ways to write the number 195:

Binary notation:	11000011X	2#11000011#
		2#1100_0011#
Octal notation:	303B	8#303#
Decimal notation:	195	10#195#
Floating point:	1.95E2	10#1.95#E2
Hex notation:	0C3H	16#C3#

The numbers followed by X, B, D, E, or H illustrate the method of writing a number followed by a radix specifier. The specifiers allowed and their meanings are:

<u>suffix</u>	<u>number system</u>	<u>radix</u>
X	binary numbers	base 2
B	octal numbers	base 8
D	decimal numbers	base 10
E	floating point	base 10
H	hexadecimal numbers	base 16

In order to avoid conflicts with identifiers, a hexadecimal constant must always start with a decimal digit (e.g., the constant C3 must be written as 0C3H).

A real constant must contain a decimal point or the letter E. An exponent may be specified, preceded by the letter e. A constant may be preceded by a sign. You should not use the suffixes for real constants.

Here are some examples:

.3 -3. 3.3 3E 3E5 3.E-5

The numbers 10#195#, 8#303#, etc., on page 61 were written by using the form:

base#number#exponent

This is a feature borrowed from the programming language ADA. The # appears as the number sign on some terminals, and as the English pound sign (£) on others. Here is an example:

```
*DISPLAY 8#100#E4 ↵
8#100#E4= 2.621440000000000E+05
*DISPLAY 100B * 8 * 8 * 8 * 8 ↵
100B * 8 * 8 * 8 * 8 = 262144
*
_
```

The 8 is the base, 100 is the number, and E4 is the exponent. So 8#100#E4 is equal to $100_8 * (10_8)^4$, that is, 262144 or 1000000B. Note that the exponent is always a base 10 number.

These are all ways
of expressing
the number 123.

→

```
10#123#
10#1.23#E2
8#173#
16#7B#
2#1111011#
2#111_1011#
```

The ADA system lets you express numbers in the bases 2 to 16. Any underline characters () in the number between the number signs (# #) will be ignored. You may write 5000 million as

10#5_000_000_000#

This will reduce your chances of having too few or too many zeros in your number!

4.2 Single-Character Constants

A single-character constant is denoted by a number sign (#) followed by an ASCII character.

Here is an example from a PLANC program. I is an integer, and CODEX is a string whose length is 40.

```
*DISPLAY ↵
I=0          PTRINT=NIL      PTRBYT= (NIL;0:0)
CODEX(1:40)  EXP= 0.0
*SET CODEX=#A; SET I=#Z ↵
*DISPLAY CODEX, I ↵
CODEX=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
I=90
```

Note that the string gets filled with A's, while the integer is assigned the ASCII value of "Z", which is 90.

NOTE:

The Debugger will convert all lowercase strings to uppercase strings.

4.3 String Constants

A string constant is preceded by and terminated with an apostrophe ('). Embedded apostrophes must be represented by a double apostrophe ('').

Here is an example with embedded apostrophes:

```
*DISPLAY CODEX ↵
CODEX=This is a testAAAAAAAAAAAAAAAAAAAAAA
*SET CODEX='Embedded 'quotes' example' ↵
*RUN ↵
Embedded 'quotes' exampleAAAAAAAAAAAAAA
*_
```

4.4 Expressions

You will mainly use expressions for the DISPLAY and BREAK command, and the LOOK-AT commands. Expressions are formed from operators and operands. In conditional expressions, >, <, =, and <> are also available.

Operands may include constants (integer and real), variable names or identifiers, array indexing, subarray specification, record component selection and the dot notation described on page 67. Variable names may be any name from the compiled language, i.e., FORTRAN variables, PLANC identifiers, or COBOL identifiers with hyphens.

The available operators include +, -, SHIFT, *, /, **, IND and ADDR. The operator ** requires an integer exponent.

A hierarchical order of precedence exists for operators when they are evaluated in expressions.

```

**
* /
SHIFT + -
ADDR IND

```

Note that ADDR and IND are higher than "." (dot) when referring to records, but are lower when the dot appears after a routine name. With operators at the same level, evaluation proceeds from left to right.

Examples:

```

*DISPLAY 4 * 2 + 4 ↵
4 * 2 + 4=12
*DISPLAY 4 * 2 + 4 ** 2 ↵
4 * 2 + 4 ** 2=24
*
_

```

In division, if both operands are integers, integer division is performed:

```

*DISPLAY 1/3 ↵
1/3=0
*DISPLAY 1/3.0 ↵
1/3.0= 3.333333333333333E-01

```

IND can be used on any item that is a pointer.

