SYMBOLIC DEBUGGER
User Guide

ND-60.158.4 EN

SYMBOLIC DEBUGGER
User Guide
ND-60.158.4 EN

PRINTING RECORD	
PRINTING	NOTES
02/82	Version 1 EN
02/83	Version 2 EN
03/85	Version 3 EN
12/86	Version 4 EN

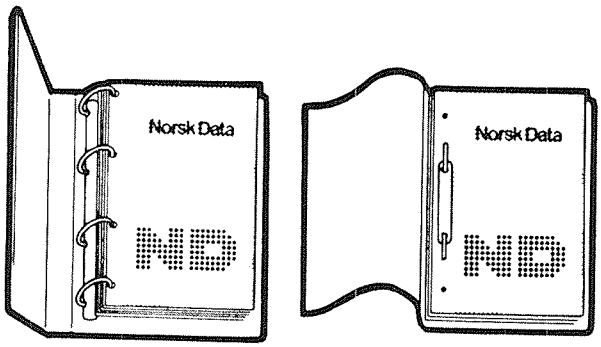
Symbolic Debugger User Guide
Publ.No. ND--60.158.4 EN

UPDATING

Manuals can be updated in two ways, new versions and revisions. New versions consist of a completely new manual which replaces the old one, and 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 give an evaluation of the manual. Both detailed and general comments are welcome.



RING BINDER OR PLASTIC COVER

The manual can be placed in a ring binder for greater protection and convenience of use. Ring binders may be ordered at a price of NKr. 45.- per binder.

The manual may also be placed in a plastic cover. This cover is more suitable for manuals of less than 100 pages than for larger manuals.

Please send your order, as well as all types of inquiries and requests for documentation 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

I would like to order

..... Ring Binders, 40 mm, at NOK 45.- per binder

..... Plastic Covers, at NOK 10.- per cover

Name: .....

Company: .....

Address: .....

Preface:

THE PRODUCT

This manual describes the following products:

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 make information for 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, Pascal or PLANC. If it is necessary to debug RT-programs, ability to use the RT-Loader is required. Advanced programming experience is is a precondition for use of some of the Symbolic Debugger commands.

THE MANUAL

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

NEW IN THIS MANUAL

The ND-100 Debugger now can be used on multi-segment reentrant programs and on RT-Programs. The related commands are described in this manual.

Chapters 1 and 2 have been completely rewritten. Chapter 1 now contains a table over all the various Symbolic Debugger commands with parameters. Chapter 2 now shows simple use of the Debugger for ND-100 and ND-500 programs, as well as introductions to RT and ND-100 multi-segment debugging.

Chapters 3 and 4 have been updated with information related to the new features mentioned above. In addition, a number of minor corrections, changes and new examples have been included.

The examples in chapter 5 are unchanged, except that it now includes one example which was previously in chapter 2.

Chapter 6, containing the error messages, now has more extensive explanations, and it includes ND-100 RT-Debugger and multi-segment errors.

Finally, the index is enlarged, and many new cross-references are added. Changes and deletions are marked with change bars in the margins, as on this page.

RELATED MANUALS

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

Ada User Guide	ND-60.198
COBOL Reference Manual	ND-60.144
FORTRAN Reference Manual	ND-60.145
ND-500 BASIC User Manual	ND-60.207
PASCAL Reference Manual	ND-60.222
PLANC Reference Manual	ND-60.117

The following manuals are also relevant:

BRF Linker User Manual	ND-60.196
ND-500 Loader/Monitor	ND-60.136
ND Relocating Loader	ND-60.066
SINTRAN III Real Time Guide	ND-60.133
SINTRAN III Real Time Loader	ND-60.051
SINTRAN III Monitor Calls	ND-60.228

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1
1.1	Symbolic Debugger Command Summary	3
2	USING THE SYMBOLIC DEBUGGER	7
2.1	A Debugging Session	10
2.2	Debugging an ND-100 Background Program	12
2.2.1	Debugging Multi-Segment Programs	14
2.3	Debugging an ND-500 Program	17
2.4	Debugging an RT-Program	18
2.5	The Capabilities of the Symbolic Debuggers	22
2.6	Advanced Features in the Debuggers	24
2.7	Additional Features in the ND-500 Debugger	25
3	COMMANDS - DETAILED DESCRIPTION	27
3.1	ACTIVE-ROUTINES (<maximum number of levels>)	29
3.2	ALIGN-LISTING <program area> <line>	30
3.3	ATTACH-REENTRANT-SEGMENT <file name> <segment name>	30
3.4	ATTACH-SEGMENT <segment number> (<W>)	31
3.5	ATTACH-SEGMENT <segment number> <prog-file name> (<W>)	31
3.6	BREAK <routine, label or line> (<count>) (<condition>)	31
3.7	BREAK-ADDRESS <program address> (<count>)	33
3.8	BREAK-RETURN	33
3.9	CHECK-OUT-MODE (<program area>)	35
3.10	COMPARE-DATA <low> <high> (<output file>)	36
3.11	COMPARE-PROGRAM <low> <high> (<output file>)	37
3.12	CONTINUE	37
3.13	DISPLAY (<item or value>)	38
3.13.1	Pascal and PLANC Records	39
3.13.2	Displaying PLANC Variant Records on the ND-500	41
3.14	DUMP-LOG (<output file>)	42
3.15	ENABLED-TRAPS	43
3.16	EXIT	43
3.17	FIND-SCOPE <program address>	43
3.18	FORMATS-DISPLAY <formats (A,D,F,H,O or combinations)>	44
3.19	FORMATS-LOOK-AT <formats (A,D,F,H,I,O or combinations)>	44
3.20	GET-BREAK-STATUS	45
3.21	GUARD <item or address> (<(*not*) low (: high)>)	45
3.22	HELP <command name>	46
3.23	INCLUDE-COMMANDS <file name>	47
3.24	INVOKE <routine> (< (parameter,...,parameter) >)	48
3.25	LOCAL-TRAP-DISABLE (<trap conditions>)	50
3.26	LOCAL-TRAP-ENABLE (<trap conditions>)	51
3.27	LOG-CALLS <program area>	52

Section		Page
3.27.1	LOG-CALLS and CHECK-OUT-MODE	53
3.27.2	LOG-CALLS and GUARD	53
3.27.3	LOG-CALLS and STEP	53
3.28	LOG-LINES <program area>	54
3.28.1	LOG-LINES and GUARD	54
3.28.2	LOG-LINES and STEP	55
3.29	LOOK-AT-DATA <data address> (<count>) (<output file>)	56
3.30	LOOK-AT Subcommands	61
3.31	LOOK-AT-PROGRAM <program address> (<count>) (<output file>)	65
3.32	LOOK-AT-REGISTER <register name> (<count>) (<output file>)	66
3.33	LOOK-AT-STACK <B register> (<count>) (<output file>)	66
3.34	MACRO <name> <body>	69
3.35	MULTIPLE-BREAK-MODE <ON/OFF>	71
3.36	PLACE <file name> (<W>)	71
3.37	PROGRAM-INFORMATION	72
3.38	REENTRANT-PLACE <Reentrant-name>	72
3.39	RESERVE-TERMINAL <logical device number>	72
3.40	RESET-BREAKS (<program area>)	73
3.41	RT-PLACE <program name> (<W>)	74
3.42	RUN (<program address>)	74
3.43	SCOPE (<module, routine or other item>)	75
3.44	SEGMENT-INFORMATION	76
3.45	SEGMENT-WRITE-PERMIT <segment number>	76
3.46	SEGMENT-WRITE-PROTECT <segment number>	76
3.47	SET <variable> (=) <value>	77
3.48	STACK-INSTRUCTIONS (<low>) (<high>)	78
3.49	STEP (<count>) (<low>) (<high>)	79
3.50	USER-ESCAPE <on/off>	80
4	SYMBOLIC DEBUGGER PARAMETERS	81
4.1	Numeric Constants	83
4.2	Single-Character Constants	85
4.3	String Constants	85
4.4	Expressions	86
4.5	Named Items	88
4.6	Program Area	91
4.7	Program Address	91
4.8	Data Address	92
4.9	Format Specifier	93
4.10	File Name	93
5	EXAMPLES	95
5.1	An Example Using FORTRAN-100	97
5.2	A PLANC Example	98
5.3	Another Example in PLANC	100
5.4	Using a File as a Segment	105
5.5	Using a File as a Segment for a COMMON Area	106

Section Page

CHAPTER 1

INTRODUCTION

1. INTRODUCTION

The Symbolic Debugger is an interactive tool for testing programs written in higher-level languages such as FORTRAN, COBOL, PASCAL and PLANC.

If there is one or more Debugger segments on the segment file in your version of SINTRAN III, the Symbolic Debugger is available on your ND-100. One program can be debugged per Debugger segment, so that if SINTRAN has three such segments, three programs or less can be debugged simultaneously.

You can debug ND-100 multi-segment programs, but you have to be logged in as user SYSTEM to be able to do this.

The Debugger can also debug RT-programs on the ND-100. In this case, only one person (who must be logged in as user SYSTEM or RT) can use it at a time.

There are no such limitations on the ND-500.

The Symbolic Debugger contains a set of powerful 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 pages.

1.1. Symbolic Debugger Command Summary

The Symbolic Debugger commands may be abbreviated, even more than SINTRAN commands. The reason is that some of them have priority in addition to their names. (Commands with priority are marked with + in the table below.)

The priority works as follows: If you type a D as your command, that could be an abbreviation for both the DISPLAY and the DUMP commands. But instead of telling you that the command D is ambiguous, the Symbolic Debugger will execute the DISPLAY command, which has priority. (As you see, it is marked with a + below.) To do a DUMP, you must type a DU, which makes it clear that this cannot be a DISPLAY command.

Here is a table of all the commands available in the various Symbolic Debuggers. Error messges are found in chapter 6, page 111.

ND-100	RT-Deb	ND-500	Commands	Parameters
			Commands, with parameters. Parameters within < ... > will be prompted for, while parameters within (< ... >) will not be prompted for. Commands marked with a + have higher priority (see above) than the others.	
●	●	●	@	<SINTRAN III command>
●	●	●	+ACTIVE-ROUTINES	(<Maximum number of levels>)
●	●	●	ALIGN-LISTING	<Program area> <Line>
●			ATTACH-REENTRANT-SEGMENT	<File name> <Segment name>
	●	●	ATTACH-SEGMENT	<Segment number> (<File name>) (<W>)
●	●	●	+BREAK	<Routine, label or line> (<Count>) (<Condition>)
●	●	●	BREAK-ADDRESS	<Program address> (<Count>)
●	●	●	BREAK-RETURN	
●		●	CHECK-OUT-MODE	(<Program area>)
●	●	●	COMPARE-DATA	<Low> <High> (<Output file>)
●	●	●	COMPARE-PROGRAM	<Low> <High> (<Output file>)
●	●	●	+CONTINUE	
●	●	●	+DISPLAY	(<Item or value>)
●		●	DUMP-LOG	(<Output file>)
		●	ENABLED-TRAPS	
●	●	●	+EXIT	
●	●	●	FIND-SCOPE	<Program address>
●	●	●	+FORMATS-DISPLAY	<Formats (A,D,F,H,O or combinations)>
●	●	●	FORMATS-LOOK-AT	<Formats (A,D,F,H,I,O or combinations)>
	●		GET-BREAK-STATUS	
●		●	GUARD	<Item or address> (<("Not") Low (: High)>)

(Cont.)

Commands, with parameters. Parameters within < ... > will be prompted for, while parameters within (< ... >) will not be prompted for. Commands marked with a + have higher priority (see above) then the others.

ND-100	RTDeb	ND-500	Commands	Parameters
●	●	●	HELP	<Command name>
●		●	INCLUDE-COMMANDS	<File name>
●		●	INVOKE	<Routine> (<(parameter,..,parameter)>)
		●	LOCAL-TRAP-DISABLE	(<Trap conditions>)
		●	LOCAL-TRAP-ENABLE	(<Trap conditions>)
●		●	LOG-CALLS	<Program area>
●		●	LOG-LINES	<Program area>
●	●	●	LOOK-AT-DATA	<Data address> (<Count>) (<Output file>)
●	●	●	+LOOK-AT-PROGRAM	<Program address> (<Count>) (<Output file>)
●	●	●	LOOK-AT-REGISTER	<Register name> (<Count>) (<Output file>)
●	●	●	LOOK-AT-STACK	<B register> (<Count>) (<Output file>)
●	●	●	+MACRO	<Name> <Body>
		●	MULTIPLE-BREAK-MODE	(<On/Off>)
●			PLACE	<File name> (<W>)
●			+PROGRAM-INFORMATION	
●			REENTRANT-PLACE	<Reentrant-name>
●		●	RESERVE-TERMINAL	<Logical device number>
●	●	●	RESET-BREAKS	(<Program area>)
	●		RT-PLACE	<Program name> (<W>)
●	●	●	+RUN	(<Program address>)
●	●	●	SCOPE	(<Module, routine or other item>)
	●	●	SEGMENT-INFORMATION	
●	●		SEGMENT-WRITE-PERMIT	<Segment number>
●	●		SEGMENT-WRITE-PROTECT	<Segment number>
●	●	●	SET	<Variable> <Value>
●			STACK-INSTRUCTIONS	(<Low>) (<High>)
●		●	+STEP	(<Count>) (<Low>) (<High>)
		●	USER-ESCAPE	(<On/Off>)

CHAPTER 2

USING THE SYMBOLIC DEBUGGER

2. USING THE SYMBOLIC DEBUGGER

This chapter will show basic use of the three different debuggers, including brief presentations of compilation and loading. For detailed descriptions of the various compilers and loaders, please consult the manuals that deal with each individual product.

