ND FORTRAN Reference Manual

ND-60.145.8 EN

Norsk Data

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

ND FORTRAN
Reference Manual

ND-60.145.8 EN

Scanned by Jonny Oddene for Sintran Data © 2011

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

Copyright©1988 by Norsk Data A.S

Version 1	*May 1981*
Version 2	*November 1981*
Version 3	*March 1982*
Version 4	*July 1982*
Version 5	*June 1983*
Version 6	*March 1984*
Version 7	*June 1986*
Version 7A	*September 1986*
Version 8	*August 1987*

Send all documentation requests to:

Norsk Data A.S
Graphic Centre
P.O. Box 25 – Bogerud
N-0621 Oslo 6
NORWAY

Scanned by Jonny Oddene for Sintran Data © 2011

PREFACE

THE PRODUCT

FORTRAN is a high-level programming language used mainly for solving scientific problems on digital computers. Norsk Data provides compilers for FORTRAN on their machines. This manual describes the language and facilities of the following compilers:

NORD-10/ND-100 ANSI 77 FORTRAN - ND-210191 Release G

ND-500 ANSI 77 FORTRAN - ND-210190 Release K

ND-500 FORTRAN Crosscompiler - ND-210863 Release G

The language supported is that of ANSI X3.9 - 1978 FORTRAN 77 with a very few restrictions, as noted on page viii, and a certain number of extensions which are described in the main part of the manual.

THE READER

This manual is intended for programmers who are writing FORTRAN programs for ND-100 or ND-500 computers. It includes complete and formal descriptions of the language, and the facilities it offers.

PREREQUISITE KNOWLEDGE

The reader must have a basic knowledge of data processing techniques and have some experience with FORTRAN.

Scanned by Jonny Oddene for Sintran Data © 2011

RELATED MANUALS

The related manuals are:

ND Relocating Loader	ND-60.066
BRF-LINKER User Manual	ND-60.196
Symbolic Debugger User's Guide	ND-60.158
SINTRAN III Reference Manual	ND-60.128
ND-500 Loader/Monitor	ND-60.136

For writing real-time programs in FORTRAN, the following manuals are recommended:

SINTRAN III Real Time Guide	ND-60.133
SINTRAN III Real Time Loader	ND-60.051

HOW TO USE THE MANUAL

The description is given in the order in which the statement types appear in the written programs.

The manual is intended for reference purposes and is organized as a progressive description of the features of ND FORTRAN. Chapter 13 lists the available compiler commands. Examples are included in the text and a sample program is provided with extensive notes for the programmer wanting an overview of the FORTRAN language, (see Section 1.4.). Supplementary information is given in the appendices at the end.

Scanned by Jonny Oddene for Sintran Data © 2011

RESTRICTIONS, DEVIATIONS, AND INCOMPATIBILITIES

The following items differ slightly from ANSI X3.9 - 1978 FORTRAN 77:

1. Blank COMMON cannot be expanded during the loading process.

2. The RECL option of the OPEN statement gives the length in bytes, as required by the ANSI standard for both formatted and unformatted files. However, on the ND-100 this length must be an even number.

The following are the limits on certain features:

1. The lengths of character strings must be less than 32767 on the ND-500 and 2047 on the ND-100. This applies to the lengths of all variables, constants, expressions and intermediate results.

2. The number of dimensions of an array during debugging must be less than 8.

3. The maximum depth of INCLUDE'd text files is 5.

The maximum size of a program unit, or length of statements, or complexity of expression are too heavily dependent on content for any rules to be given.

The following are known incompatibilities with the NORD-10/ND-100 (P.D. number FTN-2090) and NORD-50 compilers (P.D. number FTN-2159) and associated libraries.

1. RECL option is in bytes.

2. Variables used in the specification of adjustable bounds may be changed within the function or subroutine without modifying the values used for bounds.

Scanned by Jonny Oddene for Sintran Data © 2011

3. Variables used in the specification of the final value and increment of DO-loops may be changed without affecting the number of times a DO-loop is executed.

4. Records in a file are counted from 1 instead of 0. However, the FIRSTREC option in the OPEN statement may be used to override this.

5. Some compiler commands have been changed.

6. Some new options have been added to OPEN statements, IOSTAT, FORM, BLANC, FACTOR, IOCONVERT, TYPE, MODE, PARITY, FIRSTREC and BUFFER-SIZE

7. If the first character of a record of a non-print file is a $, then the FTN-2090 and FTN-2159 compilers are used to supress the LF and CR characters. This compiler will only do so if the file is a PRINT file.

8. The parameters to the monitor calls must now be exactly as given in Section 13.1.

9. Character dummy arguments in subroutines are now taken to be exactly as long as declared in the subroutine. To pick up the length of the actual argument, a length of (*) must be specified for the dummy argument.

10. If a variable in a DATA-statement is an array and the corresponding constant is a Hollerith constant, the Hollerith constant is filled in the first array element even if it is longer than the length of the array element.

Scanned by Jonny Oddene for Sintran Data © 2011

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1
1.1	THE NOTATION	3
1.2	FORTRAN CHARACTER SET	4
1.3	FORTRAN TERMS AND CONCEPTS	6
1.3.1	Lines	7
1.3.2	Statements	9
1.3.3	Program Units and Procedures	10
1.3.4	Required Order of Statements and Lines	11
1.4	NOTES ON A SAMPLE PROGRAM	13
2	DATA TYPES, CONSTANTS, VARIABLES, ARRAYS AND SUBSTRINGS	25
2.1	DATA TYPES	27
2.1.1	Type rules for identifiers	28
2.2	CONSTANTS	29
2.2.1	Integer constants	29
2.2.2	Real constants	31
2.2.3	Double-precision constants	32
2.2.4	Complex constants	33
2.2.5	Logical constants	34
2.2.6	Character constants	34
2.3	VARIABLES	36
2.4	ARRAYS	37
2.4.1	Array elements	38
2.4.2	Order of stored array elements	39
2.4.3	Adjustable arrays	39
2.4.4	Assumed-size arrays	40
2.4.5	Actual and dummy array declarators	40
2.5	CHARACTER SUBSTRINGS	41
3	SPECIFICATION STATEMENTS	43
3.1	THE DIMENSION STATEMENT	45
3.2	THE EQUIVALENCE STATEMENT	48
3.2.1	Array Names and Array Element Names	49
3.2.2	Character Variables in EQUIVALENCE Statements	49

Scanned by Jonny Oddene for Sintran Data © 2011

Section Page

3.2.3 Restrictions on EQUIVALENCE Statements 50
3.3 THE COMMON STATEMENT 52
3.3.1 COMMON Block Storage Sequences 53
3.3.2 Differences between Named COMMON and Blank COMMON . . 54
3.3.3 Restrictions on COMMON and EQUIVALENCE 55
3.3.4 COMMON Blocks in APT 55
3.4 TYPE STATEMENTS 57
3.4.1 INTEGER, REAL, DOUBLE PRECISION, NUMERIC, COMPLEX and LOGICAL Type Statements 58
3.4.2 CHARACTER Type Statement 61
3.5 THE IMPLICIT STATEMENT 66
3.6 THE PARAMETER STATEMENT 68
3.7 THE EXTERNAL STATEMENT 70
3.8 THE INTRINSIC STATEMENT 71
3.9 THE SAVE STATEMENT 72
3.10 THE ASSEMBLY STATEMENT 74

4 THE DATA STATEMENT 75

4.1 DATA STATEMENT RESTRICTIONS 77
4.2 IMPLIED DO IN A DATA STATEMENT 80

5 EXPRESSIONS 83

5.1 ARITHMETIC EXPRESSIONS 85
5.1.1 Interpretation of Results for Arithmetic Expressions 89
5.1.2 Arithmetic Constant Expressions 92
5.2 CHARACTER EXPRESSIONS 93
5.2.1 CHARACTER Constant Expressions 95
5.3 RELATIONAL EXPRESSIONS 96
5.3.1 Arithmetic Relational Expressions 96
5.3.2 CHARACTER Relational Expressions 98
5.3.3 LOGICAL Relational Expressions 98
5.4 LOGICAL EXPRESSIONS 99
5.4.1 LOGICAL Constant Expressions 103
5.5 EVALUATION OF EXPRESSIONS 103
5.5.1 The Use of Parentheses 103
5.5.2 Precedence of Operators 104
5.5.3 Location of Operators within an Expression 104
5.6 CONSTANT EXPRESSIONS 106

Scanned by Jonny Oddene for Sintran Data © 2011

Section Page

6 ARRAY EXPRESSIONS 107

6.1 ARITHMETIC ARRAY EXPRESSIONS 109
6.1.1 Interpretation of Results for Arithmetic Array Expressions 111
6.2 CHARACTER ARRAY EXPRESSIONS 113
6.3 RELATIONAL ARRAY EXPRESSIONS 115
6.3.1 Arithmetic Relational Array Expressions 115
6.3.2 CHARACTER Relational Array Expressions 116
6.3.3 LOGICAL Relational Array Expressions 116
6.4 LOGICAL ARRAY EXPRESSIONS 117
6.5 EVALUATION OF EXPRESSIONS 119
6.5.1 The Use of Parentheses 119
6.5.2 Precedence of Operators 119

7 ASSIGNMENT STATEMENTS 121

7.1 ARITHMETIC ASSIGNMENT STATEMENT 123
7.2 LOGICAL ASSIGNMENT STATEMENT 125
7.3 STATEMENT LABEL ASSIGNMENT (ASSIGN) STATEMENT 126
7.4 CHARACTER ASSIGNMENT STATEMENT 127

8 CONTROL STATEMENTS 129

8.1 UNCONDITIONAL GO TO STATEMENT 132
8.2 COMPUTED GO TO STATEMENT 133
8.3 ASSIGNED GO TO STATEMENT 135
8.4 ARITHMETIC IF STATEMENT 137
8.5 LOGICAL IF STATEMENT 138
8.6 THE BLOCK IF, ELSEIF, ELSE, AND ENDIF STATEMENTS . . . 139
8.6.1 The ELSEIF Statement 139
8.6.2 The ELSE Statement 140
8.6.3 The ENDIF Statement 141
8.6.4 Examples of Block IF, ELSEIF, ELSE and ENDIF Statements 141
8.7 THE DO STATEMENT 144
8.7.1 Execution of a DO Statement 146
8.7.2 The DO FOR ... ENDDO Statements 148

Scanned by Jonny Oddene for Sintran Data © 2011

Section Page

8.7.3 The DO WHILE ... ENDDO Statements 149
8.8 THE CONTINUE STATEMENT 151
8.9 THE STOP STATEMENT 152
8.10 THE PAUSE STATEMENT 153
8.11 THE END STATEMENT 154

9 INPUT/OUTPUT STATEMENTS 155

9.1 I/O TERMS AND CONCEPTS 157
9.1.1 Records 157
9.1.2 Files 158
9.1.2.1 File Format 159
9.1.2.2 File Access 160
9.1.3 Units 162
9.1.4 Format Specifier and Identifier 163
9.1.5 End-of-File Specifier 164
9.1.6 Error Specifier 165
9.1.7 Input/Output Status Specifier 166
9.1.8 Record Specifier 167
9.2 DATA TRANSFER OPERATIONS 168
9.2.1 Input/Output Lists 168
9.2.1.1 Implied DO Lists 169
9.2.2 Formatted and Unformatted Data Transfer 170
9.2.3 List-Directed Input/Output 170
9.2.3.1 List-Directed Input 171
9.2.3.2 List-Directed Output 173
9.2.4 The READ Statement 174
9.2.5 The WRITE Statement 176
9.2.5.1 Printing of Formatted Records 178
9.2.6 The PRINT Statement 180
9.2.7 The INPUT Statement 181
9.2.8 The OUTPUT Statement 181
9.3 FILE OPEN AND CLOSE 182
9.3.1 The OPEN Statement 182
9.3.2 The CLOSE Statement 194
9.4 FILE POSITIONING 195
9.4.1 The BACKSPACE Statement 195
9.4.2 The ENDFILE Statement 196
9.4.3 The REWIND Statement 197
9.5 THE INQUIRE STATEMENT 198

Scanned by Jonny Oddene for Sintran Data © 2011

Section Page

10 FORMAT SPECIFICATIONS 207

10.1 FORMAT SPECIFICATION METHODS 209
10.2 FORMAT DESCRIPTORS 210
10.2.1 Interaction between the Format Descriptors and the I/O List 212
10.2.2 Editing Provided by the Format Descriptors 214
10.2.2.1 Numeric Editing 214
10.2.2.2 The I and J Format Descriptors 215
10.2.2.3 REAL and DOUBLE PRECISION 216
10.2.2.4 The F Format Descriptor 217
10.2.2.5 Scale Factor: The P Format Descriptor 218
10.2.2.6 The E and D Format Descriptors 220
10.2.2.7 The G Format Descriptor 222
10.2.2.8 COMPLEX Data 223
10.2.2.9 S, SP and SS Format Descriptors 223
10.2.2.10 The BN and BZ Format Descriptors 223
10.2.2.11 The Text Format Descriptor 224
10.2.2.12 The H Format Descriptor 224
10.2.2.13 The T, TL, TR, and rX Format Descriptors 225
10.2.2.14 The Slash, /, Format Descriptor 226
10.2.2.15 The L Format Descriptor 226
10.2.2.16 The A Format Descriptor 227
10.2.2.17 The O Format Descriptor 228
10.2.2.18 The Z Format Descriptor 229

11 FUNCTIONS AND SUBROUTINES 231

11.1 DUMMY AND ACTUAL ARGUMENTS 234
11.1.1 Variables as Dummy Arguments 238
11.1.2 Arrays as Dummy Arguments 239
11.1.3 Procedures as Dummy Arguments 241
11.1.4 Asterisks as Dummy Arguments/Alternative Return Arguments 243
11.2 INTRINSIC FUNCTIONS 244
11.2.1 Specific Names and Generic Names 244
11.2.2 Referencing an INTRINSIC Function 246
11.3 STATEMENT FUNCTIONS 258
11.3.1 Statement Function Restrictions 259
11.3.2 Referencing a Statement Function 260

Scanned by Jonny Oddene for Sintran Data © 2011

Section Page

11.4 EXTERNAL FUNCTIONS 261
11.4.1 Actual Arguments for an External Function 262
11.4.2 Function Subprogram Restrictions 262
11.5 SUBROUTINES 264
11.5.1 Subroutine Reference 265
11.5.2 Subroutine Subprogram Restrictions 266
11.6 THE ENTRY STATEMENT 266
11.6.1 ENTRY Statement Restrictions 268
11.7 THE RETURN STATEMENT 270
11.7.1 Execution of a RETURN Statement 270

12 MAIN PROGRAM 273

12.1 THE PROGRAM STATEMENT 275

13 BLOCK DATA SUBPROGRAM 277

13.1 BLOCK DATA SUBPROGRAM RESTRICTIONS 279

14 ADVANCED FORTRAN PROGRAMMING 281

14.1 EFFICIENT PROGRAMMING TECHNIQUES 283
14.1.1 Loops 283
14.1.2 Loop Control Variable 284
14.1.3 Array Operations 284
14.1.4 Actual Argument Data Types 285
14.1.5 CHARACTER and Hollerith 286
14.1.6 CHARACTER Alignment - ND-100 287
14.1.7 File Accessing 287
14.1.8 I/O Buffer Allocation 288

APPENDIX

A ASCII CHARACTER SET 291

B ERROR MESSAGES 299

Scanned by Jonny Oddene for Sintran Data © 2011

Section		Page
C	MONITOR CALLS	323
D	LIBRARY UTILIITY FUNCTIONS	363
E	STORAGE MAPPING	391
F	INTERFACES TO OTHER LANGUAGE PROGRAMS	405
G	HOLLERITH	439
Index		1

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 1

INTRODUCTION

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

The FORTRAN language described in this manual is in accordance with the American National Standard Institute's FORTRAN 77. The full language has been implemented, except for items listed on page 7; a certain number of ND FORTRAN extensions are noted in the text.

1.1 THE NOTATION

The notation used throughout the manual to describe the FORTRAN statements and constructs is listed below:

1. Square brackets, [], indicate optional items.

2. An ellipsis, ..., following square brackets specifies that the preceding optional items may appear one or more times in succession.

3. Round brackets, (), are part of FORTRAN and must be coded where shown.

4. Blanks are used to improve readability, but unless otherwise noted have no significance.

5. Grey shading, over text, has been used to highlight any divergence from the ANSI FORTRAN 77 standard, including variations and ND extensions.

   Note that the grey shading has been used in Chapters 1 through 11 only.

6. Windows are used to call attention to the importance of commands.

Scanned by Jonny Oddene for Sintran Data © 2011

1.2 FORTRAN CHARACTER SET

The FORTRAN character set consists of twenty-six letters, ten digits, and thirteen special characters.

A letter is one of the twenty-six characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

A digit is one of the ten characters:

0123456789

An alphanumeric character is a letter or a digit.

A Special Character is one of the following characters:

CHARACTER	*MEANING*
	Blank
=	*Equals*
+	*Plus*
-	*Minus*
*	*Asterisk*
/	*Slash*
(	*Left Parenthesis*
)	*Right Parenthesis*
,	*Comma*
.	*Decimal Point*
$	*Currency Symbol*
'	*Apostrophe*
:	*Colon*

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN the following special characters may be used:

;	*semicolon*
%	*percent*
_	*underscore*
!	*exclamation point*
&	*ampersand*
"	*double quotation mark*

The collating sequence is the ASCII sequence described in Appendix A.

In ND FORTRAN, lowercase letters are allowed in addition to uppercase letters, as specified in ANSI FORTRAN 77. If they occur in a character string or Hollerith constant, they retain their lowercase values. Otherwise, they are converted to uppercase.

Scanned by Jonny Oddene for Sintran Data © 2011

1.3 FORTRAN TERMS AND CONCEPTS

The basic language elements of FORTRAN, i.e., syntactic items, are constants, symbolic names, statement labels, keywords, operators, and special characters. These are all formed from the letters, digits, and special characters of the FORTRAN character set previously described in this chapter. The form of a constant is described in Section 2.2. on page 29.

A symbolic name is a sequence of one to six letters or digits, the first of which must be a letter. It can be used to identify a global item, i.e., an item known to the whole executable program. The following are global items:

- a common block
- an external function
- a subroutine
- a main program
- a block data subprogram

A symbolic name can also be used to identify a local item - one whose scope is only that of the program unit in which it appears, as listed below:

- an array
- a variable
- a constant
- a statement function
- an intrinsic function
- a dummy procedure

In ND FORTRAN, symbolic names may be longer than six characters, and the first 31 are used as the unique identification. Any character except the first may be an underscore.

A keyword is a word that is recognized by the compiler. Keywords appear in capital letters throughout this manual.

Scanned by Jonny Oddene for Sintran Data © 2011

Operators are described in Chapter 5, which begins on page 85.

1.3.1 Lines

A line in a program unit is a sequence of 72 characters. The character positions in a line are called columns and are numbered consecutively 1, 2, through 72, the sequential order being from left to right. Lines are ordered in the same sequence as they are presented to the compiler.

An initial line is any line that is not a comment line and contains the character blank or the digit zero, in column 6. Columns 1 to 5 may contain a statement label or they may all be blank.

A continuation line is any line containing any character of the FORTRAN character set other than a blank or a zero in column 6, and containing only blanks in columns 1 through 5. A statement must not have more than nineteen continuation lines.

With ND FORTRAN, some of these strict requirements are lifted. An initial line may start at any column except column 6. A label need not be restricted to columns 1 to 5, and a statement may begin before column 7.

In ND FORTRAN the ampersand sign (&), may be used to indicate that the next line is a continuation line.

A comment line is any line containing a **C** or an asterisk in column 1, or containing only blank characters in columns 1 through 72. The remaining columns may contain any character which the compiler can accept. Comment lines may appear anywhere within the program unit.

Scanned by Jonny Oddene for Sintran Data © 2011

If the first character of a statement is a percent sign (%), then the whole statement is treated as a comment. So, for example, you can write:

```
% THIS IS A COMMENT
A=B; % THIS IS ANOTHER
% AND THIS ONE IS C      CONTINUED
```

Note that semicolons (;) do not terminate comments.

In ND FORTRAN, either a percent sign or an exclamation point (!), may be used for inline comments.

Example:

```
      IF (A.EQ.B .AND.          ! comment
     *    C.EQ.D)  THEN         ! comment
           A=C                  ! comment
      ENDIF
```

Tab characters found in the program text are interpreted as a sequence of blanks up to the next tab position. The tab positions are the same as the default positions for the QED editor, i.e., at columns 8, 14, 30, 40, 50, 60, 70, 80; but beyond column 80, a tab character is treated as a blank. This is true even within character strings, H-format format items, and Hollerith strings.

The form-feed character (14 octal) is also treated specially by the compiler. A blank is substituted for the form-feed, and then the next line will be printed at the beginning of a new page on the source listing This retains compatibility with previous implementatons, but it is discouraged as normal practice. The EJECT command should be used instead.

Lines may be of any length but only columns 1 to 72 may contain program statements. Characters beyond column 72 are listed, but ignored.

Scanned by Jonny Oddene for Sintran Data © 2011

1.3.2 Statements

An ANSI FORTRAN 77 source program consists of a set of statements composed of keywords and other syntactic items as described above. Most statements begin with a keyword which is then used as the statement identifier. The exceptions are assignment and statement function statements.

There are two basic types of statements, executable and nonexecutable.

Executable statements specify the actions to be taken during execution of a program, i.e., the computation of values, input and output operations, transfer of control within one program unit or between program units etc. Executable statements are normally executed in the sequence they appear in the program unit. They may be labeled, and references to labels may be used to alter the sequence of execution.

Nonexecutable statements specify characteristics, arrangement, and initial values of data. They can also contain editing information, specify statement functions, classify program units, and specify entry points within subprograms. Nonexecutable statements are not part of the execution sequence; they may be labeled but such labels cannot be used to control the execution sequence.

A statement is written on one or more lines, the first of which is called an initial line. Succeeding lines, if any, are called continuation lines, Section 1.3.1. on page 7.

A statement label is a sequence of one to five digits, one of which must be nonzero, and is used to identify a statement. The statement label may be placed anywhere in columns 1 through 5 of the initial line of the statement, Section 1.3.1. on page 7.

Statement labels provide a means of referring to individual statements. Any statement can be labeled but the only ones which can be referred to are labeled executable statements and FORMAT statements.

Scanned by Jonny Oddene for Sintran Data © 2011

The same statement label must not be given to more than one statement in a program unit. Blanks and leading zeros are not significant in distinguishing between statement labels.

In ND FORTRAN more than one statement may appear on a line provided they are separated by a semicolon (;). These extra statements on a line can also have labels prefixed to them.

1.3.3 Program Units and Procedures

A program unit consists of a sequence of statements and optional comment lines. It is either a main program or a subprogram.

A main program contains the first executable statement of the executable program. Its first statement can be a PROGRAM statement but not a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

A subprogram is a program unit having a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

A procedure is an intrinsic function, statement function, subroutine, or an external function. Subroutines and external functions are called external procedures.

Scanned by Jonny Oddene for Sintran Data © 2011

1.3.4 Required Order of Statements and Lines

Within a program unit, the required order of statements and comment lines, as described in ANSI FORTRAN 77, is summarized in the diagram below:

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY Statements</td><td rowspan="2">PARAMETER Statements</td><td>IMPLICIT Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement-Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="3">END Statement</td></tr>
</table>

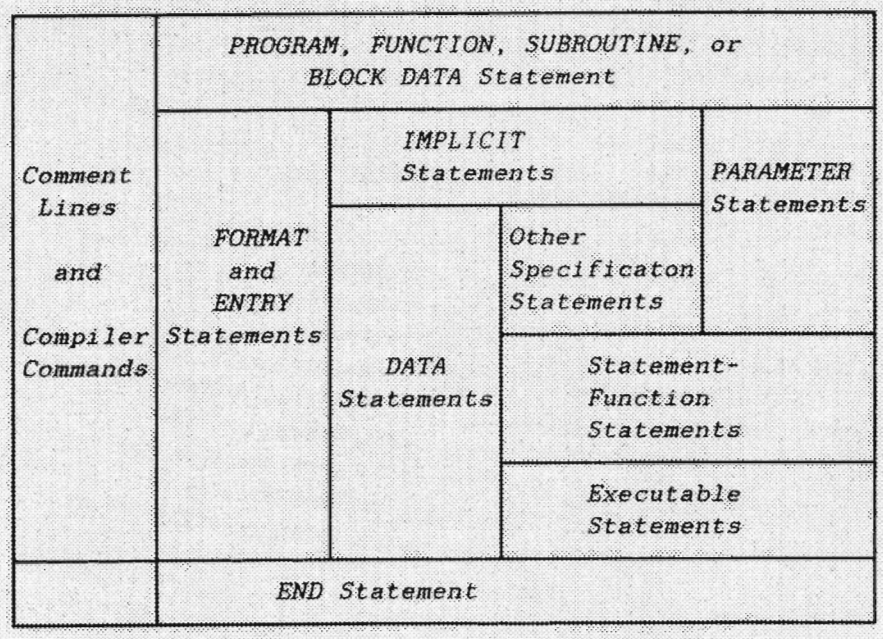
In the diagram, vertical lines delineate varieties of statements that may be interspersed. For example, FORMAT statements can be interspersed with statement function-statements and executable statements.

Horizontal lines delineate the kinds of statements that must not be interspersed. For example, statement-function statements cannot be interspersed with executable statements.

Note that the END statement is also an executable statement and must only appear as the last statement of a program unit.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, the rules for the required order of statements have been relaxed somewhat as illustrated below:



In ND FORTRAN, DATA statements are also allowed among the specification statements, but must follow all IMPLICIT statements.

Comment lines may follow the END statement.

Compiler commands may appear anywhere in the source program.

Scanned by Jonny Oddene for Sintran Data © 2011

1.4 NOTES ON A SAMPLE PROGRAM

This section contains an example of a complete FORTRAN program. The example illustrates a number of different features of the language.

The numbers to the left of the FORTRAN statements are line numbers, that are added by the compiler, to the listing of the source program.

The example is followed by detailed comments explaining each line of the program.

Scanned by Jonny Oddene for Sintran Data © 2011

```
 1*            PROGRAM SAMPLE
 2*            IMPLICIT INTEGER(R,O,Y,G,B)
 3*            PARAMETER (RED=1,ORANGE=2,YELLOW=3,GREEN=4,BLUE=5)
 4*            INTEGER N(5),M
 5*            CHARACTER GROUP*1,COLOUR*5
 6*            COMMON/SHARE/AV
 7*            REAL X(5,20),R,Y,AV
 8*            DATA COLOUR/'ROYGB'/,X,N/100*0.0,5*0/
 9*            IUNIT=60
10*            OPEN (UNIT=IUNIT,FILE='READINGS:DATA',STATUS='OLD',
11*           1       FORM='FORMATTED',ACCESS='SEQUENTIAL',
12*           2       ISOTAT=IERNAM,ERR=900)
13*            M=0
14*     10     READ (IUNIT,500,END=40) GROUP,R
15*            K=INDEX(COLOUR,GROUP)
16*            M=M+1
17*            IF(K.EQ.0) THEN
18*               WRITE(1,*) 'INVALID GROUP IDENTIFIER'
19*            ELSE
20*               N(K)=N(K)+1
21*               X(K,N(K))=R
22*               IF (K.EQ.ORANGE.OR.K.EQ.GREEN) THEN
23*                  N(K-1)=N(K-1)+1
24*                  X(K-1,N(K-1))=R
25*                  N(K+1)=N(K+1)+1
26*                  X(K+1,N(K+1))=R
27*               ENDIF
28*            ENDIF
29*            GOTO 10
30*
31*     40     CONTINUE
32*            WRITE(1,'(I6,''READINGS'')') M
33*            DO 50 K=RED,BLUE
34*               CALL AVRAGE(X,N,K)
35*               SDEV=VAR(X,N,K)
36*               WRITE(1,510) COLOUR(K:K),N(K),AV,SDEV
37*     50     CONTINUE
38*            GOTO 999
39*
40*     500    FORMAT(A1,F5.2)
41*     510    FORMAT(1H,A4,I5,F9.2,'ST DEV:',E10.3)
42*     900    CONTINUE
43*            WRITE(1,*)'OPEN ERROR - CODE IS:',IERNAM
44*     999    CONTINUE
45*            END
46*
47*            SUBROUTINE AVRAGE(X,N,K)
```

Scanned by Jonny Oddene for Sintran Data © 2011

```
48*            COMMON/SHARE/AV
49*            DIMENSION X(5,*),N(*)
50*            R=0.0
51*            DO 10 I=1,N(K)
52*               R=R+X(K,I)
53*     10     CONTINUE
54*            AV=R/N(K)
55*            END
56*
57*            REAL FUNCTION VAR(V,M,J)
58*            DIMENSION V(5,*),M(*)
59*            COMMON/SHARE/AV
60*            VAR=0.0
61*            DO 10 I=1,M(J)
62*               VAR=VAR+(V(J,I)-AV)**2
63*     10     CONTINUE
64*            VAR=SQRT(VAR/(M(J)-1))
65*            END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Line 1: This line identifies the main entry point of the program.

Lines 2-7: This part defines the variables used in the program. This section must precede the description of what the program does (the 'algorithm').

Line 2: If variables have not been given a type explicitly, then they receive their types according to the first letter of their name. Here, initial letters R, O, Y, G, B will imply that the variables are of type INTEGER.

Line 3: This statement assigns values to certain names. These names are not normal variables, but are used to give consistent names to constants. For example, here the name GREEN will mean the constant 4. The constants are of type INTEGER because their first letters appear in an implicit statement (see line 2).

Line 4: Here an array is defined as having 5 elements and name N; also a simple variable is defined called M. Both these items are declared to be of type INTEGER.

Line 5: Two items of type CHARACTER are declared in this line: one of length 1 and one of length 5. Since this statement explicitly gives GROUP a type, the IMPLICIT statement (line 2) does not apply.

Line 6: Here we have a COMMON block called SHARE. It is known outside this program unit, and enables variables to be shared between program units (see lines 48, 59). The block contains only one variable called AV in this program unit.

Line 7: This defines 4 items to be of type REAL. One of these (X) is a two-dimensional array. The first subscript varies from 1 to 5 and the second from 1 to 20. The name AV is the same as the one in the COMMON block (line 6),and this statement declares this COMMON variable to be of type REAL.

Scanned by Jonny Oddene for Sintran Data © 2011

Line 8: This statement gives values to 3 variables initially. Before the program starts to execute, the variable COLOUR will have the value 'ROYGB'; all the 100 elements of X and the 5 of N are given the initial value zero.

Line 9: This is the first executable statement and gives the value 60 to the variable IUNIT. Since this variable has not been declared, and the IMPLICIT statement (line 2) does not contain the letter I, the default type is derived from the I-N rule. This rule states that all undeclared variables beginning with the letters I, J, K, L, M, N are INTEGER and the rest are REAL.

Lines 10-12: These 3 lines form one statement. The continuation lines 11 and 12 have a character in column 6 which is neither zero nor blank (in this case 1 and 2) which defines them as continuation lines. This OPEN statement prepares a file called READINGS:DATA for sequential access, and it is shown containing formatted data. If an error should occur (e.g. the file does not exist) then the program will continue at the statement labeled 900 (see line 42). Subsequent I/O statements on this file will use the same unit number (see line 14).

Line 13: A simple assignment of zero to M.

Line 14: This statement reads a single record from the file identified by the unit number IUNIT. In this case, this is associated with the file READINGS:DATA by means of the OPEN statement (see line 10). The record is interpreted according to the FORMAT statement at label 500 (see line 40). If there are no more records left, then the program continues at statement label 40 (see line 31). Two variables are read in, GROUP and R. This statement has a label (10) which other statements can reference (see line 29).

Line 15: The INDEX intrinsic function is invoked with actual parameters COLOUR and GROUP. This searches for the string contained in GROUP

Scanned by Jonny Oddene for Sintran Data © 2011

(let us say this is 'R') in the string contained in colour (this is 'ROYGB'). In our example, the result would be 1 (the first occurrence of 'R' in 'ROYGB' is the first character), and this would then be placed in K. K is not declared, and receives the default type INTEGER (see also line 9). INDEX is not declared since it is an intrinsic function.

Line 16: Adds one to M. (In this program M is used to count the number of records read).

Lines 17-28: Here we have a block IF construct. Line 17 (the IF) shows the test to be made. If K has the value 0, then the THEN part is executed (line 18). Otherwise control goes to the ELSE clause (line 19) and proceeds normally to the ENDIF (line 28). The indentations in the listing are purely to help the reader so that the THEN and ELSE clauses are easily seen.

Line 18: This merely writes the character constant to the user's terminal (unit 1). The * indicates that free format (also known as list-directed I/O) is to be used.

Line 19: See note on lines 17-28.

Line 20: The first statement of the ELSE clause. It adds 1 to the K'th element of array N. N keeps a count of how many of each type of reading is recorded, the type being identified by K.

Line 21: Puts the reading R into the appropriate position in the table X. X has 2 dimensions. The first subscript K identifies the group, and the second N (K) identifies which position within the group.

Lines 22-27: Another block IF; this time without an ELSE clause. If K has the value ORANGE or GREEN then the reading is also placed in the previous and succeeding groups. Otherwise nothing is done here.

Line 27: Terminates the block IF in line 22.

Scanned by Jonny Oddene for Sintran Data © 2011

Line 28: Terminates the block IF in line 17. Note how one block IF is completely nested within one clause of another block IF.

Line 29: Directs the execution of the program to label 10 (line 14) where the next record is to be read. The repeated execution of lines 14 through 29 is only halted by the END clause in line 14, which will cause execution to jump to label 40 (line 31).

Line 30: Blank lines are treated as comments. They can be placed anywhere to make the listing easier to read.

Line 31: The CONTINUE statement does nothing itself. Here it is simply used so that the label 40 can be positioned. Note that the label 40 could have been placed on the WRITE statement in line 32 instead.

Line 32: Writes to the user's terminal (unit 1). The format used is written here as a character constant, the value of which is (I6, 'READINGS'). There is only one value, M, to be written. Thus M is written according to the format item I6. It is then followed by the characters READINGS.

Lines 33-37: This is a DO-loop. It begins with the DO statement (line 33) which identifies the end as a statement label 50 (line 37). K is the control variable of the loop. It starts with the value RED, and increases each time the loop is repeated until it is greater than BLUE. Since no increment is specified, it is taken to be 1. Thus after control has passed through the lines 34 to 37, K is increased by 1 and control resumes at line 34. When K exceeds BLUE, program execution leaves the loop, and continues after statement label 50 (i.e., at line 38).

Line 34: This is how subroutines are called. The name AVRAGE has no declaration, and because it occurs in a CALL statement, it is by default the name of an EXTERNAL program unit, known as

Scanned by Jonny Oddene for Sintran Data © 2011

a SUBROUTINE. It has 3 actual parameters (see also line 47).

Line 35: The variable SDEV receives the value returned by the function VAR. VAR is not declared as an array but appears followed by a parameter list. It is therefore by default EXTERNAL, and a FUNCTION. It returns a single value, and the type of this value is implied in the normal way as for variables; in this case it is REAL because the letter V is not in the range I-N, nor does it appear in an IMPLICIT statement. There are three actual parameters X, N and K. The name SDEV is not declared, but is implicitly a variable of type REAL.

Line 36: This line writes the results of the computations to unit 1 (the user's terminal) according to the format at label 510. The first value written is the group letter, which is the substring taken from COLOUR starting and ending with the K'th (i.e., just one) character. The next value is the count of readings in each group taken from the array N. Then the average which was computed by the subroutine AVRAGE and left in the COMMON block. And finally the standard deviation as calculated by VAR and returned to SDEV in line 35.

Line 37: The end of the DO-loop which starts at line 33. Once again, the CONTINUE statement is simply in order to place the statement label here.

Line 38: A simple jump to avoid the error-handling routine to label 999 (line 44).

Line 39: Another blank line of no significance.

Line 40: Defines the format of the input records (used in line 14). There is a field of length 1 used as a literal character (A1); and a field of width 5 treated as a fixed-point number, with an implied position of the decimal point 2 digits from the right-hand end if no point is present explicitly.

Line 41: Defines the output format, consisting of 6

Scanned by Jonny Oddene for Sintran Data © 2011

separate fields. "1H" puts a blank in the first position.

Since the user's terminal is being written to (see line 36), this first character is used as a "forms control character"; a blank means start on the next line. Then follow data formats of type character (A4), integer (I5), and fixed-point (F9.2). Next is a literal string and finally a field with an exponent (E10.3).

Line 42: The start of the error handling. The statement label 900 is referred to by line 12.

Line 43: An error message is written in free format to the user's terminal (unit 1). If an error in the OPEN statement is found, the IOSTAT status specifier indicates that an error code should be stored in the variable IERNAM. This is then written out by means of this WRITE statement.

Line 44: A CONTINUE statement to hold the position of label 999.

Line 45: An END statement marks the end of this program unit. The lines 1 to 45 could be compiled as a separate job.

Line 46: Insignificant blank line.

Line 47: A new program unit is started. It is a SUBROUTINE with the name AVRAGE and uses 3 dummy arguments called X, N, and K.

Line 48: A COMMON block is defined called SHARE, containing one variable called AV. (The name SHARE is what connects this COMMON block with the one in the other program units 3 (lines 6 and 59).)

Scanned by Jonny Oddene for Sintran Data © 2011

Line 49: Declares X and N to be arrays. Since they are dummy arguments, the last upper bounds can be left free; this is what the * means.

Lines 48-49: There are no type statements here, so all variables will take the implicit types defined by their initial letters. In this program unit there are no IMPLICIT statements, therefore only the I-N rule is used. (Compare with line 9. Note that line 2 is no longer valid. Its range stopped with the END at line 45.)

Line 50: Initializes the REAL variable R to zero.

Lines 51-53: A DO-loop to add up the N(K) values in X from X (K,1) to X (K, N (K)). The sum is accumulated in the variable R.

Line 54: Compute the average and place it in the variable AV, in the COMMON block where it is available to the other program units.

Line 55: Terminate this program unit. When program execution reaches this point, it returns to where the program unit was called from and continues from there. (In this example there is only one point where a CALL statement is used, line 34.)

Line 56: Another blank line.

Line 57: VAR is declared to be the symbolic name of a FUNCTION which returns a REAL value and uses 3 dummy arguments called V, M, and J. By comparing the call in line 35 with this definition, it can be seen that the dummy argument V is a reference to the actual argument X, similarly that M refers to N, and J to K.

Scanned by Jonny Oddene for Sintran Data © 2011

Lines 58-59: The same comments apply as for lines 48-49.

Line 60: Initializes the return value to zero, VAR being the name of this FUNCTION.

Lines 61-63: A DO-loop to sum the squares of deviations for the J'th group. Note that it is assumed that AV has been set before the function is invoked.

Line 64: An extraction of the square root completes the evaluation of the standard deviation. SQRT is an intrinsic function and here the actual argument is an expression. In this expression, the numerator is REAL, but the denominator is of type INTEGER, so it is converted to REAL before the division is done.

Line 65: The END of this program unit. When the execution comes here, the value in VAR is taken as the value of the function and is sent back to the program unit that called the function.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 2

DATA TYPES, CONSTANTS, VARIABLES, ARRAYS AND SUBSTRINGS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

2.1 DATA TYPES

There are six data types defined in ANSI FORTRAN 77:

- *INTEGER*
- *REAL*
- *DOUBLE PRECISION*
- *COMPLEX*
- *LOGICAL*
- *CHARACTER*

In ND FORTRAN, there are further types:

*INTEGER*1 (ND-500 only), INTEGER*2, INTEGER*4*

DOUBLE INTEGER

*REAL*4, REAL*6 REAL*8*

*COMPLEX*8, COMPLEX*12 COMPLEX*16, DOUBLE COMPLEX*

*LOGICAL*1 (ND-500 only), LOGICAL*2, LOGICAL*4*

NUMERIC (ND-500 only)

These are fully described in Section 3.4. on page 57.

Each type has its own internal representation; for storage mapping see Appendix E. Appendix E also describes the default data types for the ND-100 and the ND-500.

Scanned by Jonny Oddene for Sintran Data © 2011

2.1.1 Type rules for identifiers

A symbolic name identifying a constant, variable, array, external function, or statement function can have its type declared in a Type statement, see Section 3.4. on page 57.

In the absence of an explicit declaration in a Type statement, the type is implied by the first letter of the name. A first letter of I, J, K, L, M, or N implies type integer and any other letter implies type real, unless an IMPLICIT statement is used to change the default implied type, see Section 3.5. on page 66.

The data type of an array element name is the same as the type of its array name. The data type of a function name is the type of the data item supplied by the function reference in an expression.

Scanned by Jonny Oddene for Sintran Data © 2011

2.2 CONSTANTS

A constant is an arithmetic constant, logical constant, or character constant. Constants do not change their value during execution of the object program. A PARAMETER statement enables a constant to be given a symbolic name, see Section 3.6. on page 68.

The value range for each type of constant is not specified in the ANSI FORTRAN 77 standard, and varies according to machine implementation. The ranges given below for ND FORTRAN apply to the ND-100 unless otherwise noted.

2.2.1 Integer constants

The form of an integer constant is an optional sign followed by a string of digits.

In ND FORTRAN, integers have either the type INTEGER*2 or INTEGER*4, see the Type statement (Section 3.4 on page 57). On the ND-500, the default integer type is INTEGER*4, and on the ND-100, it is INTEGER*2. These defaults can be changed by the DEFAULT command, see Chapter 3 in the ND FORTRAN User Guide, ND-60.265.

The values must lie between -2147483648 and +2147483647 inclusive. If the number lies within the inclusive range: -32768 to +32767 and the number of digits used is 5 or less, then the data type is the default INTEGER type. Otherwise it is INTEGER*4.

Scanned by Jonny Oddene for Sintran Data © 2011

Example:

```
      0     is INTEGER
  32000     is INTEGER
   -127     is INTEGER
1234567     is INTEGER*4
 -98765     is INTEGER*4     ( < -32768)
 000002     is INTEGER*4 ( > 5 digits used)
```

An integer data item is always an exact representation of an integer value.

In ND FORTRAN, integers can be represented as octal numbers. These are a string of digits in the range 0 to 7 inclusive, followed by the letter B. Octal numbers must be unsigned and are stored as positive numbers.

The values must lie in the range 0 to 37777777777B. If the number lies within the range 0 to 177777B and the number of digits used is 6 or less, then the data type is the default INTEGER. Otherwise it is INTEGER*4.

Example:

```
      1B   is INTEGER
  123456B  is INTEGER
  345670B  is INTEGER*4 (> 177777B)
0000002B   is INTEGER*4 (> 6 octal digits used)
```

In ND FORTRAN, an integer constant may be written as a hexadecimal constant. This is a string of hexadecimal digits starting with a decimal digit and ending with a X.

Example:

```
0A0X
1ABCX
```

Scanned by Jonny Oddene for Sintran Data © 2011

2.2.2 Real constants

The form of a basic real constant is an optional sign, an integer part, a decimal point, and a fractional part, in that order. Both the integer part and the fractional part are strings of digits; either of these parts may be omitted but not both.

A real exponent consists of the letter E followed by an optionally signed integer constant. A real exponent denotes a power of ten.

A real constant takes any of the forms:

- Basic real constant.
- Basic real constant followed by a real exponent.
- Integer constant followed by a real exponent.

The value of a real constant containing a real exponent is the product of the constant preceding the E and the power of ten indicated by the integer following it.

In ND FORTRAN, the absolute value of a real constant must be zero or lie between 10 ** -76 and 10 ** +76.

With the ND-500 and the optional 32-bit floating-point hardware on the NORD-10 and ND-100, all intermediate results during the execution of a program must also lie within this range, the accuracy being 6-7 decimal digits. The 48-bit floating-point hardware on the ND-100 can allow the range to extend from 10**-4932 to 10**+4932, with an accuracy of 9 digits, although for consistency and compatibility the range of real constants is restricted as before from 10**-76 to 10**+76. This limit is also imposed on output values.

Scanned by Jonny Oddene for Sintran Data © 2011

Examples of real constants are:

```
   0.
   3.1415927
-728.998
   -.1
     10E43
   0.2718283E+1
1557.4077E-3
  +1.E-10
```

A real value is an approximation to the actual value of a mathematical expression.

2.2.3 Double-precision constants

The form of a double-precision exponent is the letter D followed by an optionally signed integer constant. The exponent denotes a power of ten. A double-precision exponent is identical to a real exponent apart from the use of a D instead of an E.

A double-precision constant can take one of the forms:

- Real constant followed by a double-precision exponent.
- Integer constant followed by a double-precision exponent.

The value of a double-precision constant is the product of the constant preceding the D and the power of ten indicated by the integer which follows it.

In ND FORTRAN, the range of values of double-precision data items is the same as for real data items, but the accuracy is greater, being 16 decimal digits.

Scanned by Jonny Oddene for Sintran Data © 2011

Examples:

```
   2.302585092994046D0
   -.1D20
+123.4D-04
   0.12345678901234567890123456789D+21
```

Note that more digits than those of the accuracy limit may be written, the value of the constant being suitably approximated.

The range of double-precision exponents is -76 to +76.

2.2.4 Complex constants

The form of a complex constant is a left parenthesis followed by an ordered pair of real or integer constants separated by a comma, and followed by a right parenthesis. The first constant of the pair is the real part of the complex constant and the second is the imaginary part.

Example:

```
(0, 1)
(0.0, 1.0)
(3.1415927, 0)
(2.71828, 1.0E10)
(-1, +2.3E-1)
```

In ND FORTRAN, a COMPLEX*16 constant is written as a parenthesised pair of integer, real, or double precision constants, at least one of which is double precision.

Example:

```
(0, 1.D0)
(3.14159D-1, 1.4142D+1)
```

Scanned by Jonny Oddene for Sintran Data © 2011

2.2.5 Logical constants

The forms and values of a logical constant are:

FORM	*VALUE*
.TRUE.	*true*
.FALSE.	*false*

In ND FORTRAN, the default data type of a logical constant depends on the computer. Thus:

ND-500 uses LOGICAL*4 NORD-10 and ND-100 uses LOGICAL*2.

However, the default may be changed by the DEFAULT command, see Chapter 3 in the ND FORTRAN User Guide, ND-60.265.

2.2.6 Character constants

The form of a character constant is an apostrophe followed by a string of characters followed by an apostrophe. The string may contain any ASCII characters except CR (octal 15), LF (octal 12) or HT (octal 11).

The delimiting apostrophes are not part of the data item. Embedded apostrophes are represented by two consecutive apostrophes without intervening blanks. In a character constant, embedded blanks between the delimiting apostrophes are significant.

The length of a character constant is the number of characters between the delimiting apostrophes, except that each pair of consecutive apostrophes counts as a single character. The delimiting apostrophes are not counted.The length of a character constant must be greater than zero.

Scanned by Jonny Oddene for Sintran Data © 2011

However, in ND FORTRAN, a string of characters may be of length zero.

Examples:

String as Written	*Value*
'ABC'	*ABC*
' I AM GREAT'	*I AM GREAT*
'I''M THE GREATEST'	*I'M THE GREATEST*
''	*empty (ND FORTRAN extension)*

See Appendix E for the internal representation of character strings.

Scanned by Jonny Oddene for Sintran Data © 2011

2.3 VARIABLES

A variable is a data item having both a name and a type. Its value can be changed during the execution of a program.

Its name is a symbolic name and its type can be optionally specified by the appearance of the symbolic name in a Type statement. Otherwise its type is implied as being INTEGER or REAL by the first letter of its name, (see Section 2.1.1 on page 28), unless this is overidden by use of the IMPLICIT statement.

During the execution of a program, a variable may contain a defined or an undefined value. Before a value has been assigned to it, a variable will contain an undefined value, and any reference to it will produce an unpredictable result.

Scanned by Jonny Oddene for Sintran Data © 2011

2.4 ARRAYS

An array is an ordered set of data identified by an array name. Array names are symbolic names which must conform to the rules given in Section 1.3. on page 6.

The number of data items (or elements - see Section 2.4.1 on the next page) in an array is given by an array declarator having the form:

```
a (d [, d] .....)
```

where

a is the symbolic name of the array.

d is a dimension declarator, the number of these specified being equal to the number of dimensions of the array.

The form of the dimension declarator is:

```
[ d1 : ] d2
```

where

d_1 is the lower dimension bound.

d_2 is the upper dimension bound.

The lower and upper dimension bounds are arithmetic expressions and are described in Section 3.1 on page 45.

Scanned by Jonny Oddene for Sintran Data © 2011

Examples of array declarators:

TABLE (2, 3, 4)
ARRAY (M1 : M2, M3 : M4)

The size of an array is equal to the product of the sizes of the dimensions specified for that array by its array declarator. Thus, in the first of the above examples the array size would be 2x3x4 = 24. The size of an array is equal to the number of elements it contains.

2.4.1 Array elements

Each item of data in the array is known as an array element. An array element name, by which an array element is referenced, is the array name qualified by a subscript. The form of an array element name is:

a (s [, s] ...)

where

a	is the array name.
(s [, s] ...)	is a subscript where each *s* is an integer expression, referred to as a subscript expression.

A subscript expression can contain array element references and function references. The number of subscript expressions in the subscript must equal the number of dimensions declared for the array (see above).

Examples of array element names:

*TABLE (I/K ** 2, L)*
*ARRAY1 (I + ARRAY2 (J*K, L) , M)*

In ND FORTRAN, reference to a multi-dimensioned array may also be made as though it were an array of only one dimension. In this case, the array element referenced is given by the order

Scanned by Jonny Oddene for Sintran Data © 2011

in which the array elements are stored, see next section. The subscript of the first element being the lower bound of the first dimension.

2.4.2 Order of stored array elements

The elements of an array are arranged in storage in ascending order with the value of the first subscript varying most rapidly.

For example, elements of the array:

I (2, 3)

are stored in the order:

I (1,1), I (2,1), I (1,2), I (2,2), I (1,3), I (2,3)

2.4.3 Adjustable arrays

An adjustable array has an adjustable array declarator, i.e. one having dimension declarators containing variable names.

Note that adjustable arrays may only be used for dummy argument declarations within subprograms, see Section 11.1 on page 234.

Scanned by Jonny Oddene for Sintran Data © 2011

2.4.4 Assumed-size arrays

An assumed-size array is a constant array declarator or an adjustable array declarator, except that the upper dimension bound of the last dimension is an asterisk. The asterisk means that there is no declared upper limit of the array index. This does not change the requirement that the dummy argument must be wholly contained within the actual argument.

Note that assumed-size arrays may only be used for dummy argument declarations within subprograms, see Section 11.1 on page 234.

2.4.5 Actual and dummy array declarators

Each array declarator is either an actual array declarator or a dummy array declarator.

An actual array declarator is one in which each of the dimension bound expressions (see Section 3.1 on page 45). is an integer constant expression. A dummy array declarator on the other hand, may be a constant array declarator, an adjustable array declarator or an assumed-size array declarator.

For more detailed descriptions, see Section 11.1. on page 234.

Scanned by Jonny Oddene for Sintran Data © 2011

2.5 CHARACTER SUBSTRINGS

A character substring is a contiguous portion of a character variable or character array element. The name of the substring may be referenced and have values assigned to it.

The substring name can take the forms:

```
v ( [ e1 ] : [ e2 ] )
```

or

```
a ( s [, s ] ...)([ e1 ] : [ e2 ])
```

where

v is a character variable name.

a (s [, s]...) is a character array element name.

e_1, e_2 are each integer expressions called substring expressions.

e_1 indicates the leftmost character position of the substring and e_2 the rightmost. For example, A (2:4) specifies the characters in positions 2 through 4 of the character variable A, while B (4,3) (1:6) specifies characters in positions 1 through 6 of the character array element B (4,3).

e_1 and e_2 must be within the limits:
$1 \leq e_1 \leq e_2 \leq$ maximum string length

If e_2 exceeds the maximum string length, results are unpredictable.

If e_1 is omitted, a value of 1 (one) is assumed for it.

Scanned by Jonny Oddene for Sintran Data © 2011

If e_2 is omitted, then its assumed value is that of the length of the character variable or array element. Both e_1 and e_2 may be omitted.

A substring expression may be any integer expression. It can contain array element references and function references.

In ND FORTRAN there are two special values to note:

- If e_1 is a constant expression whose value is -1, then it is interpreted as the position of the first non-blank character in the string.
- If e_2 is a constant expression whose value is -1, then it is interpreted as the position of the last non-blank character in the string.

Thus, A (-1:-1) strips off leading and trailing blanks.

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 3

SPECIFICATION STATEMENTS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

FORTRAN specification statements specify storage allocation, type characteristics,and data arrangement. The different specification statements are:

- DIMENSION
- EQUIVALENCE
- COMMON
- Type statements
- IMPLICIT
- PARAMETER
- EXTERNAL
- INTRINSIC
- SAVE
- ASSEMBLY

All specification statements are non-executable.

3.1 THE DIMENSION STATEMENT

The DIMENSION statement provides the symbolic names and dimension specifications of arrays. Its form is:

```
DIMENSION ad [, ad]...
```

where

each *ad* is an array declarator of the form a(d,[, d]...), Section 2.4, on page 37. Note that array declarators may also appear in COMMON statements and Type statements.

Scanned by Jonny Oddene for Sintran Data © 2011

Each *a* appearing in a DIMENSION statement is the symbolic name of an array in the same program unit. Each *d* is a dimension declarator, and the number of dimensions of the array is the number of dimension declarators in the array declarator. The minimum number of dimensions is one and the maximum is seven.

Note that in ND FORTRAN, the upper limit on the number of dimensions in an array is not applicable, except that arrays with more than 7 dimensions cannot be handled by the Symbolic Debugger.

The form of a dimension declarator is also given in Section 2.4, on page 37. Each dimension may be expressed as having two bounds, a lower and an upper, separated by a colon. The value of either bound may be positive, negative, or zero. If only the upper bound is given, then the value of the lower bound is one.

Dimension bounds are arithmetic expressions in which all constants (or their symbolic names) and variables are of type integer. The upper dimension bound of the last dimension may be an asterisk. The array declarator containing an asterisk in its last dimension bound may or may not be adjustable, see Section 2.4.3, on page 39. In an adjustable array, those dimension declarators that contain a variable name are called adjustable dimensions.

For example, in the statement:

DIMENSION PAGE (60), PROF (10, 12)

the array PAGE has 60 elements and 1 dimension. PROF is a two-dimensional array whose total size is:

10 x 12 = 120 elements

Scanned by Jonny Oddene for Sintran Data © 2011

The statement:

```
DIMENSION TABLE (-1 : 10, 0 : 9)
```

defines a two-dimensional array called TABLE. The first subscript may vary from -1 to 10 (i.e., 12 values) and the second subscript varies from 0 to 9 (i.e., 10 values) giving a total size of 120 elements.

The following code:

```
SUBROUTINE SUB (A, ROWS, COLS)
INTEGER ROWS, COLS
DIMENSION A (ROWS, COLS)
```

defines a dummy argument as an adjustable array whose size is given by further dummy arguments.

For example, if ROWS = 4 and COLS = 5 on one entry to SUB, then the size of A is 4 x 5 = 20 elements with the bounds of 4 and 5 remaining constant for this invocation, even though ROWS or COLS may receive new values during it. If, when it is called next time, ROWS = 3 and COLS = 2, then these bounds will hold for this new invocation.

In the next example:

```
SUBROUTINE CALC (TAB)
COMMON/CM/LEN
DIMENSION TAB ( 0 : LEN*(LEN + 1)/2,*)
```

TAB is an assumed-size array. The first upper bound is an integer expression, and the second upper bound is left free. Note that in these two last cases the bounds of the arrays are redetermined each time the subroutine is invoked, but that they remain fixed throughout each invocation.

Scanned by Jonny Oddene for Sintran Data © 2011

3.2 THE EQUIVALENCE STATEMENT

The EQUIVALENCE statement is used to specify that storage is shared by two or more variables, arrays, or character substrings.

An EQUIVALENCE statement has the form:

```
EQUIVALENCE (list) [, (list)] ....
```

Each *list* must contain at least two names. Names of dummy arguments are not allowed. Any subscript or substring expression in the *list* must be an integer constant expression.

If equivalenced items are of different data types, no type conversion is performed.

Example:

```
INTEGER*4 INT4
LOGICAL*4 LOG4
LOGICAL*2 LOG2
DOUBLE PRECISION RL8
EQUIVALENCE (INT4,LOG4),(LOG2,RL8)
```

The first pair of variables in the EQUIVALENCE statement, INT4 and LOG4, requires exactly the same storage, they will overlap exactly. The second pair, require different amounts of storage, LOG2 requires 16 bits and RL8 requires 64 bits, but LOG2 and RL8 will begin at exactly the same place in memory.

There are restrictions on strict ANSI FORTRAN 77 due to the architecture of the NORD-10 and ND-100 machines. The reader should consult Appendix E for details. These restrictions are not applicable to the ND-500 or the NORD-10 and ND-100 with 32-bit floating-point hardware. Storage mapping is fully described in Appendix E.

Scanned by Jonny Oddene for Sintran Data © 2011

3.2.1 Array Names and Array Element Names

If an array element name occurs in an EQUIVALENCE statement, the number of subscript expressions must be the same as the number of dimensions specified in the array declarator for that array name. The use of an array name unqualified by a subscript in an EQUIVALENCE statement has the same effect as specifying the first element of the array.

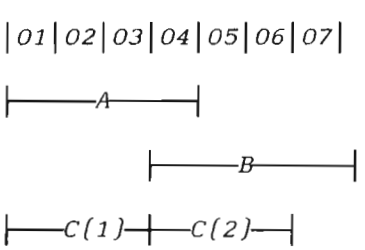
3.2.2 Character Variables in EQUIVALENCE Statements

Items of type CHARACTER may be equivalenced only with other items of type CHARACTER.

Example:

```
CHARACTER A*4, B*4, C(2)*3
EQUIVALENCE (A, C (1)), (B, C (2))
```

The sharing of storage can be illustrated as follows:



Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, the restriction on equivalencing CHARACTER only with CHARACTER is lifted. However, an arithmetic or logical item may not begin on an odd byte boundary on the ND-100, but the following is acceptable:

```
INTEGER K
CHARACTER*10 C
EQUIVALENCE (K, C (2 : 3))
```

since C can start at an odd byte so that K will start at an even byte.

However:

```
INTEGER K, N
CHARACTER*10, C
EQUIVALENCE (K, C (1 : 2)), (N, C (2 : 3))
```

is not allowed, since there is no way of avoiding one of either K or N starting at an odd byte.

On the ND-500, this situation produces an extension message, not an error.

3.2.3 Restrictions on EQUIVALENCE Statements

An EQUIVALENCE statement must not specify that consecutive storage units are to be nonconsecutive as in:

```
REAL A (2)
DOUBLE PRECISION D(2)
EQUIVALENCE (A (1), D (1)), (A (2), D (2))
```

Scanned by Jonny Oddene for Sintran Data © 2011

Nor may the same storage unit be specified more than once in a storage sequence, as in:

```
DIMENSION A (2)
EQUIVALENCE (A (1), B), (A (2), B)
```

However, there are several ways of specifying essentially the same equivalence information.

Example:

```
REAL A (20), B (10)
EQUIVALENCE (A (1), B (1)), (A (2), B (2))
```

Also, more than one list can refer to the same storage unit as in:

```
EQUIVALENCE (A,B,C), (A,D), (B,E,F), (C,G), (E,H)
```

which is the same as:

```
EQUIVALENCE (A,B,C,D,E,F,G,H)
```

Scanned by Jonny Oddene for Sintran Data © 2011

3.3 THE COMMON STATEMENT

The COMMON statement enables storage to be shared by different program units. This allows the program units to define and reference data without using arguments.

The form of the statement is:

```
COMMON [/[cb]/] list[[,]/[cb]/list]
```

where

cb is a COMMON block name.

list is a list of variable names, array names, and array declarators.

If the COMMON block name is omitted, the blank COMMON block will be used. If the first COMMON block name is omitted, the first two slashes are optional.

In each COMMON statement, items whose names appear in a list are declared to be in the immediately preceding COMMON block. If the first COMMON block name is omitted, then the items in the first list appear in blank COMMON.

Any COMMON block (including blank COMMON) may occur more than once in one or more COMMON statements within a program unit. The list following each successive appearance of the same COMMON block name is treated as a continuation of the list for that COMMON block.

Only one appearance of a symbolic name as a variable name, array name, or array declarator is permitted in all such lists within a program unit.

Dummy arguments must not appear in the list.

If a character variable or a character array is in a COMMON block then all items in the block must be of type character.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, the mixing of entities of character and non-character data types in one COMMON block is allowed.

3.3.1 COMMON Block Storage Sequences

During compilation of a source program, a storage sequence is formed for all items in the lists for a particular COMMON block. The order of the sequence is the same as the order of appearance of the lists. The sequence is further extended to include the storage for any storage sequence associated with it by equivalence association.

Storage sequences of all COMMON blocks with the same sequences have the same first storage unit. The storage sequences of all blank COMMON blocks also have the same first storage unit as each other. This results in the association of data in different program units.

For example, with the following code:

```
PROGRAM MAIN
COMMON / CM / MA, MB, MC
.....
END

SUBROUTINE SUB
COMMON / CM / NA, NB, NC
.....
END
```

MA and NA will share the same storage, as will the pairs MB, NB and MC, NC.

For details on the amount of storage occupied by each variable and the way in which COMMON blocks are mapped, see Appendix E.

Scanned by Jonny Oddene for Sintran Data © 2011

3.3.2 Differences between Named COMMON and Blank COMMON

- COMMON blocks of the same name must have the same size wherever they appear. Blank COMMON blocks may be of different sizes.

- Items in named COMMON blocks may be initially defined by a DATA statement in a BLOCK DATA subprogram.

  Note that in the ANSI FORTRAN 77 standard, initialization of named COMMON blocks is restricted to BLOCK DATA subprograms and blank COMMON blocks cannot be initialized by DATA statements.

  In ND FORTRAN both named and blank COMMON blocks may be initialized in any program unit.

- Execution of RETURN and END statements can cause items in named COMMON blocks to become undefined but not items in blank COMMON.

  In ND FORTRAN, items in COMMON blocks retain their values between calls, irrespective of whether they are named or not.

Scanned by Jonny Oddene for Sintran Data © 2011

3.3.3 Restrictions on COMMON and EQUIVALENCE

An EQUIVALENCE statement must not cause the association of the storage of two different COMMON blocks in the same program unit. Furthermore, EQUIVALENCE association must not cause storage extension to precede that of the storage of the first item in a COMMON statement.

Example:

```
COMMON / X / A
REAL B (2)
EQUIVALENCE (A, B (2))
```

is not permitted.

3.3.4 COMMON Blocks in APT

In ND FORTRAN for the NORD-10 and ND-100, a COMMON block can be placed at a user-defined location in the APT (Alternative Page Table). When doing this, all other data areas will reside on the NPT (Normal Page Table) along with the program code. The user has most of the responsibility for setting access to the APT correctly.

Such a COMMON block is defined by naming the block in the following way:

```
*n
```

where *n* is the address in the APT where the block should start.

Scanned by Jonny Oddene for Sintran Data © 2011

n must be a decimal or octal integer constant, not a symbolic name.

Example:

```
COMMON/*1000B/A, B, C
```

Places A, B and C in the APT from octal address 1000.

Variables in APT COMMON cannot be used as freely as other variables, due to the difficulties of addressing them. They are restricted to the following uses:

- In expressions
- On the left of assignment statements
- As actual arguments to subroutines and functions, if they are simple variables or array elements
- In WRITE and READ statements (but not arrays)
- Any data type, but not character

The user must set the alternative page table before an APT COMMON is accessed, by:

```
CALL ALTON (n)
```

where

n is the page table to use. (See also the SINTRAN Reference Manual, ND-60.128)

Before using any of the file subsystem monitor calls (e.g. RFILE), the APT must be disabled by:

```
CALL ALTOF
```

and then reinstated by:

```
CALL ALTON (n)
```

Scanned by Jonny Oddene for Sintran Data © 2011

as before.

Use of I/O through the normal FORTRAN statements READ, WRITE, PRINT, OPEN, CLOSE, BACKSPACE, ENDFILE and REWIND is automatically protected.

3.4 TYPE STATEMENTS

A Type statement is used to override or confirm an implicit type. It may also provide dimension information.

The appearance of the name of a variable, array, statement function, external function, or a constant in a Type statement, specifies the data type for that name for all occurrences of it in a program unit.

The name of a main program, subroutine, or block data subprogram must not be used in a Type statement.

Scanned by Jonny Oddene for Sintran Data © 2011

3.4.1 INTEGER, REAL, DOUBLE PRECISION, NUMERIC, COMPLEX and LOGICAL Type Statements

These statements have the form:

```
type var [/value/] [,var[/value/]]...
```

where

type is one of INTEGER, REAL, DOUBLE PRECISION, NUMERIC (fw,sc), COMPLEX and LOGICAL.

NUMERIC (fw,sc) is also used to specify entities of packed decimal format also known as BCD (Binary Coded Decimal). This is a fixed format, where *fw* (field width) specifies the number of digits in the entity, and *sc* (scaling factor) specifies the number of digits to the right of the decimal point. This type should be used mainly when mixing routines within COBOL and FORTRAN.

var is a variable name, array name, array declarator, function name, dummy-procedure name, or the symbolic name of a constant.

value is a constant, a symbolic name of a constant or INTRINSIC functions (see Section 5.1.2 on page 92 and Section 5.4.1 on page 103) with constant expressions as parameters. If *var* is an array, then *value* means a list of values, one for each element of the array; if the *list* contains less values than required for the entire array, the rest of the array will be initialized to zero; the *list* may contain a constant or symbolic name of a constant prefixed by a repetition factor (followed by an asterisk, as in the DATA statement). This is an ND FORTRAN extension.

In ND FORTRAN, additional types are allowed. The implications

Scanned by Jonny Oddene for Sintran Data © 2011

for the use of storage are fully explained in Appendix E.

The additional types are:

*INTEGER*1*	*occupies 1 byte of storage (ND-500 only)*
*INTEGER*2*	*occupies 2 bytes of storage*
*INTEGER*4*	*occupies 4 bytes of storage*
DOUBLE INTEGER	*occupies 4 bytes of storage*

The above types behave in the same way as type INTEGER. In particular, they can be used in expressions for subscripts etc. DOUBLE INTEGER and INTEGER*4 are identical in all respects.

*REAL*4*	*the same as REAL*
*REAL*6*	*the same as REAL*
*REAL*8*	*the same as DOUBLE PRECISION*
NUMERIC (fw,sc)	*(ND-500 only)*
*COMPLEX*8*	*the same as COMPLEX*
*COMPLEX*12*	*the same as COMPLEX*
*COMPLEX*16*	*COMPLEX values with DOUBLE PRECISION accuracy*
DOUBLE COMPLEX	*the same as COMPLEX*16*
*LOGICAL*1*	*occupies 1 byte of storage (ND-500 only)*
*LOGICAL*2*	*occupies 2 bytes of storage*
*LOGICAL*4*	*occupies 4 bytes of storage*

During the evaluation of an arithmetic expression, the order of implied conversion is:

*INTEGER*1*
*INTEGER*2*
*INTEGER*4*
REAL
DOUBLE PRECISION
NUMERIC (fw,sc)
COMPLEX

Scanned by Jonny Oddene for Sintran Data © 2011

DOUBLE COMPLEX

The CMPLX intrinsic function applied to a DOUBLE PRECISION argument gives a DOUBLE COMPLEX result. It should be used to retain accuracy when mixing DOUBLE PRECISION and COMPLEX operands.

There is an equivalent hierarchy for logical expressions. The order of implied conversion is:

*LOGICAL*1*

*LOGICAL*2*

*LOGICAL*4*

Scanned by Jonny Oddene for Sintran Data © 2011

3.4.2 CHARACTER Type Statement

The form of this statement is:

```
CHARACTER [*length [,]] name [/value/][,name[/value/]]...
```

where

name can take the form of:

```
v [*length]
```

or

```
a [(d)][*length]
```

v is a variable name, function name, dummy procedure name, or the symbolic name of a constant.

a is an array name.

a (d) is an array declarator.

length is the length (number of characters) of the associated name. It is one of the following:

- An unsigned, non-zero, decimal integer constant.
- An integer constant expression within parentheses and having a positive value.
- An asterisk in parentheses, (*).

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, if the expression is only the name of a symbolic constant, the surrounding parentheses can be omitted.

For example, in ANSI FORTRAN 77:

```
PARAMETER (LEN = 10)
CHARACTER C*(LEN)
```

could be written in ND FORTRAN as:

```
PARAMETER (LEN = 10)
CHARACTER C*LEN
```

In ND FORTRAN also, if the expression is not of type INTEGER, it will be converted to type INTEGER.

value is a constant, a symbolic name of a constant or the INTRINSIC function CHAR with a constant expression as parameter. If *name* is an array, then *value* means a list of values, one for each element of the array; if the list contains fewer values than required for the entire array, the rest of the array will be initialized to zero; the list may contain a constant or symbolic name prefixed by a repetition factor (followed by an asterisk, as in the DATA statement). This is an ND FORTRAN extension.

A length specification immediately following the word CHARACTER applies to each item in the statement without a length specification of its own. If this length specification does not appear, then the default length is one.

Scanned by Jonny Oddene for Sintran Data © 2011

Example:

```
CHARACTER*3 A, B*4, C
```

defines A, B and C as character strings of lengths 3, 4 and 3 respectively. Also:

```
CHARACTER A, B*4, C
```

gives A, B and C lengths of 1, 4 and 1 respectively.

A length specification must be an integer constant expression except for external functions, dummy arguments of external procedures, or character constants having a symbolic name.

If the length of a dummy argument is declared as (*) then it assumes the length of the associated actual argument for each reference of the subroutine or function. (When the associated actual argument is an array name then the length of an element of the array is assumed.)

Example:

```
SUBROUTINE S (C)
CHARACTER C*(*)
.....
END

PROGRAM MAIN
CHARACTER A*4, B*9
CALL S(A) CALL S(B)
.....
END
```

In the above code, the first time S is called, the dummy argument C identifies with A, and so has a length of 4; the second time, it takes the length of B, i.e., 9.

If the length of an external function is declared in a function subprogram as (*) then the function name must appear in a FUNCTION or ENTRY statement in the same subprogram. On execution of such a function reference, the assumed length is that specified in the referencing program.

Scanned by Jonny Oddene for Sintran Data © 2011

The length given for a character function in a referencing program must be an integer constant expression that agrees with the length given in the specifying subprogram.

Example:

```
FUNCTION NAME
CHARACTER *(*) NAME
.....
NAME = TAB (I)
RETURN
END

SUBROUTINE PERSON
EXTERNAL NAME
CHARACTER NAME*25, PN*25
.....
PN=NAME ()
.....
END

SUBROUTINE FIRM
EXTERNAL NAME
CHARACTER*35 NAME, FN
.....
FN=NAME ()
.....
END
```

In the above, when NAME is called from PERSON, its length is 25. When it is called from FIRM, its length is 35. Within both PERSON and FIRM, NAME must be declared with a constant length and not with an asterisk (*).

If a character constant with a symbolic name has its length declared as (*), then the constant assumes the length of its corresponding constant expression in a PARAMETER statement.

For example, in the code:

```
CHARACTER HEAD *(*)
PARAMETER (HEAD = 'TOTALS-BY-MONTH')
```

the length of HEAD becomes 15.

Scanned by Jonny Oddene for Sintran Data © 2011

A character statement function or the character dummy argument of a statement function must have a length which is an integer constant expression.

For example, if we have:

```
CHARACTER DIGITS*10, MNAMS*50
CHARACTER*3 MONTH, DAY*2, DATE*6, DD*2, DM*3
DATA DIGITS/'0123456789'/, MNAMS/'JAN FEB MAR...DEC'/
DAY (I) = DIGITS (I/10+1:I/10+1)//DIGITS (MOD(I,10)+1:MOD(I,10)+1)
MONTH (I) = MNAMS (3*I-2 : 3*I)
DATE (DD, DM) = DD // '-' // DM
```

then the statement functions DAY, MONTH and DATE must be of known fixed length, as must the dummy arguments of DATE, i.e., DD and DM.

In ND FORTRAN there are restrictions on the maximum length of items of type CHARACTER. On the ND-500, the maximum is 32767, and on NORD-10/ND-100 it is 2047. The upper limits apply to all CHARACTER items, including the maximum lengths of all CHARACTER expressions that are not assigned (i.e., used as actual parameters or as operands to relational operators).

See Appendix E for the internal representation of CHARACTER data.

Scanned by Jonny Oddene for Sintran Data © 2011

3.5 THE IMPLICIT STATEMENT

An IMPLICIT statement is used to change or confirm default implied data types, based on the initial letter of the symbolic name of a constant, variable, array, external function, or statement function.

The statement has the form:

```
IMPLICIT type (a[,a]...)[,type(a[,a]....)]
```

where

type is one of INTEGER, REAL, DOUBLE PRECISION, NUMERIC (fw,sc), COMPLEX, LOGICAL or CHARACTER [* length].

In ND FORTRAN, type may also be one of the ND extensions, see Section 3.4. on page 54.

a is a single letter or range of single letters in alphabetical order. A range is denoted by the first and last letter of the range separated by a minus.

length is the length of a character item and must be either an unsigned, non-zero, integer constant, or a positive integer constant expression in parentheses. Its default value is one.

Example:

IMPLICIT COMPLEX (C)

ensures that all untyped names beginning with a C will be of type COMPLEX.

Scanned by Jonny Oddene for Sintran Data © 2011

An IMPLICIT statement specifies a type for all:

- variables
- arrays
- symbolic names of constants
- external functions
- statement functions

based on the first letter of the name. The normal defaults for types can be expressed as:

IMPLICIT REAL (A-H, O-Z), INTEGER (I-N)

Example:

VARIABLE NAME	*IMPLICIT VARIABLE TYPE*
x123	*REAL*
horse	*REAL*
insect	*INTEGER*
c	*REAL*
J	*INTEGER*

An IMPLICIT statement does not change the type of any intrinsic function, and its scope is that of the program unit containing it.

Type specification by an IMPLICIT statement may be overridden in all cases by a type statement. An explicit type specification in a FUNCTION statement overrides an IMPLICIT statement for the name of that function subprogram. Note that the length is also overridden when a particular name appears in a CHARACTER or CHARACTER FUNCTION statement.

IMPLICIT statements must precede all other specification statements in a program unit except a PARAMETER statement. More than one IMPLICIT statement may be used in a program unit.

Scanned by Jonny Oddene for Sintran Data © 2011

3.6 THE PARAMETER STATEMENT

A PARAMETER statement is used to give a constant a symbolic name.

The form of a PARAMETER statement is:

```
PARAMETER (p = e [, p = e]....)
```

where

p is a symbolic name of a constant.

e is a constant expression.

The assignment to *p* is made according to the rules for the assignment statements, see Chapter 7.

When *p* is of type integer, real, double precision, or complex, then the corresponding expression must be an arithmetic constant expression. If *p* is of type character or logical, the corresponding expression must be a constant expression of type character or logical respectively.

p must not be defined more than once in a program unit. Furthermore, if it is not of default implied type, then its type must be specified by a Type statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement.

If *p* is of type character and of length other than the default length of one, its length must be also defined prior to its first appearance in a PARAMETER statement.

For example, the following code:

```
PARAMETER (PI = 3.141593)
COMPLEX J
PARAMETER (J = (0.,1.), ROOT2 = 1.4142)
PARAMETER (RADIAN = 180/PI)
```

defines three REAL symbolic constants and one COMPLEX one.

Scanned by Jonny Oddene for Sintran Data © 2011

In the sequence:

```
PARAMETER (MAX = 100)
IMPLICIT REAL (M)
DIMENSION MATRIX (MAX, MAX)
PARAMETER (MAGFLD = 0.82)
```

the IMPLICIT statement does not apply to the PARAMETER statement that precedes it. Thus MATRIX and MAGFLD are both of type REAL but MAX is of default type (i.e., INTEGER).

Scanned by Jonny Oddene for Sintran Data © 2011

3.7 THE EXTERNAL STATEMENT

An EXTERNAL statement is used to identify an external or dummy procedure and to permit its symbolic name to be used as an actual argument.

The form of an EXTERNAL statement is:

```
EXTERNAL proc [,proc]...
```

where

each *proc* is the name of an external procedure, dummy procedure, or block data subprogram.

When a name appears in an EXTERNAL statement it is declared to be an external procedure, dummy procedure, or block data subprogram name. If an external procedure name or dummy procedure name is used as an actual argument, it must appear in an EXTERNAL statement in the same program unit. A statement function name cannot appear in an EXTERNAL statement.

The name of an intrinsic function appearing in an EXTERNAL statement becomes the name of some external procedure, whereupon an intrinsic function of the same name cannot be referenced in the program unit.

Only one occurrence of a symbolic name is allowed in all of the EXTERNAL statements of a program unit.

Scanned by Jonny Oddene for Sintran Data © 2011

3.8 THE INTRINSIC STATEMENT

An INTRINSIC statement is used to identify the name of an INTRINSIC function and to permit the use of this name as an actual argument.

The form of this statement is:

```
INTRINSIC fname [,fname]...
```

where

each *fname* is an INTRINSIC function name.

If a specific name of an INTRINSIC function is used as an actual argument, it must appear in an INTRINSIC statement in the same program unit. For the INTRINSIC function names which must not be used as actual arguments, see Section 11.2 on page 244.

If a generic function name appears in an INTRINSIC statement, it does not loose its generic property.

A symbolic name may only appear once in all of the INTRINSIC statements of a program unit and it must not occur in this unit in both an EXTERNAL and an INTRINSIC statement.

Scanned by Jonny Oddene for Sintran Data © 2011

3.9 THE SAVE STATEMENT

A SAVE statement retains the defined values of items after execution of a RETURN or END statement in a subprogram.

It has the form:

```
SAVE [ a [,a ]...]
```

where

each *a* is a named common block name preceded and followed by a slash, or a variable name, or an array name. (Dummy argument names, procedure names, and names of items in a common block must not appear.)

A SAVE statement without a list is treated as though it contained the names of all allowable items within the program unit. The appearance of a common block name preceded and followed by a slash has the effect of specifying all of the items in that block.

When a common block name occurs in a SAVE statement in a subprogram then it must occur in a SAVE statement in every subprogram in which the common block appears.

If a named common block appears in a SAVE statement of a subprogram, then the current value of items in the common block storage sequence when a RETURN or END statement is executed, are made available to the next program unit specifying that common block.

If a named common block is specified in the main program unit, then the current values of the common block storage sequence become available to each subprogram specifying that common block; a SAVE statement in this program has no effect.

If a local item appearing in a SAVE statement but not in a common block is in a defined state when a RETURN or END statement is executed, then this item is defined with the same value at the next reference to the same subprogram.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, this statement is checked for valid syntax, but is otherwise ignored. If the REENTRANT compiler option is OFF or the FIXED-DATA-AREA compile option is ON, all variables are saved, and so the statement is redundant. If the REENTRANT option is ON or the FIXED-DATA-AREA compile option is OFF, all common variables are saved but other variables are not.

NOTE
Reentrant programs do not conform to the ANSI FORTRAN 77 standard if they contain SAVE statements referring to local variables.

Scanned by Jonny Oddene for Sintran Data © 2011

3.10 THE ASSEMBLY STATEMENT

This statement is an ND FORTRAN extension and is used to modify the calling sequence for EXTERNAL subroutines and functions.

The form of the statement is:

ASSEMBLY name [,*name*]...

Each *name* is the name of an externally compiled routine or function, or the name of a dummy argument. The names cannot also appear in an EXTERNAL statement. They may be used either as subroutines or as functions, with the exception of character functions.

When these functions are called, their actual arguments must obey certain restrictions:

- There can be no more than 4 of them.
- They must be INTEGER*2 or non-character array names.

The statement is checked for syntax on the ND-500, but the names are then assumed to be EXTERNAL names or arguments.

The ASSEMBLY statement modifies the calling sequence to EXTERNAL program units on ND-100 programs. It can be used where the external routine is written in MAC, NPL, or PLANC with the SPECIAL option, see Appendix F for details of the calling sequences.

On the ND-500, the statement is exactly equivalent to an EXTERNAL statement.

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 4

THE DATA STATEMENT

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

A DATA statement is used to provide initial values for variables, arrays, array elements and substrings.

The form of a DATA statement is:

```
DATA namelist/valuelist/ [[,] namelist/valuelist / ]..
```

where

namelist is a list of names of variables, arrays, array elements, and substrings, together with implied DO lists.

valuelist consists of a list of constants and/or symbolic names of constants, each of which may be prefixed by a repetition factor.

In ND FORTRAN, valuelist may consist of some INTRINSIC functions with constant expressions as parameters (see Sections 5.1.2 on page 92, 5.2.1 on page 95 and 5.4.1 on page 103).

4.1 DATA STATEMENT RESTRICTIONS

Each namelist and valuelist must contain the same number of items. There is a one-to-one correspondence between the items in the two lists.

If an array name without a subscript appears in the list, then there must be one constant for each element of that array.

Initialization must not occur more than once for variables, array elements, or substrings.

Scanned by Jonny Oddene for Sintran Data © 2011

Each constant value from the valuelist is used to initialize the corresponding element from the namelist according to the rules of a normal FORTRAN assignment statement, see Chapter 7 which starts on page 123.

In ND FORTRAN, DATA statements may precede other specification statements. However, in this case, the variables in the DATA statements must have their data types defined in preceding declaration statements or they will receive implicit data types.

Examples of simple DATA statements:

- *DATA I/10/*

This assigns a value of 10 to the integer variable I before execution of the program.

- *DATA PI/3.1415927/E/2.7182818/*

  is the same as

  DATA PI,E/3.1415927, 2.7182818/

- To initialize a 6-element array to the values 1, 2, 3, 4, 5, and 6, we may write:

```
REAL X(6)
DATA X/1, 2, 3, 4, 5, 6 /
```

- To zero an array, the following could be used:

```
REAL STATS (10, 10)
DATA STATS /100*0./
```

Scanned by Jonny Oddene for Sintran Data © 2011

- For multidimensional array names, the implied order of elements is with the first subscript varying most rapidly. Thus:

```
DIMENSION A (3, 3)
DATA A/11, 21, 31, 12, 22, 32, 13, 23, 33/
```

will produce an array with the values:

```
A(1,1)=11, A(2,1)=21, A(3,1)=31, A(1,2)=12,...
```

Note that replication factors can cut across name-list items. Thus:

```
DIMENSION A(8), B(8)
DATA A, B/1, 14*0, -1/
```

will set A(1) to 1, B(8) to -1, and all other elements of A and B to 0.

Scanned by Jonny Oddene for Sintran Data © 2011

4.2 IMPLIED DO IN A DATA STATEMENT

An implied DO list may appear in a DATA statement namelist, see Section 8.7 on page 144. It is written as:

$$(dlist,\ I = m_1,\ m_2\ [,\ m_3])$$

where

dlist is a list of array element names, and it may contain other implied DO lists.

I is the name of an integer variable, here called the implied DO-variable.

m_1, m_2, m_3 and the subscripts in the *dlist* are each an integer constant expression or an integer expression containing only constants and the implied DO-variable.

An iteration count and the values of the implied DO-variable are established from m_1, m_2, and m_3 exactly as for a DO-loop, see Section 8.7 on page 144, except that the iteration count must be positive.

Example:

To initialize the even elements of a one-dimensional array to +1, and the odd elements to -1, you may write:

```
INTEGER SGN(20)
DATA (SGN(I),I=2,20,2)/10*+1/
```

Scanned by Jonny Oddene for Sintran Data © 2011

```
(SGN (I),I=1,20,2)/10*-1/
```

or to create a character string of alternating A's and B's:

```
CHARACTER C*40
DATA (C(2*K-1:2*K),K=1,20)/20*'AB'/
```

To initialize only the diagonal elements of a square array:

```
DIMENSION Q (10, 10)
DATA (Q(N,N),N =1,10)/10*1.1/
```

The default ordering of a two-dimensional array is by columns. To set data in by rows, you can write:

```
DIMENSION A(3,3)
DATA ((A (I,J),J=1,3)I=1,3)/11,12,13,21,22,23,31,32,33/
```

which will set up A as in the last example in the previous section. Note the ordering of the loops. The innermost one varies most often.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 5

EXPRESSIONS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

An expression is formed from operands, operators and parentheses. This chapter describes the formation, interpretation, and evaluation rules for the various types of expressions. These may be:

- Arithmetic
- Character
- Relational
- Logical

5.1 ARITHMETIC EXPRESSIONS

The simplest forms of arithmetic expressions are unsigned arithmetic constants, symbolic names of arithmetic constants, and arithmetic types of variables, array elements, and function references.

Examples:

99	*(arithmetic constant)*
IV	*(integer variable)*
TABLE (2,3,4)	*(array element)*
LOG (X+Y)	*(function reference)*

More complicated arithmetic expressions can be formed by using one or more arithmetic operands together with arithmetic operators and parentheses.

Scanned by Jonny Oddene for Sintran Data © 2011

The arithmetic operators are:

OPERATOR	*MEANING*
**	*Exponentiation*
/	*Division*
*	*Multiplication*
-	*Subtraction (or negation)*
+	*Addition*

All the above operators are binary, i.e. used with two operands. The - and the + are also available as unary operators, i.e. they can be used with only one operand.

There is a precedence among the arithmetic operators which determines the order in which the operands are to be combined (unless the order is changed by the use of parentheses) as follows:

OPERATOR	*PRECEDENCE*
**	*highest*
* and /	*intermediate*
+ and -	*lowest (unary and binary)*

Within each precedence level, the order is assumed to be from left to right, except with exponentiation which is evaluated from right to left.

Scanned by Jonny Oddene for Sintran Data © 2011

The arithmetic operands are:

- unsigned arithmetic constants
- symbolic names of arithmetic constants
- arithmetic variables
- arithmetic array elements
- function references
- arithmetic expressions enclosed in parentheses
- any of the above operands combined by means of arithmetic operators to form arithmetic expressions.

Examples:

If X, Y, Z, A, and B are variables:

X+Y	*Forms the sum of X and Y.*
X-Y	*Subtracts Y from X.*
X+Y+Z	*Adds together X, Y, and Z.*
X+Y-Z	*Adds X and Y, and then subtracts Z from the result (see the general notes below).*
*X*Y/Z*	*Multiplies X and Y before dividing the result by Z (see the general notes below).*
*X/Z*Y*	*First divides Z into X, and then multiplies the result by Y.*
*X*Y+Z*	*Multiplies X and Y, and then adds Z to the result.*
*Z+X*Y*	*Multiplies X and Y, and then adds the result to Z; the order here is determined by operator precedences. * is performed first, followed by the +, as in example 7.*
*X**Y**Z*	*Raises Y to the power of Z first, and then X is raised to the power of this result.*

Scanned by Jonny Oddene for Sintran Data © 2011

*-A**2*	*Since the operator ** has precedence in this example, its operands will be combined first. Thus, the expression will be interpreted as:*
	*- (A ** 2) .*

Expressions containing two consecutive arithmetic operators, such as A**-B or A+-B, are not allowed. However, expressions such as A**(-B) are permitted.

In ND FORTRAN such juxtaposing of signs is allowed when the second of them is a unary + or -. Thus, A+-B is evaluated as A+(-B), and X---Y as X-(-(-Y)) etc.

If the order dictated by the precedence rules is not the order required, then parts of an expression may be written within parentheses. Parts thus enclosed are then evaluated as a whole expression before being used as an operand.

Example:

X+Y/Z	*Y is divided by Z, and then the result is added to X (precedence rules).*
(X+Y)/Z	*Ensures that X is added to Y before the result is divided by Z.*
(X+Y)/(X+Z)	*Here X+Y and X+Z will be computed separately and then the result of X+Y will be divided by the result of X+Z. Note that in this case there is no stipulation as to whether X+Y or X+Z is evaluated first.*

While the symbols +, -, *, /, and ** represent the usual mathematical operations, the reader should be aware that the underlying computing hardware has fixed limits as to the precision and accuracy of the respresentation of values and of the results of operations. These are described for each machine in Appendix E.

Scanned by Jonny Oddene for Sintran Data © 2011

Note that the order of operations on the computing hardware is such that the result would be mathematically exact if the hardware were mathematically precise. If a particular order of operations is vital for numerical accuracy, it is best to use parentheses to force the order.

Example:

X+Y+Z	*Represents the sum of X, Y, and Z. The computation may add X to Y and then add Z, or it may add Y to Z and then add X.*
(X+Y)+Z	*Ensures that X and Y are added together first, before adding Z to this result.*

5.1.1 Interpretation of Results for Arithmetic Expressions

When the operator + or - operates on a single operand, the data type of the result is the same as that of the operand.

When an arithmetic operator operates on a pair of operands, then, except for exponentiation, the data type of the result is as follows:

- If the types of the two operands are the same, then the data type of the result will be the data type of these operands.
- If the types of the two operands are different, the operand of lower data type (see below) is converted to the data type of the other operand. Thus, the data type of the result will be that of the operand with the higher data type.

Scanned by Jonny Oddene for Sintran Data © 2011

The hierarchical order of the data types is:

DATA TYPE	*ORDER*
integer	*lowest*
real	
double precision	
numeric	
complex	*highest*

Note that the conversion takes place before the operation is performed, and that the operators are defined only for operands of equivalent type. The conversions are defined by the INTRINSIC functions REAL, DBLE, and CMPLX. See Section 11.2.2 on page 244.

Example:

If I, R, D, and C are variables of type INTEGER, REAL, DOUBLE PRECISION, and COMPLEX respectively, then:

- *The result of the expression I+I is of type INTEGER.*
- *I*R will cause I to be converted to type REAL before the multiplication, and the result is of type REAL.*
- *(D/I)+R will first cause conversion of I to DOUBLE PRECISION; then the division will occur, then R will be converted to DOUBLE PRECISION, and finally the addition will take place giving a result of type DOUBLE PRECISION.*
- *D/I+R will have exactly the same effect as the previous example, since the precedence rule for operators implies that division occurs before addition.*
- *R*C will produce a result of type COMPLEX.*

Scanned by Jonny Oddene for Sintran Data © 2011

For the exponentiation operator, if the exponent (i.e. the right-hand operand) is of type integer, then the data type of the result is the same as that of the left-hand operand. Otherwise conversion takes place as given above for the case of two arithmetic operands.

Example:

If I and R are variables of data types INTEGER and REAL respectively, then:

- *I**I* — *Has a result of type INTEGER.*
- *R**I* — *Is an expression of type REAL, (but note that I is not converted here).*
- *I**R* — *Is of type REAL, and I is converted to REAL.*
- *R**R* — *Is of type REAL.*

NOTE:

If the exponent is of type INTEGER, then exponentiation can be defined as repeated multiplications, so that every value of the base (i.e. left-hand operand) is admissible. (except zero if the exponent is negative.) But if the exponent is not of type INTEGER, A**B is defined as EXP(B*LOG(A)), where EXP and LOG are the INTRINSIC functions described in Section 11.2.2 on page 244. In particular, note that LOG is not defined for negative values of its argument. It is important to realize that the difference in definition is dependent on the type of the exponent and not on its value. Thus the expression (-3.0)**(2.0) constitutes an error, whereas (-3.0)**2 does not.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, there is one exception to the rule that if the right-hand operand with a ** operator is of type INTEGER then the result type is the same as the type of the left-hand operand. If the right-hand operand is INTEGER*4 and the left operand is INTEGER*2, then the result is INTEGER*4.

5.1.2 Arithmetic Constant Expressions

An arithmetic constant expression is an arithmetic expression in which each operand is an arithmetic constant, a symbolic name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses. Any arithmetic operator is allowed; the operator ** is valid only if the exponent is of type INTEGER.

In ND FORTRAN, any of the following INTRINSIC functions, may be used:

MIN, MAX, MOD, ABS, ICHAR, NINT, ANINT, DIM, DPROD, CMPLX, CONJG, IMAG or LEN,

provided that all parameters are constant expressions.

Scanned by Jonny Oddene for Sintran Data © 2011

5.2 CHARACTER EXPRESSIONS

The simplest form of a character expression is a character constant or the symbolic name of a character constant, or a character-type variable, array element, substring, or function reference. More complicated character expressions are formed by using one or more character operands together with the character operator and parentheses.

Evaluation of a character expression produces a character-type result.

The character operator is: // which represents concatenation.

The result of 'AB' // 'CD' is 'ABCD'.

If a character variable is of unknown length, then there are certain restrictions on its use in character expressions, in that it can be used only in character assignment statements, and even then, only when it directly forms part of the final result.

Example:

If we have the following declarations:

```
SUBROUTINE SR(C)
CHARACTER C*(*), A*100, B*10
```

then C is of unknown length, i.e. its length is taken from the actual parameter.

You are allowed to write:

```
A = C//B
```

because the final result length is constrained by the length of A.

Scanned by Jonny Oddene for Sintran Data © 2011

But you cannot write:

```
CALL X(C//B)
```

because the actual argument is an expression whose length is not constrained.

Similarly, the following expressions are allowed:

```
A = C (I:J) // C (1:N)
B = A (1:3) // C (4:7)
A = (C (1:N) // B) // (B (2:N) // C)
```

but none of these expressions can be used as actual arguments, or as part of a relational expression (even though they may then form part of an assignment statement).

Note that a symbolic constant always has a known length since a declaration with length (*) means: use the length of the constant expression assigned to it by a PARAMETER statement.

Example:

```
CHARACTER ALPHA* (*)
PARAMETER (ALPHA = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

implies no restrictions on the use of ALPHA due to its length (*).

Character operands are:

- character constants
- symbolic names of character constants
- character variables
- character array elements
- character substrings
- character function references
- character expressions enclosed in parentheses
- any combination of the above operands using the character operator.

Scanned by Jonny Oddene for Sintran Data © 2011

A character expression is a sequence of one or more operands separated by the character operator, (concatenation operator, //). The evaluation is from left to right. Thus the expression:

'AB' // 'CD' // 'EF'
is the same as:

('AB' // 'CD') // 'EF'

5.2.1 CHARACTER Constant Expressions

A character constant expression is a character expression in which each operand is a character constant, the symbolic name of a character constant or a character constant expression enclosed in parentheses. The only operand allowed is the concatenation operator, //.

In ND FORTRAN, the INTRINSIC function, CHAR, may be used, provided that the parameter is a constant expression.

Scanned by Jonny Oddene for Sintran Data © 2011

5.3 RELATIONAL EXPRESSIONS

A relational expression is used to compare the values of two arithmetic expressions or two character expressions.

In ND FORTRAN, it is also possible to compare logical expressions for equality or non-equality.

Relational expressions may form part of logical expressions and, on evaluation, they produce a result of type logical, i.e. a value .TRUE. or .FALSE. .

The relational operators are:

OPERATOR	*MEANING*
.LT.	*Less than*
.LE.	*Less than or equal to*
.EQ.	*Equal to*
.NE.	*Not equal to*
.GT.	*Greater than*
.GE.	*Greater than or equal to*

5.3.1 Arithmetic Relational Expressions

The form of an arithmetic relational expression is:

e_1 *rel* e_2

where

rel is a relational operator.

e_1 and e_2 are each an integer, real, double precision, numeric or complex expression.

Scanned by Jonny Oddene for Sintran Data © 2011

A complex operand is permitted only when the relational operator is .EQ. or .NE. .

Before the comparison is carried out, the operands are converted, if necessary, to make them of the same type. The rules are the same as those for the common arithmetic operators (see Section 5.1 on page 86).

Example:

If I, R, D, and C are variables of type INTEGER, REAL, DOUBLE PRECISION, and COMPLEX respectively, then:

- *I.LE.0 will give a value .TRUE. if I has the value zero or a value less than zero.*
- *R.GT. 0 - 10 will yield .TRUE. if the value of R is greater than -10.*
- *D.LT.R will convert R to DOUBLE PRECISION and the comparison will yield .TRUE. if the value of D is less than R.*
- *1.0.LE.I will convert I to REAL since the constant 1.0 is REAL (note the extra dot).*

For the relationship between arithmetic and relational operators, see Section 5.5 on page 103.

Scanned by Jonny Oddene for Sintran Data © 2011

5.3.2 CHARACTER Relational Expressions

Character relational expressions have the form:

e_1 *rel* e_2

where

rel is a relational operator.

e_1, e_2 are character expressions.

NOTE:

e_1 is considered to be less than e_2 if its value precedes that of e_2 in the collating sequence (see Appendix A).

If the lengths e_1 and e_2 are unequal, then for comparison purposes the shorter string is extended to the right and filled with blanks.

5.3.3 LOGICAL Relational Expressions

In ND FORTRAN, logical quantities can be compared by .EQ. and .NE. . These operators have the usual precedence of relational operators and perform their usual functions. If two logical expressions are both .TRUE. or both .FALSE., then comparing them with .EQ. will give .TRUE., etc.

Scanned by Jonny Oddene for Sintran Data © 2011

5.4 LOGICAL EXPRESSIONS

Evaluation of a logical expression produces a logical result, i.e. with a value of .TRUE. or .FALSE.

In its simplest form, a logical expression is a logical constant (or the symbolic name of one), or a logical variable, array element, function reference, or it can also be a relational expression.

More complicated expressions can be formed using one or more logical operands combined with logical operators and parentheses.

The logical operators are:

OPERATOR	*MEANING*
.NOT.	*logical negation*
.AND.	*logical conjunction*
.OR.	*inclusive or*
.EQV.	*logical equivalence*
.NEQV.	*logical non-equivalence (exclusive or)*

The operator .NOT. is unary, i.e. used with one operand. The other operators are binary, i.e. used with two operands.

Scanned by Jonny Oddene for Sintran Data © 2011

The logical operators have a precedence order, i.e. the order in which operands are to be evaluated, unless this is changed by the use of parentheses.

OPERATOR	*PRECEDENCE*
.NOT.	*highest*
.AND.	
.OR.	
.EQV. or .NEQV.	*lowest*

For example, in:

A .OR. B .AND. C

the .AND. operator has higher precedence than the .OR. operator. Therefore the interpretation of the above expression is the same as the following:

A .OR. (B .AND. C)

The values of expressions involving the above operators is shown below, where X_1 and X_2 are logical operands:

X_1	.NOT. X_1
.TRUE.	.FALSE.
.FALSE.	.TRUE.

X_1	X_2	X_1 .AND. X_2	X_1 .OR. X_2
.TRUE.	.TRUE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.	.FALSE.

Scanned by Jonny Oddene for Sintran Data © 2011

X_1	X_2	X_1 .EQV. X_2	X_1 .NEQV. X_2
.TRUE.	.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.

The logical operands are:

- logical constants
- symbolic names of logical constants
- logical variables
- logical array elements
- logical function references
- relational expressions
- logical expressions enclosed in parentheses
- any of the above operands combined by means of logical operators to form logical expressions.

For examples of how these combine with relational and arithmetic operators, see Section 5.5 on page 103.

The data type of the result of an operator which returns a logical result is LOGICAL.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, the result is the default LOGICAL data type, i.e. LOGICAL*2 for the ND-100 and LOGICAL*4 for the ND-500. The default can be changed by the DEFAULT command (see Chapter 3 in the ND FORTRAN User Guide, ND-60.265). Automatic conversion between the LOGICAL types occurs on assignment. If an operator has operands of different types, conversion according to Section 3.4.1, on page 58, will be done before the operation takes place.

If the operands are arithmetic, the normal conversions and precedence rules apply (see Section 5.1, on page 86).

In ND FORTRAN, the logical operators can also be used on entities of type INTEGER, i.e. they treat the integer value as a string of bits, operating on corresponding bits. A bit whose value is 1 is interpreted as .TRUE. and a bit whose value is 0 is interpreted as .FALSE. .

Thus, if A and B are integer variables with values:

A = 1010B

B = 1100B

then:

A .AND. B is *1000B*

A .OR. B is *1110B*

A .NEQV. B is *0110B*

If A and B are INTEGER*2, then:

.NOT. A is *176767B*

A .EQV. B is *177667B*

It is important to note that although these operators produce integer results when operating on integers, they still have the same precedence as the logical operators.

Scanned by Jonny Oddene for Sintran Data © 2011

5.4.1 LOGICAL Constant Expressions

A logical constant expression is a logical expression in which each operand is a logical constant, the symbolic name of a logical constant, a relational expression in which each operand is a constant expression, or a logical constant expression enclosed in parentheses. Any logical operator or relational operator is allowed.

In ND FORTRAN, you can to use any of the INTRINSIC functions: LGE, LGT, LLE or LLT, provided that all parameters are constant expressions.

5.5 EVALUATION OF EXPRESSIONS

This section applies to arithmetic, character, relational, and logical expressions. The order of evaluation of expressions is determined by:

- The use of parentheses
- The established precedence among the various operators
- The location of operators within an expression

5.5.1 The Use of Parentheses

Expressions within parentheses are evaluated first. Where parenthetical expressions are nested (one contained within another), the innermost expression is evaluated first, followed by the next innermost, and so on, until the outermost parenthetical expression has also been evaluated. If more than one operator is contained in an expression within parentheses, the computation proceeds according to the precedence rules for the operators.

Scanned by Jonny Oddene for Sintran Data © 2011

5.5.2 Precedence of Operators

The hierarchy of precedence among the arithmetic operators (see Section 5.1 on page 86), and logical operators (see Section 5.4 on page 99), has already been discussed. There is only one character operator and no precedence has been established among the relational operators.

Precedence among the various types is as follows:

OPERATOR	*PRECEDENCE*
Arithmetic	*Highest*
Character	
Relational	
Logical	*Lowest*

An expression may contain more than one kind of operator.

For example, the logical expression:

L .OR. A+B .GE. C

where *A*, *B*, and *C* are of type real and *L* is of type logical, contains an arithmetic operator, a relational operator, and a logical operator. This expression would be interpreted in the same way as:

L .OR. ((A+B) .GE. C)

5.5.3 Location of Operators within an Expression

When a series of exponentiation operators occurs within an expression, the order of evaluation is from right to left.

All other operations are computed from left to right when there is more than one occurrence within an expression of operators at the

Scanned by Jonny Oddene for Sintran Data © 2011

same hierarchical level.

Example:

Using the variable names and types below:

I, K	*integer*
R, S	*real*
L, M	*logical*
G, H	*character*

these more complex expressions will be interpreted as follows:

I + K	*Simple arithmetic of type INTEGER.*
L .OR. M	*Simple LOGICAL.*
R .LT. S	*Relational giving result of type LOGICAL.*
G // H	*Simple character expression of type CHARACTER.*
(I + 1) .EQ. K	*Compares K with (I+1) giving a result of type LOGICAL.*
I + 1 .EQ. K	*K is compared with (I+1) since arithmetic operators are evaluated before relational operators.*
*R * 2 .GT. S + 10.*	*Compares (R*2) with (S+10.) yielding a result of type LOGICAL.*
I .EQ. 3 .OR. R .LT. S	*Performs a comparison between I and 3, and a comparison between R and S. These two logical results are then combined with the .OR. operator to give a logical result.*

Note that the order in each example above is described for explanatory purposes only so that an expression can be correctly interpreted. However, the actual order of interpretation is not fixed, so long as the result is mathematically and logically equivalent. In reality, it could be the case that part of an expression is not evaluated at all.

Scanned by Jonny Oddene for Sintran Data © 2011

Consider the following:

```
IF (I .EQ. 1 .OR. K .EQ. 4) GO TO 10
```

If I has the value 1, then the expression in brackets is known to be true after testing I for 1. The testing of K for 4 can be skipped in this case and control can pass to 10 immediately.

Note further, that any function called during the evaluation of an expression should not modify any values used elsewhere in the expression since the order of evaluation of the operands of an expression is not defined. The results of such misuse may differ from machine to machine, or even depend on the optimization level employed. The only exception is that a function will not be called until its actual arguments have been evaluated. This can be relied upon.

5.6 CONSTANT EXPRESSIONS

A constant expression is one of the following:

- an arithmetic constant expression (see Section 5.1.2 on page 92)
- a character constant expression (see Section 5.2.1 on page 95)
- a logical constant expression (see Section 5.4.1 on page 103)

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 6

ARRAY EXPRESSIONS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

An array expression is formed of operands, operators and parentheses. This chapter describes the formation, interpretation, and evaluation rules for the various types of array expressions. These may be:

- Arithmetic
- Relational
- Logical

All arrays that form an array expression must have only one dimension and be of the same length. An array expression results in an array.

6.1 ARITHMETIC ARRAY EXPRESSIONS

An arithmetic array expression is any set of arithmetic array operands combined with arithmetic array operators and parentheses. An arithmetic array operand may be either an arithmetic array or another arithmetic array expression.

Examples:

A

A + B

A + (B-C)

where *A*, *B* and *C* are arithmetic arrays.

Scanned by Jonny Oddene for Sintran Data © 2011

The arithmetic array operators are:

OPERATOR	MEANING
/	Division
*	Multiplication
-	Subtraction (or negation)
+	Addition

All the above operators are binary, i.e., used with two operands. The - and the + are also available as unary, i.e., used with only one operand.

There is a precedence among the arithmetic array operators which determines the order in which the operands are to be combined (unless the order is changed by the use of parentheses) as follows:

OPERATOR	PRECEDENCE
* and /	highest
+ and -	lowest (unary and binary)

Within each precedence level, the order is assumed to be from left to right.

The arithmetic array operands are:

- arithmetic arrays
- arithmetic expressions enclosed in parentheses.
- or any of the above operands combined by arithmetic array operators to form arithmetic array expressions.

Scanned by Jonny Oddene for Sintran Data © 2011

Examples:

If A and B are arrays of the same length and x is a variable:

A + x will add x to each element of A

*A + x*2* will add x*2 to each element of A

A + B will add A to B, one element from each at a time.

6.1.1 Interpretation of Results for Arithmetic Array Expressions

When the operator + or - operates on a single operand, the data type of the result is the same as that of the operand.

When an arithmetic array operator operates on a pair of operands, then the type of the result is as follows:

If the types of the two operands are the same then the data type of the result will be the data type of these operands.

If the types of the two operands are different, the operand of lower data type (see below) is converted to the data type of the other operand. Thus, the higher order data type will be that of the result.

Scanned by Jonny Oddene for Sintran Data © 2011

The hierarchical order of the data types is:

DATA TYPE	*ORDER*
integer	*lowest*
real	
double precision	
complex	*highest*

Note that the conversion takes place before the operation is performed, and that the operators are defined only for operands of equivalent type. The conversions are defined by the INTRINSIC functions REAL, DBLE, and CMPLX, see Section 11.2 on page 244.

Scanned by Jonny Oddene for Sintran Data © 2011

6.2 CHARACTER ARRAY EXPRESSIONS

A character array expression is an expression where one or more operands is an array. A character array expression is evaluated on an element by element basis.

The character array operator is:

//

which represents concatenation.

The result of 'AB' // 'CD' is 'ABCD'.

If a character array is of unknown length, then there are certain restrictions on its use in character array expressions, in that it can be used only in character assignment statements, and even then, only when it directly forms part of the final result.

Example:

If we have the following declarations:

```
SUBROUTINE SR(C)
CHARACTER C(10)*(*), A(10)*100,  B(10)*10
```

then C is of unknown length, i.e. its length is taken from the actual parameter.

You are allowed to write:

```
A = C//B
```

because the final result length is constrained by the length of A.

Scanned by Jonny Oddene for Sintran Data © 2011

But you cannot write:

CALL X(C//B)

because the actual argument is an expression whose length is not constrained.

Scanned by Jonny Oddene for Sintran Data © 2011

6.3 RELATIONAL ARRAY EXPRESSIONS

A relational array expression is an expression where at least one of the operands is an array expression. The values are compared on an element by element basis.

It is also possible to compare logical array expressions for equality or non-equality.

The relational array operators are:

OPERATOR	*MEANING*
.EQ.	*Equal to*
.NE.	*Not equal to*

6.3.1 Arithmetic Relational Array Expressions

The form of an arithmetic relational array expression is:

e_1 *rel* e_2

where *rel* is a relational operator.

e_1, e_2 are each an integer, real, double precision, numeric, complex array expression or other expression of same type, but at least one must be an array expression.

Before the comparison is carried out, the operands are converted, if necessary, to make them have the same type. The rules are the same as those for the common arithmetic operators, see Section 5.1. on page 86.

Scanned by Jonny Oddene for Sintran Data © 2011

6.3.2 CHARACTER Relational Array Expressions

Character relational array expressions have the form:

e_1 *rel* e_2

where *rel* is a relational operator.

e_1, e_2 are character expressions or character arrays, but at least one must be a character array.

If the element lengths e_1 and e_2 are unequal, then for comparison purposes the shorter string is extended to the right and filled with blanks.

6.3.3 LOGICAL Relational Array Expressions

Logical quantities can be compared by .EQ. and .NE.. These operators have the usual precedence of relational operators and perform the usual function.

Scanned by Jonny Oddene for Sintran Data © 2011

6.4 LOGICAL ARRAY EXPRESSIONS

A logical array expression is an expression where at least one of the operands is an array expression. The logical array expression is evaluated on an element by element basis.

A logical array expression is any set of logical array operands combined with logical array operators and parentheses. A logical array operand may be a logical array, a relational array expression or a logical array expression.

The logical array operators are:

OPERATOR	*MEANING*
.NOT.	*logical negation*
.AND.	*logical conjunction*
.OR.	*inclusive or*
.EQV.	*logical equivalence*
.NEQV.	*logical non-equivalence (exclusive or)*

The operator .NOT. is unary, i.e., used with one operand. The other operators are binary, i.e., used with two operands.

The logical array operators have a precedence order, i.e., the order in which operands are to be evaluated, unless this is changed by the use of parentheses.

OPERATOR	*PRECEDENCE*
.NOT.	*highest*
.AND.	
.OR.	
.EQV. or .NEQV.	*lowest*

Scanned by Jonny Oddene for Sintran Data © 2011

The logical array operands are:

- logical arrays
- relational array expressions
- logical expressions enclosed in parentheses

or any of the above operands combined by logical operators to form logical array expressions.

The data type of the result of an operator which returns a logical result is LOGICAL.

The result is the default LOGICAL data type, i.e. LOGICAL*2 for the ND-100 and LOGICAL*4 for the ND-500. The default can be changed by the DEFAULT command, see Chapter 3 on the ND FORTRAN User Guide, ND-60.265. Automatic conversion between the LOGICAL types occurs on assignment. If an operator has operands of different types, conversion according to Section 3.4 on page 58 will be done before the operation takes place.

If the operands are arithmetic, the normal conversions and precedence rules apply, see Section 5.1. on page 86.

The logical operators can also be used on type INTEGER, i.e., they treat the integer value as a string of bits, operating on corresponding bits. A bit whose value is 1 is interpreted as .TRUE. and a bit whose value is 0 as .FALSE.

Scanned by Jonny Oddene for Sintran Data © 2011

6.5 EVALUATION OF EXPRESSIONS

This section applies to arithmetic, relational, and logical array expressions. The order of evaluation of expressions is determined by:

- The use of parentheses
- The established precedence among the various operators
- The location of operators within an array expression

6.5.1 The Use of Parentheses

Array expressions within parentheses are evaluated first. Where parenthetical array expressions are nested (one contained within another), the innermost array expression is evaluated first followed by the next innermost, and so on, until the outermost parenthetical array expression has also been evaluated. If more than one operator is contained in an array expression within parentheses, the computation proceeds according to the precedence rules for the operators.

6.5.2 Precedence of Operators

The hierarchy of precedence among the arithmetic array operators, see Section 5.1 on page 86, and logical array operators, see Section 5.4 on page 99, has already been discussed. No precedence has been established among the relational array operators. Precedence among the various types is as follows:

OPERATOR	*PRECEDENCE*
Arithmetic	*Highest*
Relational	
Logical	*Lowest*

Scanned by Jonny Oddene for Sintran Data © 2011

An array expression may contain more than one kind of array operator, for example, the logical array expression:

```
L .OR. A+B .NE. C
```

where *A*, *B*, and *C* are arrays of type real and *L* is an array of type logical, contains an arithmetic array operator, a relational array operator, and a logical array operator. This array expression would be interpreted in the same way as:

```
L .OR. [(A+B) .NE. C]
```

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 7

ASSIGNMENT STATEMENTS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

Execution of an assignment statement causes a specific value to be given to one or more variables and/or array elements.

In ND FORTRAN it is also possible to assign an array expression or other expressions to an array. These arrays must have only one dimension and the same size.

There are four kinds of assignment statements:

- Arithmetic
- Logical
- Statement Label (ASSIGN)
- Character

7.1 ARITHMETIC ASSIGNMENT STATEMENT

The form of an arithmetic assignment statement is:

```
v = e
```

where

v is the name of a variable array or array element of type INTEGER, REAL, DOUBLE PRECISION, COMPLEX or NUMERIC.

e is an arithmetic expression.

If *v* is an array, then *e* may also be an arithmetic array expression.

Scanned by Jonny Oddene for Sintran Data © 2011

Examples:

A, B and C are arrays of the same size:

```
A = 0 A = B + C
```

Upon execution of an arithmetic assignment statement, the expression e is evaluated according to the rules in Section 5.5 on page 103, it is then converted to the type of *v*, with the resultant value being assigned to *v* according to the rules given in the table below:

TYPE OF V	*VALUE ASSIGNED*
Integer	*INT (e)*
Real	*REAL (e)*
Double Precision	*DBLE (e)*
Complex	*CMPLX (e)*

where the functions in the VALUE ASSIGNED column are INTRINSIC functions described in the table in Section 11.2 on page 244.

Scanned by Jonny Oddene for Sintran Data © 2011

7.2 LOGICAL ASSIGNMENT STATEMENT

The form of a logical assignment statement is:

```
v = e
```

where

v is the name of a logical variable, logical array or logical array element.

e is a logical expression

If *v* is an array then *e* may also be a logical array expression.

Upon execution of a logical assignment statement, the expression *e* is evaluated and its resultant value is assigned to *v*. *e* must have a value of either true or false.

Scanned by Jonny Oddene for Sintran Data © 2011

7.3 STATEMENT LABEL ASSIGNMENT (ASSIGN) STATEMENT

The form of a statement label assignment statement is:

```
ASSIGN s TO i
```

where

s is a statement label.

i is an integer variable name.

Execution of an ASSIGN statement causes *s* to be assigned to *i*. *s* must be the label of a statement appearing in the same program unit as the ASSIGN statement, and it must also be the label of an executable statement or a FORMAT statement.

Execution of a statement label assignment statement is the only way that a variable may be given a statement label value.

A variable must be defined with a statement label value when referenced in an assigned GO TO statement or as a format identifier in an I/O statement. While possessing a statement label value, the variable must not be referenced in any other way. An integer variable may be subsequently redefined with the same or a different statement label value or with an integer value.

In ND FORTRAN, *i* must be INTEGER*4 on ND-500 and INTEGER*2 on ND-100.