Here is an example of IND and ADDR. In the following example, CURRENT is a PLANC pointer to a record. IND lists all the elements of the record pointed to by CURRENT. By inspecting the data address pointed to by CURRENT, we can also see the area where the record itself is stored.

```
*DISPLAY IND(CURRENT) ↵
IND(CURRENT)= NAME(1:20) RESULT= 2.80000000
LEFT=NIL RIGHT=001032B
*DISPLAY CURRENT ↵
CURRENT=001010B
*LOOK-AT-DATA ADDR(CURRENT) ↵
D 000024B: 001010B 520 ↵
D 001010B: 000142B 98 b ↵
D 001011B: 067542B 28514 ob ↵
*
```

ADDR can be used on any item that has an address.

```
*DISPLAY ADDR(I) ↵
ADDR(I)=01000000100B
*DISPLAY ADDR(PTRBYT) ↵
ADDR(PTRBYT)=01000000110B

*LOOK-AT-DATA ADDR(CODEX) ↵
D 01000000027B: 000000000000B 0
D 01000000033B: 000000000000B 0 ↵
```

Here is an example of how you use SHIFT:

```
*SET DEC = 20 ↵
*DISPLAY DEC SHIFT -1 ↵
DEC SHIFT -1=10
*DISPLAY DEC SHIFT -2 ↵
DEC SHIFT -2=5
*
```

Here are examples of conditional expressions:

```
*BREAK CALC CURRENT <> NIL ↵
*BREAK MAIN.75 X < 0 ↵
```


4.5 Named Items

By named items we mean:

- Modules
- Routines
- Labels
- Lines
- Variables
- Identifiers
- Names
- Statements

In PLANC and SIMULA, a named item is specified by a sequence of names separated by dots (.), corresponding to the static Module/Routine nesting in a program.

Here is an example from PLANC:

```

MODULE MOD1
INTEGER: J
...
    ROUTINE VOID,VOID: ROUT1
    INTEGER: I
    LABEL: RETRY
    RETRY:    I =: ATTEMPTS
    ...
    ENDROUTINE
...
    ROUTINE VOID,VOID: ROUT2
    INTEGER: I
        ROUTINE VOID,VOID: ROUT5
        ...
        ENDROUTINE
    ...
    ENDROUTINE
PROGRAM main
...
ENDROUTINE
ENDMODULE
    
```

In the PLANC example, the various routines can be specified as:

```
MOD1.ROUT1
MOD1.ROUT2
MOD1.ROUT2.ROUT5
```

The two I's can be specified by:

```
MOD1.ROUT1.I and MOD1.ROUT2.I
```

The label RETRY can be specified by:

```
ROUT1.RETRY
```

Line 50 in the main program can be specified by:

```
MAIN.50
```

However, in order to simplify the specifications, the name search is always done according to the "current scope". This means that if you are in MOD1.ROUT2, you can write I, instead of MOD1.ROUT2.I.

The current scope always refers to the point where the last breakpoint occurred (unless the scope is explicitly changed by the FIND-SCOPE or the SCOPE command).

Consider once more the above example and assume the current scope to be: MOD1.ROUT2, that is, inside the body of ROUT2. The name I causes the debugger to find the I declared in ROUT2, while ROUT1.I (or MOD1.ROUT1.I) must be used in order to find the I declared in ROUT1. The name J causes the debugger to search ROUT2 (with no success) and then the entire module where the global J is found.

In FORTRAN, a \$ (dollar) sign is appended by the compiler in front of labels. For example, in:

```
10      GO TO 20
```

the label "10" is known to the debugger as "\$10".

Note therefore that:

	BREAK \$10	breaks at label 10
while	BREAK 10	breaks at line 10

FORTTRAN statement functions cannot be referred to in the debugger (since they are expanded in-line by the compiler at the point of invocation).

4.6 Program Area

We have an argument of the form:

name (<:name>)

where name is a routine, module, label, or line number.

Here is an example:

```
*LOG-LINES ↵
PROGRAM AREA: MAIN.12 : $800 ↵
*_
```

This specifies a program area starting at line 12 and ending at the FORTRAN label 800. Note that the second parameter is optional. If no last item is given, it is considered to be equal to the first.

Here is another example:

```
*LOG-LINES MAIN.110 ↵
*LOG-LINES MAIN.PROCINP.2 : MAIN.PROCINP.10 ↵
*LOG-LINES ENTER ↵
*STEP ↵
```

The program will execute until it encounters line 110 of MAIN, lines 2 to 10 of PROCINP or the label/routine called ENTER.

4.7 Program Address

A program address can be given as an octal number or in the form:

ADDR(routine-name.line-number)

Example:

```
*SCOPE ↵
M1.MPROG.10
*LOOK-AT-PROGRAM 100B ↵
P 01000000100B: W3 =: R.044B:S
P 01000000102B: W STZ R.050B:S ↵
*LOOK-AT-PROGRAM ↵
PROGRAM ADDRESS: ADDR(LIST_PERSON) ↵
P 003160B: 146547B -12953 MG COPY AD1 SL DX ↵
*_
```

When we wrote ADDR(LIST_PERSON), we get the start address of the routine LIST_PERSON.