The ND debuggers contain many useful commands, as you saw in the table in the preceding chapter. An overview of advanced features is given on page 24.

You need, however, only to understand four commands to use the debuggers:

1) BREAK <routine, label or line>

   instructs the debugger to stop execution of your program before code belonging to a subroutine, following a label or a line number is executed

2) RUN

   starts execution of your program. You will not come back to the debugger before a breakpoint is reached

3) DISPLAY <item>

   tells the debugger to show you what values the variables in your program have at the current breakpoint (i.e., before the execution of the instruction immediately after the breakpoint)

4) EXIT

   terminates the debugging session

In addition, you need a listing of the source code of the program, for instance the listing generated by the compiler.

We will assume that all files used in the following examples exist before the examples are run. If you do not know how to create files with the "<file name>" notation, talk to somebody who knows, or read about it in the SINTRAN III documentation.

The examples use the following FORTRAN program:

```
 1          program BUBBLE
 2          integer ToSort(0:9)
 3          data ToSort/9,8,7,6,5,4,3,2,1,0/
 4          integer i, j, k
 5          do 100 i = 1, 9
 6              do 200 j = 9, i, -1
 7                  if (ToSort(j) .lt. ToSort(j-1)) then
 8                      k = ToSort(j-1)
 9                      ToSort(j-1) = ToSort(j)
10                      ToSort(j) = k
11                  endif
12  200         continue
13  100     continue
14          end
```

Here, the numbers to the left are not part of the program. They are line numbers, as generated by compilers, pretty-printers etc. These line numbers are used extensively as location references when you debug programs, together with subroutine or module names etc.

The program sorts the integer array named ToSort, which has ten elements containing integer numbers. The numbers are in reverse order from the outset, but the program will put them in the right order. We use the debuggers to see how the numbers wander through the array as the sorting proceeds. The program does not contain any errors. You will learn to use the debuggers to observe the execution of a program, and need no errors to do that.

In the examples that follow, your input to the computer is underlined, and a ↵ denotes the final carriage return for each of your commands. Comments on the examples are enclosed in boxes near the right margin,

like this one. (You will see more such boxes later on!)

As the example is finished, a _ (underscore) shows where the cursor is placed.

2.1. A Debugging Session

Here you see how you use a debugger to observe the BUBBLE program. This is exactly how the debugging session will appear when you use the ND-100 and the ND-500 debuggers. The next three subsections will show you how you prepare a program for debugging with the three different debuggers.

When using the RT-Debugger, the procedure for restarting after a breakpoint has been reached is more complicated than is shown here, for natural reasons. We will deal with that in the section about the RT-Debugger on page 18.

In what follows, we presume that the program has been compiled with the Debug option on and loaded as an ordinary program, and that you have started the appropriate Debugger to look at the program. You will see how this is done on page 12 for the ND-100 and on page 17 for the ND-500.

This is what you do after you have started either the ND-100 or the ND-500 Debugger for your program.

```
*break 7 ↵
*run ↵

Break at BUBBLE.7
*display i,j,ToSort ↵
I=1
J=9
TOSORT=9 8 7 6 5 4 3 2 1 0
*break 10 ↵
*run ↵

Break at BUBBLE.10
*display i,j,k,ToSort ↵
I=1
J=9
K=1
TOSORT=9 8 7 6 5 4 3 2 0 0
*break 12 ↵
*run ↵

Break at BUBBLE.12
*display i,j,ToSort ↵
I=1
J=9
TOSORT=9 8 7 6 5 4 3 2 0 1
*run ↵

Break at BUBBLE.12
*display i,j,ToSort ↵
I=1
J=8
TOSORT=9 8 7 6 5 4 3 0 2 1
*b 13 ↵
*r ↵

Break at BUBBLE.13
*d i,j,ToSort ↵

I=1
J=0
```

First set a BREAK at line 7, then ask the Debugger to run the program until it hits the break. At the breakpoint, display the control variables for the loops and the array ToSort. Then put breakpoint at line 10, to see if execution reaches it and to display the variables. Also look at the variable k, which is used for intermediate storage as array contents are swapped in neighbouring array elements.

Move the break to the end of the innermost loop.

The two last elements in ToSort have exchanged places. The smallest integer in the array, 0, has moved one position towards the first element in the array, which is where it should end up if the program works. One more execution of the loop moves the 0 one step further to the left. Move the breakpoint to the end of the outermost loop. Here, we have used the ultimate command abbreviations!

```
TOSORT=0 9 8 7 6 5 4 3 2 1
*r ↵

Break at BUBBLE.13
*d i,j,ToSort ↵
I=2
J=1
TOSORT=0 1 9 8 7 6 5 4 3 2
*b 14 ↵
*r ↵

Break at BUBBLE.14
*d i,j,ToSort ↵
I=10
J=8
TOSORT=0 1 2 3 4 5 6 7 8 9
*exit ↵

@_
```

Two complete executions of the outer loop show that both the smallest elements in the array have been moved to their correct locations. Move the breakpoint to the last line in the program -

run it, and you will see that the array is sorted in the correct sequence. Leave the Debugger.

2.2. Debugging an ND-100 Background Program

The following :MODE-file will compile the BUBBLE program and generate debug information for the ND-100 Debugger. (If you don't know what a :MODE-file is, talk to somebody who does, or read the SINTRAN documentation.)

```
@fortran-100
debug
compile fortran-bubble terminal fortran-bubble
exit
@brf-linker
program-file fortran-bubble
load fortran-bubble fortran-2bank
exit
```

After this :MODE-file has been run, just type

```
@debugger fortran-bubble ↵
```

on your terminal, and you are ready to debug, as shown on page 11.

This is how you prepare the BUBBLE program for use with the ND-100 Debugger, shown in detail.

```
@fortran-100 ↵
ND-100/NORD-10 ANSI 77 FORTRAN COMPILER - MAY 9, 1986
FTN: separate off ↵
```

Forcing the compiler to generate one-bank code. You may want to make two-bank code instead - then you must load the FORTRAN-2BANK library afterwards.

```
FTN: debug ↵
```

This is the command that tells the compiler to make debug information for your program.

```
FTN: compile fortran-bubble terminal fortran-bubble ↵
```

The source code is taken from the file FORTRAN-BUBBLE:SYMB, output from the compilation is directed to the terminal, and the object code is put on the file FORTRAN-BUBBLE:BRF.

```
ND-100/NORD-10 ANSI 77 FORTRAN COMPILER - MAY 9, 1986
SOURCE FILE:    FORTRAN-BUBBLE:SYMB

    1*              program bubble
    2*              integer ToSort(0:9)
    3*              .
    4*              .
     .              .
     .
```

This is the output from the compiler, telling you how the compilation proceeds. The line numbers can be used when debugging.

```
- CPU TIME USED: 0.7 SECONDS.  14 LINES COMPILED.
- NO MESSAGES
- PROGRAM SIZE=63 DATA SIZE=29 COMMON SIZE=0
FTN: exit ↵
```

Leave the compiler, and enter the BRF-Linker to build an executable program.

```
@brf-linker ↵
- BRF Linker -  10721B00
Brl: program-file fortran-bubble ↵
Brl: load fortran-bubble fortran-1bank ↵
```

You must specify which :PROG-file the executable program should be built on, before you can load the relocatable program and the FORTRAN-1BANK library. What follows is output from the linker.

```
FREE: P 000077-177777      D 000035-177777      DEBUG 000130
FORTRAN-1BANK-E0 48-BIT FLOATING
PLANC-1BANK-F00
FREE: P 005265-177777      D 005466-177777      DEBUG 000130
Brl: exit ↵
```

Done. Now you can start the ND-100 Debugger.

```
@debugger fortran-bubble ↵
FORTRAN PROGRAM.  BUBBLE.1
*
```

You are ready to observe the program execution with the ND-100 Debugger.

2.2.1. Debugging Multi-Segment Programs

The ND-100 Symbolic Debugger can now be used with programs that have been loaded onto SINTRAN III's segment file as multi-segment programs, in addition to ordinary programs residing on :PROG-files. (Make sure you do not confuse ND-100's segments with ND-500's segments! They are very different things.)

Using multiple segments enables you to load bigger programs than was previously possible (that is unless you previously used the somewhat limited overlay technique). However, there is a limitation on the multi-segment technique: You cannot have more than 64 K global data, no matter how many segments you use.

In order to use the Debugger with multi-segment programs, you must be able to log in as either user SYSTEM or user RT on your computer. Otherwise, you cannot dump the program onto the segment file, and you cannot use the commands in the Debugger that are used with segments.

In this section, we will look at how a simple COBOL program called A-COBOL that uses a subprogram B-COBOL is loaded onto two different segments and debugged. A-COBOL and B-COBOL must be on separate files. However, before the debugging can start, the programs must be compiled, loaded and linked to the other segment, and dumped on the segment file.

The program which is built will be known to the computer as person, and the segments used will be called acobol and bcobol. The *PROG-files reside on the fictitious user area OWN-USER, where user SYSTEM must have the appropriate read- and write-privileges. The segments are cleared before they can be used.

The preparation for debugging is done by the following :MODE-file:

```
@delete-reentrant person
@clear-reentrant-segment bcobol
@cobol-100
debug
compile (own-user)a-cobol,1,(own-user)a-cobol
exit
@cobol-100
debug
compile (own-user)b-cobol,1,(own-user)b-cobol
exit
@brf-linker
program-file (own-user)a-cobol/acobol
load (own-user)a-cobol cobol-2bank
exit
@brf-linker
program-file (own-user)b-cobol/bcobol
define #dclc 4000b d
link-to (own-user)a-cobol
load (own-user)b-cobol cobol-2bank
exit
@dump-program-reentrant person,(own-user)a-cobol,acobol
@load-reentrant-segment (own-user)b-cobol,bcobol
```

For details concerning loading and execution of multi-segment programs, see the BRF-Linker User Manual, ND-60.196.

The program we now have on the segment-file can be started by typing person as your response to the SINTRAN prompt. It consists of a main program, which does nothing more than to initialize the data for a record called PERSON with such things as first name, middle initials, christian name, sex and age. The subprogram is then called with these data as parameters, processes them and then returns to the main program, which exits.

There are two new commands which become mandatory when you debug multi-segment programs:

1) ATTACH-REENTRANT-SEGMENT <file-name>,

   which links a :PROG-file with the segment it has been dumped to. The can be either a name or a number

2) REENTRANT-PLACE <reentrant-name>

   which prepares the Debugger for work on the main program segment

Let us look at how the Debugger is connected to the program on the segments, and how we can use the BREAK, RUN and DISPLAY afterwards.

```
@debug ↵
```

The Debugger is entered without any :PROG-file specification. Remember to be on either user Area SYSTEM or RT!

```
ND-100 SYMBOLIC DEBUGGER.  VERSION F.
*attach-reentrant-segment (own-user)a-cobol,acobol ↵
```

The :PROG-file built by the BRF-Linker and the name of the segment where the main program resides is attached.

```
*reentrant-place person ↵
```

Then the reentrant program system is initiated for the Debugger.

```
COBOL PROGRAM.  (Segment 156B) A-COBOL.1
```

The Debugger tells what kind of program it is, the number of the segment it is on, and that we are currently the first line of the program with the internal name A-COBOL.

```
*attach-reentrant-segment (own-user)b-cobol,bcobol ↵
```

We also attach the segment where the subroutine is.

```
*break b-cobol.18 ↵
*run ↵

Break at (Segment 157B) B-COBOL.18
```

Now, we can BREAK, RUN and DISPLAY! We set a break at the 18the line of the subroutine b-cobol, which is on the bcobol segment, and run. The Debugger breaks in segment 157B.

```
*display age ↵
AGE=28
*set age=45 ↵
*break a-cobol.18 ↵
*run ↵
```

The person has the age of 28. We use Debugger command set to change the age to 45. Then we set a break in the 18. line of the main program, and run.

```
Break at (Segment 156B) A-COBOL.18
*display age ↵
AGE=45

*__
```

The Debugger breaks in the segment of the main program. We see that the age of our hapless person has now been changed to 45 years.