Scanned by Jonny Oddene for Sintran Data © 2011

7.4 CHARACTER ASSIGNMENT STATEMENT

The form of a character assignment statement is:

```
v = e
```

where

v is the name of a CHARACTER variable, CHARACTER array, CHARACTER array element, or a CHARACTER substring.

e is a CHARACTER expression.

If *v* is an array, then *e* may also be a CHARACTER array.

Execution of a character assignment statement causes the expression *e* to be evaluated, and the result assigned to *v*. If any of the character positions defined by *v* are referenced in *e*, the results are undefined. *v* and *e* may have different lengths. If the length of *v* is greater than the length of *e*, then the effect is to extend to the right with the blank characters until it has the same length as *v*. If *v* is shorter than *e* then *e* is truncated from the right until its length equals that of *v*.

Example:

```
CHARACTER A*2, B*4  A = B
```

only the substring B(1:2) must be defined.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 8

CONTROL STATEMENTS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

Control Statements enable the normal sequence of statement execution to be altered. There are sixteen control statements.

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO
- Arithmetic IF
- Logical IF
- Block IF
- ELSEIF
- ELSE
- ENDIF
- DO
- CONTINUE
- STOP
- PAUSE
- END
- CALL
- RETURN

The CALL and RETURN statements are described in Sections 11.5.1 on page 265 and 11.7 on page 270 respectively.

Scanned by Jonny Oddene for Sintran Data © 2011

8.1 UNCONDITIONAL GO TO STATEMENT

The unconditional GO TO statement transfers control of the program to the statement specified. It has the form:

```
GO TO s
```

where

s is the statement label of an executable statement appearing in the same program unit as the unconditional GO TO statement.

On execution, control is transferred so that the statement identified by the statement label is executed next.

Scanned by Jonny Oddene for Sintran Data © 2011

8.2 COMPUTED GO TO STATEMENT

A computed GO TO statement has the form:

```
GO TO (s [,s] ...) [,]i
```

where

s is the statement label of an executable statement appearing in the same program unit as the computed GO TO statement. The same statement label may appear more than once in the same computed GO TO statement.

i is an integer expression.

NOTE:
Although the ANSI FORTRAN 77 standard states that *i* should be of the above type, in ND FORTRAN, *i* can be any arithmetic expression that can be converted to type integer.

Execution of a computed GO TO statement causes a transfer of control to the statement having the *i*th statement label in the list of statement labels. This will only occur if $1<i<n$ where n is the number of labels in the list. If *i* is outside this range, the execution sequence is as if a CONTINUE statement were executed, i.e. control passes to the statement immediately following the computed GO TO.

Scanned by Jonny Oddene for Sintran Data © 2011

Example:

```
      INTEGER RECTYP
C Read next record on file. Sets rectyp to integer code
  5   CALL INPUT
C Decide what to do by looking at type of record in rectyp
      GO TO (10, 20, 30, 30, 50), RECTYP
C Error as record type has invalid value
      CALL INVALR
      GO TO 5
C Rectyp is 1 - good employees are paid
  10  CALL PAY
      GO TO 5
C Rectyp is 2 - he gets a rise
  20  CALL UPPAY
      GO TO 5
C Rectyp is 3 or 4 - change name or address etc.
  30  CALL UPDAT
      GO TO 5
C Rectyp is 5 - he is fired
  50  CALL DELETE
      GO TO 5
```

Scanned by Jonny Oddene for Sintran Data © 2011

8.3 ASSIGNED GO TO STATEMENT

The form of this statement is:

```
GO TO i [[,] (s[,s]...)]
```

where

s is the statement label of an executable statement appearing in the same program unit as the assigned GO TO statement. The same statement label may occur more than once in the same assigned GO TO statement.

i is an integer variable name.

At the time of execution of this statement, *i* must have the value of a statement label appearing in the same program unit. Assigned GO TO statements must be logically preceded by an ASSIGN statement, within the same program unit, which will set the value of *i*. Execution of the assigned GO TO statement then transfers control so that the statement identified by *i* is executed next.

If the parenthesized list is present then the statement label assigned to *i* must be one of those in the list.

Scanned by Jonny Oddene for Sintran Data © 2011

Example:

```
  5   CALL INPUT
C Normal case
      ASSIGN 10 TO KLAB
C See if it could be a small one
      IF (AREA.LT.100..AND.WIDTH.LT.10.) ASSIGN 20 TO KLAB
C Perhaps it is large
      IF (AREA.GT.10E4.OR.WIDTH.GT.100.) ASSIGN 30 TO KLAB
C Decide how to process
      5000 GO TO KLAB, (10, 20, 30)
C Normal
  10  CALL NORM
      GO TO 5
C Small case
  20  CALL SMALL
      GO TO 5
C Large case
  30  CALL LARGE
      GO TO 5
```

The statement labeled 5000 could also have been written as:

GO TO KLAB (10, 20, 30)

or,

GO TO KLAB

or,

GO TO KLAB (10, 20, 30, 5000, 5)

If the list is given, then all of 10, 20, and 30 must be included since otherwise the compiler may generate incorrect code. It relies on the list to determine the possible flow of control from this point in the program. The best code will result when the list is exactly correct, so that it does not include any labels that cannot be reached.

Scanned by Jonny Oddene for Sintran Data © 2011

8.4 ARITHMETIC IF STATEMENT

This statement has the form:

```
IF (e) s1 , s2 , s3
```

where

e is an integer, real, numeric or double precision expression.

s_1, s_2, s_3 are each the statement label of an executable statement in the same program unit as the arithmetic IF statement. The same statement label may appear more than once in the group of statement labels.

Execution of the arithmetic IF statement causes evaluation of the expression *e*, followed by a transfer of control. One of the statements identified by s_1, s_2 or s_3 is executed next; which one depends on whether the value of *e* is less than zero, equal to zero, or greater than zero respectively.

Example:

```
C Check to see if it will fit
      IF (SIZE - LIMIT) 30, 20, 10
C Size > limit, so it will not fit
  10   CALL ERROR
C Exactly at limit - issue warning
  20   CALL WARN
C Fits easily - process it
  30   CALL PROCESS
```

Scanned by Jonny Oddene for Sintran Data © 2011

8.5 LOGICAL IF STATEMENT

The form of this statement is:

```
IF (e) sta
```

where

e is a logical expression.

sta is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement.

Execution of this statement causes evaluation of the expression *e*. If the value of e is true then statement *sta* is executed.

If the value of *e* is false, the statement *sta* is not executed. Program execution then proceeds as if a CONTINUE statement were executed, i.e. control passes to the statement immediately following the logical IF.

Example:

```
C If debugging, write intermediate values
      IF (DBUG) WRITE (1,*) ALPHA, VAL, I
C If it is negative, cannot continue
      IF (RESULT .LT.0) STOP 16
C Find first element in the range -1 to +1
      DO 10 I = 1, N
      IF (A(I).GE. -1 .AND.A(I).LE+1) GO TO 20
  10   CONTINUE
  20   CALL PROC (A(I))
```

Scanned by Jonny Oddene for Sintran Data © 2011

8.6 THE BLOCK IF, ELSEIF, ELSE, AND ENDIF STATEMENTS

These statements are used to control the execution sequence. The block IF statement and its corresponding ENDIF statement forms a single unit. The ELSEIF and ELSE statements may be optionally combined with the block IF and ENDIF statements to provide alternative paths for the sequence of execution.

The form of a block IF is:

```
IF (e) THEN
    .
    .
    .
ENDIF
```

where

e is a logical expression.

Upon execution of a block IF statement, the expression *e* is evaluated. If the value of *e* is true, the execution sequence continues with the next executable statement following the block IF statement. Statements between the next (if any) ELSEIF or ELSE statement and the corresponding ENDIF will not then be executed. If false, control is transferred to the next ELSEIF or ELSE statements, if any, or to the ENDIF statement corresponding to the block IF statement.

8.6.1 The ELSEIF Statement

The form of an ELSEIF statement is:

```
ELSEIF (e) THEN
```

where

e is a logical expression.

Scanned by Jonny Oddene for Sintran Data © 2011

Execution of this statement causes *e* to be evaluated. If the value of *e* is true, then the execution sequence continues with the next executable statement following the ELSEIF statement. Again, statements between the next ELSEIF or ELSE statements, if any, and the ENDIF statement of this unit, will not then be executed. If there are no executable statements between this statement and the next ELSE IF, ELSE, or ENDIF statements, then control will be transferred to the ENDIF statement. If the value of *e* is false, control is transferred to the next ELSEIF, ELSE, or ENDIF statement of this unit.

8.6.2 The ELSE Statement

The form of an ELSE statement is:

The execution of an ELSE statement has no effect. The ELSE statement shows where control passes to if all expressions in the IF and ELSEIF statements in this unit are false, see note on statement labels at the end of the next section.

Scanned by Jonny Oddene for Sintran Data © 2011

8.6.3 The ENDIF Statement

This statement has the form:

```
ENDIF
```

Execution of an ENDIF statement has no effect. For each block IF statement there must be a corresponding ENDIF statement in the same program unit.

> In ND FORTRAN, statement labels on ELSEIF and ELSE statements can be referenced. A GO TO statement will transfer control to a point immediately prior to the evaluation of *e* in ELSEIF statements.

8.6.4 Examples of Block IF, ELSEIF, ELSE and ENDIF Statements

- ```
  C Test for fit on a page
      IF (CURLIN + N.GT.LINPAG) THEN
          CALL NEWPAG
          CURLIN = 0
      ENDIF
  ```

- ```
  C Adjust payment
      IF (TAXED) THEN
          NET = GROSS-TAX (GROSS)
      ELSE
          NET = GROSS
      ENDIF
  ```

Scanned by Jonny Oddene for Sintran Data © 2011

```
C Compute area of figure
    IF (N .EQ. 3) THEN
        S = (A + B + C) /2.0
        AREA = SQRT ((S-A)*(S-B)*(S-C)*S)
    ELSEIF (N .EQ. 4) THEN
        AREA = A*B
    ELSE
        AREA = PI*A**2
    ENDIF
```

```
C Check signatures
    IF (AMOUNT .GE. 10000) THEN
        IF (NSIG .NE. 2) THEN
            CALL NOGOOD
        ELSE
            CALL BIGCHK
        ENDIF
    ELSEIF (AMOUNT .GE. 100) THEN
            CALL MIDCHK
    ENDIF
C If passed, pay it
    IF (OK) THEN
        CALL PAYIT
    ELSE
        CALL ABORT
        WRITE(1,*)'ERROR IN CHEQUE',AMOUNT,INVOIC,NSIG,CNUM
    ENDIF
```

Scanned by Jonny Oddene for Sintran Data © 2011

As can be seen from the last example, block IF constructs can be nested. They may be nested to any depth, but each nested block IF must be wholly contained between:

- The IF ... THEN statement and the next occuring ELSEIF ... THEN, ELSE, or ENDIF statements of the next outermost block IF construct.

or,

- The ELSEIF ... THEN statement and the next occurring ELSEIF ... THEN, ELSE, or ENDIF statements of the next outermost block IF construct.

or,

- The ELSE statement and the next occurring ENDIF statement of the next outermost block IF construct.

Scanned by Jonny Oddene for Sintran Data © 2011

8.7 THE DO STATEMENT

A DO statement specifies a loop, called the DO-loop, which can be used for coding iterative procedures.

This statement has the form:

DO s [,] i = e_1 , e_2 [,e_3]

where

s is the statement label of an executable statement. This statement is called the terminal statement of the DO-loop and it must appear in the same program unit as the DO statement.

i is the name of an integer, real, or double-precision variable called the DO-variable.

e_1,e_2,e_3 are each an integer, real, or double-precision expressions

NOTE:

The terminal statement of a DO-loop must not be a control statement with the exception of logical IF, CONTINUE, PAUSE, or the CALL statement. If it is a logical IF statement, then this may contain any executable statement except DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement.

The label on the terminal statement is inside the loop. If several loops have the same terminal statement, then the label is in the innermost of the loops. Thus, program control can only jump to this label from within the innermost loop.

The range of a DO-loop is that of all executable statements following the relevant DO statement, up to and including the associated terminal statement.

A 'nested' DO statement, i.e., one whose range is contained entirely within the range of another DO statement, may have the same terminal statement as the outer DO-loop.

Scanned by Jonny Oddene for Sintran Data © 2011

If a block IF statement appears within the range of a DO-loop, its corresponding ENDIF statement must also do so.

This can be illustrated as follows:

```
C Find maximum and minimum values
      MX = 0
      MN = 0
      DO 10 I = 1, N
           IF (A(I) .GT. MX) THEN
              MX = A (I)
           ELSEIF (A(I) .LT. MN) THEN
                  MN = A (I)
           ENDIF
  10   CONTINUE
```

If a DO-loop appears within a block IF ... ENDIF unit, then the range of the DO-loop must be contained within the unit. Furthermore, it must be contained entirely between ELSEIF or ELSE statements, if any, and the next ELSEIF, ELSE or ENDIF statement in this block IF ... ENDIF unit.

For example:

```
C Get the sum of the elements of one of three different arrays
      X=0.0
      IF(M .LE. 0) THEN
C Sum elements of array "A"
        DO 10 I = 1,NELS
           X=X+A(NELS)
  10  CONTINUE
      ELSEIF(M .GE. 5) THEN
C Sum elements of array "B"
        DO 20 I = 1,NELS
           X=X+B(NELS)
  20  CONTINUE
        ELSE
C Sum elements of array "C"
         DO 30 I = 1,NELS
            X=X+C(NELS)
  30  CONTINUE
      ENDIF
```

Scanned by Jonny Oddene for Sintran Data © 2011

8.7.1 Execution of a DO Statement

A DO statement is executed in the following stages:

1. e_1, e_2, and e_3 are evaluated (including, if necessary, conversion to the type of the DO-variable). These values will be known from now on as the initial parameter, terminal parameter, and incremental parameter respectively. If e_3 does not appear, then the incremental parameter is given the value of one. (It must not be zero.)

2. The DO-variable, i, takes the value of the initial parameter.

3. The following test is performed to determine whether the loop should be terminated:

   If the incremental parameter >0, then the loop is terminated, if i> terminal parameter.

   If the incremental parameter <0, then the loop is terminated if i< terminal parameter.

   If the DO-loop is to be terminated, control passes to the next executable statement following the terminal statement or, if there is another DO-loop sharing its terminal statement with this one, then control passes to the incrementing stage for the next outer DO.

4. If the loop has not been terminated, the statements within the loop are executed.

5. At the end of the loop, the DO-variable is incremented by the value of the incremental parameter. (Note that if the incremental parameter <0, the DO-variable will, in fact, decrease.)

6. The loop control processing begins again at stage 3.

NOTE:

It is perfectly possible for the body of the loop not to be executed at all. This happens if the terminating conditions are satisfied on the first entry to loop control processing at stage 3.

Scanned by Jonny Oddene for Sintran Data © 2011

Examples:

- ```
  C Initialize array to zero
        DO 10 I = 1, N
  10        A(I) = 0
  ```

- ```
  C Copy upper diagonal to lower
        DO  20  I=2, N
            DO  10  J = I+1, N
             A (J, I) = A (I, J)
  10    CONTINUE
  20 CONTINUE
  ```

- ```
  C Find maximum values by rows
        DO 20 I = 1, N
           XMX (I) = A (I, 1)
           DO 20 J= 2, N
              IF (A(I,J) .GT. XMX (I)) XMX (I) = A(I,J)
  20    CONTINUE
  ```

- ```
  C The sieve of eratosthenes
        LOGICAL P(2 : 1000)
  C Initialize prime array
        DO 10 I = 2,1000
  10       P(I) = .TRUE.
  C Run through all candidates
        DO 30 I=2, SQRT (1000+1)
  C If it is a prime, then mark off all multiples
              IF (P(I)) THEN
                DO 20 K = 2*I, 1000, I
  20                 P(K) = .FALSE.
              ENDIF
  30    CONTINUE
  ```

Scanned by Jonny Oddene for Sintran Data © 2011

```
C Set diagonal to sum of row to the left
      DO 20 I = 1, N
         S = 0
         DO 10 K = 1, I
            S = S + A (I, K)
10       CONTINUE
C K now contains the final value, I-1, plus one increment,
C i.e. the value of I
      A (K, K) = S
20       CONTINUE
```

8.7.2 The DO FOR ... ENDDO Statements

In ND FORTRAN the DO statement can have the form:

```
DO [FOR] i = e1 , e2 [,e3]
```

where

i, e_1, e_2 and e_3 are each an integer, real or double-precision expression.

The end of the DO-loop is represented by the statement:

```
ENDDO
```

Scanned by Jonny Oddene for Sintran Data © 2011

Using the ND FORTRAN extension form of the DO statement, the last example of the previous section could be written as:

```
DO FOR I = 1, N
   S = 0
   DO FOR K = 1, I
      S = S + A (I, K)
   ENDDO
   A (K, K) = S
ENDDO
```

or,

```
DO I = 1, N
   S = 0
   DO K = 1, I
      S = S + A (I, K)
   ENDDO
      A (K, K) = S
ENDDO
```

8.7.3 The DO WHILE ... ENDDO Statements

In ND FORTRAN, there is a further iterative programming construct delimited by the DO WHILE and ENDDO statements. This takes the form:

```
DO WHILE (e)
   .
   .
   .
ENDDO
```

where

e is a logical expression.

Scanned by Jonny Oddene for Sintran Data © 2011

Upon execution of the DO WHILE statement, *e* is evaluated. If *e* is true, control passes to the next executable statement. If *e* is false, control passes to the next executable statement following the delimiting ENDDO statement.

When the ENDDO statement is executed, control is returned to its corresponding DO WHILE statement for re-evaluation of *e*.

This construct allows the natural programming of loops which terminate only when certain conditions are met, rather than a specific number of repetitions.

For example:

The DO WHILE ... ENDDO construct may be used to read a sequential file and process each and every record, in which case, the following loop could be constructed:

```
C Read first record
      CALL INPUT
C Test for end-of-file
      DO WHILE (.NOT. EFLAG)
C Process record
           CALL PROCS
C Read next record
           CALL INPUT
C Loop and re-test for end-of-file
      ENDDO
```

Note that the test is executed at the start of the loop. In this example, if there are no records in the file, the loop will not be executed.

Scanned by Jonny Oddene for Sintran Data © 2011

8.8 THE CONTINUE STATEMENT

The form of a CONTINUE statement is:

```
CONTINUE
```

This statement may appear anywhere within the program. Its execution has no effect and the statement is commonly used to provide a loop termination to avoid ending with a GO TO, STOP, PAUSE, RETURN, Arithmetic IF, another DO statement, or a Logical IF statement containing any of the these.

Scanned by Jonny Oddene for Sintran Data © 2011

8.9 THE STOP STATEMENT

This statement has the form:

```
STOP [n]
```

where

n is an integer constant of up to five digits (decimal) or a character constant.

In ND FORTRAN, *n* may be any integer expression.

Execution of a STOP statement causes termination of the executable program. At the time of termination the text, STOP *n* is printed out on the message output file, i.e. the user's terminal for background programs, and the system console for RT-programs.

When execution terminates, all files which have not been permanently opened are closed.

Example:

```
STOP 16
STOP 'CANNOT OPEN FILE' // FILENM
```

Scanned by Jonny Oddene for Sintran Data © 2011

8.10 THE PAUSE STATEMENT

The form of the PAUSE statement is:

PAUSE [n]

where

n is an integer constant of up to five digits (decimal) or a character constant.

In ND FORTRAN, *n* may be any integer expression.

Execution of a PAUSE statement suspends execution of the program and the text PAUSE *n* is printed on the message output file.

In ND FORTRAN, execution resumes when the program receives a carriage return from the SINTRAN logical device number 1, the user's terminal for background programs, and the system console for RT-programs. If execution is resumed, it is as if a CONTINUE statement had been executed, i.e. control passes to the statement immediately following the PAUSE.

Example:

```
PAUSE 224
PAUSE 'PLEASE MOUNT TAPE'
```

Scanned by Jonny Oddene for Sintran Data © 2011

8.11 THE END STATEMENT

The form of this statement is:

END

It is used to indicate that the end of the sequence of statements and comment lines of a program unit has been reached. If executed in a function or subroutine program, it has the effect of a RETURN statement; in a main program, it terminates execution of the executable program and hence causes all files to be closed.

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 9

INPUT/OUTPUT STATEMENTS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

9.1 I/O TERMS AND CONCEPTS

Input statements control the transfer of data from external media or from an internal file into internal storage. This process is called reading. Output statements control the transfer of data from internal storage to external media or to an internal file. This process is called writing.

In addition to data transfer statements, other statements perform file control, device control, or inquiry.

These are the input/output statements:

- READ
- WRITE
- PRINT
- OPEN
- CLOSE
- BACKSPACE
- ENDFILE
- REWIND
- INQUIRE

The READ, WRITE and PRINT statements are data transfer statements. The OPEN and CLOSE statements are file control statements. The BACKSPACE, ENDFILE and REWIND statements are device control statements. The INQUIRE statement performs file inquiry.

9.1.1 Records

A record is a sequence of values or characters which is considered as a single unit by the device it is being read from or written to. It may correspond to a physical entity, such as a punched

Scanned by Jonny Oddene for Sintran Data © 2011

card, but not necessarily. For instance, input from a terminal is separated into records by the return key.

There are three types of records:

- Formatted
- Unformatted
- Endfile

A FORMAT statement contains a set of format specifications defining the layout of a record and the form of the data fields within the record (see Chapter 10.1 on page 209, for a complete description of the FORMAT statement). Format specifications may also be stored in an array or variable of type CHARACTER rather than in a FORMAT statement.

A formatted record is one which is transferred under the control of a format specification as outlined above. Other records are unformatted records. During unformatted transfers, data is transferred on a one-to-one basis between external media (or internal files) and internal storage with no conversion or formatting operations involved.

An endfile record is written by using the ENDFILE statement. An endfile record may only occur as the last record of a file.

9.1.2 Files

A file is a sequence of records; it may be internal or external.

Internal files provide a means of transferring and converting data within internal storage. An internal file has the following properties:

- The file is a character variable, character array, character array element, or a character substring.
- A record of an internal file is a character variable, array element, or a substring.

Scanned by Jonny Oddene for Sintran Data © 2011

- If the file is a character variable, character array element, or character substring, it consists of a single record whose length is the same as the length of the variable, array element, or substring respectively.

- If the file is a character array, it is treated as a sequence of character array elements. Each array element is a record of the file. The ordering of the records of the file is the same as the ordering of the array elements in the array, see Section 2.4.2. on page 39. Every record of the file has the same length, which is the length of an array element in the array.

- If the number of characters written in a record is less than the length of the record, the remaining portion is filled with blanks.

- An internal file is always positioned at the beginning of the first record prior to data transfer.

An external file is a collection of records stored on an external storage medium, e.g., a disk.

9.1.2.1 File Format

An unformatted file consists of unformatted records, while a formatted file has formatted records as its components. Both types can have an end-file record, as the last record in the file.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, formatted files have records of a single length if the RECL = specifier is present in the OPEN statement. If the RECL = specifier is not used in the OPEN statement, then records may be of varying lengths.

If a formatted file is also a PRINT file, then the record layout is as described in Section 9.2.5.1. on page 178.

If a formatted file is not a print file, then each record is followed by the pair of characters (CR,LF). All these control characters must be included in the RECL count if it is specified.

9.1.2.2 File Access

For an external file there are two access methods, sequential and direct.

The method of accessing the file is determined when the file is connected to a unit. An internal file must be accessed sequentially, as must also magnetic tapes and character devices, i.e. terminals and internal devices.

SEQUENTIAL ACCESS

The order of the records on the file is the order in which they were written. Each I/O statement executed in sequential mode transfers the record immediately following the previous record transferred from the accessed source file.

The records of the file are either all formatted or all unformatted (except that the last record of the file may be an endfile record). A record that has not been written since the file was created must not be read.

Scanned by Jonny Oddene for Sintran Data © 2011

DIRECT ACCESS

All records of the file have the same length. They must be either all formatted or all unformatted.

Each record of the file is uniquely identified by a positive integer called the record number which is specified when the record is written. Once established, this number cannot be changed. Although a record may not be deleted it can, however, be rewritten.

The order of the records on the file is the order of their record number.

Records need not be read or written in the order of their record number. Any record may be written into the file while it is connected to a unit. For example, you may write record 3 even though records 1 and 2 have not been written. Any record may be read from the file provided that the record has been written.

See the OPEN statement, Section 9.3.1 on page 182, for further information on the ND FORTRAN implementation and extensions.

Scanned by Jonny Oddene for Sintran Data © 2011

9.1.3 Units

A UNIT is a means of referring to a file. A unit specifier has the form:

[*UNIT=*] *u*

where

u is an external unit identifier (to refer to external files) or an internal file identifier.

If the optional characters **UNIT** are omitted from the unit specifier then this specifier must be the first item in a list of specifiers.

An external unit identifier can be:

- A positive or zero integer expression
- An asterisk, identifying a particular unit that is preconnected for formatted sequential access, see Chapter 3 on the ND FORTRAN User Guide, ND-60.265.

In the example:

```
SUBROUTINE A
READ (6) X
    .
    .
    .
SUBROUTINE B
N = 6
REWIND N
```

the value 6 used in both program units identifies the same external unit.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, the values that the external unit identifier may take are:

0	:	on output, this is a dummy unit and all output is lost; on input, data is taken from the command line of SINTRAN, i.e. one record terminated by a carriage return. This implies that the same editing possibilities used on SINTRAN commands can be used on input strings.
1	:	user's terminal (system console for RT execution).
2-127	:	available for defining files to be OPEN'ed in the FORTRAN program, otherwise taken as a SINTRAN logical device number.
128-32767	:	a SINTRAN logical device number.

An internal file identifier is the name of a character variable, character array, character array element, or character substring.

In ND FORTRAN, non-character arrays may also be used.

Internal files provide a means of transferring and converting data within internal storage.

9.1.4 Format Specifier and Identifier

A format specifier has the form:

[*FMT* =] *f*

where

f is a format identifier.

If the optional characters **FMT** are omitted then the format specifier must be the second item in a list of specifiers. In this case the first item must be a unit specifier without the optional

Scanned by Jonny Oddene for Sintran Data © 2011

characters UNIT=.

A format identifier identifies the format type, see Chapter 10 on page 209, and it must be one of the following:

- FORMAT statement label in the current program unit
- The name of an array containing the format specifications
- Any CHARACTER expression, except a CHARACTER expression involving concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is a symbolic name of a constant
- An asterisk, implying list-directed formatting
- An integer variable name that has been assigned the statement label of a FORMAT statement that appears in the same program unit as the format identifier

---

9.1.5 End-of-File Specifier

An end-of-file specifier has the following form:

```
END = label
```

where

label is a statement label appearing in the current program unit.

If a READ statement (see Section 9.2.4 on page 174) contains an end-of-file specifier and an end-of-file condition but no error condition is encountered during its execution, then the following will result:

- Execution of the READ statement terminates.

Scanned by Jonny Oddene for Sintran Data © 2011

- If the READ statement contains an I/O status specifier, this will be set as specified in Section 9.1.7. on page 166.
- Execution continues with the statement having the designated label.

Example:

```
READ ( 10, 5, END = 70) TABLE I, J, K
```

Detection of an end-of-file condition during execution of this statement causes transfer of control to statement 70. All items in the input list, following the detection, of an end-of-file condition, and all implied DO indices on input lists will have unpredictable values.

An end-of-file condition will occur if an endfile record is encountered during the reading of a file connected for sequential access.

9.1.6 Error Specifier

The form of an error specifier is:

ERR = label

where

label is a statement label appearing in the current program unit.

Scanned by Jonny Oddene for Sintran Data © 2011

If an error condition occurs during execution of an I/O statement containing an error specifier the following will result:

- Execution of the I/O statement terminates.
- The position of the file pointer specified in the statement becomes undefined.
- If the statement contains an I/O status specifier, this will be set as specified below.
- Execution continues with the statement having the designated label.

9.1.7 Input/Output Status Specifier

The form of an input/output status specifier is:

```
IOSTAT = s
```

where

s is a variable or array element of integer type.

Execution of an I/O statement containing this specifier causes it to be set as follows:

zero - if neither an error nor an end-of-file condition is encountered.

positive number - when an error condition occurs.

negative number - when an end-of-file but no error condition occurs.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, on the ND-100, *s* must be of type INTEGER*2. On the ND-500, *s* must be of type INTEGER*4. The error codes stored in *s* are the standard FORTRAN/SINTRAN/500 Loader/Monitor error codes. They are listed in Appendix D of this manual, in Appendix C of the SINTRAN III Reference Manual (ND-60.128) and 500 Loader/Monitor Manual (ND-60.136).

If an error condition occurs, and there is no ERR= specified, (or an end-of-file condition and no END=) and no IOSTAT= specified, then the program is aborted.

NOTE:

In routines compiled with STANDARD-CHECK set OFF (see Chapter 3 on the ND FORTRAN User Guide, ND-60.265, there is a reserved variable called ERRCODE which takes the absolute value of IOSTAT after the execution of an I/O statement. This ensures compatibility with the previous FORTRAN compiler.

In routines compiled with STANDARD-CHECK ON, the name ERRCODE is not reserved and is treated like any other variable.

In ND FORTRAN when the end-of-file condition is encountered, IOSTAT will take the value 3 with the sign bit set (i.e. 100003B on the ND-100 and 20000000003B on the ND-500), but ERRCODE will be set to +3.

9.1.8 Record Specifier

A record specifier has the following form:

```
REC = rn
```

where

rn is an integer expression whose value is positive. It specifies the number of the record to be read or written in a file connected for direct access.

Scanned by Jonny Oddene for Sintran Data © 2011

9.2 DATA TRANSFER OPERATIONS

Data transfer is the function of the I/O statements READ, WRITE and PRINT. The transfer of data occurs between storage and peripheral devices and/or between storage locations.

The storage locations are identified by an input/output list.

The type and format of external data (on input or output) may be controlled by using format specifications.

9.2.1 Input/Output Lists

An I/O list specifies the names of the variables, arrays, array elements, or character substrings to which input data is to be assigned or from which output data is to be obtained.

The list is processed one item at a time, the transfer of each item is completed before it is started for the next.

Example:

Suppose N is an integer and A is a one-dimensional array of type REAL, then the code:

```
N = 3
READ (5) N, A (N)
```

means that the value in the input stream on unit 5 is assigned to N. Suppose this value is 10. The next value on the input stream is assigned to the element A(10). Note that the most recently read value of N is used.

Implied DO lists (described below) which specify sets of array elements, may also be included in I/O lists.

Scanned by Jonny Oddene for Sintran Data © 2011

9.2.1.1 Implied DO Lists

When an array name appears in an I/O list, all elements of the array are transferred in the order in which they are stored, see Section 2.4.2. on page 39. Specific sets of array elements may be specified in the I/O list either individually or in the form of an implied DO list.

The implied DO takes the same general form as that of a DO statement:

```
(iolist , I = e1, e2 [,e3])
```

where

iolist is an I/O list which may contain further implied DO lists to an arbitrary depth of nesting.

I the index control variable representing a subscript appearing in the subscript list.

e1, e2, e3 are the indexing parameters specifying the initial, terminal and incremental values controlling the range of I (*e1, e2, e3* are each an INTEGER, REAL or DOUBLE PRECISION expression). If *e3* does not appear, its value defaults to 1 (one).

Example:

```
      REAL A(2,3)
  10  FORMAT (6F10.3)
      READ (1,10) A
```

The READ statement will read A in the following order:

A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3) i.e.

first subscript varies most rapidly.

The same effect is achieved by the following statement:

READ (1,10) ((A(I,J), I = 1,2), J = 1,3) i.e. the

innermost loop varies most rapidly.

Scanned by Jonny Oddene for Sintran Data © 2011

If you need to vary the other subscript most often, use the following form: *READ (1,10) ((A (I,J), J = 1,3), I = 1,2)*

9.2.2 Formatted and Unformatted Data Transfer

I/O statements which include format specifications enable the user to convert the data being transferred into a different form. This may be required on output, for example, to make the data easier to read.

During formatted data transfer, data is transferred with editing between the items specified by the I/O list and the file. The record at the current position and possibly additional records are read or written. The editing between the internal representation and the character strings of a record, or sequence of records, is directed by a format specification. This specification may be contained in a FORMAT statement or in an array. If the format identifier is an * (asterisk), this indicates list-directed input/output, see the next section.

Unformatted data transfer is used for intermediate files for internal use on disk and tape units. During unformatted data transfer, data is transferred without editing between the current record and the items specified by the I/O list. Exactly one record is read or written.

9.2.3 List-Directed Input/Output

If the format identifier contained in an I/O statement is an asterisk, this causes the transfer operation to be list-directed. List-directed input/output may also be called free-format.

Scanned by Jonny Oddene for Sintran Data © 2011

Note:

In this case, a record specifier must not be present.

Data for list-directed transfers should consist of alternate constants and delimiters. Delimiters may be one of the following:

- A comma optionally preceded or followed by one or more blanks.
- A slash, optionally preceded or followed by one or more blanks.
- One or more blanks between two constants (or following the last constant).

9.2.3.1 List-Directed Input

The form of the input value must be acceptable for the type of the input list item. Values which are consistent with format specifications (see Chapter 10, on page 209), are also acceptable in list-directed input except in the following cases:

- When the list item is of type REAL or DOUBLE PRECISION, the corresponding input form should be numeric and suitable for F editing, see Section 10.2.2.4. on page 217.
- For list items of type CHARACTER, the corresponding input constants should be enclosed in single quotes, i.e. 'ABC'. Each quote within a CHARACTER constant must be represented by two consecutive quotes. The constant may be continued over as many records as needed. The characters blank, comma, and slash, which are otherwise delimiters, may appear in CHARACTER constants. If the lengths of the list item and CHARACTER constant differ, the result is as for the CHARACTER assignment statement, Section 7.4. on page 127.

Scanned by Jonny Oddene for Sintran Data © 2011

- When the corresponding list item is of type COMPLEX, the pair of constants being input must be enclosed in parentheses and separated from each other by a comma. Each constant should be numeric as in the first rule above.

In ND FORTRAN, parentheses are not required. A pair of constants can be separated by spaces instead of comma.

- Null values on input are represented by two consecutive commas with no intervening constant(s). If a null value appears in the data, its corresponding list element will retain its old value and definition status.

- When all the items in the I/O list have been assigned, any remaining input data is ignored.

- A slash encountered in the input stream causes the current input statement to terminate. Any remaining items in the I/O list will retain their old values and definition status.

- The input values for List Directed Input can contain repetition groups of the form :

*v*c or v**

where

v is the repetition factor.

c is a constant.

For example:

*3*2.7, 2* , 2*'ABC'*

which is the same as:

2.7, 2.7, 2.7, , ,'ABC','ABC'

Scanned by Jonny Oddene for Sintran Data © 2011

Note:

Blanks are never used as zeros, and embedded blanks are not permitted in constants except within CHARACTER constants as described in the second point in the list above.

9.2.3.2 List-Directed Output

The form of the values produced is the same as that for input except in the cases of CHARACTER constants given here. The values are separated by one or more blanks.

CHARACTER constants are not delimited by apostrophes on output.

Each output record begins with a blank character to provide carriage control when the record is printed.

If successive values are identical, no replication factors are employed.

The internal values are converted on output according to the formats:

I16	*INTEGER*
E16.7	*REAL*
D16.9	*DOUBLE PRECISION*
2E16.7	*COMPLEX*
A	*CHARACTER*
L16	*LOGICAL*
D16.9	*NUMERIC*

Scanned by Jonny Oddene for Sintran Data © 2011

9.2.4 The READ Statement

The READ statement causes data to be transferred from external media to internal storage, or from an internal file to internal storage. The forms of the READ statement are as follows:

UNFORMATTED READ

- Form:

```
READ (u[,arglist])[iolist]
```

Note:

The form READ (u) will cause one unformatted input record to be skipped.

FORMATTED READ

- Form 1:

```
READ f [,iolist]
```

Note:

This statement reads from the standard input device which can be set in the UNIT command, see the ND FORTRAN User Guide, ND-60.265.

Scanned by Jonny Oddene for Sintran Data © 2011

- Form 2:

```
READ (u,f[,arglist])[iolist]
```

- Form 3 (List-directed):

```
READ (u,*[,arglist])[iolist]
```

where

u is a unit specifier (see Section 9.1.3 on page 162)

f is the format specifier (see Section 9.1.4 on page 163)

iolist, when present, is an input list specifying the data items whose values are to be transferred. A data item in an input list must be one of the following:

- a variable
- an array
- an array element
- a character substring

arglist is a list of optional items, separated by commas, in which each of the following items may appear no more than once:

REC=rn (see Section 9.1.8, on page 167)

IOSTAT=s (see Section 9.1.7, on page 166)

ERR=label (see Section 9.1.6, on page 165)

END=label (see Section 9.1.5, on page 164)

If *arglist* contains a record specifier, the file should be opened for direct access.
arglist cannot contain both a record specifier and an end-of-file specifier.

Scanned by Jonny Oddene for Sintran Data © 2011

If the format identifier is an asterisk, the statement is a list-directed input statement and a record specifier must not be present.

In ND FORTRAN, the record specifier can be used with sequential access to reposition the file for READ or WRITE statements. Thus, the same part of the file can be read several times, and part of a file can be updated. For such an operation to be possible, the file must reside on a direct access device.

Also, in such a case it is permitted to specify both END= and REC= in the same arglist.

9.2.5 The WRITE Statement

The WRITE statement transfers data from internal storage to external media or from internal storage to internal files. The forms of the WRITE statement are as follows:

UNFORMATTED WRITE

- Form:

```
WRITE (u[,arglist])[iolist]
```

FORMATTED WRITE

- Form 1:

```
WRITE f[,iolist]
```

Scanned by Jonny Oddene for Sintran Data © 2011

- Form 2:

```
WRITE (u,f[,arglist])[iolist]
```

- Form 3 (List-Directed):

```
WRITE (u,*[,arglist])[iolist]
```

where

u is a unit specifier (see Section 9.1.3 on page 162).

f is the format specifier (see Section 9.1.4 on page 163).

iolist, when present, is an output list identifying the data items whose values are to be transferred. A data item in an output list must be one of those:

- a variable
- an array
- an array element
- a character substring
- any other expressions except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is the symbolic name of a constant.

arglist is a list of optional items, separated by commas, in which each of the following items may appear no more than once:

REC=rn (See Section 9.1.8, on page 167)

IOSTAT=s (See Section 9.1.7, on page 166)

ERR=label (See Section 9.1.6, on page 165)

If *arglist* contains a record specifier, the statement is a direct access output statement (See the READ statement earlier). If not, it is a sequential access output statement.