4.8 Data Address

Data addresses can also be given as octal numbers or in the form:

ADDR(variable)

Example:

```

*LOOK-AT-DATA ↵
DATA ADDRESS: ADDR(CODEX) ↵
D 01000000027B: 000000000000B      0      ↵
D 01000000033B: 000000000000B      0      ↵
*
_

```

Here is an example of using a data address to guard part of a string variable:

```

*LOG-LINES... ↵
*DISPLAY ADDR(NAMN) ↵
ADDR(NAMN)=(000266B;0:7)
*GUARD ↵
ITEM OR ADDRESS: 266B ↵
*RUN ↵
GUARD VIOLATION AT MAIN.62
*
_

```

Here is another example:

```

*LOOK-AT-DATA ADDR(CURRENT.NAME) ↵
D 001010B: 000142B      98  b  ↵
*SET CURRENT.NAME='DEBUGGER' ↵
*LOOK-AT-DATA ADDR(CURRENT.NAME) 5 ↵
D 001010B: 000104B      68  D
D 001011B: 042502B    17730 EB
D 001012B: 052507B    21831 UG
D 001013B: 043505B    18245 GE
D 001014B: 051040B    21024 R  ↵
*
_

```

4.9 Format Specifier

A format specifier (also called a radix specifier) is one or more of the following letters:

O - Octal	A - ASCII
D - Decimal	F - Floating point
H - Hexadecimal	I - Instruction (disassembly)

Here is an example:

```
*FORMATS-DISPLAY O D H ↵
*DISPLAY 8#101# ↵
8#101#=65 41H 101B
*
_
```

4.10 File Name

The file name will not be checked to see if the syntax is correct. A file name is terminated by Carriage Return, space, comma, or semicolon. If the file is already open, the octal file number can be used in place of the file name (octal number without B).

```
@OPEN-FILE TEMP:DATA W ↵
FILE NUMBER IS 000103
@LIST-OPEN-FILES ↵

FILE NUMBER 000100 : (PACK-ONE:SCRATCH)SCRATCH05:DATA;1
FILE NUMBER 000101 : (PACK-TWO:DEBUG)EX:SYMB;1
FILE NUMBER 000102 : (PACK-TWO:DEBUG)FORMAT:TEXT;1
FILE NUMBER 000103 : (PACK-TWO:DEBUG)TEMP:DATA;1

@DEBUGGER-100 TEST ↵

FORTRAN PROGRAM. CONVERT.1 ↵
*DISPLAY ↵
ERRCODE=0      NAMN      DEC= 0      VALUE= 0
COUNTER=0      I=0      BITS(1:16)
*LOOK--DATA ADDR(NAMN) 20 103 ↵
*LOOK--DATA ADDR(BITS) 16 TEST:DATA ↵
*
_
```

In the above example, output is sent to file number 103 (TEMP:DATA) and to TEST:DATA. The numbers 20 and 16 indicate the number of addresses that are written.

CHAPTER 5

EXAMPLES

5 EXAMPLES

5.1 An Example Using FORTRAN-100

Below is a small FORTRAN program which will be used as an example. All the program does is to write the numbers one to six and their squares.

```
ND-100/NORD-10 ANSI 77 FORTRAN COMPILER - 203053D
9:23  3 DEC 1984
SOURCE FILE:  TEST:SYMB
```

```
1*          PROGRAM SQRS
2*          INTEGER I
3*          REAL R
4*          DO 10 I = 1,6
5*          R = REAL(I)*I
6*          WRITE (1,'(X,I5,4X,F12.2)')I,R
7*          10  CONTINUE
8*          END
```

CROSS REFERENCE

The displacement of the data relative to the B register.

I	INTEGER* 2	VARIABLE	-172	2	4	5	5	6
REAL	REAL * 6	INTRINSIC		5				
R	REAL * 6	VARIABLE	-171	3	5	6		
SQRS		PROGRAM		1				
\$10	STATEMENT	LABEL	AT	7	4			

The lines in your program where the variables or references appear.

CALL HIERARCHY

```
1  SQRS
2    1 REAL
```

```
@DEBUGGER ↵
ND-100 SYMBOLIC DEBUGGER.  VERSION D.
*PLACE TEST ↵
FORTRAN PROGRAM.  SQRS.1
*BREAK $10 3 ↵
```

This will break the third
time label 10 is found.

```
*RUN ↵
  1          1.00
  2          4.00
  3          9.00
```

BREAK AT SQRS.7

```
*DISPLAY ↵
ERRCODE=0      I=3
*BREAK $10 I > 5 ↵
```

R= 9.00000000

This will break at label 10
when I is greater than 5.

```
*RUN ↵
  4          16.00
  5          25.00
  6          36.00
```

CONDITIONAL BREAK AT SQRS.7

```
*DISPLAY ↵
ERRCODE=0      I=6
*RUN ↵
```

R= 3.60000000E+01

PROGRAM TERMINATED AT SQRS.8

```
*EXIT ↵
```

5.2 A PLANC Example

Here is the program listing for PLANC-MYPROG:SYMB:

```

1      MODULE M1
2      INTEGER ARRAY : stack (0:100)
3      PROGRAM : myprog
4      INTEGER : i, k, m, sum
5      INISTACK stack
6          1 =: i
7          2 =: k
8      i + k =: k =: m
9      k + m =: sum
10     IF sum > (m + k) THEN
11         output(1, '', 'ERROR')
12     ELSE
13         output(1, '', sum)
14     ENDIF
15     OUTPUT(1, '', 'End of myprog')
16     ENDRoutine
17     ENDMODULE
18     $EOF