2.3. Debugging an ND-500 Program

The following :MODE-file will compile the BUBBLE program and generate debug information for the ND-500 Debugger. (If you don't know what a :MODE-file is, talk to somebody who does, or read the SINTRAN documentation.)

```
@fortran-500
debug-mode
compile fortran-bubble terminal fortran-bubble
exit
@linkage-loader
set-domain fortran-bubble
open-segment fortran-bubble,,,
load-segment fortran-bubble
end
exit
```

After you have run this file, you must do the following to start debugging:

```
@nd-500 ↵
N500: debugger fortran-bubble ↵
```

Now you can proceed with the debugging, as shown on page 11.

This is how you prepare the BUBBLE program for use with the ND-500 Debugger, shown in detail.

```
@fortran-500 ↵
ND-500 ANSI 77 FORTRAN COMPILER - 203054I
FTN: debug-mode ↵
```

This makes the compiler generate debug information.

```
FTN: compile fortran-bubble terminal fortran-bubble ↵
```

The source code is taken from the file FORTRAN-BUBBLE:SYMB, output from the compilation is directed to the terminal, and the object code is put on the file FORTRAN-BUBBLE:NRF.

```
ND-500 ANSI 77 FORTRAN COMPILER - 203054I
SOURCE FILE:   FORTRAN-BUBBLE:SYMB

    1*            program bubble
    2*            integer ToSort(0:9)
    3*            .
    4*            .
     .            .
     .
```

This is the output from the compiler, telling you how the compilation proceeds. The line numbers can be used when debugging.

```
- CPU TIME USED: 0.2 SECONDS.  14 LINES COMPILED.
- NO MESSAGES
- PROGRAM SIZE=71 DATA SIZE=152 COMMON SIZE=0
FTN: exit ↵
```

Leave the compiler, and enter the Linkage-Loader to build an executable program.

```
@linkage-loader ↵
ND-Linkage-Loader -  H.00        3. April      1986 Time:  0:00
N11 entered:                     4. July       1986 Time: 12: 6
N11: set-domain fortran-bubble ↵
N11: open-segment fortran-bubble,,, ↵
N11: load-segment fortran-bubble ↵
```

The program is loaded on the domain FORTRAN-BUBBLE, segment FORTRAN-BUBBLE. Since the FORTRAN-LIBRARY is on an auto-link segment, it does not need to be included explicitly in the loading sequence.

```
Program:...113 P01   Data:.....234 D01   Debug inf:..327 Bytes
N11: exit ↵
Segment no.....30        is linked

@nd-500 ↵
```

Leave the compiler, and enter the ND-500 Monitor to start the debugger.

```
ND-500 MONITOR  Version H00 86. 5. 6 / 86. 5.14

N500: debugger fortran-bubble ↵
```

The debugger is implemented as a part of the ND-500 Monitor, and is started by this command.

```
ND-500 SYMBOLIC DEBUGGER.  VERSION F.  MAY 12, 1986.
FORTRAN PROGRAM.  BUBBLE.1
*
```

Now you may proceed with the debugging, as shown earlier in this chapter.

2.4. Debugging an RT-Program

In this section, you will learn to use the RT-Debugger. It is assumed that you know how to use the RT-Loader to make RT-programs (also known as foreground programs), so only a few relevant points concerning the preparation of an RT-program for debugging will be presented here.

(If you have an ordinary ND-100 Debugger version F or later, then you make an RT-Debugger by dumping the :BPUN-file that the Debugger comes on with start address 2, restart address 3 on the segment file, instead of dumping it with addresses 0 and 1 as you would with the background Debugger. These operations can only be carried out if you have SYSTEM privileges.)

The same :MODE-file that compiles and loads the BUBBLE program for the ND-100 and generates debug information is used when you prepare an RT-program for debugging. The RT-Debugger will use the loaded :PROG-file as well as the code on the RT-program segment when it is used.

You must be logged in on SINTRAN III User Areas RT or SYSTEM to use the RT-Debugger. Only one terminal at a time can use it. The following :MODE-file loads the BUBBLE program as an RT-program:

```
@rt-loader
clear-segment 200
yes
new-segment 200,,,,,
read-progfile (debugger)fortran-bubble 200,,,
declare-program bubble,,,
change-rt-description bubble 30 200,,11,,,,,
exit
```

The :PROG-file is read into segment 200 and an RT-program name BUBBLE is declared. Then BUBBLE's RT-Description is changed: it is given priority 30, it is on segment 200, then there is an empty parameter before you specify 11 as the address where execution will start. This address can be taken from the loading session with the BRF-Linker: After you have loaded the file where the main program begins, you give the BRF-Linker command LIST-ENTRIES-DEFINED. In this case, you will find that BUBBLE has the address 11.

(An alternative way to make the same RT-program would be to load the file (DEBUGGER)FORTRAN-BUBBLE:BRF with the RT-Loader command LOAD, followed by another load of the FORTRAN-1BANK:BRF file. This will generate code precisely similar to that on the FORTRAN-BUBBLE:PROG file. Then, you would have to set the priority for BUBBLE with the SINTRAN command @PRIOR after you have left the RT-Loader.)

When using the RT-Debugger, you need a few more commands than when debugging a background program. These commands are:

1) ATTACH-SEGMENT <segment number> <program file>

   which tells the RT-Debugger which segment and :PROG-file it will be dealing with

2) RT-PLACE <RT-program name>

   to get the appropriate RT-description placed in your register block.

3) GET-BREAK-STATUS

   which retrieves the information about the last RT-break that has occurred in your computer, provided that you have set the break for the current debug session to the same place in the program as what SINTRAN has stored.

This is what a debugging session may look like from terminal 39 on your computer, and if you can use the terminal as your computer's error-device.

```
@set-error-device 39 ↵
```

Make your own terminal act as error-device, if you are allowed to log in as user SYSTEM. Then SINTRAN will send the messages that RT-program breaks have been reached to your terminal. Enter the RT-Loader.

```
@rt-loader ↵

REAL TIME LOADER,  SINTRAN III VSX - K

*clear-segment 200 ↵

RT-PROGRAMS ON SEGMENT:

 BUBBLE
```

Clear the segment you want to use.

```
DELETING THIS RT-PROGRAM(S)? yes ↵
*new-segment 200,,,,, ↵
*read-progfile (debugger)fortran-bubble 200,, ↵
```

Reading the code into your segment directly from the :PROG-file. Declare BUBBLE as an RT-program.

```
*declare-program bubble,,, ↵
*change-rt-description bubble 30 200,,11,,,,, ↵
```

Make the right RT-description for your program: Set priority, segment-number and start-address. Now, you can leave the RT-Loader and start the RT-Debugger. (You get the start-address from the BRF-Linker during loading to the :PROG-file.)

```
*exit ↵
@rt-debugger ↵
ND-100 SYMBOLIC DEBUGGER.  RT VERSION F.  FEBRUARY 19, 1986.

*attach-segment 200b (debugger)fortran-bubble ↵
```

```
*rt-place bubble ↵
```

Tell the RT-Debugger which segment it will find the code for the RT-program on, and associate the RT-program name BUBBLE with it.

```
*break 12 ↵
*run ↵
```

Then set a breakpoint at line 12, and start the program.

Since the terminal we use is also the ERROR-DEVICE, we now see the message that SINTRAN gives to notify that an RT-breakpoint has been reached.

```
  BREAKPOINT IN BUBBLE AT ADDRESS       54B

*get-break-status ↵

Break at BUBBLE.12
*dis tosort ↵
TOSORT=9 8 7 6 5 4 3 2 0 1
```

This command must be given to retrieve information about the RT-breakpoint from SINTRAN.

Now, variables can be inspected etc. We see that the sort has started!

Giving a SINTRAN command from the RT-Debugger to check the status of the BUBBLE program.

```
*@list-rt-desc bubble ↵

ACTIVE     I/O-WAIT  ......  ....  ....  .....  .....  .....

        SEGMENTS 1  AND  2    REENT  NPIT APIT RING PRIORITY
INITIAL  :     200B       0B            1B   1B   0      30B
ACTUAL   :     200B       0B            1B   1B   0      30B

START ADDRESS:      11B   LAST STARTED:     10 SECS
ND-100 CPU TIME USED:     0 BASIC TIME UNITS

   P       X       T       A       D       L       S       B
 000054 000011 000001 000001 000017 001237 000100 000300

*break 13 ↵
*run ↵
*exit ↵
```

Now, note how you can leave the RT-Debugger, wait for an RT-break to occur, go back into the RT-Debugger and look at the status of the program.

```
  BREAKPOINT IN BUBBLE AT ADDRESS       62B

@rt-debugger ↵
ND-100 SYMBOLIC DEBUGGER.  RT VERSION F.  FEBRUARY 19, 1986.
*attach-segment 200b (debugger)for-bubble ↵
```

```
*break 13 ↵

*get-break-status ↵

Break at BUBBLE.13

*dis tosort ↵
TOSORT=0 9 8 7 6 5 4 3 2 1
*@list-rt-desc bubble ↵

ACTIVE      I/O-WAIT  ......  ....  ....  .....  .....  .....

        SEGMENTS 1  AND  2     REENT  NPIT APIT RING PRIORITY
INITIAL  :     200B       0B             1B   1B   0      100B
ACTUAL   :     200B       0B             1B   1B   0      100B

START ADDRESS:      11B   LAST STARTED:     22 SECS
ND-100 CPU TIME USED:     0 BASIC TIME UNITS

   P      X      T      A      D      L      S      B
 000062 000001 000001 000000 000017 001237 000100 000300


*break 13 ↵
*run ↵

  BREAKPOINT IN BUBBLE AT ADDRESS       62B

*get-break-status ↵

Break at BUBBLE.13
*dis tosort ↵
TOSORT=0 9 8 7 6 5 4 3 2 1
*exit ↵

@set-error-device 7 ↵

@__
```

Picking up the segment and the :PROG-file again. Retrieve the break-status from SINTRAN, after having set the break-point to where it was before you gave the last run-command.

Look at the status of the program, as before.

You can continue to enter and leave the debugger like this until you are sure your program is all right.

Remember to give back the ERROR-DEVICE, if you have used it!

2.5. The Capabilities of the Symbolic Debuggers

Now that you have a basic idea about how the various Symbolic Debuggers can be used, we will introduce some definitions of terms used in this manual, give some hints concerning debugging, and then an overview of the commands in the Debuggers and how they can be used. It is not very comprehensive. You must consult the following chapters for more advanced details. An overview of advanced features is given on page 24.

In the first part of this chapter, you learnt about debugging by breakpoints. The commands you need to debug by breakpoints are:

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

   telling the Debugger to stop at a certain location, called a breakpoint, in your program, optionally after having passed it the number of times specified in <count> or if the <condition> on the variables in the program is met

2) DISPLAY (<item or value>)

   to inspect the contents of variables etc.

3) RUN (<program address>)

   to execute the program until the next breakpoint is reached or the program exits by itself. The optional parameter is included so that you can resume execution from another location than where you are at present

4) EXIT

   to stop the debugging session. (It is recommended that you don't use the ESC key.)

There is an alternative to setting BREAKs at the places where you want execution to stop, if you are not debugging RT-programs. That is to step through the program one line at a time or one subroutine call at a time, and watch the state of the program as execution proceeds.

If you choose to use this strategy, the commands that you will need are

1) A LOG command, such as

   a) LOG-LINES <program area>

      to make step-points on every line in the <program area> that you specify in your source code

   b) LOG-CALLS

      to make step-point on every subroutine call in the <program area> that you specify in your source code

2) STEP (<count>)

   where count is the number of step-points you want to pass before the program is stopped. (The Debuggers can also step through code one machine instruction at a time. In that case the <count> is set to -1, and no LOG-command is needed.)

3) DISPLAY (<item or value>)

   to inspect the contents of variables etc.

4) EXIT to leave the Debugger

At this stage, you need to know precisely what a "breakpoint" and a "step-point" are.

A breakpoint is an address in your program where you want execution to stop. You may define an address indirectly as the place where code from one specific line or subroutine in your source code begins, or it may be an absolute program address. A RUN command executes your program up to and including the last instruction before the address of the breakpoint. (If you want to stop at an absolute address, use BREAK-ADDRESS.)