Scanned by Jonny Oddene for Sintran Data © 2011

If the format identifier is an asterisk, the statement is a list-directed output statement and a record specifier is not allowed.

9.2.5.1 Printing of Formatted Records

The transfer of information in a formatted record to certain devices determined by the processor is called printing. If a formatted record is printed, the first character of the record is not printed. The remaining characters of the record, if any, are printed on one line beginning at the left margin.

The first character of such a record determines vertical spacing as follows:

CHARACTER	*VERTICAL SPACING BEFORE PRINTING*	*EXTERNAL OUTPUT*
Blank	*One line*	*LF record CR*
0	*Two lines*	*LF CR LF record CR*
1	*To first line of new page*	*FF LF record CR*
+	*No advance*	*record CR*
$	*No advance, CR suppressed*	*record*

Note that the character $ in the above table is an ND FORTRAN extension. It can be used, for example, when writing to a terminal. The print head or cursor will remain at the end of the line, thus enabling answers to questions to follow on the same line.

Any other character occurring in the first position is treated as a blank.

If there are no characters in the record, the vertical spacing is one line and no characters other than blank are printed on that line.

Scanned by Jonny Oddene for Sintran Data © 2011

A PRINT statement does not imply that printing will occur, and a WRITE statement does not imply that printing will not occur.

In ND-FORTRAN printing is used if the file is opened with ACCESS = 'PRINT' or 'PRINT-APPEND' or if the file is a PRINT file. The formatting must be different from List-Directed Formatting.

The following logical SINTRAN device numbers are PRINT files. All numbers are octal:

- Less than 100 except:

3	fast punch
20 and 21	cassette
25,33,40,41	magnetic tape controller 1
32,34	magnetic tape controller 2

- 200 to 277, 700 to 777, 1040 to 1077, 2000 to 2077 — see SINTRAN manual ND 60.128 for descriptions

- 100 to 127 — if they are spooling files

Scanned by Jonny Oddene for Sintran Data © 2011

9.2.6 The PRINT Statement

The PRINT statement causes data to be transferred from internal storage to the standard output device. This can be defined by the UNIT command, see the ND FORTRAN User Guide, ND-60.265. It is used only for sequential formatted data transfer. The PRINT statement takes the following forms:

- Form 1:

```
PRINT f[,iolist]
```

- Form 2:

```
PRINT*[,iolist]
```

where

f is the format specifier (see Section 9.1.4 on page 163).

iolist if present, is the output list identifying the data items whose values are to be transferred.

Scanned by Jonny Oddene for Sintran Data © 2011

9.2.7 The INPUT Statement

In ND FORTRAN, list-directed input can be specified by the statement:

```
INPUT (u[,arglist])[iolist]
```

This is exactly equivalent to the List-Directed form (Form 3) of the READ statement, see Section 9.2.4. on page 174.

9.2.8 The OUTPUT Statement

In ND FORTRAN, list-directed output can be specified by the statement:

```
OUTPUT (u[,arglist])[iolist]
```

This is exactly equivalent to the List-Directed form (Form 3) of the WRITE statement, see Section 9.2.5. on page 176.

Scanned by Jonny Oddene for Sintran Data © 2011

9.3 FILE OPEN AND CLOSE

This section covers connecting and disconnecting files, creating them, and establishing of parameters for I/O operations. The statements used for this are OPEN and CLOSE.

9.3.1 The OPEN Statement

The OPEN statement can connect an existing file to a unit, create a file that is preconnected, create a file and connect it to a unit, or change the specifiers of a connection between a file and a unit. It has the form:

```
OPEN (u[,arglist])
```

where

u is a unit specifier (see Section 9.1.3 on page 162.

arglist is a list of optional items, separated by commas, in which each of the following items appear no more than once:

IOSTAT	= s	(Section 9.1.7, page 166)
ERR	= label	(Section 9.1.6, page 165)
FILE	= file	
STATUS	= sta	
ACCESS	= acc	
FORM	= fm	
RECL	= rl	
BLANK	= bl	

Scanned by Jonny Oddene for Sintran Data © 2011

```
PARITY      = par    (ND FORTRAN Extension)
FIRSTREC    = value  (ND FORTRAN Extension)
FACTOR      = fac    (ND FORTRAN Extension)
IOCONVERT   = ioco   (ND FORTRAN Extension)
TYPE        = ty     (ND FORTRAN Extension)
MODE        = seg    (ND FORTRAN Extension)
BUFFER_SIZE = bs     (ND FORTRAN Extension)
```

If the form UNIT= is used for the unit specifier, it may appear anywhere in *arglist*. If UNIT= is omitted, *u* must be the first specifier in the list.

When executing as RT programs (ND-100 only), all programs on a particular segment must use different unit numbers. Care should be taken when opening and using logical devices shared among programs on the same segment. Otherwise, there are no restrictions on the I/O facilities available to RT-programs.

To make the most efficient use of the various I/O options, refer to Chapter 15, Advanced FORTRAN Programming in this manual.

The specifiers not previously described in **arglist** are described in the remainder of this section:

FILE = file

where

file is a character expression whose value is the name of the file acceptable to SINTRAN and is to be connected to the specified unit. The default file type is SYMB.

If no file is specified, the actual open monitor call is not executed, but the number must be within the range of legal unit numbers to OPEN (1-127), otherwise an error condition will occur.

Scanned by Jonny Oddene for Sintran Data © 2011

If a file is specified, the unit number is used in subsequent I/O statements to refer to this file. If this is the case, the unit number must be positive and less than 128.

STATUS = sta

where

sta is a character expression whose value is OLD, NEW, SCRATCH or UNKNOWN. If OLD is specified, the file must

exist; correspondingly, the file must not exist if NEW is specified. If the specifier is omitted, a value of UNKNOWN is assumed. If UNKNOWN is specified, the file is created if it does not exist.

Successful execution of an OPEN statement with NEW specified creates the file and changes the status to OLD.

If SCRATCH is specified with an unnamed file, the file is connected to the specified unit for use by the executable program. The file is deleted at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program.

In ND-FORTRAN named files are allowed to be used also if SCRATCH is specified. The file with specified name will be used (if it does not exist, it will be created). This file will be deleted by CLOSE statement or termination of the program.

Scanned by Jonny Oddene for Sintran Data © 2011

ACCESS = acc

where

acc is a character expression whose value is one of the following:

- SEQUENTIAL
- DIRECT

and this determines the access method for the connection of the file. The default is SEQUENTIAL. SEQUENTIAL or DIRECT access should be used if the file is to be accessed through FORTRAN READ/WRITE statements.

Note:

If DIRECT or SEQUENTIAL access is used, the monitor call (76) SETBS must not be used for this file.

In ND FORTRAN the following values are also acceptable:

W	Sequential output (WRITE statements)
R	Sequential input (READ statements)
WX	Random input or output (for RFILE/WFILE use)
RX	Random input (for RFILE use)
RW	Sequential input or output (READ/WRITE statements)

Scanned by Jonny Oddene for Sintran Data © 2011

WA	Sequential output appending to an existing file (WRITE statements)
WC	Random input or output to contiguous files (for RFILE/WFILE use)
RC	Random input from contiguous files (for RFILE use)
D	Random input or output direct (for RFILE/WFILE use)
DC	Random input or output direct closed (for RFILE/WFILE use, ND-500 only) NOTE: Direct closed means that such a file remains closed during all file transfers. FORTRAN does not set the maximum byte pointer for DC files.
READ	Random input (READ statements)
WRITE	Random input or output (READ/WRITE statements)
PRINT	Sequential output, first character interpreted as a forms control character (WRITE statement)
PRINT-APPEND	Sequential output appending to an existing file. First character interpreted as a form control character (WRITE statement).

SPECIAL for use of monitor calls (i.e. SETBS, RFILE, WFILE or MAGTP)

FORM = fm

where

fm is a character expression with the value FORMATTED or UNFORMATTED. The value determines whether the file is being connected for formatted or unformatted I/O. The default with direct access is UNFORMATTED, with sequential access it is FORMATTED.

Scanned by Jonny Oddene for Sintran Data © 2011

RECL = rl

where

rl is an integer value which must be positive. It gives the length in characters (bytes) of each record, in the file to be connected with DIRECT access.

In ND FORTRAN, the RECL= specifier may always be used, whatever the value of the ACCESS= specifier.

In ND FORTRAN, SEQUENTIAL access used in conjunction with the RECL= specifier, may be used with either FORMATTED or UNFORMATTED I/O. Thus, all records in a SEQUENTIAL file can be made the same size.

In FORMATTED files, data in a record is generally terminated by a CR LF (carriage return, line feed) pair of characters.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, the record delimiter CRLF is included in the record. This means that the record must be specified as being two characters longer than the number of characters to be transferred. By using the compiler command: RUNTIME-STANDARD-MODE, you can avoid including the delimiters in the record. In either case, the information in the file and the number of bytes in the file are exactly the same.

If a FORMATTED file has a RECL= specifier in the OPEN statement, and data to be output is shorter than the specified record length, the record is padded with blanks.

In ND FORTRAN, a record which is output and is shorter than the specified RECL value, will have undefined characters at the end of the record, as it is stored on a file. This applies to both FORMATTED and UNFORMATTED files. However, if a record is read from a FORMATTED file, it will be padded with blanks after the read operation.

If a FORMATTED file has a RECL= specifier in the OPEN statement, and data to be output is longer than the specified record length, the record is truncated.

In ND FORTRAN, in both FORMATTED and UNFORMATTED I/O with a RECL= specifier in the OPEN statement, a request to output more data than the RECL= specifier value, will result in a physical record being output, which is a multiple of the RECL= specifier value.

Scanned by Jonny Oddene for Sintran Data © 2011

The following table summarizes the possible outcomes of I/O, in ND FORTRAN:

	RECL= specified	*no RECL=*
FORMATTED	*data CR LF uuu*	*data CR LF*
UNFORMATTED	*data uuu*	*data*

where

data is the record to be read or written by the program.

uuu is an undefined part of the record.

BUFFER_SIZE = bs

where

bs is an integer value, which gives the number of bytes in the buffer. The smallest value is 2048 bytes (1 page); the value must be a potence of 2. The BUFFER-SIZE used by FORTRAN can be smaller than specified if there is not enough contiguous space in buffer pool. Use of bigger buffer is most effective for big contiguous files. In FORTRAN-100 this is a dummy parameter.

Scanned by Jonny Oddene for Sintran Data © 2011

BLANK = bl

where

bl is a character expression whose value is NULL or ZERO. It is valid only for files being connected for formatted I/O and it determines the treatment of blanks. If NULL is specified, then all blank characters in numeric input fields are ignored (except that a field of all blanks has a value of zero). If ZERO is specified, then all blanks are treated as zeros.
The default value is NULL.

The following specifiers are ND FORTRAN Extensions:

PARITY = par

where

par is a character expression which indicates how the "parity" bit is to be handled. The "parity" bit is the left-most bit of the character read or written. It applies only to formatted (including list-directed) transfers of data. The possible values and meanings are:

IGNORE no action is taken on either input or output.

SET the parity bit is set to zero on input, and to even parity on output. (This is compatible with the previous FORTRAN implementation.)

REMOVE the parity bit is set to zero on input, but is left untouched on output.

If this parameter is not specified, the value is taken to be REMOVE unless the device is a terminal, in which it is taken to be SET.

REMOVE is supplied as an easy way of converting files from the old to the new form.

Scanned by Jonny Oddene for Sintran Data © 2011

FIRSTREC = value

where *value* is an expression of type integer. Value determines the number assigned to the first record on a file. The default value, if the option is omitted, is 1. Previous compilers used 0 to indicate the first record. Only 0 and 1 are allowed as valid values; any other value will give unpredictable results.

In ND FORTRAN, this option can be used to maintain compatibility with previous ND FORTRAN compilers with regard to the numbering of records for a direct access file.

FACTOR = fac

where *fac* is an integer expression (with legal values of 1, 2, or 4). This parameter specifies the modification to the "amount" factor in the monitor calls SETBS, RFILE, WFILE, and MAGTP. The monitor calls have as argument(s) the length of the area read or written except for MAGTP function codes 26B and 27B where the amount is the exact number of bytes. The given amount parameters (or return parameters) in these monitor calls are adjusted by the value of *fac* before and sometimes after the monitor calls are executed. A value of fac = 1 indicates that the amount parameter is to be interpreted as a number of bytes, fac = 2 means the number of 16-bit words, while fac = 4 means the number of 32-bit words.

Scanned by Jonny Oddene for Sintran Data © 2011

The default is fac = 2 for the ND-100 and fac = 4 for the ND-500, i.e. the number of words in both cases.

IOCONVERT = ioco

where

ioco is a character expression whose value is CONVERT or FORCE. The ioco parameter value indicates the handling of formatted I/O when either:

- the I/O list element is of type REAL or COMPLEX; and the format specifier is an I

  or,

- the I/O list element is of type INTEGER and the format specification is F, E or G.

A value of CONVERT indicates that a conversion REAL/INTEGER or INTEGER/REAL, is to take place if the specification would not otherwise apply.

For the ND-500, and the ND-100 with the 32-bit floating-point option, a value of FORCE means that formatting is to be performed according to the format specification regardless of the type of the I/O list element (INTEGER*4/REAL*4). The default is FORCE.

It is an error to specify FORCE on an ND-100 with the 48-bit floating-point processor; the default here is CONVERT.

Scanned by Jonny Oddene for Sintran Data © 2011

TYPE = ty

where

ty is a CHARACTER string. The first four characters (or the whole string if its length is less than four) are used to change the default SINTRAN file type from SYMB to some other value.

MODE = mo (ND-500 only)

where

mo is a CHARACTER string with the value SEGMENT or BUFFER.

When SEGMENT is used it results in the file being used as a segment (ND-500 only). This will normally achieve more efficient I/O usage.

Accesses 'SEQUENTIAL' and 'DIRECT' have as default MODE='SEGMENT'. This mode can be changed by use of MODE='BUFFER'.

The use of this parameter will have no effect on the way a program executes, other than speed, providing only the following I/O statements are used:

READ, WRITE, ENDFILE, BACKSPACE, REWIND, OPEN and CLOSE.

Note:

No monitor call (in particular RFILE or WFILE) can be used for a file being used as a segment except for MON 416B (WSEGN).

Scanned by Jonny Oddene for Sintran Data © 2011

9.3.2 The CLOSE Statement

A CLOSE statement is used to terminate the connection of a file to a unit.

It has the form:

```
CLOSE (u[,arglist])
```

where

u is a unit specifier (see Section 9.1.3, on page 162)

arglist is a list of optional items, separated by commas, in which each of the following items may appear no more than once:

IOSTAT = s (see Section 9.1.7, on page 166)

ERR = label (see Section 9.1,6, on page 165)

STATUS = sta

where *sta* is a character expression whose value is KEEP or DELETE.

The unit to be deleted must be explicitly specified. If unit in the following: CLOSE (u, status = 'DELETE') has a negative value, no files are deleted.

In ND FORTRAN, the following values for u have special meanings:

-1 means close all files opened for this terminal, except those permanently opened.

-2 means close all files, even those permanently opened.

-3 means that all files opened by the ND-500 Monitor or the ND-500 program, will be closed.

Scanned by Jonny Oddene for Sintran Data © 2011

9.4 FILE POSITIONING

The statements used for positioning are BACKSPACE, ENDFILE and REWIND. The operations performed by these statements are normally used for sequential files on disk or magnetic tape devices.

9.4.1 The BACKSPACE Statement

The BACKSPACE statement will cause a file, connected to a specified unit, to be positioned at the start of the preceding record. If there is no preceding record, the file position remains unchanged.

Format:

```
BACKSPACE u
```

or

```
BACKSPACE (u[,arglist])
```

where

u is a unit specifier

Scanned by Jonny Oddene for Sintran Data © 2011

arglist is a list of the following optional items, separated by commas, as given below:

IOSTAT = s (see Section 9.1.7 on page 166)

ERR =label (see Section 9.1.6 on page 165)

In ND FORTRAN, a SINTRAN logical device number may be used instead of a unit specifier.

If the file was opened with a RECL parameter, then this parameter is used to identify the position of the previous record. If the file is a formatted file, the statement will execute slowly unless RECL is specified.

9.4.2 The ENDFILE Statement

The ENDFILE statement is used to write an endfile record as the next record of the file. This record will define the end of the file that contains it.

Format:

ENDFILE u

or

ENDFILE (u[,arglist])

where

u is a unit specifier

Scanned by Jonny Oddene for Sintran Data © 2011

arglist is a list of the following optional items, separated by commas, as given below:

IOSTAT = s (see Section 9.1.7 on page 166)

ERR = label (see Section 9.1.6 on page 165)

In ND FORTRAN, a SINTRAN logical device number may be used instead of a unit specifier.

After execution of an ENDFILE statement, a BACKSPACE or REWIND statement must be used to reposition the file prior to execution of any data transfer I/O statement.

NOTE:

An ENDFILE statement will not automatically be performed before rewinding.

9.4.3 The REWIND Statement

Execution of a REWIND statement causes the specified file to be positioned at its initial point (the load- point mark on a magnetic tape). If the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Format:

REWIND u

or

REWIND (u[,arglist])

where

u is a unit specifier

Scanned by Jonny Oddene for Sintran Data © 2011

9.5 THE INQUIRE STATEMENT

The INQUIRE statement may be used to inquire about properties of a particular named file or of the connection to a particular unit.

The INQUIRE statement may be executed before, during, or after a file is connected to a unit. All values assigned by the INQUIRE statement are those that are current at the time the INQUIRE statement is executed.

The two forms of the INQUIRE statement are:

- INQUIRE by file:

```
INQUIRE(FILE=filename[,arglist])
```

where

filename is a character expression whose value, when any trailing blanks are removed, specifies the name of the file being inquired about.

arglist is a list of optional specifiers, taken from the table given on the next page. The specifiers must be separated by commas, and each may occur no more than once.

The specifier FILE= may appear anywhere in *arglist*.

Scanned by Jonny Oddene for Sintran Data © 2011

- INQUIRE by unit:

```
INQUIRE(u[,arglist])
```

where

u is a unit specifier (see Section 9.1.3 on page 162)

arglist is a list of optional specifiers taken from the table given on the next page. The specifiers must be separated by commas, and each may occur no more than once.

If the form UNIT= is used for the unit specifier, it may appear anywhere in *arglist*. If UNIT= is omitted, *u* must be the first specifier in the list.

The following inquiry specifiers may be used in either form of the INQUIRE statement, i.e. for arglist above:

IOSTAT	= s	(see Section 9.1.7, page 166)
ERR	= label	(see Section 9.1.8, page 165)
ACCESS	= acc	
BLANK	= blnk	
DIRECT	= dir	
EXIST	= ex	
FORM	= fm	
FORMATTED	= fmt	
NAME	= fn	
NAMED	= nmd	
NEXTREC	= nr	
NUMBER	= num	
OPENED	= od	
RECL	= rcl	
SEQUENTIAL	= seq	
UNFORMATTED	= unf	

Scanned by Jonny Oddene for Sintran Data © 2011

The specifiers are described in the rest of this chapter:

ACCESS=acc

where

acc is a character variable or a character array element.

acc will be assigned the value SEQUENTIAL if the file is connected for sequential access, or the value DIRECT if the file is connected for direct access.

Scanned by Jonny Oddene for Sintran Data © 2011

If there is no connection, **acc** becomes undefined.

BLANK=blnk

where

blnk is a character variable or a character array element.

blnk will be assigned the value NULL if null blank control is in effect for a file connected for formatted I/O, or the value ZERO if zero blank control is in effect for a file connected for formatted I/O.

If there is no connection, or if the connection is not for formatted I/O, *blnk* becomes undefined.

DIRECT=dir

where

dir is a character variable or a character array element.

dir will be assigned the value YES if DIRECT is one of the allowed access methods for the file, or the value NO if DIRECT is not one of the allowed access methods for the file.

If it is not possible to determine whether DIRECT is allowed as an access method for the file, *dir* will be assigned the value UNKNOWN.

Scanned by Jonny Oddene for Sintran Data © 2011

EXIST=ex

where

ex is a logical variable or a logical array element.

For INQUIRE by file, *ex* will be assigned the value .TRUE. if the file with the specified file name exists, or the value .FALSE. otherwise.

For INQUIRE by unit, *ex* will be assigned the value .TRUE. if the specified unit exists, or the value .FALSE. otherwise.

FORM=fm

where

fm is a character variable or a character array element.

fm will be assigned the value FORMATTED if the file is connected as formatted I/O, or UNFORMATTED if the file is connected for unformatted I/O.

If there is no connection, *fm* becomes undefined.

FORMATTED=fmt

where

fmt is a character variable or a character array element.

fmt will be assigned the value yes if formatted is an allowed form for the file, or the value NO if FORMATTED is not an allowed form for the file.

If it is not possible to determine whether FORMATTED is an allowed form for the file, *fmt* will be assigned the value UNKNOWN.

Scanned by Jonny Oddene for Sintran Data © 2011

NAME=nme

where

nme is a character variable or a character array element.

nme will be assigned the name of the file being inquired about, i.e. the file named in the FILE= specifier, or connected by a UNIT= specifier.

The value assigned to *nme* will not necessarily be identical to the name in the FILE= specifier. The value assigned to *nme* is a fully qualified file name, which is suitable for use in the FILE= specifier of the OPEN statement.

In ND FORTRAN, the value assigned to *nme* is made up of, (directory:user)name:type;version, while the name used in the FILE= specifier may be an abbreviated form of the name, using the usual rules for abbreviation.

NAMED=nmd

where

nmd is a logical variable or a logical array element.

nmd will be assigned a value .TRUE. if the specified file has a name, or the value .FALSE. otherwise.

Scanned by Jonny Oddene for Sintran Data © 2011

NEXTREC=nr

where

nr is an integer variable or an integer array element.

nr will be assigned an integer value, plus one, of the record number of the last record read, or written, to a file connected for direct access. If the file is connected, but no records have been read or written, *nr* will be assigned 1.

If the file is not connected for direct access, or the position of the file is indeterminate because of a previous error condition, *nr* becomes undefined.

NUMBER=num

where

num is an integer variable or an integer array element.

num will be assigned the value of the unit currently connected to the specified file.

If there is no unit connected to the specified file, *num* becomes undefined.

Note:

If the form of the INQUIRE statement is inquired by file, and the UNIT= specifier is set to -1, then the SINTRAN logical device number will be assigned to the NUMBER= identifier. ¨font=9;

Scanned by Jonny Oddene for Sintran Data © 2011

OPENED=od

where

od is a logical variable or a logical array element.

od will be assigned the value .TRUE. if either the specified file (specified by the FILE= specifier) or the unit specified, is currently open, or the value .FALSE. if the file or unit is not open.

RECL=rcl

where

rcl is an integer variable or an integer array element.

rcl will be assigned the value of the record length of a file connected for direct access. The value is in bytes, whether the file has been connected for formatted or unformatted I/O.

If the file is not connected, or if the file is connected for other than direct access, *rcl* becomes undefined.

In ND FORTRAN, the RECL= specifier in the INQUIRE statement is always defined if the RECL= specifier in the corresponding OPEN statement is defined.

SEQUENTIAL=seq

where

seq is a character variable or a character array element.

seq will be assigned the value YES if sequential is one of the allowed access methods for the file, or the value NO if SEQUENTIAL is not one of the allowed access methods for the file. If it is not possible to determine whether SEQUENTIAL is allowed as an access method for the file, *seq* will be assigned the value UNKNOWN.

Scanned by Jonny Oddene for Sintran Data © 2011

UNFORMATTED=unf

where

unf is a character variable or a character array element.

unf will be assigned the value YES if UNFORMATTED is an allowed form for the file, or the value NO if UNFORMATTED is not an allowed form for the file.

If it is not possible to determine whether UNFORMATTED is an allowed form for the file, *unf* will be assigned the value UNKNOWN.

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 10

FORMAT SPECIFICATIONS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

A format used in conjunction with formatted I/O statements provides information that directs the editing between the internal representation and the character strings of one or a sequence of records in the file.

A format specification provides explicit editing information. An asterisk (*) as a format identifier in an I/O statement indicates list-directed input/output, see Section 9.2.3 on page 170.

10.1 FORMAT SPECIFICATION METHODS

Format specifications may be given either:

- in FORMAT statements, or
- as arrays of CHARACTER strings, CHARACTER variables, or other CHARACTER expressions.

The FORMAT statement has the form:

```
FORMAT (F1, F2, F3, ....., Fn)
```

where

F1, F2,... etc. are format descriptors, described in the next section.

The comma used to separate the descriptors may be omitted in the following circumstances:

- before or after a slash or colon format descriptor
- between a P format descriptor and an F, E, D, or G descriptor which follows immediately after it

Scanned by Jonny Oddene for Sintran Data © 2011

The FORMAT statement must be labeled.

With character format specifications, as in the second instance above, the expression must contain format descriptor(s) enclosed in parentheses.

10.2 FORMAT DESCRIPTORS

These descriptors describe the record structure of the data, the format of the fields within the record, and the conversion, scaling and editing of data within specific fields. A list is given on the following page.

Scanned by Jonny Oddene for Sintran Data © 2011

DESCRIPTORS	COMMENTS
rFw.d	
rEw.d	
rEw.dEe	Floating-point numeric field descriptors
rDw.d	
rGw.d	
rGw.d.Ee	
rIw	
rIw.m	INTEGER field descriptors
rJw	**(ND FORTRAN Extension)**
rLw	LOGICAL field descriptor
rA	
rAw	Alphanumeric data field descriptor
rOw	**Octal format descriptor (ND FORTRAN Extension)**
rZw	**Hexadecimal format descriptor (ND FORTRAN Extension)**
nHs	
'text'	Text descriptors
text	**(ND FORTRAN Extension)**
rX	Field formatting descriptor
kP	Numerical scale factor descriptor
:	Format control terminating descriptor
/	Record delimiting descriptor
Tc	
TLc	Positional editing descriptors
TRc	
S	
SP	Optionally positive sign editing descriptors
SS	
BN	
BZ	Blank interpretation descriptors

Scanned by Jonny Oddene for Sintran Data © 2011

Explanation:

r is a repetition factor and is a nonzero unsigned integer constant.

d and *m* are unsigned integer constants.

w, *e*, *n* and *c* are nonzero unsigned integer constants.
k is an optionally signed integer constant.

s is a string of characters.

NOTE:

w is known as the field width and is the size in characters of the field, the part of a record read on input or written on output under the control of a format specification.

In addition, repetition of groups of format descriptors can be achieved by parentheses, e.g. r(F1, F2, ... ,Fn) where *Fi* are format descriptors.

In ND FORTRAN the maximum depth of nesting of these parentheses is 5.

The following sections provide detailed descriptions of the various types of format descriptors and the manner in which they are written and employed.

10.2.1 Interaction between the Format Descriptors and the I/O List

The execution of an I/O statement specifying a formatted data transfer operation will initiate format control. The contents of the I/O list and the format specifications are scanned in step. Whenever format control encounters a repetition factor in a format descriptor, it determines whether there is a corresponding item in the I/O list. If there is, it transmits appropriately edited information between the item and the record. If not, format control terminates.

Scanned by Jonny Oddene for Sintran Data © 2011

A list item of type complex will require two corresponding format descriptors of type F, E, D, or G.

For P, X, T, TL, TR, S, SP, SS, H, BN, BZ, slash, colon, or text format descriptors there are no corresponding items in the I/O list, and format control communicates information directly to or from the record.

If format control encounters the rightmost parenthesis of a complete format specification and another list item is not found, format control terminates. It also terminates if a colon descriptor is encountered in the format specification and another list item is not found.

If the end of the format specification is reached and more items remain in the list, a new record is established and the scan process is restarted. It restarts either at the first item in the format specification or, if parenthesised, with the last set of descriptors within the format specification. (That is, restarting at the first left parenthesis to the left in the format specification just acted upon.)

A record is terminated by one of the following:

- A slash format descriptor.
- The rightmost parenthesis of the FORMAT.
- The end of the I/O list is encountered, and the rest of the format descriptors require I/O list items.
- A colon descriptor is encountered, and there are no more items in the I/O.

On input, only a single slash, /, will cause an additional record to be read. A record is skipped when two slashes, //, are encountered or a slash is followed by the end of the format specifications.

If the record ends, due to the end of the format specifications or a slash within them, then any data left in the input record is ignored. If the input record is exhausted before the data transfers are completed, then the transfer proceeds as if the record were extended with blanks.

Scanned by Jonny Oddene for Sintran Data © 2011

On output, an additional record is written only when a slash, /, is encountered in the format specifications. Two consecutive slashes or one slash followed by the end of the specifications will cause an empty record to be written.

If the file is an internal file, then a record is determined by the length of the internal data item. For non-CHARACTER arrays, and for CHARACTER variables, the file contains just one record. For CHARACTER arrays, each element is a record, the order of access being the same as the order of implied subscripting, with the first subscript varying most rapidly.

In ND FORTRAN, records on a formatted external file are delimited by the ASCII carriage return character (octal 15). Line-feeds (octal 12) which immediately follow a carriage return are ignored on input.

In ND FORTRAN, the maximum length of a formatted record is 256, excluding the delimiters.

10.2.2 Editing Provided by the Format Descriptors

10.2.2.1 Numeric Editing

The I, F, E, D and G descriptors are used for the I/O of INTEGER, REAL, DOUBLE PRECISION and COMPLEX data. The following rules apply:

- On input, leading blanks are not significant. The interpretation of other blanks depends on whether any BLANK = (see the OPEN statement) specifier and whether any BN or BZ control is currently in effect. Plus signs may be omitted. A field of all blanks is considered to be zero.

Scanned by Jonny Oddene for Sintran Data © 2011

- On input, with F, E, D and G editing, a decimal point appearing in the input field overrides its specification in a format descriptor.

- On output, the representation of a zero or positive value in the field may be prefixed with a plus, as controlled by the S, SP and SS descriptors. A negative internal value will be prefixed by a minus in the field.

- On output, the representation in the field is right justified. After editing, if the number of characters is less than the field width, leading blanks will be inserted. If the number of characters exceeds the field width then the entire field of width **w** is filled with asterisks.

10.2.2.2 The I and J Format Descriptors

The **Iw**, **Iw.m**, and **Jw** descriptors are for INTEGER editing, where the field for editing occupies **w** positions. The specified I/O list item must be of type INTEGER.

In ND FORTRAN, if the list item is of type REAL, DOUBLE PRECISION or COMPLEX, then the editing will be performed according to the IOCONVERT specifier in the OPEN statement.

In ND FORTRAN, the Iw descriptor can also be used for editing items of type NUMERIC.

In the input field, the character string must be in the form of an optionally signed integer constant. On input an **Iw.m** descriptor is treated identically to an **Iw** descriptor.

Scanned by Jonny Oddene for Sintran Data © 2011

The output field for the **Iw** descriptor will consist of leading blanks, if any, a minus sign if the internal value is negative, or an optional plus if the internal value is positive. This is followed by the magnitude of the internal value expressed as an unsigned integer constant and must consist of at least one digit.

The output field for the **Iw.m** descriptor is the same as for the **Iw** descriptor except that the unsigned integer constant consists of at least m digits and, if necessary, leading zeros. The value of **m** must not be greater than **w**. If it is zero and the internal value is also zero, the output field will consist only of blanks, regardless of sign control.

In ND FORTRAN, the Jw is treated like Iw except that there is no zero suppression at the start of the output field. If a sign is to be output, it occupies the first position, otherwise a digit fills this place.

Examples:

VALUE	*FORMAT*	*OUTPUT*
1	*I1*	*1*
1234	*I5*	*1234*
-1234	*I5*	*-1234*
0	*I5*	*0*
1234	*I5.0*	*1234*
0	*I5.0*	
12	*I5.4*	*0012*
-12	*I5.4*	*-0012*
0	*I5.4*	*0000*

10.2.2.3 REAL and DOUBLE PRECISION

The F, E, D and G format descriptors specify the editing of REAL, DOUBLE PRECISION, and COMPLEX data. An I/O list item corresponding to one of these descriptors must also be REAL, DOUBLE PRECISION or COMPLEX.

Scanned by Jonny Oddene for Sintran Data © 2011

In ND FORTRAN, if the list item is of type INTEGER, then the editing will be performed according to the IOCONVERT specifier in the OPEN statement.

In ND FORTRAN, the F, E and D descriptors can also be used for editing items of type NUMERIC.

10.2.2.4 The F Format Descriptor

The **Fw.d** descriptor implies that the field contains **w** positions, the fractional part of which consists of **d** digits.

The input field consists of an optional sign, followed by a string of digits optionally containing a decimal point. If there is no decimal point, the rightmost digits are interpreted as the fractional part of the value. The basic form may be followed by an exponent of the form:

- Signed integer constant, or
- E followed by zero or more blanks, followed by an optionally signed integer constant, or
- D followed by zero or more blanks, followed by an optionally signed integer constant.

An exponent containing an E is processed identically to an exponent containing a D.

The output field consists of blanks, if necessary, followed by a minus if the internal value is negative, or an optional plus otherwise. This is followed by a string of digits containing a decimal point and representing the magnitude of the internal value, modified by any established scale factor, and rounded to d digits. Leading zeros are suppressed up to the decimal point, i.e. if the value lies between zero and one, the first non-blank character represents the position of the decimal point.

Scanned by Jonny Oddene for Sintran Data © 2011

Examples:

VALUE	*FORMAT*	*OUTPUT*
1.2	*F5.0*	*1.*
-1.2	*F5.0*	*-1.*
1.2	*F5.1*	*1.2*
0.4	*F5.2*	*.40*
-0.4	*F5.2*	*-.40*
0.	*F5.2*	*.00*
1.	*F5.2*	*1.00*
-1.	*F5.2*	*-1.00*

10.2.2.5 Scale Factor: The P Format Descriptor

The P format descriptor specifies the scale factor in the form:

where

k is called the scale factor and is an optionally signed constant.

The value of the scale factor is zero at the beginning of execution of each I/O statement. It applies to all subsequently interpreted F, E, D and G descriptors until another scale factor is encountered. It has the following effect upon the editing:

- With F, E, and D format descriptors on input (provided that no exponent exists in the field) and the F format descriptor on output, the externally represented number equals the internally represented number multiplied by 10 ** k.
- On input, with F, E, D and G format descriptors, the scale factor has no effect if there is an exponent in the field.

Scanned by Jonny Oddene for Sintran Data © 2011

- On output, with E and D format descriptors, the basic real constant part of the quantity (optional sign, integer part, decimal point and fractional part) is multiplied by 10 ** k. The exponent is reduced by **k**.

- On output, with G editing, the scale factor has no effect unless the magnitude of the value is outside the range for F editing. If the use of E editing is required, the scale factor has the same effect as using the E format descriptor on output.

Example:

```
      REAL REALARR (4)
  100 FORMAT (1X, F12.4, 2PF12.4, F12.4, -2PF12.4)
  200 FORMAT (1X, E12.4, 1X, 2PE12.4, 1X, -1PE12.4, 1X, OPE 12.4)
  300 FORMAT (1X, 4F12.4, ///)
  400 FORMAT (1X, 4 (E12.4, 1X))
  500 FORMAT (F12.4, 2PF12.4, -1PF12.4, -2PF12.4)
  600 FORMAT (E12.4, 2PE13.4, -1PE13.4, OPE13.4)
      READ (1,500) REALARR
      WRITE (1,100) REALARR
      WRITE (1,300) REALARR
      READ (1,600) REALARR
      WRITE (1,200) REALARR
      WRITE (1,400) REALARR
```

Input and Output With F Editing:

1.6	.16E+1	160.E-2	1.6
1.6000	160.0000	160.0000	1.6000
1.6000	1.6000	1.6000	160.0000

Scanned by Jonny Oddene for Sintran Data © 2011

Input and Output With E Editing:

2.5	*.25E+1*	*2.5*	*250.E-2*
.2500E+01	*25.000E-01*	*.0250E+03*	*.2500E+01*
.2500E+01	*.2500E+01*	*.2500E+02*	*.2500E+01*

10.2.2.6 The E and D Format Descriptors

The Ew.d, Dw.d and Ew.dEe descriptors indicate that the external field occupies w positions, the fractional part of which consists of **d** digits (unless a scale factor of greater than one is in effect) and the exponent part consists of **e** digits. The **e** has no effect on input.

The form of the input field is the same as that for the F format descriptor described above.

With a zero scale factor the form of the output would be:

[±] . X_1 X_2 -----*Xd exp*

X_1 X_2 -----*Xd*

being the most significant digits after rounding.

exp is a decimal exponent, which for the value 76 or less will be of the form:

$E \pm n_1 \; n_2$

or (for Dw.d descriptors only):

$D \pm n_1 \; n_2$

where

n is a decimal digit.

Scanned by Jonny Oddene for Sintran Data © 2011

For the Ew.dEe descriptor, the form of the exponent is:

$$E \pm n_1 \; n_2 \text{ ---- } ne$$

where the absolute value of the exponent must be:

$\leq (10 ** e) - 1$

The scale factor **k**, described above, controls the decimal normalisation.

There are two cases to consider:

- $-d < k \leq 0$. The output field will contain (taking the absolute value of **k**) **k** leading zeros and **d-k** significant digits after the decimal point.
- $0 < k < d + 2$. The output field will contain **k** significant digits to the left of the decimal point and **d-k+1** significant digits to the right of the decimal point.

Other values of **k** are not accepted.

Examples:

VALUE	*FORMAT*	*OUTPUT*
0.	*E12.4*	*.0000E+00*
123.	*E12.4*	*.1230E+03*
-123.	*E12.4*	*-.1230E+03*
123.	*E12.4E1*	*.1230E+3*
-123.	*E12.4E1*	*-.1230E+3*

Scanned by Jonny Oddene for Sintran Data © 2011

10.2.2.7 The G Format Descriptor

The Gw.d and Gw.dEe descriptors indicate that **w** is the width of the external field, the fractional part of which contains d digits unless a scale factor of greater than one is in effect. The exponent part consists of **e** digits.

On input, the editing performed by the G format descriptor is the same as that for F described earlier.

On output, the editing depends on the magnitude of the internal value, N, in the following way:

If N <0.1 or N> 10**d, then editing performed by Gw.d on output is the same as that by kPEw.d, and for Gw.dEe the result is the same as that when using kPEw.dEe.