```


Here is an example of how you could debug it on the ND-500:

```

@ND-500 ↵
ND-500 MONITOR VERSION C 82.11.22 / 82.12.16
N500:DEBUGGER TEST ↵
PLANC PROGRAM. M1.MYPROG.3
*LOG-LINES... ↵
*CHECK-OUT-MODE ↵
*BREAK 9 ↵
*RUN ↵

BREAK AT MYPROG.9
*DISPLAY ↵
I=1          SUM=0          K= 3          M= 3
*BREAK 15 ↵
*RUN ↵
6
BREAK AT MYPROG.15
*DISPLAY ↵
I=1          SUM=6          K= 3          M= 3
*LOOK-AT-DATA ADDR(SUM) ↵
D 01000000044B: 00000000006B          6
D 01000000050B: 01000000004B 134217732  ↵
*DUMP-LOG ↵
MYPROG.11 12 16

*EXIT ↵

```

Since CHECK-OUT-MODE was used, only the lines not executed are listed.

5.3 Another Example in PLANC

Here is a more substantial program; it sorts an array quickly.

The program that follows consists of two separate modules in two files. It was compiled and loaded as follows:

```
@PLANC-100 ↵
*DEBUG-MODE ↵
*COMPILE SORTER SORTER:LIST SORTER ↵
@PLANC-100 ↵
*DEBUG-MODE ↵
*COMPILE TESTSORT TESTSORT:LIST TESTSORT ↵
@DELETE-FILE SORT-EXAMPLE:PROG ↵
@BRF-LINKER ↵
Brl: PROG-FILE "SORT-EXAMPLE" ↵
Brl: LOAD SORTER ↵
FREE: P 000156-177777      DEBUG 000300
Brl: LOAD TESTSORT ↵
FREE: P 002404-177777      DEBUG 000515
Brl: LOAD PLANC-1BANK ↵
FREE: P 005460-177777      DEBUG 000515
Brl: EXIT ↵
```

Note that if you had both modules on one file and compiled them, you would get different line numbers than in this example.

Here is SORTER:LIST.

```

1  MODULE sorter
2  EXPORT quiksort
3  ROUTINE VOID,VOID (INTEGER2 ARRAY) : quiksort(arr)
4  INTEGER: low, high, marker, top, temp
5  % set up the boundaries of the operation
6  MAXINDEX(arr,1) =: top
7  MININDEX(arr,1) =: marker
8  % marker is the one to place in position
9  % continue with the largest part in this stack element
10 DO WHILE top-marker > 0    % until no more to do
11 % search for position into which to put the marker
12     marker + 1 =: low; top =: high
13 % set search limits
14     DO
15 % find the first in the upper part
16 % which belongs in the lower part
17         DO WHILE arr(marker) < arr(high)
18             high - 1 =: high
19         ENDDO
20 % find the first in the lower part
21 % which should be in the upper part
22         DO WHILE arr(marker) > arr(low)
23             low + 1 =: low
24         ENDDO
25 % might have found right position now
26         WHILE low < high
27 % reverse the elements found in upper and lower parts
28             arr(high)=:temp; arr(low)=:arr(high); temp =:arr(low)
29 % and continue the search on reduced parts
30             low + 1 =: low
31             high - 1 =: high
32         ENDDO
33 % now put the marker in the middle position
34 % isolated by low and high
35         arr(marker) =: temp; arr(high) =: arr(marker);
36         temp =: arr(high)
37 % stack space is saved by recursing
38 % for the larger of the parts only
39         IF high-marker < top-high THEN
40             quiksort(arr(marker : high - 1))
41             high + 1 =: marker
42         ELSE
43             high - 1 =: top
44             quiksort(arr(high + 1 : top))
45         ENDIF
46 % repeat the sorting on the reduced array
47     ENDDO
48 ENDRoutine
49 ENDMODULE
50 $EOF

```


Here is the other module from the file TESTSORT:LIST.

```
1  MODULE testsort
2  IMPORT (ROUTINE VOID,VOID(INTEGER2 ARRAY) : quiksort)
3  INTEGER ARRAY: stack(0:1000)
4  INTEGER: max := 10
5  % length of array to sort
6  INTEGER: seed := 579, mult := 5181
7  % random number generator
8
9  PROGRAM: main
10 INTEGER2 ARRAY POINTER: iap
11 INTEGER: i
12   INISTACK stack
13   OUTPUT(1,'A','$START VALUES$')
14   NEW INTEGER2 ARRAY(0:max) =: iap
15   FOR i IN IND(iap) DO
16 % set random values in array
17   seed * mult =: seed =: IND(iap)(i)
18   ENDFOR
19   FOR i IN IND(iap) DO
20   OUTPUT(1,'I6',IND(iap)(i))
21   ENDFOR
22   OUTPUT(1,'A','$SORTED VALUES$')
23   quiksort(IND(iap))
24   FOR i IN IND(iap) DO
25   OUTPUT(1,'I6',IND(iap)(I))
26   ENDFOR
27 ENDROUTINE
28 ENDMODULE
29 $EOF
```

By writing \$EOF, you do not need to give the EXIT command to the PLANC compiler.

@DEBUGGER ↵

ND-100 SYMBOLIC DEBUGGER. VERSION D.