A step-point is a place where code generated from one specific line or subroutine in your source code begins. You must give either a LOG-LINES or a LOG-CALLS command to make your step-points, and then you move from one step-point to the next using the STEP command. (The CONTINUE and RUN commands ignore step-points.)

If you do a lot of debugging, you may want to be aware that stopping at step-points is much more time-consuming than using breakpoints, especially in the ND-100. Executing code with many step-points may hamper the execution of the programs of other users significantly.

Debugging by step-points is a looser way of debugging than debugging by breakpoints. Many programmers will use it when they work on a program that they do not know very well, to get an idea of how it works, or when they miss clues to what the problem in their program is. On the other hand, better knowledge of the code is a premise when debugging by use of one single breakpoint, so it is recommendable to use step-points as little as possible, and always to know your code well enough to be able to debug by breakpoints. However, you can use multiple breakpoints on the ND-500, as a substitute for debugging by step-points.

2.6. Advanced Features in the Debuggers

The debuggers have many features in addition to the basic ones described above. Some of them rely on step-points being available.

You can:

- control the number of times a breakpoint is executed. See the BREAK command description on page 31 for details.
- control the number of step-points to be executed. See the STEP command description on page 79 for details.
- control the values or value ranges that variables can have. You can do this with the BREAK command (see page 31) if debugging by breakpoint, or by the GUARD command if debugging by step-points (see page 45).

- change the contents of variables with the SET command, see page 77.
- display record components in addition to simple variables. See the DISPLAY command on page 38.
- display data, register- and stack contents, and disassembled code with the various LOOK-AT commands. See page 56 to 70.
- find the scope of addresses in the code that resulted in errors with the FIND-SCOPE command, which is described on page 43.
- shift your point of view (scope) to different places (lines, subroutines etc.) in your program. This is done by the SCOPE command, see page 75.
- invoke subroutines, either to try them under different conditions or to use them as diagnostic tools. See the INVOKE command on page 48.
- look for parts of the code which have not been executed. This is done while debugging by step-points by the CHECK-OUT-MODE command, see page 35.
- make macros (which may take parameters) consisting of several Debugger commands with the MACRO command on page 69.
- read macros from files instead of from the keyboard. This operation is done by the INCLUDE command, see page 47.
- reserve another terminal from which the debugging can be done, so that you avoid having the debugger mess up your screen pictures. The RESERVE-TERMINAL command does this for you, see page 72.
- optimize the code on some ND-100 CPUs with the STACK-INSTRUCTIONS command, which is described on page 78.

2.7. Additional Features in the ND-500 Debugger

The ND-500 Symbolic Debugger has some features in addition to those found in the other Debuggers. These are:

1) it is possible to debug reloaded programs. The debugger checks if you have several modules with the same name in the debug information for a domain, and if so, it assumes that a reload of a multimodule system has occurred. The information for the module with the same name that was loaded last is used by the debugger. For details about reloading, see the Linkage-Loader User Guide, ND-60.182 EN.

2) three new functions:

   a) MOD, which returns the remainder from integer divisions.

   b) TYPEOF, which returns the type of the variable you are inspecting (as represented in the debug information). If the variable is a boolean, TYPEOF returns BOOLEAN, if it is a record, it returns RECORD.

   c) SPECIAL, which makes it possible to display parameter names that otherwise have special meanings to the debugger, such as "+", "IND" etc. Example:

      *DISPLAY SPECIAL 'ind' ↵

      will display the variable ind.

3) it can display PLANC variant records. See page 41.

4) the ERRCODE location on the stack can be displayed as a variable. The meaning of the various ERRCODEs can be found in appendix A of the manual SINTRAN III Monitor Calls, ND-60.228 EN.

5) you can log returns from subroutines. See the command LOG-CALLS on page 52.

6) up to 20 simultaneous breakpoints can be set. See the command MULTIPLE-BREAK-MODE on page 71.

CHAPTER 3

COMMANDS - DETAILED DESCRIPTION

3. COMMANDS - DETAILED DESCRIPTION

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

Parameters are enclosed in left and right angle brackets, < and >. 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.

The parameters that you can give to the commands described in this chapter are explained in the chapter about SYMBOLIC DEBUGGER PARAMETERS on page 83. Please refer to that chapter if you do not know what the parameter notation means.

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, if you have not specified a maximum number of levels. In the latter case, only as many routines as you have asked for will be printed. If you are debugging a multi-segment program, the segment number of the routines are printed together with the routine names.

If your program has not been started, you have no call hierarchy. The B-register that points to your stack-frame has not been initialized, and you will get an error message if you attempt to list the hierarchy.

You use the parameter if you want to set the number of levels to display to a different number than the default number.

```
*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.

Here is how you align the listing with the routine PRINT:

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

BREAK AT PRINT.804
*
```

The declaration line in the routine PRINT will be numbered 800. The rest of the lines inside the routine PRINT will be adjusted relative to the declaration line.

The declaration line is the line where the routine or main program is declared. If the declaration is split over several lines, the last of these lines is used as declaration line when you align the listing with a routine name, except in FORTRAN, where the first line is the declaration.

3.3. ATTACH-REENTRANT-SEGMENT <file name> <segment name>

This command is intended for debugging of programs that have been loaded as multi-segment programs on the ND-100. (You should not confuse SINTRAN III's segments on the ND-100 with the segments on the ND-500 - they are quite different concepts.)

The <file name> is the name of a file whose contents have been dumped on SINTRAN III's segment file as a part of a multi-segment program. If you do not specify an extension to the <file name>, the default extension is :PROG.

The <segment name> is the same as the one used while linking the multi-segment program with the BRF-Linker and then used when dumping the program to the segment file.

You can decide to change the contents of the segments or to protect them from such changes with the commands SEGMENT-WRITE-PERMIT and SEGMENT-WRITE-PROTECT, see page 76.

After you have given this command, the Debugger will display the segment number in addition to other information about the location of breakpoints, program addresses etc.

3.4. ATTACH-SEGMENT <segment number> (<W>)

This section describes the command on the ND-500.

ND-500 programs may use several segments. The debug-information for a segment other than that for your main program is not available until you have attached it with this command. After it is attached, you can set breaks on the attached <segment number>.

The command SEGMENT-INFORMATION provides a list of all active segments.

3.5. ATTACH-SEGMENT <segment number> <prog-file name> (<W>)

This section describes the command in the RT-Debugger.

When you give this command, the segment of the RT-program that you want to debug is made known to the debugger, together with the name of a program file that contains that same code as the RT-program. (Make sure that this really is so, or you are in trouble!) If you do not specify an extension to the file name, the default extension is :PROG.

You have to give the command RT-PLACE after you have given this command, to get the appropriate RT-description of the program.

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

Sets a breakpoint at the specified item, and removes the previous breakpoint set by BREAK. The position will be set according to the first parameter. See on page 88 how you specify routines and labels, and on page 91 how you specify line numbers.

When you give a RUN or CONTINUE command with an active breakpoint, the program will execute until it reaches the breakpoint or the program exits. If execution reaches a breakpoint, the location of the breakpoint is displayed, together with the segment number if you are debugging an ND-100 multi-segment program.

When source code macros or INLINE routines in PLANC or FORTRAN are expanded during compilation, they are not given line numbers. The alternative is to set breaks at program addresses with the BREAK-ADDRESS command.

If a routine name is specified, the breakpoint is set at the first executable line in the routine.

If you use an ND-500 and version F or later of the Symbolic Debugger, you can give the command MULTIPLE-BREAK-MODE ON, which will allow you to set up to 20 simultaneous breakpoints with the BREAK command. See page 71 for details.

If a positive number K is specified for the count parameter, the program will break when the breakpoint has been reached K times. Then,

the (<count>) is cleared, and you will get a break every time the breakpoint is reached afterwards. (Note that you cannot specify the (<count>) and (<condition>) parameters when multiple breaks are used.)

```
*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 conditional expression (described on page 86) is given in the last parameter, control passes to the Debugger at the breakpoint only if the condition is true and the variable is local:

```
*BREAK $10 I > 5 ↵
*RUN ↵
   4          16.00
   5          25.00
   6          36.00
CONDITIONAL BREAK AT SQRS.7
*DISPLAY I ↵
I=6
*
```

$10 specifies label 10 in the FORTRAN subroutine SQRS. Since the (<count>) parameter must be a number, the Debugger knows that the I is the start of a conditional expression.

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 example on page 91.

You can also create breakpoints by using GUARD, see page 45.

Here is an example with multiple breakpoints on the ND-500. It uses the bubble-sorting program which was used in chapter 2.

```
ND-500 MONITOR  Version H00 86. 5. 6 / 86. 5.14
N500: debugger fortran-bubble ↵
ND-500 SYMBOLIC DEBUGGER.  VERSION F.  MAY 12, 1986.
FORTRAN PROGRAM.  BUBBLE.1
*multiple-break on ↵
*break 12 ↵
*break 13 ↵
*break 14 ↵
*dis j,tosort; run ↵
J=0
TOSORT=9 8 7 6 5 4 3 2 1 0

Break at BUBBLE.12
*dis j,tosort; run ↵
```

```
J=10
TOSORT=9 8 7 6 5 4 3 2 1 0

Break at BUBBLE.12
*reset-break 12; break 6 ↵
*dis j,tosort; run ↵
J=9
TOSORT=9 8 7 6 5 4 3 2 0 1

Break at BUBBLE.13
.
.

*exit ↵
```

Note how you can remove breaks that you are not interested in any more. Here, the break at line 12 is reset.

3.7. BREAK-ADDRESS <program address> (<count>)

This command is similar to the BREAK command, except that the breakpoint is specified directly as a program address. Addresses are assumed to be in decimal unless you specify octal, hexadecimal etc. You can use the segment notation, such as 5'54B, meaning segment 5, location 54 octal. If you are debugging a multi-segment program on the ND-100, you are only allowed to use addresses on the currently active segment. 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.

3.8. 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 ↵
PLANC PROGRAM.EXAMPLE.OUTER.9
 ...
*BREAK PARALLEL ↵
*RUN ↵
*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.

3.9. CHECK-OUT-MODE (<program area>)

This command is not available in the RT-Debugger.

The CHECK-OUT-MODE command removes the step-point on each line in the specified routine/module after it has been reached. You can thus obtain a list of all lines which have never been executed, by using the DUMP-LOG command. If you are debugging an ND-100 multi-segment program, and give no parameter, only the current segment will be checked.

If no area is specified, all lines are checked.

See the examples on page 99. Note:

> Since CHECK-OUT-MODE removes step-points,
> you cannot do the following:
>
> *LOG-LINES <program area>
> *CHECK-OUT-MODE <subroutine/module name>
> *STEP
>
> You need to do this instead:
>
> *CHECK-OUT-MODE <subroutine/module name>
> *BREAK <routine, label or line>
> *RUN

If you give the commands LOG-LINES or LOG-CALLS on an area before you give the CHECK-OUT-MODE command, then only that area is checked. Default is LOG-LINES on the entire program.

If you specify LOG-CALLS, only the subroutine calls will be checked. (On the ND-500, you will also log the corresponding returns.)

This example shows how a program consisting of several subroutines was checked out:

```
*log-lines routines.insert node.13:routines.main.83 ↵
*check-out-mode ↵
*run ↵
.
.
.
Program terminated at MAIN.83
*dump-log ↵
FIND_KEY.53 54 55 56 57 58 59 60 61 62 63 64 65 66
*
```

We log the lines inside the program called routines from line insert node.13 to main.83, specify check-out-mode, and start the program. After it has terminated, we dump the log, and find that lines 53 to 66 in the subroutine find key have not been executed.

3.10. COMPARE-DATA <low> <high> (<output file>)

The data area specified by the two addresses, which are given as numbers or via ADDR, is compared to the file contents (:DSEG file on the ND-500, :PROG-files for one- and two-bank programs on the ND-100). In one-bank ND-100 programs, both program and data are compared. In multi-segment ND-100 programs, the current segments are compared. 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 ↵

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
*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.

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

3.11. COMPARE-PROGRAM <low> <high> (<output file>)

The program area specified by the lower and upper bounds is compared to the program file contents (:PSEG files on the ND-500). Modified locations are displayed with address, old contents and new contents. In multi-segment ND-100 programs, the current segments are compared.

The default output file is the terminal; the default file type is :LIST. See also COMPARE-DATA on page 36.

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
*
```

3.12. CONTINUE

Execution is resumed from the current location. If you want to specify where you want to resume execution from, use RUN. See page 74. 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.

3.13. DISPLAY (<item or value>)

Any named element in the information generated by the compilers when the DEBUG-option is on is an item. Examples of valid items are routine names, labels, and variables. Values are constants or expressions.