If 0.1 < N < 10**d, **k** (the scale factor) has no effect and the results depend upon **N** as given below where F is the format descriptor, and **n** takes the value 4 with the Gw.d descriptor or e+2 for the Gw.dEe descriptor:

N	*Conversion Type*
0.1 ≤ N < 1	*F(w-n).d*
1 ≤ N < 10	*F(w-n).d-1*
.	.
.	.
.	.
*10**(d-2) < N < 10**(d-1)*	*F(w-n).1*
*10**(d-1) < N < 10**d*	*F(w-n).0*

The output field will be followed by **n** blanks.

Note that the scale factor has no effect unless **N** is outside the range of values for effective **F** editing.

Scanned by Jonny Oddene for Sintran Data © 2011

10.2.2.8 COMPLEX Data

Since this consists of separate pairs of real values the editing is specified by two successive F, E, D, or G format descriptors.

The first descriptor will specify the real part, the second the imaginary part. The two descriptors may be different and other non-repeatable descriptors may appear between them.

10.2.2.9 S, SP and SS Format Descriptors

These edit descriptors are used to control the optional plus signs in the output fields.

If none of the edit descriptors are used, then optional plus signs will not be printed.

If an SP descriptor is encountered in a format specification, then subsequent optional plus signs will be printed.

If an SS or S descriptor is encountered, then further optional plus signs will not be printed.

10.2.2.10 The BN and BZ Format Descriptors

These descriptors specify the interpretation of non-leading blanks in numeric input fields. Such blank characters, at the beginning of the input statement execution, are interpreted as zeros or are ignored, depending upon the BLANK=specifier in the OPEN statement in effect.

Upon encountering a BN descriptor in the format specification, the non-leading blanks referred to above will be ignored.

The effect of a BZ descriptor is to treat all such blanks as zeros.

BN and BZ affect the I, F, E, D, G, O and Z editing during input.

Scanned by Jonny Oddene for Sintran Data © 2011

10.2.2.11 The Text Format Descriptor

This descriptor has the form 'text' which is equivalent to a character constant. It causes a character string (which may include blanks) to be written from the enclosed characters of the format descriptor itself. An apostrophe edit descriptor is not valid on input. The width of the field is the number of characters between the delimiting apostrophes, but not including the apostrophes themselves.

ND FORTRAN has an alternative form: *text*.

If the asterisk is used as the text delimiter, then a quote is treated as just another character. Correspondingly, if the delimiter is a quote, then the asterisk is treated as an ordinary character.

10.2.2.12 The H Format Descriptor

The descriptor has the form:

nHs

It causes the *n* characters forming the string *s* to be written on the output stream.

In ND FORTRAN the form Hs may be used for a single character.

An **H** descriptor must not be used on input.

Scanned by Jonny Oddene for Sintran Data © 2011

10.2.2.13 The T, TL, TR, and rX Format Descriptors

These descriptors control positional editing and specify at which position the next character will be transmitted to or from the record.

The position indicated by a **T** descriptor may be in either direction from the current position. On input, this allows parts of a record to be processed more than once, possibly with different editing.

On output, since this group of descriptors do not themselves cause characters to be transmitted, they do not affect the length of the record. If characters are transmitted to or beyond the position specified positions skipped are filled with blanks. The result is as if the whole record were initialized with blanks.

In the **Tc** format descriptor, **c** is the character position to which, or from which the record transmission of the next character is to occur.

With the **TLc** descriptor, the transmission is to occur at a position **c** characters backward from the current one. (If the current position should be **c**, the transmission to or from will start from position 1 (one) of the record.)

With the **TRc** descriptor, the transmission will occur at a position **c** characters forward from the present one. The rX format descriptor causes the transmission of the next character to or from a record to occur at a position **r** characters forward from the current position. On input this position may be beyond the last character of the record so long as no characters are transmitted from such positions.

In ND FORTRAN, *r* can be omitted and 1 will be assumed as its value.

Scanned by Jonny Oddene for Sintran Data © 2011

10.2.2.14 The Slash, /, Format Descriptor

This descriptor denotes the end of data transfer on the current record. The following will occur:

- On input from a sequential file, the remaining portion of the record is skipped and the next record becomes the current record.
- On output to a sequential file, a new record is created and becomes the last and current record of the file.
- For a direct access file, the record number **i** is increased by one and the file is positioned at the beginning of the record having that number. This record becomes the current record.

In ND FORTRAN, the slash format descriptor can be preceded by a replication factor.

10.2.2.15 The L Format Descriptor

This descriptor has the form:

where

w indicates that the field occupies *w* positions.
The corresponding I/O list item must be of type LOGICAL.

The input field consists of optional blanks, optionally followed by a decimal point, followed by T for true or F for false. T or F may have additional characters following them in the field.

The output field consists of w-1 blanks followed by T or F according to whether the value of the internal LOGICAL variable has the value .TRUE. or .FALSE.

Scanned by Jonny Oddene for Sintran Data © 2011

10.2.2.16 The A Format Descriptor

The A [w] format descriptor is used with CHARACTER I/O list items.

If the optional field width **w** is used then the field consists of **w** characters. If **w** is not specified, then the number of characters in the field is the same as the length of the I/O list item.

Let I be the length of the I/O list item. On input, if

$$w \geq I$$

then the rightmost I characters will be taken from the input field.

If however:

$$w < I$$

w characters will appear left justified with **I-w** trailing blanks.

On output, if:

$$w > I$$

the output field will consist of **w-I** blanks followed by **I** characters from the internal representation.

Scanned by Jonny Oddene for Sintran Data © 2011

If, however:

$w \leq I$

the output field will consist of the leftmost **w** characters of the internal representation.

Example:

VALUE	*FORMAT*	*OUTPUT*
'ABCDE'	*A*	*ABCDE*
'PQRST'	*A3*	*PQR*
'PQRST'	*A8*	*bbbPQRST*

where

b represents a space character (blank).

10.2.2.17 The O Format Descriptor

In ND FORTRAN this descriptor is used for octal I/O for any of the data types - INTEGER, REAL or DOUBLE PRECISION.

On input, w characters from the input record are treated as an octal number. The bit representation of the binary value is then placed, unconverted, in the list item, right-justified and truncated or padded with zeros on the left.

Scanned by Jonny Oddene for Sintran Data © 2011

Example:

The following example is on an ND-100 with 48-bit floating-point hardware:

INPUT (Octal)	*FORMAT*	*INTERNAL VALUE (INTEGER*2) (octal)*
137326	*O6*	*137326*
2671	*O6*	*002671*
37533235	*O8*	*133235*
		*Value (REAL*6)*
2000160000000002	*O16*	*040003 100000 000002*

On output, the internal bit representation is treated as an unsigned binary integer and is output right-justified with truncation or padding with blanks on the left in a field of width **w**.

10.2.2.18 The Z Format Descriptor

In ND FORTRAN this descriptor is used for hexadecimal I/O for any of the data types: INTEGER, REAL or DOUBLE PRECISION.

On input, **w** characters from the input record are treated as a hexadecimal number. The bit representation of the binary value is then placed, unconverted, in the list item, right-justified and truncated or padded with zeros on the left.

For example:

INPUT (Hexadecimal)	*FORMAT*	*INTERNAL VALUE (INTEGER*4) (decimal)*
2186E	*Z6*	*137326*
A67	*Z6*	*2671*
F00000001	*Z9*	*1*
		*Value (REAL*8)*
4050000000000000	*Z16*	*1.0*

Scanned by Jonny Oddene for Sintran Data © 2011

On output, the internal bit representation is treated as an unsigned binary integer and is output right-justified with truncation or padding with blanks on the left in a field of width w.

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 11

FUNCTIONS AND SUBROUTINES

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

Functions and subroutines (also known as subprograms) are procedures which can be called from within a program as many times as required. These procedures may be either internal (contained within the program in which they are referenced) or external (self-contained executable procedures that may be compiled separately).

These are the categories of procedures:

- Intrinsic functions
- Statement functions
- External functions
- Subroutines

The first three categories are referred to collectively as functions. The last two, subroutines and external functions, are both referred to as external procedures.

Section 11.2.2 contains a table of all the INTRINSIC functions, including those defined in ANSI FORTRAN 77 and a number of extra functions implemented in ND FORTRAN. Appendix D contains descriptions and a table of functions or subroutines of a more general nature, provided in the FORTRAN library. Appendix C contains complete descriptions of all the available Monitor Calls, i.e. services provided by the operating system.

Scanned by Jonny Oddene for Sintran Data © 2011

11.1 DUMMY AND ACTUAL ARGUMENTS

Some of the entities used by functions and subroutines may vary from one call to another. Such entities are represented by dummy arguments of the same type and are given in the form of a list associated with the subprogram identifier. The actual arguments, i.e., the values the entities are to take for a particular reference to the subprogram, are given in a corresponding list associated with this reference.

Example:

A function to evaluate the arithmetic mean (average) of two real numbers could be defined as:

```
FUNCTION AVER(A,B)
AVER = (A+B) / 2.0
RETURN
END
```

The first statement defines AVER as a function and indicates that it has two dummy arguments, A and B. The second statement demonstrates how to evaluate the function. The third statement shows that control is complete and is to return to the routine or program which invoked this function. The END statement indicates that the definition of this function is complete. These statements are discussed in more detail later in this chapter.

To use the function to calculate an average, the following could be written:

```
P = AVER(X, Y)
```

where

X and **Y** are the actual arguments in this invocation. The statement demonstrates the invocation of AVER and the assignment of the resulting function value to P. The actual argument X is associated with the dummy argument A, and the actual argument Y with the dummy argument B.

Scanned by Jonny Oddene for Sintran Data © 2011

The result, as defined above, is (A+B)/2.0 which, in this case, is (X+Y)/2.

The result can be used as part of an expression in the same manner as any other operand.

For example:

```
P = Q+AVER(1.0, T+V) * S
```

which evaluates the average of the constant 1.0 and T+V. It then multiplies the result by S, adds Q, and puts the resulting sum in P.

Actual arguments may be constants (or their symbolic names), function references, expressions involving operators, and expressions enclosed in parentheses, if and only if the associated dummy argument is 'read-only', i.e. its value is not changed by the subprogram.

The type of each actual argument must agree with the type of its associated dummy argument except when the actual argument is a subroutine name, see Section 11.5, on page 264 or an alternate return argument on Section 11.7 on page 270.

Example:

In the previous example demonstrating the AVER function, neither A nor B are in any way changed by the execution of AVER, consequently the use of constants and expressions is in order.

However, suppose a function called NEXTIN is defined to read the next number from a file, and returns this number in the dummy argument. Furthermore, it is a LOGICAL function and indicates whether the next value was read, by returning .TRUE. if that was the case, or .FALSE. if not. All the numbers on the file can be summed as follows:

```
      S=0
10    IF (NEXTIN(X)) THEN
         S=S+X
         GO TO 10
      ENDIF
```

Scanned by Jonny Oddene for Sintran Data © 2011

NOTE:

In ND FORTRAN this could be written more neatly as:

```
S=0
DO WHILE (NEXTIN(X))
      S=S+X
ENDDO
```

Since NEXTIN returns a value in its dummy argument, it is illegal to write NEXTIN (A+1) or NEXTIN (35*2) as there would be nowhere to set the value that NEXTIN reads.

However, an array can be used for reading into, as follows:

```
      DIMENSION A (1000)
      DO 10 I=1, 1000
         IF (.NOT.NEXTIN(A(I))) GO TO 20
   10 CONTINUE
   20 CONTINUE
C  Here I contains the index beyond the last one read.
```

Upon execution of a function or subroutine reference, an association is established between the corresponding actual and dummy arguments. The first dummy argument becomes associated with the first actual argument, the second with the second and so on.

Argument association may be carried through more than one level of procedure reference.

Argument association within a program unit terminates when a RETURN or END statement in the program unit has been executed.

Length of Character Dummy and Actual Arguments

For a character-type dummy argument, the associated actual argument (also of type character) must have a length equal to or greater than that of the dummy argument. When the lengths differ, if **e** is the length in characters of the dummy argument, then the **e**

Scanned by Jonny Oddene for Sintran Data © 2011

leftmost characters of the actual argument become associated with the dummy argument.

For an array name, the restriction on length is for the entire array and not for each array element.

Example:

In the subroutine:

```
SUBROUTINE PRNAME (NAME)
CHARACTER NAME*20
WRITE (OUT, '(5X, A)') NAME
END
```

there is a character dummy argument that is assumed to be of an exact length of 20, and it will write 20 characters on the file whatever the actual argument.

Thus if we have:

```
CHARACTER ALPHA*26
DATA ALPHA /'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/
```

then:

```
CALL PRNAME (ALPHA (7:))
```

will cause the characters 'G' to 'Z' to be written.

If the intention is to write out exactly the actual argument, then the appropriate declaration of the dummy argument is:

```
CHARACTER NAME*(*)
```

Dummy and Actual Argument of Type NUMERIC

Normally, dummy and actual arguments have to be declared with the same field width and scaling factor. In this case there is no restriction in mixed arithmetic.

Scanned by Jonny Oddene for Sintran Data © 2011

It is possible to declare a dummy argument of type NUMERIC without specifying field width and scaling factor.

Example:

```
SUBROUTINE S(N)
NUMERIC (*) N
```

In this case, the dummy argument may not be used in mixed arithmetic.

11.1.1 Variables as Dummy Arguments

A dummy argument that is a variable may be associated with an actual argument that is a variable, array element, substring, or expression.

The dummy argument may be defined or redefined with the subprogram if the actual argument is:

- a variable name
- an array element name
- a substring name

If, however, the actual argument is:

- a constant (or the symbolic name of a constant)
- a function reference
- an expression involving operators
- an expression enclosed in parentheses

then the dummy argument must not be redefined within the program.

Scanned by Jonny Oddene for Sintran Data © 2011

11.1.2 Arrays as Dummy Arguments

The number and size of dimensions of an array in an actual argument may differ from those of an array in an associated dummy argument.

If the actual argument is an array name, then the association between actual and dummy arguments occurs as if the first element of the actual argument were the actual argument.

If the actual argument is an array element name then the dummy argument is associated with an array whose first element is the actual argument.

The dummy argument must be wholly contained within the actual argument.

Example:

Suppose there is a function defined to compute the arithmetic mean (average) of an array. It contains two dummy arguments, the array and the number of elements in the array.

Thus:

```
      FUNCTION ARMEAN (A, N)
      DIMENSION A (1:N)
C  Add up the array first, then divide by the number of elements
      R=0
      DO 10 I= 1, N
         R=R+A(I)
  10  CONTINUE
      ARMEAN = R/N
      RETURN
      END
```

This function can be used to find the arithmetic mean of whole arrays or parts of them, provided the parts are contiguous.

Scanned by Jonny Oddene for Sintran Data © 2011

If we have, for example:

```
DIMENSION AGES(1:100), SIZES(1:50, 1:50)
```

then the average of all ages is:

```
ALLAGE=ARMEAN (AGES, 100)
```

or, the first 10 ages would be given by:

```
FIRST=ARMEAN (AGES (1), 10)
```

or, the last 10 ages (91 to 100 inclusive):

```
FINAL = ARMEAN (AGES (91), 10)
```

But the mean of the second, fourth, sixth ... etc., elements cannot be computed since they are not contiguous.

When using a two-dimensional (or higher) array, the dummy argument is associated with contiguous locations in the actual argument, i.e. the first subscript varies most rapidly. Thus, clearly:

```
ALLSIZE=ARMEAN (SIZES, 50*50)
```

will compute the mean of all sizes, but:

```
SINGLE=ARMEAN (SIZES (1, 1), 50)
```

will examine

SIZES(1, 1),SIZES(2, 1),SIZES(3, 1)... SIZES(50, 1).

Scanned by Jonny Oddene for Sintran Data © 2011

11.1.3 Procedures as Dummy Arguments

A dummy procedure is a dummy argument identified as a procedure. An example of its use is given below.

Example:

If a routine is required for approximate evaluation of an integral using Simpson's rule on ten intervals, then, for it also to apply to any function supplied by the caller, the definition might be as follows:

```
      FUNCTION SIMPSN (LO, HI, F)
      REAL LO, HI, F
C
C This makes it clear that F is an entry point that can be invoked
C
      EXTERNAL F C Interval size
      H = (HI - LO) / 10.0
C Add up values of functions
      R = F (LO) + F (HI)
      DO 10 I = 1, 9, 2
         R = R + R * F (LO + I * H)
  10  CONTINUE
         DO 20 I = 2, 9, 2
            R = R + 2 * F (LO + I * H)
  20  CONTINUE
C Final calculation
      SIMPSN = R * H/3.0
      RETURN
      END
```

Note that it is not mandatory to have an EXTERNAL statement, in the function SIMPSN, but it is strongly recommended, so as to make the intention clear.

Scanned by Jonny Oddene for Sintran Data © 2011

To evaluate the integral of one of your own functions (i.e. one that you have defined yourself), write:

```
FUNCTION OWN (X)
OWN=(1+X*X) ** (-1)
RETURN
END
```

and call the Simpson routine with:

```
VAL = SIMPSN (0.5, 1.0, OWN)
```

Note that in this case, i.e. defining the function OWN, the program unit containing the call to the function, SIMPSN,in the statement "VAL=...", requires a statement:

```
EXTERNAL OWN
```

If the invocation of SIMPSN is the only place OWN appears in this program unit, the EXTERNAL statement is required to inform the compiler that OWN is the name of an external procedure.

To evaluate the integral using an INTRINSIC function, for example, the trigonometric function SIN from 0 to 1 radians, the following invokes the function SIMPSN:

```
INTRINSIC SIN
QUAD=SIMPSN(0.0,1.0,SIN)
```

This program unit must contain the INTRINSIC statement to use the INTRINSIC SIN function.

The argument passed to the function SIMPSN, is the specific name of the relevant INTRINSIC function, i.e. SIN for a REAL argument giving a REAL result. It is not the generic name SIN which gives access to the variants of SIN for REAL, DOUBLE or COMPLEX type arguments.

Beware that a symbolic name passed as a dummy argument must not occur in both an INTRINSIC and an EXTERNAL statement, within the same program unit. An INTRINSIC statement will cause the supplied functions to be used. An EXTERNAL statement will cause a user written function to be used. This could be a user defined version of a SIN function to be used instead of the supplied function; note that if this is done, the generic name SIN is no longer available in this program unit.

Scanned by Jonny Oddene for Sintran Data © 2011

11.1.4 Asterisks as Dummy Arguments/Alternative Return Arguments

A dummy argument that is an asterisk may appear only in a dummy argument list of a SUBROUTINE statement or ENTRY statement in a subroutine subprogram.

An asterisk dummy argument can only be associated with an actual argument that is an alternate return argument in the relevant CALL statement, see Section 11.7 on page 270.

An alternative return actual argument must be a statement label preceded by an asterisk, as it appears within the argument list of a CALL statement, see Section 11.7.1 on page 270.

Scanned by Jonny Oddene for Sintran Data © 2011

11.2 INTRINSIC FUNCTIONS

INTRINSIC functions are supplied by FORTRAN and have special meanings. The specific names that identify the INTRINSIC functions, their generic names, function definitions, argument type, and result type appear in the table on page 225.

NOTE:

An IMPLICIT statement does not change the type of an INTRINSIC function.

11.2.1 Specific Names and Generic Names

Generic names simplify the referencing of INTRINSIC functions since the same function may be used with more than one argument type.

If a generic name is used to reference an INTRINSIC function, the result type (except for those functions performing type conversion, nearest integer, and the absolute value with a complex argument) is the same as the argument type.

Example:

For the cosine routine, whose generic name is COS, the specific names are COS, DCOS, and CCOS. If I, R, D, and C are variables of type INTEGER, REAL, DOUBLE PRECISION, and COMPLEX respectively, then:

- COS (R) will invoke the routine called COS.
- COS (D) will invoke DCOS since it requests the double precision version.
- COS (C) will invoke the complex version CCOS.

Scanned by Jonny Oddene for Sintran Data © 2011

NOTE:

In ND FORTRAN, COS(I) is allowed causing conversion of I to REAL, since COS is a specific name for the cosine function of REAL arguments.

Only a specific name may be used as an actual argument when the argument is an INTRINSIC function. (However, the names INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR, CHAR, LGE, LGT, LLE, LLT, MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1, IINT, I2INT, DFLOAT, DCMPLX, ININT, I2NINT, I2DNINT, I2ABS, IMOD, I2MOD, I2SIGN, I2DIM, IMAX0, IMIN0, IAND, I2AND, IOR, I2OR, IEOR, I2EOR, NOT, I2NOT, ISHFT, I2SHFT, IBIT, I2BIT, CLBIT, I2CLBIT, STBIT, I2STBIT, GETBF, I2GETBF, PUTBF, I2PUTBF must not be used as actual arguments.)

Otherwise, the actual arguments must agree in order, number, and type with the specifications of the table and may be any expression of the specified type. An actual argument in an INTRINSIC function reference may be any expression except a character expression of unknown length (one involving concatenation of an operand having its length given by an asterisk in parentheses, unless the operand is the symbolic name of a constant).

There is an exception to the above rule in ND FORTRAN. If the name is a specific name, and no generic selection is possible, then conversion of the actual argument is attempted. It is for this reason that COS(I) (see the previous example) is allowed. Other examples are DCOS(R), when R converts to DOUBLE PRECISION, and ALOG(I), when I converts to REAL. LOG(I) is not allowed as LOG is not a specific name.

All INTRINSIC function references except CLBIT and STBIT may have arrays as actual arguments, if the INTRINSIC function reference is part of an array expression. The INTRINSIC function will be evaluated on an element by element basis.

Scanned by Jonny Oddene for Sintran Data © 2011

Example:

```
REAL RES (10), ARG (10)
RES = SIN (ARG)
```

Another way of writing this assignment statement:

```
DO I = 1,10
   RES(I) = SIN (ARG(I))
ENDDO
```

11.2.2 Referencing an INTRINSIC Function

The reference to an INTRINSIC function uses its assigned name as an operand in an arithmetic or logical expression.

On the ND-100 with 48-bit floating-point hardware, short real variables in fact take 6 bytes. In the Table of INTRINSIC Functions, REAL*4 and COMPLEX*8 are used for short real and complex variables on all machines. Consequently, any arguments shown in the table as short real variables, REAL*4 or COMPLEX*8, should be read as REAL*6 and COMPLEX*12 on an ND-100 with 48-bit floating-point hardware. Further, any INTRINSIC function returning a short real result, e.g. ALOG, will return a 6 byte real value on an ND-100 with 48-bit floating-point hardware.

Scanned by Jonny Oddene for Sintran Data © 2011

TABLE OF INTRINSIC FUNCTIONS

INTRINSIC Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to Integer INT(a) See Note 1	1	INT	-	Any arith	Default int.
				IFIX	Real*4	Integer*4
				IDINT	Real*8	Integer*4
				IINT	Any arith	Integer*4
				I2INT	Any arith	Integer*2
	Conversion to Real See Note 2	1	REAL	REAL	Any arith	Real*4
				FLOAT	Integer*4	Real*4
				SNGL	Real*8	Real*4
	Conversion to Double See Note 3	1	DBLE	DBLE	Any arith	Real*8
				DFLOAT	Integer*4	Real*8
	Conversion to Complex See Note 4	1 or 2	CMPLX	CMPLX	Any arith	Complex*8
				DCMPLX	Any arith	Complex*16
	Conversion to Integer See Note 5	1	-	ICHAR	Character*1	Default int.
	Conversion to Character See Note 5	1	-	CHAR	Integer*2	Character*1
Truncation	INT(a) See Note 1	1	AINT	AINT	Real*4	Real*4
				DINT	Real*8	Real*8
Nearest Whole Number	INT(a+.5) if a>0 INT(a-.5) if a<0	1	ANINT	ANINT	Real*4	Real*4
				DNINT	Real*8	Real*8
Nearest Integer	INT(a+.5) if a>0 INT(a-.5) if a<0	1	NINT	-	Real*4	Default int.
				ININT	Real*4	Integer*4
				I2NINT	Real*4	Integer*2
				IDNINT	Real*8	Integer*4
				I2DNINT	Real*8	Integer*2

Scanned by Jonny Oddene for Sintran Data © 2011

INTRINSIC Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Absolute Value	$\lvert a \rvert$ See Note 6 $(ar^2+ai^2)^{1/2}$	1	ABS	IABS	Integer*4	Integer*4
				I2ABS	Integer*2	Integer*2
				ABS	Real*4	Real*4
				DABS	Real*8	Real*8
				CABS	Complex*8	Real*4
				CDABS	Complex*16	Real*8
Remainder	$a_1-INT(a_1/a_2)*a_2$ See Note 1	2	MOD	-	Default int	Default int
				IMOD	Integer*4	Integer*4
				I2MOD	Integer*2	Integer*2
				AMOD	Real*4	Real*4
				DMOD	Real*8	Real*8
Transfer of Sign	$\lvert a_1 \rvert$ if $a_2 \geq 0$ $-\lvert a_1 \rvert$ if $a_2<0$	2	SIGN	ISIGN	Integer*4	Integer*4
				I2SIGN	Integer*2	Integer*2
				SIGN	Real*4	Real*4
				DSIGN	Real*8	Real*8
Positive Difference	a_1-a_2 if $a_1>a_2$ 0 if $a_1 \leq a_2$	2	DIM	IDIM	Integer*4	Integer*4
				I2DIM	Integer*2	Integer*2
				DIM	Real*4	Real*4
				DDIM	Real*8	Real*8
Double Precision Multiply	DBLE(a_1)*DBLE(a_2)	2	DPROD	DPROD	Real*4	Real*8
Choosing Maximum Value	max(a_1,a_2,...) See Note 9	≥ 2	MAX	MAX0	Default int	Default int
				IMAX0	Integer*4	Integer*4
				I2MAX0	Integer*2	Integer*2
				AMAX1	Real*4	Real*4
				DMAX1	Real*8	Real*8
			-	AMAX0	Default int	Real*4
			-	MAX1	Real*4	Default int
Choosing Minimum Value	min(a_1,a_2,...) See Note 9	≥ 2	MIN	MIN0	Default int	Default int
				IMIN0	Integer*4	Integer*4
				I2MIN0	Integer*2	Integer*2
				AMIN1	Real*4	Real*4
				DMIN1	Real*8	Real*8
			-	AMIN0	Default int	Real*4
			-	MIN1	Real*4	Default int

Scanned by Jonny Oddene for Sintran Data © 2011

INTRINSIC Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Length	Length of Character Entity See Note 11	1	LEN	LEN	Character	Default int
Index of a Substring	Location of Substring a_2 in String a_1 See Note 10	2	INDEX	INDEX	Character	Default int
Imaginary Part of Complex Arguments	ai See Note 6	1	IMAG	AIMAG DIMAG	Complex*8 Complex*16	Real*4 Real*8
Conjugate of a Complex Argument	(ar,-ai)	1	CONJG	CONJG CDCONJG	Complex*8 Complex*16	Complex*8 Complex*16
Square Root	$\sqrt{a}$ See Note 8	1	SQRT	SQRT DSQRT CSQRT CDSQRT	Real*4 Real*8 Complex*8 Complex*16	Real*4 Real*8 Complex*8 Complex*16
Exponential	e**a	1	EXP	EXP DEXP CEXP CDEXP	Real*4 Real*8 Complex*8 Complex*16	Real*4 Real*8 Complex*8 Complex*16
Natural Logarithm	log(a) See Note 8	1	LOG	ALOG DLOG CLOG CDLOG	Real*4 Real*8 Complex*8 Complex*16	Real*4 Real*8 Complex*8 Complex*16
Common Logarithm	log10(a)	1	LOG10	ALOG10 DLOG10	Real*4 Real*8	Real*4 Real*8
Logarithm (base 2)	log2(a)	1	LOG2	ALOG2 DLOG2	Real*4 Real*8	Real*4 Real*8

Scanned by Jonny Oddene for Sintran Data © 2011

INTRINSIC Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Sine	sin(a) See Notes 7,8	1	SIN	SIN DSIN CSIN CDSIN	Real*4 Real*8 Complex*8 Complex*16	Real*4 Real*8 Complex*8 Complex*16
Cosine	cos(a) See Notes 7,8	1	COS	COS DCOS CCOS CDCOS	Real*4 Real*8 Complex*8 Complex*16	Real*4 Real*8 Complex*8 Complex*16
Tangent	tan(a) See Note 7	1	TAN	TAN DTAN	Real*4 Real*8	Real*4 Real*8
Arcsine	arcsine(a) See Note 7	1	ASIN	ASIN DASIN	Real*4 Real*8	Real*4 Real*8
Arccosine	arccosin(a) See Note 7	1	ACOS	ACOS DACOS	Real*4 Real*8	Real*4 Real*8
Arctangent	arctan(a) arctan(a_1,a_2) See Note 7	1 2	ATAN ATAN2	ATAN DATAN ATAN2 DATAN2	Real*4 Real*8 Real*4 Real*8	Real*4 Real*8 Real*4 Real*8
Hyperbolic Sine	sinh(a)	1	SINH	SINH DSINH	Real*4 Real*8	Real*4 Real*8
Hyperbolic Cosine	cosh(a)	1	COSH	COSH DCOSH	Real*4 Real*8	Real*4 Real*8
Hyperbolic Tangent	tanh(a)	1	TANH	TANH DTANH	Real*4 Real*8	Real*4 Real*8
Lexically Greater Than or Equal	$a_1 \geq a_2$ See Note 12	2	-	LGE	Character	Default logical
Lexically Greater than	$a_1 > a_2$ See Note 12	2	-	LGT	Character	Default logical

Scanned by Jonny Oddene for Sintran Data © 2011

INTRINSIC Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Lexically Less Than or Equal	$a_1 \leq a_2$ See Note 12	2	-	LLE	Character	Default logical
Lexically Less than	$a_1 < a_2$ See Note 12	2	-	LLT	Character	Default logical
AND	arg1.AND.arg2 See Note 13	2	IAND	IAND I2AND	Integer*4 Integer*2	Integer*4 Integer*2
OR	arg1.OR.arg2 See Note 13	2	IOR	IOR I2OR	Integer*4 Integer*2	Integer*4 Integer*2
Exclusive OR	arg1.NEQV.arg2 See Note 13	2	IEOR	IEOR I2EOR	Integer*4 Integer*2	Integer*4 Integer*2
NOT	logical complement	1	NOT	NOT I2NOT	Integer*4 Integer*2	Integer*4 Integer*2
Bit Shifting	shifts value left or right See Note 14	2	ISHFT	ISHFT I2SHFT	Integer*4 Integer*2	Integer*4 Integer*2
Bit Extract	0 if bit arg2 of arg1 is 0,else -1 See Note 15	2	IBIT	IBIT I2BIT	Integer*4 Integer*2	Integer*4 Integer*2
Clear bit	sets bit arg2 of arg1 to 0 See Note 15	2	CLBIT	CLBIT I2CLBIT	Integer*4 Integer*2	- -
Set bit	sets bit arg2 of arg1 to 1 See Note 15	2	STBIT	STBIT I2STBIT	Integer*4 Integer*2	- -
Get bit field	See Notes 15,16	3	GETBF	GETBF I2GETBF	Integer*4 Integer*2	Integer*4 Integer*2
Set bit field	See Notes 15,16	4	PUTBF	PUTBF I2PUTBF	Integer*4 Integer*2	Integer*4 Integer*2

Scanned by Jonny Oddene for Sintran Data © 2011

NOTES ON TABLE OF INTRINSIC FUNCTIONS

(1) For a of type integer, INT (a) = a. For a of type real or double precision, there are two cases:

- if $|a| < 1$, INT(a)=0:
- if $|a| > 1$, INT(a)

is the integer whose magnitude is the largest integer that does not exceed the magnitude of **a** and whose sign is the same as the sign of **a**. For example:

INT (-3.7) = -3

For **a** of type complex, INT (a) is the value obtained by applying the above rule to the real part of **a**.

For **a** of type real, IFIX (a) is the same as INT (a).

The result of INT is the default integer type for this compilation. (See "DEFAULT command" in the ND FORTRAN User Guide, section 3.24.1)

To convert to INTEGER*2, use I2INT, and to INTEGER*4, use IINT.

(2) For **a** of type real, REAL (a) is **a**. For **a** of type integer or double precision, **a** is converted to type REAL. If significant bits are lost, the result is truncated. If **a** has type complex, REAL (a) is the real part of **a**.

If **a** is of type integer, FLOAT (a) is the same as REAL (a).

(3) For **a** of type double precision, DBLE (a) = **a** .
For **a** type of integer or real, the result is converted to double precision so that no significant bits can be lost in conversion.

Scanned by Jonny Oddene for Sintran Data © 2011

(4) CMPLX may have one or two arguments. If there is one argument, it may be of type integer, real, double precision, or complex. If there are two arguments, they may be of type integer, real, or double precision, but must both be of the same type.

If **a** has type complex, CMPLX (a) = a. For **a** of type integer, real, or double precision, CMPLX (a) is the complex value whose real part is REAL (a) and whose imaginary part is zero.

CMPLX (a_1 , a_2) is the complex value whose real part is REAL (a_1) and whose imaginary part is REAL (a_2).

DCMPLX acts analogously with a result of DOUBLE COMPLEX.

(5) ICHAR provides a means of determining the position of a character in the collating sequence, which for ND machines is the ASCII sequence. There are 128 values in this sequence. For example, the letter A is number 65, and the first (NUL) is 0 (zero).

The argument **a** is a character of length 1.

In ND FORTRAN, if the length is greater than 1, the first character is used. No check is made to see if the value is more than 127; hence care should be taken if this is used on values where the parity bit has not been cleared.

For any characters **C1** and **C2**, (C1.LE.C2) is true if and only if (ICHAR (c_1)) .LE.ICHAR (c_2)) is true, and (c_1 .EQ.c_2) is true if and only if (ICHAR (c_1).EQ.ICHAR (c_2)) is true.

The result is of default integer type (see the ND FORTRAN User Guide, section 3.24.1).

CHAR (i) returns the character in the **i**th position of the collating sequence. The value is of type character of length one. **i** must be an integer expression whose

Scanned by Jonny Oddene for Sintran Data © 2011

value must be in the range 0<**i**<128. In ND FORTRAN, no check is made that the integer is in the restricted range; hence care must be taken if parity bits are being manipulated.

ICHAR (CHAR (i)) = i for 0<i<128 CHAR (ICHAR (c)) = c

for any character c.

(6) A complex value is expressed as an ordered pair of reals, (ar, ai), where **ar** is the real part and **ai** is the imaginary part.

(7) All angles are expressed in radians.

(8) The result of a function of type complex is the principal value.

(See the restrictions which follow these notes, on page 234.)

(9) All arguments in an INTRINSIC function reference must be of the same type.

In ND FORTRAN, arguments are automatically converted to the highest order data type, see Section 5.1.1, on page 89,according to the normal rules of operand matching for arithmetic operators.

(10) INDEX (a_1 , a_2) returns an integer value indicating the starting position within the character string $\mathbf{a}_1$ of a substring identical to string $\mathbf{a}_2$. If $\mathbf{a}_2$ occurs more than once in $\mathbf{a}_1$, the starting position of the first occurrence is returned.

If $\mathbf{a}_2$ does not occur in $\mathbf{a}_1$, the value zero is returned. Note that zero is returned if LEN (a_1)<LEN (a_2). Zero is also returned if the second argument is a null string.

Scanned by Jonny Oddene for Sintran Data © 2011

(11) The value of the argument of the LEN function need not be defined at the time the function reference is executed.

(12) LGE (a_1, a_2) returns the value .TRUE. if $a_1 = a_2$ or if a_1 follows a_2 in the collating sequence described in American National Standard Code for Information Interchange, ANSI X3.4-1977 (ASCII). Otherwise, it returns the value .FALSE..

LGT (a_1, a_2) returns the value .TRUE. if a_1 follows a_2 in the collating sequence described in ANSI X3.4 - 1977 (ASCII), and otherwise returns the value .FALSE..

LLE (a_1, a_2) returns the value .TRUE. if $a_1 = a_2$ or if a_1 precedes a_2 in the collating sequence in ANSI X3.4-1977 (ASCII), and otherwise returns the value .FALSE..

LLT (a_1, a_2) returns the value .TRUE. if a_1 precedes a_2 in the collating sequence described in ANSI X3.4 - 1977 (ASCII), and otherwise returns the value .FALSE..

If the operands for LGE, LGT, LLE, and LLT are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to match the length of the longer operand.

In ND FORTRAN, the comparison is done character by character, treating each one as an unsigned 8-bit integer.

(13) The logical operators are defined for integers in ND FORTRAN by applying the operator to each bit position of the arguments independently. These functions have identical results to the logical operators, see Section 5.4 on page 99.

(14) ISHFT (argl, arg2) shifts the bits in **argl** by **arg2** positions. If **arg2** is positive, the shift is to the left (i.e. towards the highest order bit). If **argl** is negative, the shift is to the right. In both cases, zeros are moved into the vacated bit positions.

For **argl** of type INTEGER*2, $-16 \leq$ arg2 ≤ 16 and for **argl** of type INTEGER*4, $-32 \leq$ **arg2** ≤ 32.

Scanned by Jonny Oddene for Sintran Data © 2011

(15) Bits are counted from the rightmost (least significant) bit, which is labeled 0. The leftmost bit is number 15 for INTEGER*2, and 31 for INTEGER*4.

The entry points CLBIT, I2CLBIT, STBIT, I2STBIT are subroutine entries. They may only be invoked by a CALL statement, and they return no value.

(16) GETBF and PUTBF may only be used in ND-500.

GETBF: The first argument specifies the operand where the bit field is taken from. The second argument defines the bit number where the bit field starts. The third argument specifies the number of bits in the bit field.

```
DEST = GETBF (SOURCE1, STARTB, BTWIDTH)
```

PUTBF: The second and third arguments specify the bit field as in GETBF. The fourth argument holds the bits that that are going to be stored in the first argument's bit map.

```
DEST = PUTBF (SOURCE1, STARTB, BTWIDTH, SOURCE2)
```

RESTRICTIONS ON RANGE OF ARGUMENTS AND RESULTS

Restrictions on the range of arguments and results for INTRINSIC functions when referenced by their specific names are as follows:

(1) Remainder: The result for MOD, AMOD, and DMOD is un-defined when the value of the second argument is zero.

(2) Transfer of Sign: If the value of the first argument of ISIGN, SIGN, or DSIGN is zero, the result is zero, which is neither positive nor negative.