*PLACE SORT-EXAMPLE ↵

PLANC PROGRAM. TESTSORT.MAIN.9

*SET SEED = 1 ↵

*SET MULT = 10 ↵

*BREAK QUIKSORT ↵

*RUN ↵

START VALUES

10 100 1000 10000-31072 16960-27008 -7936-13824 -7168 -6144

SORTED VALUES

BREAK AT QUIKSORT.6

*DISPLAY ARR ↵

ARR=10 100 1000 10000 -31072 16960 -27008 -7936 -13824 -7168 -6144

We see that the correct
array is being used.

*RUN ↵

BREAK AT QUIKSORT.6

*ACTIVE-ROUTINES ↵

QUIKSORT.3 CALLED FROM QUIKSORT.44

QUIKSORT.3 CALLED FROM MAIN.23

MAIN.9

We see that QUIKSORT is a
recursive routine.

*DISPLAY ARR ↵

ARR=

*DISPLAY ADDR(ARR) ↵

ADDR(ARR)=(000176B;7:5)

There are no elements in
the array because the
lower bound of 7 is greater
than the upper bound of 5.

*BREAK-RETURN ↵

BREAK AT QUIKSORT.46

We break when we leave the
routine QUIKSORT.

*DISPLAY ↵

HIGH=6

ARR(0:10)

TEMP= 10

TOP= 5

LOW=7

MARKER=0

*EXIT ↵

Since the bounds of the array were wrong in the beginning of the routine QUIKSORT, we investigate the code immediately before QUIKSORT is called and find that lines 40 and 41 were transposed. They should have appeared in the order:

```
    quiksort(arr(high + 1:top))  
    high - 1 =: top
```

5.4 Using a File as a Segment

On the ND-500, you can achieve faster I/O by opening a file as a segment.

In the following example a file is opened as a segment, and the numbers 1 to 2560 are written to it:

```

C  PROGRAM FILESEG
   OPEN FILE AS SEGMENT
   INTEGER MEM (2560)
   WRITE(1,*) 'WILL ATTEMPT TO OPEN FISH:DATA FOR WX ACCESS'
   OPEN (13, FILE='FISH:DATA', ACCESS='WX',MODE='SEGMENT')
   DO 2 K = 1, 2560
     MEM(K) = K
     WRITE(13,*) MEM(K)
2  CONTINUE
   DO 3 K = 1, 10
     WRITE (1,*) K, MEM(K)
3  CONTINUE
   CLOSE(13)
   WRITE (1,*) 'END OF PROGRAM'
   END

```

No special procedures were needed to load the above program:

```

@ND LINKAGE-LOADER ←
ABORT-BATCH-ON-ERROR OFF ←
RELEASE-DOMAIN TEST ←
DELETE-DOMAIN TEST ←
SET-DOMAIN "TEST" ←
LOAD TEST,... ←
LIST-SEG TEST,... ←
END ←
EXIT ←

```

5.5 Using a File as a Segment for a COMMON Area

The following program uses the monitor call FSCNT to connect a file as a segment. It uses a common area that is placed on the file connected as a segment. Thus every time the common area is accessed, that segment will be accessed.

If you debug the program below and give the command:

```
*LOOK-AT-DATA ADDR(I) ↵
```

You will see that the address starts with 07 because the file uses segment 7.

```
PROGRAM TESTSEG
INTEGER ACC, SEGNO, IOPENF
COMMON TEKST
CHARACTER*10 TEKST(20,100)
IOPENF = 10
SEGNO = 7
WRITE(1,*) 'WILL ATTEMPT TO OPEN FISH:DATA FOR WX ACCESS'
C   The file FISH:DATA must already exist, be large enough
C   to hold the array TEKST, and contain some text.
OPEN(IOPENF,FILE='FISH:DATA', ACCESS='WX')
C   Get SINTRAN file number related to IOPENF.
I = LDN(IOPENF)
C   You must be able to read from and write to the segment.
ACC = 2
C   Don't use 0, 1, 2, 3, 26D, or 30D as SEGNO if you will debug.
CALL FSCNT(I,SEGNO,ACC,IACTNO)
C   Remember to use N11: COMMON-SEGMENT-NUMBER 7,, in loading.
WRITE(1,*) 'The following segment has been connected:'
WRITE(1,*) IACTNO
WRITE(1,*)
DO 10 J = 1, 10
  DO 20 K = 1, 100, 10
    TEKST(J,K) = 'AAAAAAAAA '
20  CONTINUE
10  CONTINUE
CALL FSDCNT(I,SEGNO)
CLOSE(IOPENF)
WRITE (1,*) 'END OF PROGRAM'
END
```


The above program can be loaded as follows:

```

@CREATE-FILE TEST:NRF 0 ↵
@ND FORTRAN ↵
DEBUG-MODE ON ↵
COMPILE TEST,TERMINAL,TEST ↵
EXIT ↵
@ND LINKAGE-LOADER ↵
ABORT-BATCH-ON-ERROR OFF ↵
RELEASE-DOMAIN TEST ↵
DELETE-DOMAIN TEST ↵
SET-DOMAIN "TEST" ↵
COMMON-SEGMENT-NUMBER 7 ↵
COMMON-SEGMENT-OPEN "TEST-SEG" F,,, ↵
LOAD TEST,,, ↵
LIST-SEG TEST,,,, ↵
END ↵
EXIT ↵
@_

```

Since segment 7 was specified in the monitor call FSCNT, segment 7 must also be specified in the COMMON-SEGMENT-NUMBER example.