You may qualify items with a dot notation to get at items not currently in scope. The items may for instance be on code that has been compiled separately. For more information about how you use the dot notation to specify places in source programs, see page 88. The dot notation can also be used to specify record components in some languages, see subsection below.

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. You will find more examples of simple use of the DISPLAY command on page 11

DISPLAY has a related command, FORMATS-DISPLAY, with which you can choose how numeric values are displayed. It can be combined with the "Ada notation" for values, so that you have a quite flexible tool for inspecting the state of the variables etc. The FORMATS-DISPLAY command is described on page 44, the Ada notation on page 83.

```
*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 cannot use blanks as separators between expressions.

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

3.13.1. Pascal and PLANC Records

Many programs in these languages use dynamic allocation of storage, with records containing pointers to other records which are not accessible through any variable names. This is the case in the following PLANC program, where the record current of type element have two pointers left and right to other elements. After execution of lines 12 and 20 in the program, these to pointers contain the addresses of two new records of type element.

(PLANC and Pascal are quite similar languages. For instance, PLANC uses the same dot notation for accessing record components as Pascal, and the USING .. ENDUSING sequence is similar to Pascal's WITH ... DO BEGIN ... END. And in PLANC, A+B =: C is the same as C := A+B; in Pascal. Two visible differences are the use of modules and explicitly named stacks in PLANC, but that does not have any practical consequences here.)

```
 1          MODULE records
 2          INTEGER ARRAY : stack(0:500)
 3          TYPE element = RECORD
 4             BYTES : name (0:6)
 5             element POINTER : left, right
 6          ENDRECORD
 7          element : current
 8          PROGRAM : useelements
 9             Inistack stack
10             USING current
11                'Current' =: name
12                New element =: left
13                USING left
14                   'Left   ' =: name
15                   NIL =: left =: right
16                ENDUSING
17                New element =: right
18                USING right
19                   'Right  ' =: name
20                   NIL =: left =: right
21                ENDUSING
22             ENDUSING
23          ENDROUTINE
24          ENDMODULE
```

The Symbolic Debugger uses the same dot notation to access record components. In addition, the operators IND and ADDR may be used to find the contents of an item pointed to by a pointer, and the address of an item. Consider this Debugger session on the program above:

```
@debug record-planc ↵
PLANC PROGRAM.  RECORDS.USEELEMENT.8
*break 20 ↵
*run ↵


Break at USEELEMENT.20
```

We want to break the program after the record components have had values assigned to them. Then, we DISPLAY their values to see the results of the assignments.

```
*display current,current.name,current.left ↵
CURRENT=          NAME(0:6)        LEFT= 000011B   RIGHT= 000017B
```

The Debugger informs us that current contains an array name with indexes in the range 0:6. Furthermore, left and right contain the values 11 and 17 octal, which are the memory addresses of the first location belonging to the records these pointers point to.

```
CURRENT.NAME=Current
CURRENT.LEFT=000011B
```

Then, we see that current.name contains the text "Current", while left points to address 11 octal.

```
*display current.left.name,current.left.right ↵
CURRENT.LEFT.NAME=Left
CURRENT.LEFT.RIGHT=NIL
```

Some more intricate accesses of record components.

```
*display ind(current.left) ↵
IND(CURRENT.LEFT)=                NAME(0:6)        LEFT= NIL
RIGHT= NIL
```

We display the record at the location pointed to by current's left pointer.

```
*display addr(current),ind(addr(current)) ↵
ADDR(CURRENT)=000765B
IND(ADDR(CURRENT))=               NAME(0:6)        LEFT= 000011B
RIGHT=000017B
*
```

Here, the Debugger tells us that current is at address 765 octal, while the object located at that address (which we now know is current itself!) has the same contents as current was found to have above.

3.13.2. Displaying PLANC Variant Records on the ND-500

PLANC allows you to create record types which are variants of a basic type. The variants will contain all the components of the basic type, plus components which are particular to the variants of the basic type that you want to use.

As an illustration, consider the following program, where the type Vehicle form the basis for the variant types Bus and Truck:

```
1          MODULE This
2          INTEGER ARRAY : Stack (0:200)
3
4          TYPE Vehicle = RECORD
5             REAL : Weight, Length, Width, Height
6          ENDRECORD
7
8          TYPE Bus = Vehicle RECORD
9             INTEGER : Seats, NumberOfCrew
10         ENDRECORD
11
12         TYPE Truck = Vehicle RECORD
13            REAL : LoadCapacity
14            BOOLEAN : Automatic
15         ENDRECORD
16
17         Bus : LocalBus := (100.0, 10.1, 3.4, 2.1, 44, 1)
18         Bus : ToursBus := (150.0, 11.3, 3.4, 2.1, 35, 3)
19         Truck : TipTruck := (50.5, 8.6, 3.2, 1.9, 45.0, TRUE)
20
21         PROGRAM : Variant
22            INISTACK Stack
23         ENDROUTINE
24         ENDMODULE
25
```

The ND-500 debugger makes the following DISPLAY commands possible:

```
*display this ↵
LOCALBUS          STACK(0:200)     TIPTRUCK          TOURSBUS
*display localbus.vehicle ↵
LOCALBUS.VEHICLE=                  WEIGHT=  1.00000E+02
LENGTH= 1.01000E+01                WIDTH=  3.40000
HEIGHT=  2.10000
*display localbus.bus ↵
LOCALBUS.BUS=              WEIGHT= 1.00000E+02
LENGTH=  1.01000E+01       WIDTH= 3.40000
HEIGHT= 2.10000            SEATS= 44
NUMBEROFCR= 1
*display tiptruck.automatic ↵
TIPTRUCK.AUTOMATIC=TRUE
```

3.14. DUMP-LOG (<output file>)

Sometimes, it is desirable to keep trace of the progress of a program, to see which source code lines and procedure calls are executed. When you use the Symbolic Debugger on the ND-100 and ND-500, the DUMP-LOG command helps you do this. The RT-Debugger has no LOG-LINES or LOG-CALLS commands, so you cannot log the progress of an RT program in the same way. For details about LOG-LINES see page 54, and see page 52 about the LOG-CALLS.

The output of the DUMP-LOG 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 52.

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 54.

If CHECK-OUT-MODE was specified last, a list of all the lines or routines (in the area specified in the CHECK-OUT-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.

If you are debugging a multi-segment ND-100 program, the segment number will be printed each time it changes.

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:

```
*CHECK-OUT-MODE ↵
*DUMP-LOG ↵
```

The default output file is the terminal; the default file type is :LIST. For details about the CHECK-OUT-MODE command, see page 35.

3.15. 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, pages 50 and 51.

3.16. EXIT

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

3.17. FIND-SCOPE <program address>

This command finds the module or routine, and the line number, that correspond to the specified program address. The current scope status is displayed. If you are debugging an ND-100 multi-segment program, this command can only be used on the currently active segment.

Addresses are specified with either ND-500 syntax, octal or hexadecimal values. For example:

```
*find-scope 1'450B ↵
ROUTINES.WRITE_TREE.49
*
```

The address originates from line 49, belonging to the routine WRITE_TREE, which is located inside a main program (or PLANC module) named ROUTINES.

The difference between FIND-SCOPE and SCOPE (see page 75) is that FIND-SCOPE needs a program address, while SCOPE has a module, routine or line number for its parameter. Furthermore, FIND-SCOPE returns the scope of an address in the executable code, while SCOPE moves you to

ROUTINES.WRITE_TREE.49 for instance, so that you can DISPLAY local variables in the routine WRITE_TREE. etc.

3.18. 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. @del; You obtain the default setting by giving an empty format specification.

Here is what the codes mean:

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

Default values are A for DISPLAYing character strings, O for pointers, D for integers and F for reals.

An example is given on page 38.

3.19. 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 when the Debugger prompts for parameters to this command.

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 56.

3.20. GET-BREAK-STATUS

This command can only be used with the RT-Debugger.

When debugging an RT-program, you may leave the RT-Debugger after you have started the program with the RUN-command. The RT-program may then run for any period of time until the code where the break was set is reached. When this happens, a message is printed on the error-device, such as:

```
BREAKPOINT IN <program name> AT <address (octal)>
```

and you can enter the RT-Debugger. It will not "remember" where the breakpoint washen you left it, so you must set a breakpoint at exactly the same location as you had before you exited the RT-Debugger. This is necessary if any debugging is to take place - now, the debugger can verify that SINTRAN III has detected the correct breakpoint. (To set the breakpoint, use the BREAK <routine, label or line> command.)

Then, give the command GET-BREAK-STATUS . The RT-debugger then responds with

```
BREAK AT <Routine name>.<source line no.>
```

and you can LOOK-AT registers, DISPLAY variables and so on, as you would in the other debuggers.

If the locations do not match, you get the error message "No active breakpoint".

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

Guard cannot be used in the RT-Debugger.

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. You can use expressions for low and high.

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

*GUARD x 0 : 10 ↵        0 to 10 is the permitted range.
*RUN ↵
GUARD VIOLATION AT MAIN.55
*DISPLAY X ↵
x=11
*RUN ↵
```

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 allowed. Array elements (packed and unpacked) and record components (packed and unpacked) may

also be specified. Composite items (arrays and records) are not allowed.

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 ↵                          This removes GUARD.
*RUN ↵
```

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.

3.22. 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 angle brackets: < >. Optional parameters are enclosed in parentheses and angle brackets: (< >).

3.23. INCLUDE-COMMANDS <file name>

If you are debugging an ND-500 program or a background program on the ND-100, this command will include the commands from the file (which has default file type :SYMB).

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.

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

You cannot use this command from the RT-Debugger.

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 you are debugging a multi-segment program on the ND-100, you can only invoke routines on the current segment, and these routines cannot change segments.

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) and constants are allowed. If the routine is a normal PLANC routine, simple variables (ENUMERATION, BOOLEAN, POINTER and INTEGER) and records are allowed.

Furthermore, all parameters that you use when you INVOKE a subroutine are given a size of one word, so simple data types of other sizes, such as REAL8 on the ND-500 or REAL4/REAL6 on the ND-100 cannot be passed as a parameters to a routine in this manner. Use the SET-command after you have INVOKEd instead. Records, strings and arrays are passed as pointers, therefore they occupy one word on the stack.

COBOL has separately compiled subprograms instead of subroutines, and these subprograms may not be INVOKEd in the manner described here.

Here is an example of how the INVOKE command can be used. The program is supposed to read a sequence of keys, which are single bytes for the sake of simplicity, and sort them into alphabetical order as they are read by the program. The contents of the resulting data structure (which is a "binary tree" in the program used as example here) is listed by the subroutine write tree called from the program at the end of execution. However, write tree can be INVOKEd at any stage during execution to see how the sorting proceeds (provided the the data structure is consistent at the time of invocation).

```
*break 64 ↵
*run ↵
```

We set a break at line 64. This is the line where insertion of a new key is initiated. Then the program is allowed to run.

```
Keys: SDFGLKHBUUREJDFNGWERTOIUYNXCVB ↵
BREAK at MAIN.64
*break 64 10 ↵
*run ↵

BREAK at MAIN.64
*invoke write tree ↵
```

After the program has read its unsorted sequence of keys, we insert 10 keys before the next break.

```
Sorted keys:

BDFGHKLSU

*reset-break ↵
*run ↵

Sorted keys:

BCDEFGHIJKLNORSTUVWXY

Program terminated at MAIN.67
```

We execute the write tree subroutine to see what the data structure contains. As we see, the keys are in alphabetical order, but duplicates are missing.

This is what we get at the end of execution.

You cannot transmit PLANC invalues to the routines that you start with the INVOKE command. Instead, move your scope inside the routine you want to start with a BREAK before the first line in the subroutine, and then use SET to give the variable @ the right value.

3.25. 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.

The Debugger shares the trap-handler with the user program. Before the Debugger starts the user program, it takes the traps PRT, BPT, ATW and SIT (29, 20, 23 and 17). It must have these traps to detect break- and step-points etc.

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
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.

The trap conditions marked with an asterisk (*) are disabled if you give the LOCAL-TRAP-DISABLE command without any parameters.

When a trap is disabled, nothing happens if the condition occurs, unless the condition is fatal. In the latter case (for instance, if a protect-violation occurs), the monitor takes over and reports the error.

3.26. 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 HELP, all available trap conditions are listed on the terminal. The trap conditions marked with an asterisk (*) are enabled if you give the command LOCAL-TRAP-ENABLE without any parameters.

Example:

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

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

If the enabled trap condition occurs, the Debugger will take over and break.

3.27. LOG-CALLS <program area>

This command cannot be used from the RT-Debugger.

The LOG-CALLS command logs all routine calls in a cyclic buffer. This buffer can be inspected by means of the DUMP-LOG command (see page 42). The buffer can hold a maximum of 200 entries.

If you do not specify any area, the whole program is checked, except when debugging multi-segment ND-100 programs, where only the current segment is checked. To log on other segments, you must first enter a break on that segment.

On the ND-500, return from subroutines is also logged.