Scanned by Jonny Oddene for Sintran Data © 2011

(3) Square Root: The value of the argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

(4) Logarithms: The value of the argument of ALOG, DLOG, ALOG10, DLOG10, ALOG2 and DLOG2 must be greater than zero. The value of the argument of CLOG must not be (0., 0.). The range of the imaginary part of the result of CLOG is:

$$pi < \text{imaginary part} \leq pi.$$

The imaginary part of the result is pi only when the real part of the argument is less than zero and the imaginary part of the argument is zero.

(5) Trigonometric functions: The values of the complex circular functions are defined as follows:

If z is complex and z = x+iy, where x, y are real, then:

$$\sin(z) = \sin(x)\cosh(y) + i\cos(x)\sinh(y)$$
$$\cos(z) = \cos(x)\cosh(y) - i\sin(x)\sinh(y)$$

The following INTRINSIC functions are ND FORTRAN extensions:

double complex functions:
CDSIN, CDCOS, CDEXP, CDLOG, CDSQRT, CDABS, CDCONJG, DCMPLX, DIMAG

log to base 2:
LOG2, ALOG2, DLOG2

integer*2 functions:
I2MIN0, I2MAX0, I2MOD, I2ABS, I2SIGN, I2DIM

integer*4 functions:
IMIN0, IMAX0, IMOD

Scanned by Jonny Oddene for Sintran Data © 2011

```
conversions:
I2NINT, IINT, I2NINT, ININT, I2DNINT, DFLOAT

generic names:
IMAG

bit operations:
IAND, I2AND, IOR, I2OR, IEOR, I2EOR, ISHFT, I2SHFT, NOT,
I2NOT, IBIT, I2BIT, CLBIT, I2CLBIT, STBIT, I2STBIT, GETBF,
PUTBF.
```

11.3 STATEMENT FUNCTIONS

A statement function is a procedure specified by a single statement that is similar in form to an arithmetic, logical, or character assignment statement. This statement is nonexecutable and not part of the normal execution sequence. However, these definitions must follow all declaratives and precede executable statements.

The general form of a statement function is:

```
symb ([arg1, arg2, ...])=e
```

where

symb is the symbolic name of the statement function.

arg is a dummy argument.

e is an expression.

The relationship between symb and e must conform to the standard assignment rules, see Chapter 7 on page 123.

Example:

The Euclidean distance between points (X1, Y1) and (X2, Y2) could be represented by:

Scanned by Jonny Oddene for Sintran Data © 2011

```
DIST(X1, Y1, X2, Y2) = SQRT((X1-X2)**2+(Y1-Y2)**2)
```

To use this function to evaluate the distance between the i-th and j-th points represented by arrays A (for the X's) and B (for the Y's), the following could be written:

```
DIST(A(I), B(I), A(J), B(J))
```

It is not necessary to restrict the operands in the expression to the statement function's dummy arguments. As an example, suppose the values A, B and C are defined in a COMMON block, then the evaluation of a quadratic expression with these coefficients could be defined as:

```
QUADR(X) = (A*X+B) * X+C
```

The dummy argument names have the scope of the statement function only.

A statement function produces only one value, that is, the result of the expression it contains.

The actual arguments must agree in order, number, and type with the corresponding dummy arguments. An actual argument may be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is the symbolic name of a constant.

In ND FORTRAN, however, conversion will be carried out, where possible, if the actual and dummy arguments are not of the same type.

11.3.1 Statement Function Restrictions

A statement function may be referenced only in the program unit that contains the statement function statement.

A statement function statement must not reference another statement function which appears in subsequent lines of the program unit.

The symbolic name identifying the statement function must not be used as a symbolic name in any specification statement (except in a Type statement for specifying the type of function) or as the name of a

Scanned by Jonny Oddene for Sintran Data © 2011

common block in the same program unit.

An external function reference in the expression **e** must not cause a dummy argument of the associated statement function to become undefined or redefined.

The symbolic name of a statement function is a local name, see Section 1.3 on page 6, and must not be the same as the name of any other entity in the program unit except the name of a common block.

The symbolic name of a statement function may not be an actual argument. It must not appear in an EXTERNAL statement.

In ND FORTRAN, if an actual argument differs in type from the corresponding dummy argument, then the actual arguments are converted to the type of the dummy argument.

A statement function statement in a function subprogram must not contain a reference to the name of the function subprogram or an entry name in the function subprogram.

11.3.2 Referencing a Statement Function

A statement function is referenced by using its function reference in an expression.

Note that if a statement function has no dummy arguments, its definition and reference must still include empty argument lists.

For example:

```
LOGICAL CONSEC
CONSEC () = ABS (X-Y) .EQ. 1
IF (CONSEC ()) THEN
ENDIF
```

Scanned by Jonny Oddene for Sintran Data © 2011

11.4 EXTERNAL FUNCTIONS

External functions are both external procedures and function subprograms. They consist of a FUNCTION statement followed by a sequence of FORTRAN statements which define desired operations. They may also contain one or more RETURN statements and must be terminated by an END statement.

The form of a FUNCTION statement is:

```
[type] FUNCTION name [([ arg1 [,arg2] ...])]
```

where

type is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, NUMERIC (fw,sc) or CHARACTER [*I] where **I**is the length of the result. **I**may have any of the forms of those allowed in the CHARACTER statement, except that an integer constant expression must not include the symbolic name of a constant. The default for **I**is one.

NOTE:

Type may be specified in a Type statement instead. The normal implicit rules apply if neither form is used.

Name is the symbolic name of the function subprogram in which the FUNCTION statement appears; it is an external function name.

arg is a dummy argument.

The symbolic name of the function subprogram must appear as a variable name in this subprogram. Its value on execution of a RETURN or END statement in the subprogram is the value of the function.

An external function may also modify one or more of its dummy arguments.

Scanned by Jonny Oddene for Sintran Data © 2011

11.4.1 Actual Arguments for an External Function

Actual arguments in the function reference must agree in order, number and type with the corresponding dummy arguments in the referenced function. An exception to this rule is when a subroutine name appears as an actual argument, because subroutine names do not have a type.

Actual arguments may be:

- An expression. (Except a character expression of unknown length, e.g. one involving concatenation of an operand whose length is given by an asterisk in parentheses, unless the operand is the symbolic name of a constant.)
- An array name.
- An intrinsic function name.
- An external procedure name.
- A dummy procedure name (see Section 11.1.3, on page 241).

Note that an actual argument in a function reference may be a dummy argument in a dummy argument list within the same subprogram.

11.4.2 Function Subprogram Restrictions

A FUNCTION statement should only appear as the first statement of a function subprogram. A function subprogram may contain any other statement except a BLOCK DATA, SUBROUTINE, or PROGRAM statement.

The symbolic name of an external function is a global name and cannot, therefore, be the same as any other global name.

In ND FORTRAN however, COMMON blocks can have the same names as external procedures.

Scanned by Jonny Oddene for Sintran Data © 2011

A function specified in a subprogram may be referenced in any other procedure subprogram or in the main pro-gram. A function subprogram must not reference itself, either directly or indirectly.

In ND FORTRAN, if the REENTRANT-MODE ON command on ND-100 or the FIXED-DATA-AREA OFF command on ND-500 is chosen, recursive calls are allowed for external procedures. Note that recursion is never allowed for statement functions.

If a function has no dummy arguments, its FUNCTION statement may omit the argument list. But its reference may not omit the list.

For example:

```
CHARACTER*1 FUNCTION NEXTCH
READ (IN, '(A1)') NEXTCH
RETURN
END
```

then the form of the invocation must be, as in this example:

```
IF (NEXTCH().EQ.'+') GO TO 10
```

Scanned by Jonny Oddene for Sintran Data © 2011

11.5 SUBROUTINES

A subroutine is an external procedure which is identified by a SUBROUTINE statement.

The SUBROUTINE statement must be the first statement of the subroutine subprogram and it has the form:

```
SUBROUTINE name [([arg1 [,arg2] ...])]
```

where

name is the symbolic name of the subroutine subprogram in which the SUBROUTINE statement appears.

arg1... is a dummy argument list consisting of variable names, array names, or procedure names. A dummy argument can also be an asterisk, see page 243.

NOTE:

If there are no dummy arguments, either of the forms, name or name(), can be used in the SUBROUTINE statement. Likewise a subroutine can be referenced, according to the form in which it was specified, by CALL name or CALL name().

Scanned by Jonny Oddene for Sintran Data © 2011

11.5.1 Subroutine Reference

A subroutine is referenced by the CALL statement which has the form:

```
CALL name [([arg1 [,arg2] ...])]
```

where

name is the symbolic name of the subroutine subprogram.

arg1... is an optional list of actual arguments.

A subroutine specified in a subprogram may be referenced within any other procedure subprogram or within the main program. A subroutine subprogram must not reference itself, either directly or indirectly.

In ND FORTRAN, if the REENTRANT-MODE ON command on an ND-100, or the FIXED-DATA-AREA OFF command on an ND-500 are chosen, a subroutine subprogram may reference itself, either directly or indirectly.

The use of a subroutine name or an alternate return specifier (see the RETURN statement on page 270, in this chapter) as an actual argument is an exception to the rule requiring agreement of type between dummy and actual arguments.

Note that an actual argument may be a dummy argument name that appears in a dummy argument list within the subprogram containing the reference. An asterisk dummy argument must not be used as an actual argument in a subprogram reference.

An actual argument may be an array expression. The dimension bounds in each array must be constants.

Scanned by Jonny Oddene for Sintran Data © 2011

11.5.2 Subroutine Subprogram Restrictions

A subroutine subprogram may contain any other statement except a BLOCK DATA, FUNCTION, or PROGRAM statement.

The symbolic name is a global name, and cannot, therefore, be the same as any other global name. See the examples earlier in this chapter.

11.6 THE ENTRY STATEMENT

The ENTRY statement enables additional entry points into an external subprogram to be specified. It may appear anywhere within a function or subroutine subprogram. However, it may not appear between a block IF and its corresponding ENDIF statement, or between a DO statement and the terminal statement of its DO-loop.

An ENTRY statement is nonexecutable. It has the form:

```
ENTRY name [([arg1 [,arg2]...])]
```

where

name is the symbolic name of an entry in a function or subroutine program and is known as an entry name.

arg1... is an optional list of dummy arguments which may be variable names, array names, dummy procedure names, or an asterisk. This last argument type is permitted only in a subroutine subprogram. A dummy procedure is defined in Section 11.1.3, on page 241.

Note that if there are no dummy arguments, either *name* or *name()* can be used in the ENTRY statement. A function specified by either form must be referenced by *name()*. A subroutine specified by either form can be referenced by a CALL statement, using either CALL name or CALL name().

The symbolic entry name may appear in a Type statement.

Scanned by Jonny Oddene for Sintran Data © 2011

Execution of the external procedure begins with the first executable statement following the relevant ENTRY statement.

An entry name can be referenced in any program unit except the one containing the entry name in an ENTRY statement.

The order, number, type, and names of the dummy arguments of the ENTRY statement may be different from the order, number, type, and names of the dummy arguments in the FUNCTION or SUBROUTINE statements and other ENTRY statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the corresponding dummy argument list of the FUNCTION, SUBROUTINE, or ENTRY statement. The use of a subroutine name or alternate return specifier (see Section 11.7, on page 270) as an actual argument is an exception to the rule requiring agreement of type.

For any particular subroutine call or function invocation at one of its entry points, only those dummy arguments specified at the point of entry can be assumed to have a value during this call/invocation.

For example:

```
SUBROUTINE SUB
    .
    .
    .
ENTRY INIT (A,B,C)
    .
    .
    .
ENTRY LOOKUP (A,X)
    .
    .
    .
ENTRY STORE (Y,A)
    .
    .
    .
END
```

If the routine was entered at entry INIT, the dummy arguments A, B and C have values. But if the entry is via LOOKUP, then only A and X, but not B nor C, will have values. Any access to B and C will give unpredictable results.

Scanned by Jonny Oddene for Sintran Data © 2011

Note that in entry STORE, the dummy argument A is defined at a different position in the list.

Within a function subprogram, all variables whose names are also the names of entries are associated with each other and with the variable whose name is also the name of the function subprogram. Therefore, any such variable that becomes defined, causes all associated variables of the same type to become defined and all associated variables of different types to become undefined.

11.6.1 ENTRY Statement Restrictions

An entry name cannot be used as a dummy argument in a FUNCTION, SUBROUTINE or ENTRY statement within the same subprogram in which it appears as an entry name. It must not appear in an EXTERNAL statement.

In ND FORTRAN, if the REENTRANT-MODE ON command on an ND-100, or the FIXED-DATA-AREA OFF command on an ND-500 is chosen, recursion is allowed.

In a function subprogram, a variable having the same name as the entry name, must not appear in any statement preceding the ENTRY statement associated with the entry name (except in a Type statement).

In a subprogram, a name used as a dummy argument in an ENTRY statement, must not appear in an executable statement preceding that ENTRY statement except in a FUNCTION, SUBROUTINE or ENTRY statement. Likewise, it must not appear in the expression of a statement function statement, unless the name is also a dummy argument of the statement function, or appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY statement preceding the statement function statement.

If a dummy argument does appear in an executable state-ment, such execution is only permitted if, during the execution of a reference to the function or subroutine, the dummy argument is in the list of dummy arguments of the procedure name referenced.

For example, if two functions have very similar algorithms barring some initialization code, then they can be written in one external function with a second entry point. The following function with two entry points searches an array of integers, IA, for either an odd or

Scanned by Jonny Oddene for Sintran Data © 2011

even value, depending on the entry point.

```
      FUNCTION IODD (IA, N)
      IODD = 0
      M = 1
      GO TO 10

      ENTRY IEVEN (IA, N)
      IEVEN = 0
      M = 0

C  Common code starts here
  10  CONTINUE
      DO 20 K=1, N
      IF (MOD (IA (K), 2).EQ.M) THEN
         IF (M.EQ.1) IODD = K
         IF (M.EQ.0) IEVEN = K
         RETURN
      ENDIF
  20  CONTINUE
      END
```

Scanned by Jonny Oddene for Sintran Data © 2011

11.7 THE RETURN STATEMENT

The RETURN statement causes control to be returned from a subprogram to the calling program unit. The form of a RETURN statement in a function subprogram is:

```
RETURN
```

The form of a RETURN statement in a subroutine sub-program is:

```
RETURN [e]
```

where

e is an integer expression.

During execution, the value of the expression *e* will select one of the alternative RETURN actual arguments specified in the relevant CALL statement, see the next section.

11.7.1 Execution of a RETURN Statement

Execution of a RETURN statement in the first of the above forms causes control to be returned to the statement of the calling program following the statement that calls the subprogram.

Execution of a RETURN statement in the second form given above also returns control to the referencing program unit and it completes the execution of the CALL statement whether **e** is specified or not.

Scanned by Jonny Oddene for Sintran Data © 2011

However, if:

$1 \leq e \leq n$

where

n is the number of asterisks in the SUBROUTINE or subroutine ENTRY statement, then *e* identifies the *e*-th asterisk in the dummy argument list. Control will now be returned to the *e*-th statement label in the argument list of the calling statement. This is known as the alternate return specifier.

Example:

The following subprogram checks a number for validity (in this case it tests for integers from 1 to 10) and takes an alternate return if it fails.

```
      SUBROUTINE VALCHK (X,*,*)
      IF (X.LT.1.OR.X.GT.10) RETURN 1
      IF (INT (X).NE.X) RETURN 2
      END
```

and it can be used as follows:

```
      CALL VALCHK (TYPE, * 90, * 91)
C  Okay in normal continuation
......
  90  STOP 'OUT OF BOUNDS'
  91  STOP 'NOT AN INTEGER'
```

Execution of a RETURN (or END) statement causes all entities within the subprogram in which it occurs to become undefined except for the following:

- Entities specified by SAVE statements.
- Entities in blank common.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 12

MAIN PROGRAM

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

A main program is a program unit which does not have a FUNCTION, SUBROUTINE, or BLOCK DATA statement as its first statement.

An executable program must have exactly one main program in it and the first executable statement of the main program is also the first executable statement of the executable program which contains it.

12.1 THE PROGRAM STATEMENT

The form of a PROGRAM statement is:

```
PROGRAM pgm
```

where

pgm is the symbolic name of the main program in which the PROGRAM statement appears.

A PROGRAM statement is not mandatory, but if it appears, it must be the first statement of the main program.

In ND FORTRAN, if the PROGRAM statement is omitted, the name #MAIN is generated.

Since the symbolic name, **pgm**, is global, it must not be used as any local name within the main program, neither may it be the name of an external procedure, BLOCK DATA subprogram, or COMMON block of the executable program in which it appears.

Scanned by Jonny Oddene for Sintran Data © 2011

A main program may not be referenced from a subprogram or from itself.

In ND FORTRAN, a program which is to be executed as an RT -program can be given a priority at compile-time by the following form of the statement:

```
PROGRAM pgm, priority
```

priority is an unsigned integer constant which must be less than 256. This is the default priority assigned by the RT loader to this program. The **priority** is ignored in backgrounds operation.

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 13

BLOCK DATA SUBPROGRAM

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

BLOCK DATA subprograms are used to provide initial values for variables and arrays in named common blocks. They are nonexecutable and more than one may appear in an executable program. The first statement of a BLOCK DATA subprogram is a BLOCK DATA statement which has the form:

```
BLOCK DATA [sub]
```

where

sub is the symbolic name of the BLOCK DATA subprogram in which the statement appears.

The optional name *sub* is global and cannot therefore be the same name as that of an external procedure, main program, COMMON block, or other BLOCK DATA subprogram in the same executable procedure. Neither can it be the same as any local name in the subprogram.

Initial values may be entered into more than one labelled COMMON block in a single subprogram of this type.

13.1 BLOCK DATA SUBPROGRAM RESTRICTIONS

The only other statements that can appear in a BLOCK DATA subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, END and Type statements. Comment lines are permitted.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

CHAPTER 14

ADVANCED FORTRAN PROGRAMMING

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

14.1 EFFICIENT PROGRAMMING TECHNIQUES

14.1.1 Loops

In most cases a DO loop will execute faster than a loop coded with IF statements and labels. The optimizing techniques used by the compiler are applied fully to DO loops, but not constructed loops.

Thus:

```
      DO 10 I = 1,100
10 SUM = SUM + A(I) * B(I)
```

is better than:

```
      I = 1
10    SUM = SUM + A(I) * B(I)
      I = I + 1
      IF (I .LE. 100) GO TO 10
```

If the looping has no natural counter for use as a control variable, then the DO WHILE should be used.

Thus:

```
      I = 1
      DO WHILE (A(I) .GT. 0.0)
         SUM = SUM + A(I) * B(I)
         I = I + 1
      ENDDO
```

Scanned by Jonny Oddene for Sintran Data © 2011

is better than:

```
      I = 1
10    IF (A(I) .GT. 0.0) GO TO 20
      SUM = SUM + A(I) * B(I)
      I = I + 1
      GO TO 10
20    CONTINUE
```

14.1.2 Loop Control Variable

A loop control variable of type INTEGER*2 will execute fastest on the ND-100, and a loop control variable of type INTEGER*4 will execute fastest on the ND-500. This is followed by INTEGER*4, REAL*4 and REAL*8 on the ND-100 and INTEGER*2, REAL*4 and REAL*8 on the ND-500. Note however, that if the natural control variable is, say, REAL, it should be used since what is gained in speed of control of the loop may be lost in doing more conversions, e.g. from an INTEGER type to the working value that is required.

On the ND-500, the differences are much less marked than on the ND-100.

14.1.3 Array Operations

- **Example 1 : Filling an array**

```
REAL A(100)
DO I = 1,100
   A(I) = 0.0
ENDDO
```

The DO-loop may be substituted by the array operation:

```
A = 0.0
```

Scanned by Jonny Oddene for Sintran Data © 2011

• **Example 2 : Moving an array**

```
REAL A(100), B(100)
DO I = 1,100
   A(I) = B(I)
ENDDO
```

The DO-loop may be substituted by:

```
A = B
```

which will execute faster

• **Example 3 : Subtraction of arrays**

```
REAL A(100), B(100), C(100)
DO I = 1,100
   A(I) = B(I) - C(I)
ENDDO
```

This may be substituted by:

```
A = B - C
```

which will be executed faster if the command USE-APF-LIBRARY ON is given.

14.1.4 Actual Argument Data Types

If there is any doubt about the data type of an expression in a subroutine call, it should be explicitly converted to the desired type by using the INTRINSIC functions, see Section 11.2. on the ND FORTRAN Reference Manual, ND-60.145. If the expression is of the correct type, there is no overhead involved, but the program is more explicit and more easily understood, which is important for later maintenance.

Scanned by Jonny Oddene for Sintran Data © 2011

Thus, if a REAL argument is needed, then:

```
INTEGER*2 I
REAL R
CALL SUBR (REAL(I + R))
```

makes it clear that a REAL argument is actually being used.

If the argument is a constant, then it can be forced to the appropriate type by using the PARAMETER statement. This defines the constant and gives an associated name. If the value is to be modified later, then only the PARAMETER statement needs to be altered.

For example:

```
INTEGER*2 LOWEST
PARAMETER (LOWEST = -32768)
CALL TEST (LOWEST)
```

14.1.5 CHARACTER and Hollerith

Since Hollerith values in FORTRAN vary greatly from one manufacturer to another, their use should be avoided if the program is to be portable without potential difficulties. The CHARACTER data type should be used instead.

Thus, the definition:

```
INTEGER*4 TITLE (5)
DATA TITLE/4HALPH,4HABET,4HICAL,4H ORD,4HER /
```

should be replaced by:

```
CHARACTER*20 TITLE
DATA TITLE/'ALPHABETICAL ORDER'/
```

Scanned by Jonny Oddene for Sintran Data © 2011

This is strongly recommended, even for transferring between the ND-100 and the ND-500 because of different word lengths and defaults.

14.1.6 CHARACTER Alignment - ND-100

Some Monitor Calls in SINTRAN require that data areas begin on a word boundary. A CHARACTER variable can be forced on to a word boundary by using an equivalence to an INTEGER variable.

For example:

```
CHARACTER*400 C
INTEGER*2 IC
EQUIVALENCE (C,IC)
```

will force the variable C to be word aligned.

14.1.7 File Accessing

Wherever possible the FORTRAN runtime system allocates buffers of default 2K bytes, and uses these for operations on all files accessed by FORTRAN programs. If a buffer is available, then access to a file will be optimal, otherwise access is one byte at a time, with consequent reduction in performance. It is strongly recommended that for normal FORTRAN files, the access types SEQUENTIAL and DIRECT are used. The runtime system will then use the most efficient method available for the particular device.

Scanned by Jonny Oddene for Sintran Data © 2011

14.1.8 I/O Buffer Allocation

Whenever possible, the FORTRAN run-time system uses buffers for the I/O statements instead of a byte-by-byte transfer of data.

- **The following applies to FORTRAN-100:**

If no buffer is available at the time of opening the file, then the access will revert to byte-by-byte operation. If the program is executed in non-reentrant mode, the buffers are allocated automatically in the space following the program (or data area if running with SEPARATE-DATA ON) and before the COMMON blocks. The maximum number of buffers is 20 and all are 2048 bytes long. They are allocated when the file is OPENed. Each file which uses a buffer will reserve one from this pool when it first requires it. The buffer will be released only when the file is closed.

If the program is reentrant (ND-100 only), then the buffers are allocated in the stack area, and each program has its own buffer pool. The allocation is done by an explicit call to a routine provided for the purpose:

```
CALL CREBUF (n)
```

where

the parameter *n* is INTEGER*2. If *n* is positive, then *n* buffers are created in the stack area.

If *n* is less than or equal to zero, no action is taken. If the buffers have already been allocated, no action is taken (i.e., only the first call to CREBUF has any effect).

Note that the FORTRAN-100 library must be loaded last of all, if buffered I/O is used.

The FORTRAN-100 library has as its last entry point, a pointer, called FREE_P, describing the area of unallocated address space, which is assumed to begin immediately following FREE_P. I/O buffers for non-reentrant programs will use this unallocated address space. For non-reentrant RT-programs, the RT-Loader command, SET-IO-BUFFERS must be used to allocate buffer space.

Scanned by Jonny Oddene for Sintran Data © 2011

- **The following applies to FORTRAN-500:**

The default buffer size is 2048 bytes. By using the BUFFER-SIZE parameter in the OPEN statement, bigger buffers can be used. The Linkage-Loader command, SET-IO-BUFFERS, must be used to allocate the space needed for the I/O buffers. The argument (octal) specifies how many buffers are to be allocated.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

A P P E N D I X A

ASCII CHARACTER SET

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

ASCII CHARACTER SET

Graphic	Octal Value		Decimal Value	ASCII Abbreviation	Comments
	Left Byte	Right Byte			
	000000	0	0	NUL	Null
	000400	1	1	SOH	Start of heading
	001000	2	2	STX	Start of text
	001400	3	3	ETX	End of text
	002000	4	4	EOT	End of transmission
	002400	5	5	ENQ	Enquiry
	003000	6	6	ACK	Acknowledge
	003400	7	7	BEL	Bell
	004000	10	8	BS	Backspace
	004400	11	9	HT	Horizontal tabulation
	005000	12	10	LF	Line feed
	005400	13	11	VT	Vertical tabulation
	006000	14	12	FF	Form feed
	006400	15	13	CR	Carriage return
	007000	16	14	SO	Shift out
	007400	17	15	SI	Shift in
	010000	20	16	DLE	Data link escape
	010400	21	17	DC1	Device control 1
	011000	22	18	DC2	Device control 2
	011400	23	19	DC3	Device control 3
	012000	24	20	DC4	Device control 4
	012400	25	21	NAK	Negative acknowledge
	013000	26	22	SYN	Synchronous idle
	013400	27	23	ETB	End of transmission block
	014000	30	24	CAN	Cancel
	014400	31	25	EM	End of medium
	015000	32	26	SUB	Substitute
	015400	33	27	ESC	Escape
	016000	34	28	FS	File separator
	016400	35	29	GS	Group separator
	017000	36	30	RS	Record separator
	017400	37	31	US	Unit separator
	020000	40	32	SP	Space
!	020400	41	33	!	Exclamation mark
"	021000	42	34	"	Quotation marks
#	021400	43	35	#	Number sign
$	022000	44	36	$	Dollar sign

Scanned by Jonny Oddene for Sintran Data © 2011

Graphic	Octal Value		Decimal Value	ASCII Abbreviation	Comments
	Left Byte	Right Byte			
%	022400	45	37	%	Percent sign
&	023000	46	38	&	Ampersand
'	023400	47	39	'	Apostrophe
(	024000	50	40	(	Opening parenthesis
)	024400	51	41	)	Closing parenthesis
*	025000	52	42	*	Asterisk
+	025400	53	43	+	Plus
,	026000	54	44	,	Comma
-	026400	55	45	-	Hyphen (Minus)
.	027000	56	46	.	Period (Decimal)
/	027400	57	47	/	Slant
0	030000	60	48	0	Zero
1	030400	61	49	1	One
2	031000	62	50	2	Two
3	031400	63	51	3	Three
4	032000	64	52	4	Four
5	032400	65	53	5	Five
6	033000	66	54	6	Six
7	033400	67	55	7	Seven
8	034000	70	56	8	Eight
9	034400	71	57	9	Nine
:	035000	72	58	:	Colon
;	035400	73	59	;	Semicolon
<	036000	74	60	<	Less than
=	036400	75	61	=	Equals
>	037000	76	62	>	Greater than
?	037400	77	63	?	Question mark
@	040000	100	64	@	Commercial at
A	040400	101	65	A	Uppercase A
B	041000	102	66	B	Uppercase B
C	041400	103	67	C	Uppercase C
D	042000	104	68	D	Uppercase D
E	042400	105	69	E	Uppercase E
F	043000	106	70	F	Uppercase F
G	043400	107	71	G	Uppercase G
H	044000	110	72	H	Uppercase H
I	044400	111	73	I	Uppercase I
J	045000	112	74	J	Uppercase J
K	045400	113	75	K	Uppercase K
L	046000	114	76	L	Uppercase L

Scanned by Jonny Oddene for Sintran Data © 2011

Graphic	Octal Value		Decimal Value	ASCII Abbreviation	Comments
	Left Byte	Right Byte			
M	046400	115	77	M	Uppercase M
N	047000	116	78	N	Uppercase N
O	047400	117	79	O	Uppercase O
P	050000	120	80	P	Uppercase P
Q	050400	121	81	Q	Uppercase Q
R	051000	122	82	R	Uppercase R
S	051400	123	83	S	Uppercase S
T	052000	124	84	T	Uppercase T
U	052400	125	85	U	Uppercase U
V	053000	126	86	V	Uppercase V
W	053400	127	87	W	Uppercase W
X	054000	130	88	X	Uppercase X
Y	054400	131	89	Y	Uppercase Y
Z	055000	132	90	Z	Uppercase Z
[	055400	133	91	[	Opening bracket
\	056000	134	92	\	Reverse slant
]	056400	135	93	]	Closing bracket
^ or ↑	057000	136	94	^ or ↑	Circumflex, up-arrow
_ or ←	057400	137	95	_, UND, BKR	Underscore, back-arrow
`	060000	140	96	`, GRA	Grave accent
a	060400	141	97	a, LCA	Lowercase a
b	061000	142	98	b, LCB	Lowercase b
c	061400	143	99	c, LCC	Lowercase c
d	062000	144	100	d, LCD	Lowercase d
e	062400	145	101	e, LCE	Lowercase e
f	063000	146	102	f, LCF	Lowercase f
g	063400	147	103	g, LCG	Lowercase g
h	064000	150	104	h, LCH	Lowercase h
i	064400	151	105	i, LCI	Lowercase i
j	065000	152	106	j, LCJ	Lowercase j
k	065400	153	107	k, LCK	Lowercase k
l	066000	154	108	l, LCL	Lowercase l
m	066400	155	109	m, LCM	Lowercase m
n	067000	156	110	n, LCN	Lowercase n
o	067400	157	111	o, LCO	Lowercase o
p	070000	160	112	p, LCP	Lowercase p
q	070400	161	113	q, LCQ	Lowercase q
r	071000	162	114	r, LCR	Lowercase r
s	071400	163	115	s, LCS	Lowercase s
t	072000	164	116	t, LCT	Lowercase t
u	072400	165	117	u, LCU	Lowercase u

Scanned by Jonny Oddene for Sintran Data © 2011

Graphic	Octal Value		Decimal Value	ASCII Abbreviation	Comments
	Left Byte	Right Byte			
v	073000	166	118	v, LCV	Lowercase v
w	073400	167	119	w, LCW	Lowercase w
x	074000	170	120	x, LCX	Lowercase x
y	074400	171	121	y, LCY	Lowercase y
z	075000	172	122	z, LCZ	Lowercase z
{	075400	173	123	{, LBR	Opening (left) brace
|	076000	174	124	|, VLN	Vertical line
}	076400	175	125	}, RBR	Closing (right) brace
~	077000	176	126	~, TIL	Tilde
	077400	177	127	DEL	Delete, rubout

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

A P P E N D I X B

ERROR MESSAGES

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

ERROR MESSAGES

B.1 COMPILER MESSAGES

During compilation, diagnostic messages will be issued for any source text which either is, or may be, erroneous. These messages appear in the program listing after the statement to which they refer, and also on the background terminal.

They fall into three categories:

- errors
- warnings
- extensions

Error messages are produced when the compiler cannot make a sensible interpretation of the program. Execution of these programs becomes impossible.

Warnings are given when there is a potential fault, but an object file is produced and execution may be possible.

Extension messages indicate where the program is using language features which are not part of the ANSI FORTRAN 77 standard. These messages are suppressed unless the STANDARD-CHECK compiler command is on.

Some messages may be preceded by some text in quotes. This may be either the name of some variables, or a part of the source program with a '?' inserted. This '?' will show where the error was detected and will usually be at or shortly after the item at fault.

Scanned by Jonny Oddene for Sintran Data © 2011

The following list of the text of the error message texts is in alphabetical order:

-1 IN SUBSTRING

ANSI FORTRAN 77 must have positive values as indexes in substring values. ND FORTRAN uses -1 to mean stripping blanks from the string.

1H ASSUMED

ANSI FCRTRAN 77 requires the length of an H-format item to be explicitly stated. ND FORTRAN allows 'HX' to mean '1HX'

1X ASSUMED

ANSI FORTRAN 77 requires the length of the X format item to be explicitly stated. ND FORTRAN allows 'X' to mean '1X'.

ALIGNMENT

A variable has been allocated to an address which cannot be supported, a eg., an INTEGER beginning at an odd-byte boundary on ND-100.

ALTERNATE RETURNS IN INTRINSIC FUNCTION

INTRINSIC functions cannot accept * return specifiers in their argument lists.

ALTERNATE RETURNS INVALID IN THIS PROGRAM UNIT

A RETURN statement was found that specified an alternate return expression when none of the entry points had * specifiers in their argument list.

ALTERNATE RETURNS ONLY ALLOWED IN SUBROUTINES

Alternate return specifiers are only valid in subroutines, not in functions.

APF-LIB WILL NOT BE USED BECAUSE OF THE ARRAY-INDEX-CHECK OPTION

Cannot have both the apf-lib option and the array-index-check option at the same time.

APT COMMON

The form of COMMON which is placed in the alternate page table on NORD-10/ND-100 is non-standard, and not available on ND-500.

Scanned by Jonny Oddene for Sintran Data © 2011

ARGUMENT(S) CONVERTED

ANSI FORTRAN 77 allows only those actual arguments which match the corresponding dummy arguments without conversion. ND FORTRAN converts, where possible, the arguments to INTRINSIC and statement functions.

ARRAYS MUST HAVE THE SAME SHAPE

All arrays in an array-operation must have the same number of

[[[[dimensions. All dimensions must be of equal size.

'x' ASSUMED

x is a character which the compiler has assumed was omitted in the indicated position.

ASSUMED BECAUSE HAS 2 ARGUMENTS

The wrong generic or specific name was given for the number of arguments (i.e., ATAN instead of ATAN2).

ASSUMED-SIZE ARRAY USED AS LIST ITEM

Assumed-size array must not be used as a list item on input/output statements.

BRANCH INTO DO/IF NEST

The compiler has found an attempt to transfer control from outside to the inside of a DO loop or structured IF construct.

CANNOT BE CALLED

The indicated item cannot be invoked. It is neither external, INTRINSIC nor a statement function.

CANNOT BE DIMENSIONED

The named item cannot have dimensions (e.g., it may have been previously declared as external).

CANNOT BE PASSED AS AN ARGUMENT

The item cannot be used as an actual argument (e.g., MAX function).

CANNOT BE SAVED

The indicated item must not occur in a SAVE statement.

Scanned by Jonny Oddene for Sintran Data © 2011

CANNOT BE USED IN A TYPE STATEMENT

A name has been explicitly given a type when this is not allowed (e.g., SUBROUTINE name).

CANNOT CONTAIN A LABEL

The indicated item is expected to have the value of a label in it (i.e., set by ASSIGN statement). The item must be an unsubscripted variable name of type INTEGER*2 on NORD-10/ND-100 and INTEGER*4 on ND-500.

CANNOT CONVERT

The requested conversion cannot be carried out (e.g., arithmetic to character).

CANNOT SELECT GENERIC ENTRY

There is no specific name for this generic entry which allows arguments of the required type.

CHARACTER AND NON-CHARACTER EQUIVALENCED

ANSI FORTRAN 77 does not allow the mixing of character items with non-character items in an equivalence list.

CHARACTER IN APT COMMON

Character variables cannot be referenced via the APT on NORD-10/ND-100.

CHARACTER VARIABLE REFERENCED IN BOTH SIDES OF "="

It is illegal to refer to the same character variable in both source and destination part of an assignment statement.

COMMON BLOCKS EXCEED MEMORY

The total memory requirements of COMMON blocks exceeds 64K words (ND-100 only).

CONFLICTING POSITIONS

A variable has been allocated to two separate places by a combination of COMMON and EQUIVALENCE lists.

CONTROL VARIABLE NOT INTEGER

In an implied DO in a DATA statement, the loop control variable must be of type integer.

Scanned by Jonny Oddene for Sintran Data © 2011

CONVERTED TO INTEGER

A non-integer arithmetic expression or constant was found where ANSI FORTRAN 77 requires an integer. ND FORTRAN allows the conversion.

DATA IN BLANK COMMON

Initializing variables in blank common by a DATA statement is an ND FORTRAN extension.

DATA IN COMMON

Initializing variables in COMMON with a DATA statement in a program unit, other than a BLOCK DATA subprogram, is an ND FORTRAN extension.

DATATYPE

The indicated data type is not in ANSI FORTRAN 77, but is a ND FORTRAN extension.

DATA STATEMENT IS ILLEGAL

Data statement must not occur in a recursive subprogram.

DECLARATION MISSING

If IMPLICIT OFF is used, every symbolic name requiring a data type must be declared in a Type statement.

DIVIDE BY ZERO

In a constant expression, an attempt was made to divide by zero.

DO HAS ZERO STEP

In a DO loop, or an implied DO loop, the step value is zero.

DO/IF NESTING ERROR

DO loops or structured IFs are not properly nested.

DOUBLY DEFINED

An attempt was made to use a name for two conflicting purposes.

DOUBLY SAVED

A variable appears in more than one SAVE statement.

Scanned by Jonny Oddene for Sintran Data © 2011

DUMMY NAME INSERTED

A name was expected. The compiler has created an internal name in order to continue processing.

EMBEDDED UNARY SIGN

ANSI FORTRAN 77 prohibits adjacent arithmetic operators, but ND FORTRAN allows it, eg., A+-B.

ENTRY NOT ALLOWED IN DO/IF NEST

Entry statements cannot appear within DO loops or structured IF constructs.

ENTRY NOT SET BEFORE RETURN

In a function subprogram, an entry point has not been assigned. Invocation of this entry point might lead to an undefined value being returned.

ERRCODE NORMAL VARIABLE IN STANDARD PROGRAMS

If STANDARD CHECK is ON, ERRCODE is treated like a normal variable. Otherwise it has a special meaning, see Section 9.1.7.

EXPRESSION MISSING

An expression was expected but not found.

EXTENDS COMMON NEGATIVELY

An EQUIVALENCE list required a variable or array to occupy storage preceding a COMMON block.

FORMAT

The indicated item is an ND FORTRAN extension.

FORMAT ERROR

A FORMAT list is incorrectly specified, e.g., missing comma.

FORMAT LABEL TARGET OF BRANCH

Control cannot be transferred to labels on FORMAT statements.