If you are going to debug a program that uses a COMMON segment, we suggest that you do not use the following common segment numbers:

0, 1, 2, 3, 26D, and 30D

That is because they are being used by the Debugger or the FORTRAN library.

CHAPTER 6

ERROR MESSAGES

6 ERROR MESSAGES

6.1 Error Messages Common to the ND-100 and the ND-500 Versions

Here are the error messages and what they mean:

AMBIGUOUS COMMAND

Self-explanatory.

ASSEMBLER ERROR

Followed by an assembler error message. This can occur when using the CODE subcommand of LOOK-AT.

ATTEMPT TO DIVIDE BY ZERO

Self-explanatory.

B REGISTER NOT INITIALISED

Unable to LOOK-AT-STACK because the B-register is not well-defined.

BAD EXPRESSION

Syntax error in expression.

BAD LINE NUMBER

Syntax error in specified line number.

BAD MODULE/ENDMODULE NESTING

Error in the debug information generated by the compiler.

BAD RECORD/ENDRECORD NESTING

Error in the debug information generated by the compiler.

BAD ROUTINE/ENDROUTINE NESTING

Error in the debug information generated by the compiler.

BAD STRING CONSTANT

Self-explanatory.

COMMAND LINE/MACRO BUFFER FULL

The command line is too long, or too many macros are defined. This message may also occur during macro expansion.

COMPONENT NOT IN SPECIFIED RECORD

Self-explanatory.

ERROR: n

Error number n from SINTRAN III or ND-500 Monitor. There is no error text for this error number.

ILLEGAL DEBUG ELEMENT TYPE; DEBUG TABLE ADDRESS: xxxxxxB

Error in the debug information generated by the compiler.

ILLEGAL DEBUG TABLE ADDRESS (xxxxxx) IN "FIND"

Internal consistency error in the Debugger.

ILLEGAL TERMINATION

Illegal termination of the command line.
You have probably used the wrong type or number of parameters.

ILLEGAL TERMINATION OF ARGUMENT

Illegal termination of a command parameter.

INDEX "n" IS OUTSIDE ARRAY

Index outside range in array access.

INDIRECTION NOT LEGAL

Indirection in LOOK-AT not legal for this step size.

LIMITS NOT LEGAL FOR THIS TYPE

Can occur in the GUARD command; low:high is not legal for this item type.

LINE TRANSLATION TABLE FULL

Too many areas specified in ALIGN-LISTING.

LINK INFORMATION INACCESSIBLE

Can occur with the BREAK-RETURN command when no well-defined return address can be found.

MODULES/ROUTINES TOO DEEPLY NESTED

Too deep nesting of modules and/or routines in the debug information.

NO DEBUG INFORMATION AVAILABLE

You probably did not compile your program, or a part of it, in debug mode. Otherwise, you may have forgotten to PLACE a :PROG.

NO SUCH COMMAND

NO SUCH REGISTER NAME

NOT A VARIANT OF THE SPECIFIED RECORD

The three above error messages are self-explanatory.

NOT FOUND

Usually preceded by a name, e.g., "SUBR" NOT FOUND. You may be in a different module or routine than you think you are.

ROUTINE INACTIVE

Routine inactive (no current stack frame allocated).

STRING CONSTANT TOO LONG

Self-explanatory.

TOO MANY INDICES

Too many indexes in the array reference.

USE LOG-CALLS OR LOG-LINES

This command requires the use of LOG-CALLS or LOG-LINES. Remember CHECK-OUT-MODE can only be used after you have specified LOG-CALLS or LOG-LINES. On the ND-100, GUARD can only be used after LOG-CALLS or LOG-LINES.

WRONG ENUMERATION TYPE NESTING

Error in the debug information generated by the compiler.

WRONG TYPE

Attempt to convert between incompatible types.

WRONG TYPE OR INACCESSIBLE

Item is of wrong type (e.g., REAL used as an array index) or inaccessible at this point in the program (e.g., local variable in inactive routine).

6.2 Error Messages Which Apply to the ND-100 Version**DATA AT DATA ADDRESS xxxxxxB IS NOT STORED ON THE PROG-FILE**

Attempt to modify a part of the data area that is not stored on the :PROG file. This can only happen when PLACE <file>,W has been used.

DATA AT PROGRAM ADDRESS xxxxxxB IS NOT STORED ON THE PROG-FILE

Attempt to modify a part of the program that is not stored on the :PROG file. Can only be happen when PLACE <file>,W has been used.

NO MORE DATA SEGMENTS AVAILABLE

Too many Debuggers are active at the same time. Each active Debugger uses one data segment. The maximum number of active Debuggers is specified when your SINTRAN III is generated. You should use the EXIT command to leave the Debugger. If you use the ESCAPE key, the data segment may not be released for use by others.