```
*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 give some examples.

3.27.1. 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.

3.27.2. LOG-CALLS and GUARD

You need to use LOG-CALLS or LOG-LINES before GUARD only 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.

3.27.3. LOG-CALLS and STEP

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

Each ↵ (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. On the ND-500, returns from subroutines are logged, too.

3.28. LOG-LINES <program area>

This command cannot be used from the RT-Debugger.

The LOG-LINES command logs all executed line numbers in a cyclic buffer. This buffer can be inspected by means of the DUMP-LOG command (see page 42). 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.

If you are debugging an ND-100 multi-segment program, only the current segment is logged. You can set breaks on any segment by first entering a break on that segment.

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

3.28.1. 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.

3.28.2. LOG-LINES and STEP

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

Each ↵ (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.

3.29. 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. The data address and the count can be specified with constants, as described on page 83. The count can also be given as an expression, see page 86. Output from these commands can be sent to the file named in the optional (<output file>) parameter. The file has default type :SYMB.

```
*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.

↵ (Carriage Return) causes an advance to the next address item without changing the contents of the current address. (An address is a 16 bit word on the ND-100 and a 32 bit word on the ND-500.) All subcommands are terminated by CR. Printing a dot (.), a semicolon (;), or EXIT returns you to the Debugger:

```
*FORMATS-LOOK-AT 0 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 65.

Here are some examples that illustrate the notation.

First, we modify the data used in the BUBBLE program used in the introductory example on page 11. This example is run on the ND-500.

```
*display tosort ↵
TOSORT=9 8 7 6 5 4 3 2 1 0
*look-at-data addr(tosort(5)) ↵
D 1'        104B: 00000000004B          4       44 ↵
D 1'        110B: 00000000003B          3       . ↵
*display tosort ↵
TOSORT=9 8 7 6 5 44 3 2 1 0
*run ↵

Program terminated at BUBBLE.14
*display tosort ↵
TOSORT=0 1 2 3 5 6 7 8 9 44
```

Then, we show some more possibilities, this time using another program on the ND-100:

```
*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 V  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 ; ↵
```

Now consider a more complex example. On page 48, we presented a program using a simple data structure, which we can use to maintain lists of keys sorted in alphabetical order. This is achieved by associating each key with pointers to other keys, which likewise have pointers associated with them. These pointers contain the addresses of other keys, or zero if they do not point to any further keys. In Pascal and PLANC, this corresponds to a RECORD with the following structure:

```
node = RECORD
   key : BYTE;
   left, right : node POINTER
ENDRECORD;
```

We make the sorted structure by making the left pointer point to keys smaller than the key of the current record, and the right pointer point to bigger keys. Now, if we follow the left pointer from node to node, the consequence of this is that we get successively smaller keys.

On page 48, we investigated this data structure with subroutines that were part of the program we were debugging. But from the above description of the slash command, it is evident that we can use it as an alternative to the DISPLAY command or invocation of such routines to inspect the data structure. This is an example of how you do this on the ND-500:

```
*break 80 ↵
*run ↵

Keys: OGHFJSDDKUIOWERTMNXCVBSDFLGKJNBQWERTUOIUHJBXCVBMN ↵
BREAK at MAIN.80
*break 80 30 ↵
*run ↵

BREAK at MAIN.80
*invoke write tree ↵


Sorted keys:
BCDEFGHIJKLMNORSTUVWX

*formats-look-at a h ↵
```

Now, we have run the program so that it has a random string of bytes to use as keys. We repeat a loop 30 times to read and insert 30 keys into the structure. Then use the INVOKE command to see what we have got in the structure thus far.

Next, the Debugger is told to output the contents of the addresses in alphabetical and hexadecimal form.

```
*look-at-data root ↲

D 1'      1570B: 4F000000H O     ↲
D 1'      1574B: 0800038CH       / ↲
D 1'      1614B: 47000000H G     ↲
D 1'      1620B: 080003B4H     4 / ↲
D 1'      1664B: 46000000H F     ↲
D 1'      1670B: 080003F0H     p / ↲
D 1'      1760B: 44000000H D     ↲
D 1'      1764B: 080004CCH     L / ↲
D 1'      2314B: 43000000H C     ↲
D 1'      2320B: 080004F4H     t / ↲
D 1'      2364B: 42000000H B     ↲
D 1'      2370B: 00000000H       . ↲
```

root is a pointer, thus it contains an address of a location in physical memory. We LOOK-AT it, and see that that address contains an 0. The next address is a pointer to the next smaller key. A / brings us there. The key is G, which is smaller than 0. Proceeding like this, we uncover an F, D, C, and a B before we find no more meaningful pointers. Which is the same as what we learned from INVOKEing the write tree routine: The smallest element in the structure is a B.

3.30. LOOK-AT Subcommands

The following subcommands apply to LOOK-AT-DATA, LOOK-AT-PROGRAM, LOOK-AT-REGISTER, and LOOK-AT-STACK.
The LOOK-AT commands may be abbreviated in the same way as the commands to the Symbolic Debugger itself, see page 3.

The LOOK-AT commands that are available in the different Debuggers. Parameters are indicated within < ... >. The + signs indicate commands which have higher priority than the others, to resolve ambiguities.

ND-100	RT-Deb	ND-500	Commands	Parameters
●	●		BREAK	
		●	BYTE	
●	●	●	CODE	<INSTRUCTION>
●	●	●	+DATA	
●	●	●	DOUBLE-FLOATING	
●	●		DOUBLE-WORD	
●	●	●	+EXIT	
●	●	●	EXTRA-FORMATS	<FORMATS A, D, F, H, I OR O>
●	●	●	FLOATING	
●	●	●	FORMATS	<FORMATS A, D, F, H, I OR O>
		●	HALF-WORD	
●	●	●	+HELP	<COMMAND NAME>
●	●	●	NEXT	
●	●	●	PREVIOUS	
●	●	●	+PROGRAM	
●	●	●	REGISTER	
		●	SEARCH	<BYTE LIST>
●	●	●	STACK	
●	●	●	WORD	

Here is how you list the subcommands:

```
*look-at-data 20 ↵
D 1'          24B: 000000000000B              0      HELP ↵
COMMAND NAME: ↵
 BYTE
 CODE                <INSTRUCTION>
+DATA
  .
  .
  .
*exit ↵
```

BREAK (ND-100 and RT)

This command is a supplement to the BREAK-ADDRESS command on the ND-100 Debugger and the RT-Debugger. It sets a breakpoint at the program address you are LOOKing at, so that your program will go into the break mode the next time it executes the instruction at that address.

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:                        000000000000B
RETURN ADDRESS:                    334020000000B
NEXT B:                            004636000000B
AUX:                               000351470018
NO. OF PARAMETERS:                 224064071308
D 00000000024B             24B: 032002537758 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 68.

WORD, FLOATING, and DOUBLE-FLOATING

DOUBLE-WORD (ND-100 and RT)

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 ; ↵
*
```

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

```
*DISPLAY ↵
X= 1.25600      Y8= 1.256000000000000          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  1.256000000000000 ; ↵
*
```

FLOATING is useful for inspecting the values of real numbers. DOUBLE-FLOATING is only helpful for real numbers stored in 2 words. It also displays 48 bit floating numbers on the ND-100.

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.

SEARCH <byte string> (ND-500 only)

This command searches for a sequence of bytes or an instruction in memory. The byte string can be input as

a) a series of numbers: SEARCH 102B,127B,177B ↵

b) an assembly instruction: SEARCH W1 := 3B ↵

c) a string of bytes inside a pair of apostrophes: SEARCH 'PETER' ↵

Note that all letters are converted to capitals before the search, so that you cannot specify SEARCH 'Peter' ↵ if you have a the string 'Peter' somewhere in memory. Use SEARCH 80,101,116,101,114 ↵ instead.

Examples:

```
*look-at-data 0 ↵
D 1'          0B: 000000000000B          0      byte ↵
D 1'          0B: 000B    0   search 80,101,116,101,114 ↵
D 1'          4B: 120B   80 P ↵
D 1'          5B: 145B  101 e ↵
D 1'          6B: 164B  116 t ↵
D 1'          7B: 145B  101 e ↵
D 1'         10B: 162B  114 r . ↵
*look-at-program 0 ↵
P 1'          0B: 0B search w1 := 3b ↵
P 1'         56B: W1 := 3B ↵
P 1'         60B: CALL 1'26343B,0B . ↵
*
```

3.31. 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! Output from this command can be sent to the file named in the optional (<output file>) parameter. The file has default type :SYMB.

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 an 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 57 and 58.

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

Inspect and modify CPU registers. This command is similar to the LOOK-AT-DATA command. Output from this command can be sent to the file named in the optional (<output file>) parameter. The file has default type :SYMB.

```
*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
PSEG:   000157B      0    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. The PSEG shows which register you are on if debugging a multi-segment program.

Here is an example on the ND-500:

```
*look-at-register b ↵
B:      01000000210B   134217864      / ↵
D 1'       210B: 000000000000B          0 ↵
D 1'       214B: 000000000000B          0      register ↵
P:      01000000614B   134218124      ↵
L:      01000000614B   134218124      ↵
B:      01000000210B   134217864      ↵
R:      01000000470B   134218040    8 . ↵
*
```

3.33. 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. Output from this command can be sent to the file named in the optional (<output file>) parameter. The file has default type :SYMB.

The stack handling related to subroutine calls is usually done by special machine instructions. (An exception is some ND-100 CPUs.) Therefore, the display of the stack looks the same on the same CPU, no matter what programming language you use.

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 ↵
```

This is what you see when you LOOK-AT the stack on an ND-100:

```
*look-at-stack,, ↵
RETURN ADDRESS:      005101B   2625  A
```

The address to which you will return when exiting this routine.

```
PREVIOUS B:          000200B    128
```

This is the B register of the previous stack frame.

```
STACK POINTER:       000021B     17
```

Points to the beginning of the next stack frame.

```
MAX. STACK:          000311B    201  I
```

Points to first word after the last element in the stack.

```
LEXIT:               004364B   2292  t
```

For special use by run-time system.

```
ERRCODE:             000000B      0
```

The error code from the current routine is returned here.

```
D 000015B     -172B: 000335B     221  ] / ↵
```

To the left, you see the address of the first local variable in the routine. Then comes -172B, the address of that location relative to the B register, followed by the contents of the location.

Now, all the various LOOK-AT subcommands can be used. By default, you will be inspecting the addresses of the parameters of the current subroutine. Next will give you the stack frame of the next subroutine on the stack, PREVIOUS that of the previous subroutine.

```
D 000335B: 000110B      72  H ↵
D 000336B: 000000B       0    . ↵
*
```

This is what you see when you LOOK-AT the stack on an ND-500:

```
*LOOK-AT-STACK,, ↵
PREVIOUS B:                    01000000314B
```

This is the B register of the previous stack frame.

```
RETURN ADDRESS:                01000000222B
```

The address to which you will return when exiting this routine.

```
NEXT B:                        01000000224B
```

This is the B register of the next stack frame.

```
AUX:                           00000000000B
```

This will contain the error code for the subroutine that you are in.

```
NO. OF PARAMETERS:             00000000002B
```

This is the number of parameters that were transferred to the subroutine whose stack frame you are examining.

```
D 01000000604B              24B: 01000000074B PREV ↵
```

To the left, you see the address of the first local variable in the routine. Then comes 24B, the address of that location relative to the B register, followed by the contents of the location.

Finally, a LOOK-AT subcommand to inspect the stack frame of the subroutine from which the current subroutine was called is given. (Use the subcommand NEXT to LOOK-AT the next stack frame.)

```
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 ↵
*
```

The NEXT command is the opposite of the PREVIOUS command: It moves to the next stack frame on the stack.

These command are useful if your program uses many routines that call each other, such as recursive routines, since you can observe the previous routine calls and their parameters.

3.34. 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 a 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
*

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

The first parameter you give will be inserted here.

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
*
```

3.35. MULTIPLE-BREAK-MODE <ON/OFF>

This command is only available from version F of the SYMBOLIC DEBUGGER on the ND-500.

When using this command, you switch between the "ordinary" debugger mode with one breakpoint and a mode where you can handle up to 20 breakpoints simultaneously. If you use MULTIPLE-BREAK-MODE OFF, the last breakpoint that you used is removed when you set a new one. If you use MULTIPLE-BREAK-MODE ON, then the breakpoints are not removed as you set new ones, until you reach the limit of 20 breakpoints.

When MULTIPLE-BREAK-MODE is ON, you cannot use conditions in your BREAK commands.

If you enter this command without an option, you get a list of currently active breakpoints.

3.36. 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). If you do not specify an extension to the file name, the default extension is :PROG. The PLACE command cannot be used while debugging reentrant multi-segment programs.

When you do a PLACE, 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 78.

3.37. 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
*
```

If you are debugging a multi-segment program, information will be given for all segments that are currently attached.

3.38. REENTRANT-PLACE <Reentrant-name>

This command is available on the ND-100 Debugger only.

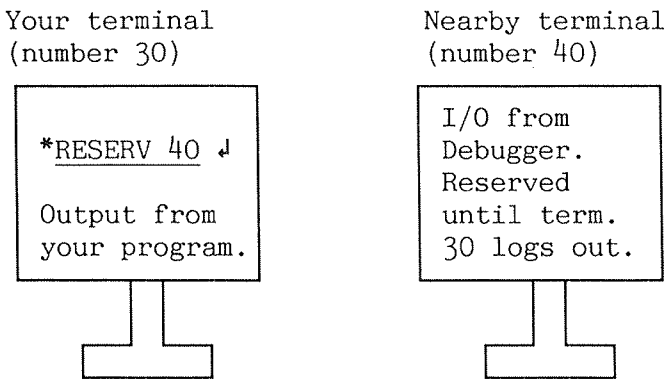
This command is used to initiate a reentrant program for the Symbolic Debugger. The reentrant program must be loaded as a multi-segment system by the BRF-Linker. Before this command can be given, the main segment of your program must be attached with the ATTACH-REENTRANT-SEGMENT command, see p. 30.

3.39. RESERVE-TERMINAL <logical device number>

This command can be used both on the ND-100 and the ND-500, but not for RT-programs.

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:



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.

3.40. 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 you are debugging multi-segment ND-100 programs, you can only reset on one segment at a time. No parameter means current segment in this case.

If a program area is specified, the breakpoints at addresses in that area are removed.

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

```
*RESET-BREAKS ↵
*RUN ↵
```

Here is how you normally remove all breakpoints and step-points:

```
*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 ↵
```

3.41. RT-PLACE <program name> (<W>)

This command only exists in the RT-Debugger. It puts the RT-description of the program you want to debug into the register block.

3.42. 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.

3.43. 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. Note that the routine that you specify must be active, which means that at least one stack frame describing an invocation of the routine must be on the stack. If you are debugging multi-segment ND-100 programs, and set the scope to another segment than the current, the new scope will be printed, but then reset to the current scope.

```
*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
*
```

3.44. SEGMENT-INFORMATION

This command is relevant to the ND-500 and to the RT-Debugger.

On the ND-500, 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    0B       (SYM-DEB)DEBUGGER
DSEG 26    0B
LINK 26    0B
PSEG 30    0B       (PACK-REM:DOMAINS)FORTRAN-LIB-H00
DSEG 30    0B
LINK 30    0B
```

When the Debugger starts, a monitor call to the ND-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. C1 and C2 are segments used by the Debugger when connecting the file as a segment. Since the Debugger uses segments 0 and 2, if you use the ND-500 Monitor call FSCNT, you must user other segments. See page 105.

On the RT-Debugger, the information that you get looks like this:

```
*segment-information ↵
   200B (OWN-USER)BUBBLE-SORT
*
```

The number of the segment of the RT-program you are currently debugging is 200 octal, while the :PROG-file that you have attached is (OWN-USER)BUBBLE-SORT:PROG.

3.45. SEGMENT-WRITE-PERMIT <segment number>

This command is used in the ND-100 Debugger and the RT-Debugger. It allows you to write data on the segment that you specify.

3.46. SEGMENT-WRITE-PROTECT <segment number>

This command is used in the ND-100 Debugger and the RT-Debugger. After you have given it, the segment is read-only - it cannot be written to.

3.47. SET <variable> (=) <value>

This command is used to set program variables. Any variable reference which has a defined address can be set. The values are formed according to the rules for expressions on page 86.

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 or an array equal to a sequence of bytes. (Note, however, that all characters are converted to capitals by the Symbolic Debuggers.) 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.

3.48. STACK-INSTRUCTIONS (<low>) (<high>)

This command, which is available on the ND-100 Debugger only, 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, which has special instructions that can be used instead of the ENTER and LEAVE subroutines that are loaded from your language's library. You may want to change you run-time system as well.

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 ↵

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.

3.49. STEP (<count>) (<low>) (<high>)

This command is not available in the RT-Debugger.

Program execution will continue to the next step-point. Step-points can be defined by LOG-LINES or LOG-CALLS, but note that step-points are not needed if you want instruction-by-instruction stepping (see below). The count parameter specifies the number of steps to take. If you use -1 as count parameter, you step through the program instruction by instruction. In this case, you can give a pair of absolute program addresses inside which you want to step as optional parameters. If you do not use this option, you will also make single-instruction steps inside subroutines belonging to, say, the run-time system called from the area you are debugging.

When you reach a step-point, the Debugger stops and outputs the current routine and line, and the segment number if you are debugging an ND-100 multi-segment program. If you then press ↵ (Carriage Return), you will continue one step at a time, as the count is cleared each time you enter a step-point. 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.

3.50. USER-ESCAPE <on/off>

This command is available on the ND-500 only.

This command enables the user to gain control when the user program is executing. With USER-ESCAPE active, control is transferred to the debugger after pressing the <ESC> key.

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 floating point 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. On the ND-500, MOD gives the remainder in integer divisions, TYPEOF returns the basic type of its parameter, and you can use reserved Debugger symbols after the SPECIAL qualifier.

  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	195D	10#195#
Floating point:		1.95E2	10#1.95#E2
Hexadecimal 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:

suffix	number system	radix
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
```

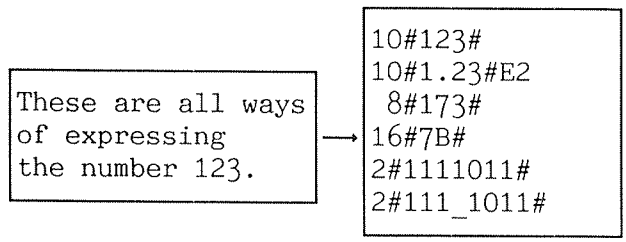
The numbers 10#195#, 8#303#, etc., on page 83 were written by using the form:

base#number#exponent

The # appears as the number sign on some terminals, and as the English pound sign (£) on others. The base is always given in decimal form. Here is an example:

```
*DISPLAY 8#100#E4 ↵
8#100#E4= 2.62144000000000E+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.



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!

(This is a feature borrowed from the programming language Ada. Ada is a trademark of the U.S. Department of Defense.)

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=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
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 testAAAAAAAAAAAAAAAAAAAAAAAAAA
*SET CODEX='Embedded ''quotes'' example' ↵
*RUN ↵
                    Embedded 'quotes' exampleAAAAAAAAAAAAAAA
*
```

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 88. 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. On the ND-500, you can also use the operators MOD (which returns the remainder from integer divisions), TYPEOF (which returns the type of its argument) and SPECIAL (which makes it possible to use special Debugger words inside quotes, such as DISPLAY SPECIAL 'ind').

Blanks can be used anywhere in expressions to separate operators and identifiers. Expressions are separated by commas. Commands are separated by semicolons.

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
*DISPLAY 2 ** 3 ** 2 ↵
2 ** 3 ** 2=64
*
```

You may use the IND and ADDR operators with or without parameters around the operand, but ensure that the value of the operand can be directly evaluated. For example:

```
Wrong: *display ind ind current.left ↵
Right: *display ind(ind current.left) ↵
```

Evaluation takes place from left to right, but the contents of parentheses are evaluated before the rest of the expression, so we use

parentheses to force the last part of the latter expression to be evaluated before the first indirection.

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
```

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
Identifiers

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

In COBOL, you may qualify with <program name>.<identifier>. In FORTRAN, you can use <routine name>.<identifier>.

By using the dot notation, you can separate variables which have the same name but are declared in different subroutines/subprograms from each other.

The dot notation is also used to retrieve record components in Pascal and PLANC. Expressions with two or more dots are evaluated from left to right.

Here is an example from PLANC. If you are not familiar with PLANC, you may want to know that a basic program unit in PLANC is a module, which consists of at least one routine (a program is a special type of routine), and that the modules may have both global and local variables with respect to the routines. The routine declarations may be nested to any level.

```
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 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
```

FORTRAN 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>)
```

with which we can specify ranges within routines/modules.

where name is a routine, module, label, or line number. If the name is a routine or module name, the range includes the lines in that routine/module. A simple example of a program area is:

```
10:20
```

meaning lines 10 to 20 in the source code.

Here is another example, showing how an area specification is used with the LOG-lines command:

```
*LOG-LINES MAIN ↵
*
```

which tells the debugger to log all lines inside the program/routine MAIN.

If an area is mandatory (as is the case with the LOG-LINES command), you will be prompted for it:

```
*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, and the way FORTRAN labels are specified using a dollar sign ($). 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
ND-500:   *LOOK-AT-PROGRAM 100B ↵
          P 01000000100B: W3 =: R.044B:S
          P 01000000102B: W STZ R.050B:S ; ↵
ND-100:   *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.

The Debugger cannot give you addresses to labels.

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: 00000000000B          0      ↵
D 01000000033B: 00000000000B          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  ; ↵
*__
```

An even more intricate example, on the ND-500 this time, and with the NAME equal to 'Current':

```
*look-at-data addr(current.name) ↵
D 1'      3730B: 10335271162B  1131770482 Curr ↵
D 1'      3734B: 14533472000B  1701737472 ent  ↵
D 1'      3740B: 010000000054B   134217772     , . ↵
*look-at-data addr(current.name(1+3)) ↵
D 1'      3734B: 14533472000B  1701737472 ent  . ↵
```

4.9. Format Specifier

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

O - Octal	A - alphanumerics (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            R=  9.00000000
*BREAK $10 I > 5 ↵
```

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            R=  3.60000000E+01
*RUN ↵

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:

```
      PROGRAM FILESEG
C     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

The first part of this chapter contains a table of all error messages from all Debuggers. Then follows subsections giving a more thorough explanation of the error messages common to all Debuggers, followed by sections on the individual Debuggers and their error messages. The table below contains references to pages where the explanations for the error message can be found.

ND-100	RT-DEB	MULTI	ND-500	This is a list of all Debugger error messages, sorted in alphabetical order. To the left of each message, you see which Debugger the message is used in. There is a reference to the pages where you can find a more thorough explanation of the error messages underneath each message.
●	●	●	●	** WARNING ** Multiple occurrences of module "N" page 115
●	●	●	●	Ambiguous command page 115
			●	Ambiguous trap condition page 121
●	●	●	●	Assembler error: "" page 115
			●	Attempt to access nonexistent data segment page 121
			●	Attempt to access nonexistent debug information page 121
			●	Attempt to access nonexistent program segment page 122
●	●	●	●	Attempt to divide by zero page 115
			●	Attempt to modify read-only segment page 122
			●	Attempt to set breakpoint on read-only segment page 122
●	●	●	●	B register not initialized page 115
●	●	●	●	Bad expression page 115
●	●	●	●	Bad line debug element; debug table address: xxxxxB page 115
●	●	●	●	Bad line number page 115
●	●	●	●	Bad module/endmodule nesting page 116
			●	Bad operand code; debug table address: xxxxxB page 122
●	●	●	●	Bad record/endrecord nesting page 116
●	●	●	●	Bad routine/endroutine nesting page 116
●	●	●	●	Bad string constant page 116

ND-100	RT-DEB	MULTI	ND-500	This is a list of all Debugger error messages, sorted in alphabetical order. To the left of each message, you see which Debugger the message is used in. There is a reference to the pages where you can find a more thorough explanation of the error messages underneath each message.
●	●	●	●	Command line/macro buffer full page 116
●	●	●	●	Component not in specified record page 116
			●	Connect called with bad access code "n" page 122
●				Data at data address xxxxxB is not stored on the prog-file page 119
●				Data at program address xxxxxB is not stored on the prog-file page 119
			●	Error in monitor call page 122
●	●	●	●	Error: "n" page 116
			●	Illegal base in numeric literal page 122
●	●	●	●	Illegal debug element type; debug table address: xxxxxB page 116
●	●	●	●	Illegal debug table address (xxxxxB in "find" page 116
	●	●		Illegal segment number..."n" page 120
●	●	●	●	Illegal termination page 117
●	●	●	●	Illegal termination of argument page 117
			●	Impossible to invoke routine; stack overflow page 122
●	●	●	●	Index "n" is outside array page 117
●	●	●	●	Indirection not legal page 117
			●	Invalid operator "<>" page 122
●	●	●	●	Limits not legal for this type page 117
●	●	●	●	Line translation table full page 117
●	●	●	●	Link-information inaccessible page 117
●	●	●	●	Modules/routines too deeply nested page 117
	●			No active breakpoint page 120