NO PROGRAM FILE SPECIFIED

You need to use the PLACE command to read in a program file.

6.3 Error Messages Which Apply to the ND-500 Version**AMBIGUOUS TRAP CONDITION****ATTEMPT TO ACCESS NONEXISTENT DATA SEGMENT****ATTEMPT TO ACCESS NONEXISTENT DEBUG INFORMATION****ATTEMPT TO ACCESS NONEXISTENT PROGRAM SEGMENT**

ATTEMPT TO MODIFY READ-ONLY SEGMENT

ATTEMPT TO SET BREAKPOINT ON READ-ONLY SEGMENT

The above 6 messages are self-explanatory.

BAD LINE DEBUG ELEMENT; DEBUG TABLE ADDRESS: xxxxxxxxxxxxB

Error in the debug information generated by the compiler.

BAD OPERAND CODE; DEBUG TABLE ADDRESS: xxxxxxxxxxxxB

Error in the debug information generated by the compiler.

ERROR IN MONITOR CALL

Error message from the ND-500 Monitor. Use the
AUTOMATIC-ERROR-MESSAGE command in the ND-500 Monitor
if further information is required.

ILLEGAL PROGRAM ADDRESS

Access attempted beyond the available address space.

IMPOSSIBLE TO INVOKE ROUTINE; STACK OVERFLOW

INVOKE command not executed; not enough room left in the
stack.

NO DSEG-FILE OPENED OR CONNECTED FOR SEGMENT nn

Self-explanatory.

NO LINK FILE OPENED OR CONNECTED FOR SEGMENT nn

Self-explanatory.

NO PSEG-FILE OPEN OR CONNECTED FOR SEGMENT nn

Self-explanatory.

NO SUCH TRAP CONDITION

Self-explanatory.

OUTSIDE DATA SEGMENT

Attempt to access beyond the available address space (on
an existing data segment).

OUTSIDE PROGRAM SEGMENT

Attempt to access beyond the available address space (on an existing program segment).

PROGRAMMED-TRAP FAILED (NOT ENABLED?)

The Debugger is unable to start your program because a programmed trap has been disabled or is not working.

SEGMENT NUMBER MUST BE IN THE RANGE 0:31

Self-explanatory.

THIS SINTRAN III COMMAND IS NOT ALLOWED FROM THE ND-500

Self-explanatory.

TOO MANY FILES OPENED

The Debugger is unable to open all the files needed.

6.4 Note on Error Returns on the ND-100

The ND-100 (if started by the Symbolic Debugger) will enter the Debugger when it stops, for instance if the stack overflows. The following messages may occur:

PROGRAM TERMINATED AT current scope

ASSERT VIOLATION AT current scope

STACK OVERFLOW AT current scope

INDEX RANGE ERROR AT current scope

WRONG NO. OF PARAMETERS AT current scope

Index

A format (ASCII)	71.
abbreviation address	46.
ACTIVE-ROUTINES	19.
ADDR	61, 66.
example	14, 66, 69, 78, 82.
address	
abbreviation	46.
data	70.
program	69.
align routine	20.
ALIGN-LISTING	20.
alternative page table	40, 47, 51.
apostrophe in addresses	46.
ASCII format	71.
ATTACH-SEGMENT	20.
break	21.
(see step point)	14.
condition	21.
conditional (example)	76.
count	21.
label	68.
line number	68.
multiple (see step point)	14, 69.
see also GUARD	25.
BREAK-ADDRESS	22.
BREAK-RETURN	22.
breakpoint	11.
multiple (see step point)	14, 69.
BRF file	10.
B register	13, 75.
change	
data address	41.
program address	26, 41, 46, 51.
CHECK-OUT-MODE	24.
after LOG-CALLS	21, 37.
after LOG-LINES	28, 39.
before BREAK	24.
before STEP	24.
example	21, 39, 78.
common area	85.
COMPARE-DATA	25.
COMPARE-PROGRAM	26.
compile	9.
example	12.
constant	61.
ADA form	62.
numeric	61.
real	62.
single-character	64.
string	64.
with exponent	62.

CONTINUE (see also RUN)	26.
count using 1 or -1	57.
CROSS-REFERENCE	12.
D format (decimal)	71.
data address	14, 70.
decimal format	71.
decimal format	14.
default file type	9.
disassembly	71.
displacement	13, 75.
DISPLAY	27.
arrays	15.
expressions	14.
IND	27.
pointer	27.
record	50.
values	14.
without parameters	14.
DOUBLE-FLOATING example	44.
DUMP-LOG	28.
example	21, 37-39.
ENABLED-TRAPS	29.
example	
compile and load	12.
program in FORTRAN	12, 75.
program in PLANC	23, 67, 77, 79.
EXIT	29.
expressions	65.
F format (floating	71.
file	
:BRF	10.
:NRF	10.
FIND-SCOPE	29.
FLOATING	
example	44.
point format	71.
format	71.
A (ASCII)	71.
D (decimal)	71.
F (floating point)	71.
H (hexadecimal)	71.
I (instruction (disassembly))	71.
O (octal)	71.
FORMATS-DISPLAY	14, 30.
FORMATS-LOOK-AT	30.
example	40, 43-45, 48.
FORTTRAN example	12, 75.
FSCNT example	85.
GUARD	31.
example	25, 31, 53, 70.
range permitted	31.
H format (hexadecimal)	71.