ND-1000	RT-DEB	MULTI	ND-5000	This is a list of all Debugger error messages, sorted in alphabetical order. To the left of each message, you see which Debugger the message is used in. There is a reference to the pages where you can find a more thorough explanation of the error messages underneath each message.
●	●	●	●	No active routines page 117
●	●	●	●	No debug-information available page 118
			●	No dseg-file opened or connected for segment "n" page 123
			●	No link-file opened or connected for segment "n" page 123
		●		No main segment attached page 120
●				No more data segments available page 119
●				No program file specified page 119
			●	No pseg-file opened or connected for segment "n" page 123
	●			No rt-program with this name page 120
	●	●		No segment attached page 120
●	●	●	●	No such command page 118
		●		No such reentrant system name "" page 121
●	●	●	●	No such register name page 118
		●		No such segment name "" page 121
			●	No such trap condition page 123
●	●	●	●	Not a variant of the specified record page 118
●	●	●	●	Not found page 118
●			-	Not write access to program file page 119
●	●	●	●	Outside program page 118
			●	Outside data segment page 123
			●	Outside program segment page 123
			●	Programmed-trap failed (not enabled?) page 123
		●		Protected command, cannot be used from user "" only from user SYSTEM or RT page 121

ND-100	RT-DEB	MULTI	ND-500	This is a list of all Debugger error messages, sorted in alphabetical order. To the left of each message, you see which Debugger the message is used in. There is a reference to the pages where you can find a more thorough explanation of the error messages underneath each message.
●				Restart impossible page 119
●	●	●	●	Routine inactive page 118
			●	Segment number must be in the range 0:31 page 123
		●		Single instruction step not allowed page 121
●	●	●	●	String constant too long page 118
●	●	●	●	Terminal occupied page 118
			●	This SINTRAN III command is not allowed from ND-500 page 123
			●	Too many files opened page 124
●	●	●	●	Too many indices page 118
			●	Too many nested include-commands page 124
			●	Too many nested macro expansions page 124
			●	Unable to switch device page 124
		●		Unknown break segment page 121
		●		Unknown break-return segment page 121
●		●	●	Use log-calls or log-lines page 120, page 121, page 124
●	●	●	●	Wrong enumeration-type nesting page 119
●	●	●	●	Wrong type or inaccessible page 119

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

Here are the error messages common to all Debuggers and what they mean:

** WARNING ** MULTIPLE OCCURRENCE OF MODULE "N"

You have several modules with the same name. The debugger will assume that you have reloaded. Only the last occurrence of "N" will be recognized as valid.

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

You are trying to divide by zero in the expression that you want to evaluate.

B REGISTER NOT INITIALIZED

Unable to LOOK-AT-STACK because the B-register is not initialized. You must have started the program before you can use the LOOK-AT-STACK command.

BAD EXPRESSION

Syntax error in expression. You may have a type conflict, for instance if you try to add an integer to a pointer. Another possibility is an uneven number of parentheses. See page 86 to find the rules for expression formation.

BAD LINE DEBUG ELEMENT; DEBUG TABLE ADDRESS: xxxxxB

Error in the debug-information generated by the compiler. It is possible that the debug information has been destroyed. Try recompilation and loading.

BAD LINE NUMBER

Syntax error in specified line number. A line number can be any valid decimal 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. Ensure that you do not have a new version of the compiler and an old version of the debugger. If nothing helps, report the error.

BAD ROUTINE/ENDROUTINE NESTING

Error in the debug information generated by the compiler. Ensure that you do not have a new version of the compiler and an old version of the debugger. If nothing helps, report the error.

BAD STRING CONSTANT

You may have forgotten the final apostrophe in the string.

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. You may have misspelled the record component name.

ERROR: n

Error number n from SINTRAN III or ND-500 Monitor. There is no error text for this error number. The right place to look to find out what this error means is the manual SINTRAN III Monitor Calls, ND-60.228.

ILLEGAL DEBUG ELEMENT TYPE; DEBUG TABLE ADDRESS: xxxxxB

Error in the debug information generated by the compiler. Ensure that you do not have a new version of the compiler and an old version of the debugger. If nothing helps, report the error.

ILLEGAL DEBUG TABLE ADDRESS (xxxxxB) IN "FIND"

Internal consistency error in the Debugger. The error should be reported.

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. The command parameter must be terminated by a "↵" (carriage return), " " (space), "," (comma) or ";" (semicolon).

INDEX "n" IS OUTSIDE ARRAY

Index outside range in array access.

INDIRECTION NOT LEGAL

Indirection in LOOK-AT not legal for this step size. In the ND-100, you may step through the code two words at a time, if you have given the LOOK-AT subcommand DOUBLE-WORD, and in the ND-500, you may step through the code/data one byte or half-word at a time, if you have given the LOOK-AT subcommands BYTE or HALF-WORD. But if you want to use the "/" (slash) command to move to a new location, you must have a word-size argument.

LIMITS NOT LEGAL FOR THIS TYPE

Can occur in the GUARD command; low:high is not legal for this item type. Limits are only legal for pointers, integers, enumerations and boolean types.

LINE TRANSLATION TABLE FULL

Too many areas specified in ALIGN-LISTING.

LINK-INFORMATION INACCESSIBLE

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

MODULES/ROUTINES TOO DEEPLY NESTED

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

NO ACTIVE ROUTINES

You do not have a subroutine in scope at present. To get an active breakpoint, you must set a breakpoint in a subroutine and RUN the program until execution stops at that breakpoint.

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-file or to give an ATTACH-SEGMENT command.

NO SUCH COMMAND

NO SUCH REGISTER NAME

NOT A VARIANT OF THE SPECIFIED RECORD

The three above error messages are self-explanatory. They are most likely to result from misspellings.

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, or your code has been compiled without the DEBUG mode on.

OUTSIDE PROGRAM

You are trying to access a program address which is outside the range of addresses that the Debugger will allow.

ROUTINE INACTIVE

Routine inactive (no current stack frame allocated).

STRING CONSTANT TOO LONG

The maximum length of a string constant is 80 characters.

TERMINAL OCCUPIED

An RT-program or another user is using the terminal that you want to reserve for debugging your program. Make the user or RT-program release the terminal, or reserve another.

TOO MANY INDICES

Too many indices in the array reference as compared to the array declaration.

WRONG ENUMERATION-TYPE NESTING

Error in the debug information generated by the compiler. Ensure that you do not have a new version of the compiler and an old version of the debugger. If nothing helps, report the error.

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 "" 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 "" 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 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.

NOT WRITE ACCESS TO PROGRAM FILE

Self-explanatory.

RESTART IMPOSSIBLE

Execution of the program has terminated, and it cannot be restarted. Exit the debugger and start over again.

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.

6.3. Error Messages Which Apply to RT Debugging

ILLEGAL SEGMENT NUMBER..."n"

SINTRAN III does not recognize this as a valid segment number.

NO ACTIVE BREAKPOINT

No ND-100 RT program has reached a breakpoint yet.

NO RT-PROGRAM WITH THIS NAME

Self-explanatory.

NO SEGMENT ATTACHED

Self-explanatory.

6.4. Error Messages Which Apply to ND-100 Multi-Segment Programs

ILLEGAL SEGMENT NUMBER..."n"

SINTRAN III does not recognize this as a valid segment number.

NO MAIN SEGMENT ATTACHED (ATTACH-REENTRANT-SEGMENT)

You are trying to start your multi-segment system without having specified your main program segment. You must give an ATTACH-REENTRANT-SEGMENT command.

NO SEGMENT ATTACHED

You must attach your reentrant segments with the ATTACH-REENTRANT-SEGMENT command before trying to do a REENTRANT-PLACE.

NO SUCH REENTRANT SYSTEM NAME

NO SUCH SEGMENT NAME ""

These messages are likely to be caused by misspellings or erroneous deletions of segments etc..

PROTECTED COMMAND, CANNOT BE USED FROM USER "" ONLY FROM USER SYSTEM OR RT

You must be logged in on SINTRAN user SYSTEM or RT to be able to debug multi-segment systems.

SINGLE INSTRUCTION STEP NOT ALLOWED

You cannot single-step across a segment boundary.

UNKNOWN BREAK SEGMENT

You are trying to break on a segment which is not attached.

UNKNOWN BREAK-RETURN SEGMENT

You are trying to return to a segment which is not attached.

USE LOG-CALLS OR LOG-LINES

This command requires the use of LOG-CALLS or LOG-LINES. On the ND-100, GUARD can only be used after LOG-CALLS or LOG-LINES.

6.5. Error Messages Which Apply to the ND-500 Version

AMBIGUOUS TRAP CONDITION

You must type enough characters to make the name of the trap unambiguous.

ATTEMPT TO ACCESS NONEXISTENT DATA SEGMENT

You can find the data address ranges from the load map that the Linkage-Loader prints.

ATTEMPT TO ACCESS NONEXISTENT DEBUG INFORMATION

You can find the debug information address ranges from the load map that the Linkage-Loader prints.

ATTEMPT TO ACCESS NONEXISTENT PROGRAM SEGMENT

You can find the program address ranges from the load map that the Linkage-Loader prints.

ATTEMPT TO MODIFY READ-ONLY SEGMENT

ATTEMPT TO SET BREAKPOINT ON READ-ONLY SEGMENT

The two errors above should be reported.

BAD OPERAND CODE; DEBUG TABLE ADDRESS: xxxxxB

Error in the debug information generated by the compiler. Ensure that you do not have a new version of the compiler and an old version of the debugger. If nothing helps, report the error.

CONNECT CALLED WITH BAD ACCESS CODE " "

Internal consistency error in the debugger. Ensure that you do not have a new version of the compiler and an old version of the debugger. If nothing helps, report the error.

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. The right place to look to find out what this error means is the manual SINTRAN III Monitor Calls, ND-60.228.

ILLEGAL BASE IN NUMERIC LITERAL

When using Ada-syntax in constants, the base must be in the range 2:16.

IMPOSSIBLE TO INVOKE ROUTINE; STACK OVERFLOW

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

INVALID OPERATOR "<>"

Try "><" instead.

NO DSEG-FILE OPENED OR CONNECTED FOR SEGMENT ""

NO LINK FILE OPENED OR CONNECTED FOR SEGMENT ""

NO PSEG-FILE OPEN OR CONNECTED FOR SEGMENT ""

The three error messages above are internal consistency errors that should be reported.

NO SUCH TRAP CONDITION

Check the manual for trap conditions. You will also find an in-depth explanation of the trap system in ND-500 Reference Manual, ND-05.009.

OUTSIDE DATA SEGMENT

Attempt to access beyond the available address space (on an existing data segment). A very likely source of trouble here is an uninitialized pointer address.

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 the "programmed-trap" (no. 29) has been disabled or is not working. This error should be reported.

SEGMENT NUMBER MUST BE IN THE RANGE 0:31

An ND-500 program consists of one domain with up to 32 different segments, numbered 0:31 (decimal). The first five bits in an address gives the segment number. You have tried to give a segment number outside this five-bit range.

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

Some SINTRAN III commands cannot be given via the command processor of the Symbolic Debugger, so the Debugger gives this error message instead.

TOO MANY FILES OPENED

The Debugger is unable to open all the files needed. You cannot open more than 957 files when running under the Debugger.

TOO MANY NESTED INCLUDE-COMMANDS

When reading macros from files and executing them, nested execution of INCLUDE-commands is possible, but only 6 times.

TOO MANY NESTED MACRO-EXPANSION

Macros may contain other macros, but such nesting can only occur 5 times.

UNABLE TO SWITCH DEVICE

You are trying to reserve another terminal, but your version of SINTRAN does not support this Debugger feature. It is only possible to give the RESERVE-TERMINAL command if you have SINTRAN III version K or later versions.

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.

6.6. 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

After these messages have occurred, the Debugger can still be used, but you cannot run the program.

INDEX LIST

********** **SEND US YOUR COMMENTS!!!** **********

Are you frustrated because of unclear information in this manual? 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 the manual by rearranging the contents? You could also tell us if you like the manual!

********** **HELP YOURSELF BY HELPING US!!** **********

Manual name: Symbolic Debugger User Guide

Manual number: ND–60.158.4 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 ? _____

_____

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

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

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

Answer from Norsk Data _____

Answered by _____ Date _____

Norsk Data A.S
Documentation Department
P.O. Box 25, Bogerud
0621 Oslo6, Norway