HELP	32.
hexadecimal format	14, 71.
H format (hexadecimal)	14.
I format (instruction (disassembly))	71.
INCLUDE-COMMANDS	33.
IND	61, 66.
example	66.
inspect	
data address	14.
program address	26, 69.
instruction format (disassembly)	71.
INTEGER* 2	13.
INVOKE	34.
label	13, 68.
library	10, 11.
LINKAGE-LOADER	10.
load	10.
example (ND-100)	10, 12.
example (ND-500)	10.
LOCAL-TRAP-DISABLE	35.
LOCAL-TRAP-ENABLE	36.
LOG-CALLS	37.
LOG-LINES	38.
advice	39.
example with CHECK-OUT-MODE	28, 39.
example with DUMP-LOG	38.
example with GUARD	25, 39, 53.
example with STEP	23, 57.
reset	53.
with CHECK-OUT-MODE	24.
with GUARD	39.
with STEP	39.
LOOK-AT-DATA	14, 40, 43.
LOOK-AT-PROGRAM	46.
example	26, 69.
LOOK-AT-REGISTER	47.
LOOK-AT-STACK	48.
MACRO	49.
parameter	50.
memory area	69.
multiple step points	14, 69.
NRF file	10.
numbers	
binary	62.
decimal	62.
hexadecimal	62.
octal	62.
O format (octal)	71.
octal format	14, 71.
O format (octal)	14.
patch	
data	41.

program	41, 46, 51, 56.
permitted range	31.
PLACE	51.
to patch :PROG file	51, 56.
PLANC example	22, 23, 67, 77, 79.
pointer	
display	27.
example	66.
precedence	65.
program	
address	69.
area	69.
execution improvement (ND-100)	56.
memory	69.
PROGRAM-INFORMATION	51.
radix specifier	62, 71.
range in GUARD	31.
Register B	13, 75.
RESERVE-TERMINAL	52.
RESET-BREAKS	15, 53.
RUN	11, 54.
example	54.
SCOPE	54.
segment number	46.
SEGMENT-INFORMATION	55.
SET	55.
SHIFT	61.
example	66.
specifier	
format	71.
radix	62.
STACK-INSTRUCTIONS	56.
STEP	11, 14, 57.
point	11.
step point multiple	14, 69.
string	
constant	64.
GUARD	70.
trace	57.
undo	
GUARD	31.
LOG-LINES	53.

NOTICE

The information in this manual is subject to change without notice. Norsk Data A.S assumes no responsibility for any errors that may appear in this manual. Norsk Data A.S assumes no responsibility for the use or reliability of its software on equipment that is not furnished or supported by Norsk Data A.S.

This manual is protected by copyright. It may not be photocopied, reproduced or translated without the prior consent of Norsk Data A.S.

Copyright © 1985 by Norsk Data A.S

UPDATING

PRINTING RECORD	
PRINTING	NOTES
02.82	Version 01
02.83	Version 02
03.85	Version 3

Manual name: Symbolic Debugger User Guide
Manual number: ND-60.158.3 EN
Date: 03.85

Manuals can be updated in two ways, new versions and revisions. New versions consist of a completely new manual which replaces the old one. New versions incorporate all revisions since the previous version. Revisions consist of one or more single pages to be merged into the manual by the user, each revised page being listed on the new printing record sent out with the revision. The old printing record should be replaced by the new one.

New versions and revisions are announced in the ND Customer Support Information and can be ordered from the address below.

The reader's comments form at the back of this manual can be used both to report errors in the manual and to give an evaluation of the manual. Both detailed and general comments are welcome.

All types of inquiry and requests for documentation should be sent to the local ND office, or (in Norway) to:

Norsk Data A.S
Graphic Center
P.O. Box 25, Bogerud
N-0621 Oslo 6, Norway

SEND US YOUR COMMENTS!



Are you frustrated because of unclear information in our manuals? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card - and an answer to your comments.

Please let us know if you:

- find errors
- cannot understand information
- cannot find information
- find needless information.

Do you think we could improve our manuals by rearranging the contents? You could also tell us if you like the manual.

Send to: Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
N-0621 Oslo 6
Norway

NOTE!

This form is primarily for documentation errors. Software and system errors should be reported on Customer System Reports.

Manual name: Symbolic Debugger User Guide Manual number: ND-60.158.3 EN

What problems do you have? (use extra pages if needed) _____

Do you have suggestions for improving this manual? _____

Your name: _____ Date: _____

Company: _____ Position: _____

Address: _____

What are you using this manual for? _____

Norsk Data's answer will be found on the reverse side. →

Answer from Norsk Data _____

Answered by _____ Date _____

Norsk Data A.S

Documentation Department
P.O. Box 25, Bogerud
0621 Oslo6, Norway

Systems that put people first

NORSK DATA A.S OLAF HELSETS VEI 5 P.O. BOX 25 BOGERUD 0621 OSLO 6 NORWAY
TEL.: 02 - 29 54 00 - TELEX: 18284 NDN