HAS INVALID BOUNDS

The named variable cannot have the declared bounds. E.g., a local variable was given non-constant bounds, or the upper bound was less than the lower.

HAS INVALID LENGTH

The named variable cannot have the declared length. I.e., a local character string was declared with a non-constant length.

Scanned by Jonny Oddene for Sintran Data © 2011

HOLLERITH CONSTANT

Hollerith constants are not part of the ANSI FORTRAN 77 standard. Appendix G describes how ND FORTRAN implements them. Character strings should be used instead of Hollerith constants wherever possible.

IGNORED IN BLOCK DATA

The indicated item is not valid in a BLOCK DATA subprogram, and has been ignored in order to continue processing.

ILLEGAL INDEX IN IMPLIED DO

Array-index error in implied DO.

IN A DIFFERENT COMMON BLOCK

In an EQUIVALENCE statement, two items in a single list are in different COMMON blocks.

INCLUDES NESTED TOO DEEPLY

The maximum depth of nesting for $INCLUDE commands is 5.

INCOMPLETE CHARACTER/HOLLERITH STRING

The end of a statement occurred before the end of a string. Possible causes are: missing quote, or wrong count before Hollerith H, or statement extends beyond column 72.

INEFFICIENT ACCESS MODE IN OPEN

This applies to the ND-500 only. The R, W, and RW access modes are very inefficient. If possible, DIRECT or SEQUENTIAL should be used.

INTEGER INVALID OR OUT OF RANGE

An unacceptable integer constant has been found. The valid values depend on the context.

INTEGER MISSING, 1 ASSUMED

An integer was expected. The compiler assumes a value of 1 in order to continue processing.

INTERNAL FILES NEED A FORMAT

Using unformatted I/O on internal files is not allowed.

INTRINSIC FUNCTION

The named function is an ND FORTRAN extension.

Scanned by Jonny Oddene for Sintran Data © 2011

INVALID AS A DUMMY ARGUMENT

The specified item cannot be a dummy argument.

INVALID AS FUNCTION/SUBROUTINE NAME

Cannot refer to a subprogram as both function and subroutine.

INVALID CHARACTER, STATEMENT IGNORED

A character is found which is not in the FORTRAN character set. Compilation continues with the next statement.

INVALID CHARACTER, SUBSTRING EXPRESSION

The character substring expression exceeds the maximum string length.

INVALID CONSTANT EXPRESSION

The expression cannot be computed at compile-time.

INVALID DIMENSION EXPRESSION

Dimension bounds must be integer expressions.

INVALID DO TERMINATION

The label specified in a DO statement was found with a statement that cannot terminate a DO loop.

INVALID IF EXPRESSION

The expression cannot be used in a Logical or Arithmetic IF. e.g., it may be of type CHARACTER.

INVALID IMPLICIT RANGE

The range in an implicit range is invalid (e.g., the second letter precedes the first).

INVALID IN CONSTANT LIST

The indicated item cannot be used as a constant in a DATA statement constant list.

INVALID IN DATA LIST

The indicated item cannot be initialized in a DATA statement.

INVALID IN EQUIVALENCE LIST

The indicated item cannot share storage with any other item.

INVALID I/O LIST ITEM

Self-explanatory.

Scanned by Jonny Oddene for Sintran Data © 2011

INVALID I/O OPTION

Self-explanatory.

INVALID ITERATION

In an implied DO loop in a DATA statement, the iteration count is negative or zero.

INVALID LABEL

A label was expected but not found.

INVALID LEFT SIDE OF ASSIGNMENT

Self-explanatory.

INVALID LOOP CONTROL

The control variable of a DO loop must be an integer, real, or double-precision variable.

INVALID OPERAND

The operand cannot be used with its operator.

INVALID SUBSCRIPT EXPRESSION

A subscript must be of type INTEGER.

LABEL DEFINED, BUT NOT REFERRED TO

The label is not referred to in other statements.

LABEL MISSING

A label was expected but not found.

LABEL NOT ALLOWED WITH THIS STATEMENT

Self-explanatory.

LABEL NOT ASSIGNED

The label must be assigned to an integer variable in a statement label assignment statement.

LABEL REFERS TO ITSELF

A potential endless loop was detected.

LABEL UNDEFINED

A label was used which did not appear in the label field of any statement.

LABEL USED AS FORMAT

The label on an executable statement was found where a format label was expected.

Scanned by Jonny Oddene for Sintran Data © 2011

LINE(S) ARE NON-BLANK BEYOND COL. 72

The indicated number of lines were found which had non-blank characters in columns beyond the 72nd, and these lines formed part of a statement or command. Comment lines which extend beyond column 72 are not included in this number.

LOCAL ARRAYS EXCEED MEMORY

The total memory requirements of local arrays exceeds 64K words (ND-100 only).

LOCAL DATA IGNORED IN RE-ENTRANT MODE

Local variables cannot be initialized by DATA statements in reentrant mode.

LOGICAL OPERATION ON INTEGERS

Self-explanatory.

MISPLACED '

In an assigment statement, the left-hand side was not followed by an equals sign.

MISSING DIMENSION LIST

No dimensions were given in an array declarator.

MISSING 'END'

The end of file was found on the program test file when a program unit was still incomplete.

MISSING NAME, '£MAIN' ASSUMED

In order that the compiler may continue its processing, it has inserted the name £MAIN.

MISSING SPECIFICATION

An empty position was found in a list, e.g., 2 adjacent commas.

MISSING SUBSCRIPTS

An array name was used where it must be followed by a subscript list.

MISSING 'THEN'

The compiler assumes the keyword 'THEN' to be present, in order to continue processing the IF statement.

Scanned by Jonny Oddene for Sintran Data © 2011

MIXED LENGTH CHARACTER ENTRIES

ANSI FORTRAN 77 requires that all entry names in a function subprogram must be either of type non-character, or CHARACTER with the same length (or an *). ND FORTRAN removes this restriction completely.

MIXING CHARACTER AND NON-CHARACTER IN COMMON

ANSI FORTRAN 77 requires that all variables in a COMMON block be character, or that all are non-character.

MIXING DOUBLE PRECISION AND COMPLEX

ANSI FORTRAN 77 does not allow arithmetic operations to have one double precision and one complex operand. For the method of treatment by ND FORTRAN, see Section 4.4.

MORE THAN 6 CHARACTERS, 31 SIGNIFICANT

ANSI FORTRAN 77 restricts names to 6 characters. ND FORTRAN uses the first 31.

MORE THAN 7 DIMENSIONS

ANSI FORTRAN 77 allows no more than 7 dimensions for its arrays. ND FORTRAN can support more except within the Symbolic Debugger.

MULTIPLE ASSIGNMENT ILLEGAL FOR ARRAYS

Assignment to an array must not occur in a multiple assignment statement.

MULTI-DIMENSIONED ARRAYS NOT ALLOWED IN ARRAY-OPERATION

Only one-dimensional arrays are allowed in an array-operation.

NEITHER UNIT NOR FILE SPECIFIED

An INQUIRE statement must indicate the unit or file to be examined.

NO DO SPECIFICATION IN LIST

A parenthesised data list has no DO specification present.

NO MORE SPACE

The compiler has exhausted its work area. The program unit is too big to be compiled. Try subdividing it into subroutines, or moving DATA statements to a BLOCK DATA subprogram.

NON-STANDARD CHARACTER

ND FORTRAN allows an underscore character (_) in symbolic names.

NON-STANDARD CONTINUATION

"&" has been used as a continuation mark.

Scanned by Jonny Oddene for Sintran Data © 2011

NON-STANDARD EQUIVALENCING

Two data items share storage in a way which may make the execution of the program diverge from the ANSI FORTRAN 77 definition. E.g., REAL and INTEGER arrays overlapping on a ND-100.

NON-STANDARD EXPRESSION

Self-explanatory.

NON-STANDARD INTERNAL FILE OR FORMAT

Self-explanatory.

NON-STANDARD LABEL FIELD

ANSI FORTRAN 77 allows labels only in columns 1 to 5 inclusive.

NON-STANDARD REDEFINITION

A name is used, both as an external name or entry, and as a dummy argument in a statement function.

NOT A FUNCTION

Self-explanatory.

NOT A LOGICAL EXPRESSION

In a structured IF, the expressions controlling each of the ELSE IF's must be logical expressions.

NOT ALLOWED IN DIMENSION EXPRESSION

Self-explanatory.

NOT ALLOWED IN LOGICAL IF

The indicated statement cannot be part of a Logical IF statement.

Scanned by Jonny Oddene for Sintran Data © 2011

NOT ENOUGH CONSTANTS

There were more items to be initialized in a DATA statement data list, than constants in the corresponding constant list.

NOT INTEGER CONSTANT EXPRESSION

Self-explanatory.

NOT INTRINSIC, EXTERNAL ASSUMED

The name is not one of the known INTRINSIC functions. It is assumed to be an external function in order to continue processing.

NOT SYMBOLIC CONSTANT

A name found where a constant expression should appear, was not a constant.

NO UNIT SPECIFIER

All I/O statements must specify the unit on which they operate.

NULL STATEMENT

ANSI FORTRAN 77 does not allow completely empty statements. E.g., after a Logical IF.

OCTAL CONSTANT

Octal constants are an ND FORTRAN extension.

OPTION

The indicated option is an ND FORTRAN extension.

OUT OF DATA BEFORE CONSTANTS

In a DATA statement, the list of initialized items was shorter than the list of constants.

OUT OF RANGE

The value on the right side of the assignment operator is too large/small for the variable on the left side.

OVERLAPPING IMPLICIT RANGES

The same letter(s) occur in more than one range in implicit specifications.

PARENTHESES ASSUMED AROUND PARAMETERS

The PARAMETER statement should have its list of symbolic constant assignments enclosed in parentheses.

Scanned by Jonny Oddene for Sintran Data © 2011

PRIORITY

Priority is valid only for ND FORTRAN programs.

REC AND END CONFLICT

The end-of-file indication can only occur in a direct access READ as an ND FORTRAN extension.

REC AND FMT

Free format I/O is only valid in SEQUENTIAL access READ and WRITE statements.

RECURSION

Recursion is valid only as an ND FORTRAN extension, and in reentrant mode.

RETURN IN PROGRAM

In ANSI FORTRAN 77, a program must terminate with a STOP statement or by reaching the END statement of the program subunit.

SAVE OF LOCALS NOT IMPLEMENTED IN REENTRANT MODE

Self-explanatory.

SEMICOLON SEPARATOR

In ANSI FORTRAN 77, only one statement can be placed on a line. ND FORTRAN allows a semicolon character (;) to separate statements on a line.

SINGLE DIMENSIONING

The ability to refer to a multidimensional array by use of a single subscript is an ND FORTRAN extension.

SPECIFICATION AFTER DATA

ND FORTRAN allows DATA statements to appear before specification statements.

STATEMENT

The indicated statement is an ND FORTRAN extension.

STATEMENT HAS TOO MANY CONTINUATION LINES

ANSI FORTRAN 77 allows 19 continuation lines in one statement. The statement must be split.

SUBSTRING OF CONSTANT

Taking the substring of a symbolic constant is a ND FORTRAN extension.

Scanned by Jonny Oddene for Sintran Data © 2011

SYMBOL NOT IN PARENTHESES A symbolic constant being used to define a length of character items must be in parentheses in ANSI FORTRAN 77.

SYNTAX ERROR IN ARITHMETIC CONSTANT

Self-explanatory.

SYNTAX ERROR, REST OF STATEMENT IGNORED

A previous syntax error has been found. Processing is continued at the next statement.

TOO FEW ITEMS

In an EQUIVALENCE statement, each list must contain at least two items.

TOO LARGE LENGTH SPECIFIER

A character variable may have a length up to 2047 on ND-100 and 32767 on ND-500.

UNRECOGNISED OR MISPLACED STATEMENT

The statement is either badly formed (e.g., a misspelled keyword) or is out of sequence (e.g., a specification follows an executable statement).

VARIABLE NOT ASSIGNED

The variable is referred to before it is assigned.

WRONG NUMBER OF ARGUMENTS

Self-explanatory.

WRONG USE OF ASSEMBLY NAME

The restrictions on use of symbolic names declared in an ASSEMBLY statement have not been observed.

ZERO LENGTH STRING

Self-explanatory.

%COMMENT

ND FORTRAN allows comments to begin with a percent (%) character.

Scanned by Jonny Oddene for Sintran Data © 2011

B.2 THE LOADER ERROR MESSAGES

These are described in the SINTRAN III Real Time Loader ND-60.051.

B.3 RUNTIME ERROR DIAGNOSTICS

The runtime error diagnostics are printed on the message output file, which is the user terminal (the SINTRAN error device for RT) in the format:

*** date time FORTRAN EXCEPTION : (nnn) line error message IN LINE ll RETURN ADDRESS aaaaaa UNIT uu DEVICE ddB

where:

nnn is the octal error number.

aaaaaa is the address in octal of the executing program of the compiled statement in which the error has occurred.

ll is the line number in decimal within the source program, of the compiled statement in which the error has occurred.

uu is the FORTRAN unit number, decimal, on which the error has occurred.

dd is the SINTRAN logical device number, octal, on which the error has occurred.

Note that on the ND-500, more information about traps and exceptions may be printed, see Appendix D.3.

If the error is serious the message ***JOB ABORTED*** is given and the control returns to the operating system.

If the error is not serious, ERRCODE is set to the value of the error code (and IOSTAT if applicable), and control returns to the FORTRAN program.

Scanned by Jonny Oddene for Sintran Data © 2011

Error Code		
Decimal	**Octal**	**Meaning (error text)**
0	0	Not used
1	1	Not used
2	2	Bad file number
3	3	End of file
4	4	Card reader error (card read)
5	5	Device not reserved
6	6	Not used
7	7	Card reader error (card not read)
8	10	Not used
9	11	Not used
10	12	End of device (time-out)
11-16	13-20	Not used
17	21	Illegal character in parameter
18	22	No such page
19	23	Not decimal number
20	24	Not octal number
21	25	You are not authorized to do this
22	26	Directory not entered
23	27	Ambiguous directory name
24	30	No such device name
25	31	Ambiguous device name
26	32	Directory entered
27	33	No such logical unit
28	34	Unit occupied
29	35	Master block transfer error
30	36	Bit file transfer error
31	37	No more tracks available
32	40	Directory not on specified unit
33	41	Files opened on this directory
34	42	Main directory not last one released
35	43	No main directory
36	44	Too long parameter
37	45	Ambiguous user name
38	46	No such user name
39	47	No such user name in main directory
40	50	Attempt to create too many users
41	51	User already exists
42	52	User has files
43	53	User is entered

Scanned by Jonny Oddene for Sintran Data © 2011

Error Code		
Decimal	**Octal**	**Meaning (error text)**
44	54	Not so much space unreserved in directory
45	55	Reserved space already used
46	56	No such file name
47	57	Ambiguous file name
48	60	Wrong password
49	61	User already entered
50	62	No user entered
51	63	Friend already exists
52	64	No such friend
53	65	Attempt to create too many friends
54	66	Attempt to create yourself as friend
55	67	Continuous space not available
56	70	Not directory access
57	71	Space not available to expand file
58	72	Space already allocated
59	73	No space in default directories
60	74	No such file version
61	75	No more pages available for this user
62	76	File already exists
63	77	Attempt to create too many files
64	100	Outside device limits
65	101	No previous version
66	102	File not continuous
67	103	File type already defined
68	104	No such access code
69	105	File already opened
70	106	Not write access
71	107	Attempt to open too many files
72	110	Not write and append access
73	111	Not read access
74	112	Not read, write and common access
75	113	Not read and write access
76	114	Not read and common access
77	115	File reserved by another user
78	116	File already opened for write
79	117	No such user index
80	120	Not append access
81	121	Attempt to open too many mass storage files

Scanned by Jonny Oddene for Sintran Data © 2011

Error Code		
Decimal	**Octal**	**Meaning (error text)**
82	122	Attempt to open too many files
83	123	Not opened for sequential write
84	124	Not opened for sequential read
85	125	Not opened for random write
86	126	Not opened for random read
87	127	File number out of range
88	130	File number already used
89	131	No more buffer space
90	132	No file opened with this number
91	133	Not mass storage file
92	134	File used for write
93	135	File used for read
94	136	File only opened for sequential read or writ
95	137	No scratch file opened
96	140	File not reserved by you
97	141	Transfer error
98	142	Reserved by RT program
99	143	No such block
100	144	Source and destination equal
101	145	Illegal on tape device
102	146	End of tape
103	147	Tape already in use
104	150	Not random access on tape files
105	151	Not last file on tape
106	152	Not tape device
107	153	Illegal address reference in monitor call
108	154	Not last record on tape
109	155	File already opened by another user
110	156	File already opened for write by another use
111	157	Missing parameter
112	160	Two pages must be left unreserved
113	161	No answer from remote computer
114	162	Device cannot be reserved
115	163	Overflow in read
116	164	DMA error
117	165	Bad datablock
118	166	Control/modus word error
119	167	Parity error

Scanned by Jonny Oddene for Sintran Data © 2011

Error Code		
Decimal	**Octal**	**Meaning (error text)**
120	170	LCR error
121	171	Device error(read-last-status to get status)
122	172	No device buffer available
123	173	Illegal mass storage unit number
124	174	Illegal parameter
125	175	Write-protect violation
126	176	Error detected by read after write
127	177	No EOF mark found
128	200	Cassette not in position
129	201	Illegal function code
130	202	Time-out (no datablock found)
131	203	Paper fault
132	204	Device not ready
133	205	Device already reserved
134	206	Not peripheral file
135	207	No such queue entry
136	210	No so much space left
137	211	No spooling for this device
138	212	No such queue
139	213	Queue empty
140	214	Queue full
141	215	Not last used by you
142	216	No such channel name
143	217	No remote connection
144	220	Illegal channel
145	221	Channel already reserved on remote computer
146	222	No remote file processor
147	223	Formatting error
148	224	Incompatible device sizes
149	225	Remote Processor not available
150	226	Tape format error
151	227	Block count error
152	230	Volume not on specified unit
153	231	Not deleted record
154	232	Device error
155	233	Error in object entry
156	234	Odd number of bytes (right byte in last word insignificant)
157-256	234-400	Not used

Scanned by Jonny Oddene for Sintran Data © 2011

Error Code		
Decimal	**Octal**	**Meaning (error text)**
257	401	Fatal formatting system error. This is a system error due to software or hardware errors.
258	402	Too low parentheses level in format. A maximum of 5 levels is permitted.
259	403	Illegal character in format
260	404	Illegal termination of format
261	405	Output record size exceeded. A maximum of 256 characters is permitted.
262	406	Format requires greater input record
263	407	Integer overflow on input. The result will be 21474836847 or -214748368 for INTEGER*4, and 32767 or -32768 for INTEGER*2.
264	410	Input record size exceeded. A maximum of 256 characters is permitted.
265	411	Backspace illegal
266	412	Bad character on input. The input field is ignored and the result will be zero.
267	413	Real overflow on input. The result will be 1.0E76.
268	414	Real underflow on input. The result will be 0.0.
269	415	String does not start on a word boundary
270	416	Real overflow on output
271	417	Formar specification does not apply
272	420	Overflow in exponent on input
273	421	Wrong number of parameter in call
274	422	Too many files opened (ND-100 only)
276	424	Mixing of FORMATTED/UNFORMATTED illegal
277	425	No more buffers available
278	426	Non-fatal error. Result of FORTRAN system or hardware error. ND-500 only.
279	427	Fatal error (I/O). Result of FORTRAN system or hardware error. ND-500 only.
280	430	I/O error without special handling

Scanned by Jonny Oddene for Sintran Data © 2011

Error Code		
Decimal	**Octal**	**Meaning (error text)**
281	431	Zero base and negative exponent. The result will be 21474836847 for integers and 1.0E76 for reals.
282	432	Base elss than zero in exponentiation. The result will be 0.0.
283	433	Overflow in exponentiation. The result will be 1.0E76.
284	434	Neg. arg. in square-root. The result will be 0.0.
285	435	Too large arg. in sine. The result will be 0.0.
286	436	Too large arg. in cosine. The result will be 0.0.
287	437	Too large arg. in exp-function. The result will be 1.0E76.
288	440	Zero or neg. arg. in logarithm. The result will be -1.0E76.
289	441	Both args. zero in arc-tan. The result will be 0.0.
290-293	442-445	Not used
294	446	Too large arg. in hyperb. sine. The result will be 1.0E76.
295	447	Too large arg. in hyperb. cosine. The result will be 1.0E76.
296	450	Too large arg. in square-root or complex abs or square-root.
297-301	451-455	Not used
302	456	Illegal arg. in arc-sine/cosine. The result will be 0.0.
303	457	Illegal arg. in tan. The result will be 0.0.

Scanned by Jonny Oddene for Sintran Data © 2011

A P P E N D I X C

MONITOR CALLS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

MONITOR CALLS

C.1 INTRODUCTION TO USING MONITOR CALLS

If you want to communicate directly with the SINTRAN operating system in a Fortran program, monitor calls are provided for this purpose. The FORTRAN language and runtime system has a variety of facilities, such as I/O statements for accessing files or handling peripheral devices. However, some situations require direct communication between a program and the SINTRAN operating system. In general, this means that the program is requesting a particular service be carried out, or that some specific item of information is required.

The monitor calls may be called by using the statements:

MONITOR_CALL(number, par-1, ..., par-n)
or
MONITOR_CALL('name', par-1, ..., par-n)

where

number is the monitor call number

'name' is the name of the monitor call.

Further explanation of each monitor call is given in the manual SINTRAN III Monitor Calls, ND-60.228.

Some monitor call routines are provided in the FORTRAN library. These may be called from a FORTRAN program as either a subroutine or a function subprogram. The main difference is that in using a monitor call as a function, a value is returned indicating the result of carrying out the request.

Most monitor calls may be used either as a function or a subroutine. However, some may only be used as a function since the function value is the information which was requested, e.g. the monitor call TUSED returns the CPU time used by a terminal since a logon.

Scanned by Jonny Oddene for Sintran Data © 2011

If a function returns a status code, it is strongly recommended that this status be tested. If a monitor call is called as a subroutine, then the status (e.g. error conditions) must be detected in a different way than with functions. The system variable ERRCODE, which is an ND FORTRAN extension, may be used with many of the monitor calls (both functions and subroutines) to detect errors. If ERRCODE is used to detect errors from monitor calls, the program must not be compiled in STANDARD-CHECK ON mode.

Two examples, one of a monitor call used as a subroutine, and another of a monitor call used as a function, are given below.

Example - a monitor call used as a subroutine

To set the system time and date, use the monitor call CLOCK as a subroutine:

```
C Declarations

      INTEGER PARAMS(7),BUNITS,SECONDS
      INTEGER MINUTES,HOURS,DAYOFMTH,MONTH,YEAR

C Set up some convenient variable names for the time and date.

      EQUIVALENCE (PARAMS(1),BUNITS),(PARAMS(2),SECONDS)
      EQUIVALENCE (PARAMS(3),MINUTES),(PARAMS(4),HOURS)
      EQUIVALENCE (PARAMS(5),DAYOFMTH),(PARAMS(6),MONTH)
      EQUIVALENCE (PARAMS(7),YEAR)

C Use the monitor call to get the system time and date.

      CALL CLOCK(PARAMS)
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example - a monitor call used as a function

To read information directly from a device, use the monitor call INCH as a function:

```
C Declarations (including the monitor call to be used as a
C function)

      INTEGER INUNIT,ONECHAR
      INTEGER INCH

C Read one character from the device which is connected to the
C FORTRAN unit number in INUNIT.

      ONECHAR=INCH(INUNIT)

C Check the system variable, ERRCODE, to see if all went well.

      IF(ERRCODE .NE. 0) GO TO 10

C Continue processing.

      ...     .

C An error occurred; terminate.

10    CONTINUE
      STOP
```

Scanned by Jonny Oddene for Sintran Data © 2011

NOTES

1. The system variable ERRCODE contains a value upon return from many monitor calls. The value returned indicates whether the service requested has been successfully carried out or an error or some unusual condition has arisen.

2. If a monitor call is used as a function, the function name must be declared as a specific data type, to define for the compiler the precise way that this variable name will be used.

All monitor calls which may be used in FORTRAN are listed in the Table of monitor calls, in section C.3. For each monitor call, the table describes the name which must be used, whether the monitor call is typically used as a function or a subroutine, the number of parameters and their corresponding data types, and whether ERRCODE contains a value upon return.

More detailed information on all available monitor calls can be found in the manual SINTRAN III Monitor Calls, ND-60.228.

The monitor calls available to FORTRAN programs, i.e. supplied in the FORTRAN library, are limited to those described in section C.3, which does not include all those available from the SINTRAN operating system. Note that the name to be used for a specific monitor call in FORTRAN could be different from the name used in the SINTRAN III Monitor Calls manual. The example above shows how the monitor call INCH is used; in the SINTRAN III Monitor Calls manual this monitor call is named INBT. The table in section C.3 has the names to be used in FORTRAN.

Scanned by Jonny Oddene for Sintran Data © 2011

C.2 COMMONLY USED MONITOR CALLS

This section illustrates some of the more commonly used monitor calls. Each monitor call which has been selected has a brief explanation of its function and an example of the way in which it can be called from a FORTRAN program.

Summary of the monitor calls in this section

Alphabetical order:			Numerical order:		
Name	No.	Example	No.	Name	Example
ABORT	105	(C.2.1)	1	INCH	(C.2.2)
BRKM	4	(C.2.3)	2	OUTCH	(C.2.2)
CLOCK	113	(C.2.1)	3	ECHOM	(C.2.3)
COMND	70	(C.2.1)	4	BRKM	(C.2.3)
ECHOM	3	(C.2.3)	11	TIME	(C.2.1)
ERMSG	64	(C.2.6)	32	MSGE	(C.2.3)
HOLD	104	(C.2.1)	64	ERMSG	(C.2.6)
INCH	1	(C.2.2)	66	ISIZE	(C.2.3)
ISIZE	66	(C.2.3)	70	COMND	(C.2.1)
MAGTP	144	(C.2.3)	73	SMAXD	(C.2.4)
MSG	32	(C.2.3)	75	REABT	(C.2.4)
OUTCH	2	(C.2.2)	76	SETBS	(C.2.4)
OUTST	162	(C.2.2)	100	RT	(C.2.1)
REABT	75	(C.2.4)	104	HOLD	(C.2.1)
RFILE	117	(C.2.5)	105	ABORT	(C.2.1)
RSIO	143	(C.2.5)	113	CLOCK	(C.2.1)
RT	100	(C.2.1)	114	TUSED	(C.2.1)
RTWT	135	(C.2.1)	117	RFILE	(C.2.5)
SETBS	76	(C.2.4)	120	WFILE	(C.2.5)
SMAX	73	(C.2.4)	121	WAITF	(C.2.5)
TIME	11	(C.2.1)	135	RTWTF	(C.2.1)
TUSED	114	(C.2.1)	140	WHDEV	(C.2.2)
WAITF	121	(C.2.5)	143	RSIOV	(C.2.5)
WFILE	120	(C.2.5)	144	MAGTP	(C.2.3)
WHDEV	140	(C.2.2)	162	OUTST	(C.2.2)

Scanned by Jonny Oddene for Sintran Data © 2011

C.2.1 PROGRAM ADMINISTRATOR

TIME - MON 11

Read the current internal time, in basic time units, i.e. 20 milliseconds per unit or some other value which has been set for the operating system.

```
C Get the current internal time in basic time units.

      INTEGER*4 TIME,CURRENTTIME,DUMMY

      CURRENTTIME=TIME(DUMMY)
```

Scanned by Jonny Oddene for Sintran Data © 2011

COMND - MON 70

Transfer the contents of a string to the SINTRAN command buffer. The string will be executed by SINTRAN.

```
C Request that SINTRAN execute a command to create a file during
C execution of this program.

C Define a variable to hold the command string.

      CHARACTER*40 COMMAND

C On the ND-100 the character string must start on a word
C boundary. An equivalence with an integer variable will
C accomplish this.

      INTEGER I
      EQUIVALENCE(COMMAND,I)

C Set the contents of the string variable to a SINTRAN command.
C Note that the character string does not begin with an @
C character. Furthermore, the string must be terminated by an
C apostrophe.

      COMMAND='CREATE-FILE A-NEW-FILE:SYMB 100'''

C Request that SINTRAN execute the contents of "command".

      CALL COMND(COMMAND)
```

Scanned by Jonny Oddene for Sintran Data © 2011

RT - MON 100

Put an entry in the execution queue, to request execution of an RT program.

```
C Put a request for execution of the RT program in the execution
C queue.

      EXTERNAL RTJOB

      CALL RT(RTJOB)
```

Scanned by Jonny Oddene for Sintran Data © 2011

HOLD - MON 104

Place the calling program in a waiting state for a specified period. The program will continue execution after return from the call to HOLD.

```
C Wait for 30 seconds, then output a message, including a count,
C once every ten time units for a limited period of time.

      INTEGER TERMINAL,COUNT,TUNITS,SECONDS
      INTEGER LONGWAIT,MSGWAIT,MSGLIMIT

C Initialize the terminal's logical unit number.

      DATA TERMINAL/1/

C Initialize the time unit indicators.

      DATA TUNITS/1/,SECONDS/2/

C Initialize the wait times, and the number of times the message
C loop should iterate.

      DATA LONGWAIT/30/,MSGWAIT/10/,MSGLIMIT/500/

C Place the program in a wait state for 30 seconds.

      CALL HOLD(LONGWAIT,SECONDS)

C Output a message with a count every tenth basic time unit.

       DO 10 COUNT = 1,MSGLIMIT
       WRITE(TERMINAL,100) COUNT
  100  FORMAT(X,'Still alive in here! Message number: ',I5)
      CALL HOLD(MSGWAIT,TUNITS)

C Do some other processing.

 10   CONTINUE
```

Scanned by Jonny Oddene for Sintran Data © 2011

ABORT - MON 105

Stop an RT program by setting it in a passive state. The program will be removed from the time or execution queue, all resources will be released and further periodic execution will be prevented.

```
C Stop the RT program COLLECT.

      EXTERNAL COLLECT

      CALL ABORT(COLLECT)
```

CLOCK - MON 113

Get the current time and date from the operating system.

```
C Get the system time and date and print them in a pleasant
C format.

      INTEGER PARAMS(7)

C Get the time and date.

      CALL CLOCK(PARAMS)

C Print the time and date.

      WRITE(PRINTFILE,100) (PARAMS(I), I=2,7)
 100  FORMAT(X,'The time is : ',I2,'.',I2,'.',I2,/,
     *       X,'The date is : ',I2,'/',I2,'/',I4)
```

Scanned by Jonny Oddene for Sintran Data © 2011

TUSED - MON 114

Compute the CPU time used by a part of a Fortran program. The CPU time used by the user's terminal since the last logon is computed at two points in time (before and after the part in which we are interested), and the difference reveals the time used by the program part.

```
C Compute the time required to complete execution of a loop.

      INTEGER*4 CPUSTART, CPUFINISH, LOOPTIME, TUSED

C Get the CPU time prior to the loop.

      CPUSTART=TUSED(DUMMY)

C The loop we want to time.

      DO

C Something is done here.

      ENDDO

C The loop has finished. Compute how long it took.

      CPUFINISH=TUSED(DUMMY)
      LOOPTIME=CPUFINISH-CPUSTART
```

Scanned by Jonny Oddene for Sintran Data © 2011

RTWT - MON 135

Set the RT program issuing the call in a wait state until it is restarted, e.g. by another program calling RT (MON 100). The program which issued the RTWT call will restart at the statement immediately following the RTWT call.

```
PROGRAM HOPE

C Set this program in a wait state, hoping that it is restarted
C sooner or later.

      INTEGER TERMINAL

C Initialize the terminal's logical unit number.

      DATA TERMINAL/1/

C Take a rest for a while.

      CALL RTWT

C After being restarted, carry on from here. Tell the user.

      WRITE(TERMINAL,*)'Hurray, we are off again!'
```

Scanned by Jonny Oddene for Sintran Data © 2011

C.2.2 RETRIEVING/CHANGING DEVICE INFORMATION

INCH - MON 1

Read one byte from a device. If the device is a data link or word-oriented internal device, read one word.

```
C Read one byte into the variable ICHAR from a device. The device
C must have been opened and its Fortran unit number stored in the
C variable IFNUM (or a SINTRAN LDN may be used).

      ICHAR=INCH(IFNUM)

C If not successful, print a File System error message.

      IF (ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)
```

OUTCH - MON 2

Write one byte to a device or to a file. If the device is a data link or word oriented internal device, write one word.

```
C Write one byte from ICHAR to a file. The file must have been
C opened and its Fortran unit number stored in the variable IFNUM
C (or a SINTRAN LDN may be used).

      CALL OUTCH(IFNUM,ICHAR)

C If not successful, print a File System error message.
C
      IF (ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)
```

Scanned by Jonny Oddene for Sintran Data © 2011

WHDEV - MON 140

Check whether or not a logical device is reserved.

```
C Check whether or not a logical device is free for output, so
C that this program can continue processing.

      INTEGER TERMINAL,LOGDEVICE,OUTPUT,FREE
      INTEGER RTDESC, WHDEV

C Initialize the terminal's logical unit and logical device
C numbers.

      DATA TERMINAL/1/,LOGDEVICE/10/

C Initialize parameter values.

      DATA OUTPUT/1/,FREE/0/

C Get the device information.

      RTDESC=WHDEV(LOGDEVICE,OUTPUT)

C Stop if the device is not free for output. Otherwise carry
C on.

      IF(RTDESC .NE. FREE) THEN
        WRITE(TERMINAL,*) 'Sorry, device already
     *reserved'
        STOP
      ELSE

C Use the device.

      ENDIF
```

Scanned by Jonny Oddene for Sintran Data © 2011

OUTST - MON 162

Write a string of characters to a peripheral device.

```
C Write a message to the user's terminal.

      INTEGER IMSG,LENGTH,LOGUNIT
      CHARACTER*50 MESSAGE

C Equivalence to an integer so the message starts on a word
C boundary.

      EQUIVALENCE (IMSG,MESSAGE)

C Initialize the message and its length in bytes.

      DATA MESSAGE/'Hello from your friendly program'/
      DATA LENGTH/50/

C Initialize the logical unit number for the terminal.

      DATA LOGUNIT/1/

C Write the message.

      MONSTATUS=OUTST(LOGUNIT,MESSAGE,LENGTH)

C Check that all went well.

      IF(MONSTATUS .NE. 0)  CALL ERMSG(MONSTATUS)
```

Scanned by Jonny Oddene for Sintran Data © 2011

C.2.3 DEVICE HANDLING

ECHOM - MON 3

Set the echo strategy for a terminal.

```
C Set the echo strategy to not echo any characters (e.g. for
C entering a password), then reset the echo strategy.

      INTEGER TERMINAL,SUPPRESS,NORMAL

C Initialize the terminal logical unit number, and the echo
C strategies.

      DATA TERMINAL/1/,SUPPRESS/-1/,NORMAL/1/

C Set no echo for password processing.

      CALL ECHOM(TERMINAL,SUPPRESS)

C Get the user password, with appropriate checks.

  ...

C Reset to echo all characters on the terminal.

      CALL ECHOM(TERMINAL,NORMAL)
```

Scanned by Jonny Oddene for Sintran Data © 2011

BRKM - MON 4

Set a specific break strategy for a terminal.

```
C Set the break strategy so that the user program may examine
C every character as it is typed on the user's terminal.

      INTEGER TERMINAL,ALLCHARS

C Initialize the terminal's logical unit number.

      DATA TERMINAL/1/

C Set the argument value for break on all characters.

      DATA ALLCHARS/0/

C Set break on all characters for the user's terminal.

      CALL BRKM(TERMINAL,ALLCHARS)
```

Scanned by Jonny Oddene for Sintran Data © 2011

MSG - MON 32

Write a character string to the user's terminal.

```
C Send a message to the terminal.

      CHARACTER*80 MESSAGE

C On the ND-100, the character string must start on a word
C boundary. An equivalence with an integer variable will
C accomplish this.

      INTEGER I
      EQUIVALENCE(MESSAGE,I)

C Initialize the message text.

      DATA MESSAGE/'Dear user, have a nice day.'''/

C Send the message.

      CALL MSG(MESSAGE)
```

Scanned by Jonny Oddene for Sintran Data © 2011

ISIZE - MON 66

Get the number of bytes currently in the terminal input buffer, i.e. those characters which have not yet been read by the user program.

```
C A terminal is being used for operator input and output of
C messages. Check whether the operator has begun typing something,
C prior to output of a message to the terminal.

      INTEGER TERMINAL
      INTEGER INCHARS

C Initialize the terminal's logical unit number.

      DATA TERMINAL/1/

C Check if the operator has begun typing on the terminal.

      INCHARS=ISIZE(TERMINAL)
      IF(INCHARS .GT. 0) GO TO 10

C If not, output a message to the terminal.

      WRITE(TERMINAL,*)'Dear user, are you still there?'

     ...

C If the operator has begun typing, process his/her input before
C printing the message on the terminal.

   10 CONTINUE
```

Scanned by Jonny Oddene for Sintran Data © 2011

MAGTP - MON 144

This monitor call reads from, writes to, or performs a variety of control functions for magnetic tape devices. It may also be used with other devices with similar characteristics to magnetic tape devices, e.g. Versatec printers/plotters or floppy disks.

```
C Carry out read and rewind operations on a magnetic tape device.

C Define variables.

      INTEGER MAGTP, DATA(100), LOGUNIT
      INTEGER READREC, REWIND
      INTEGER STATUS, LENGTH, WORDSREAD, DUMMY

C Initialize the logical unit number and required functions.

      DATA LOGUNIT/32/
      DATA READREC/0/, REWIND/13B/

C Read a record, say 50 words.

      LENGTH=50
      STATUS=MAGTP(READREC,DATA,LOGUNIT,LENGTH,WORDSREAD)

C If all is not well, exit to error processing.

      IF(STATUS .NE. 0) GO TO ...

C Rewind the tape.

      STATUS=MAGTP(REWIND,LOGUNIT)

C If all is not well, exit to error processing.

      IF(STATUS .NE. 0) GO TO ...
```

Scanned by Jonny Oddene for Sintran Data © 2011

C.2.4 RETRIEVING/CHANGING FILE SYSTEM INFORMATION

SMAX - MON 73 and **REABT - MON 75**

SMAX sets the value of the maximum byte pointer of a file.

REABT reads the byte pointer as it would be used for the next sequential access of a mass storage file.

```
C After writing some records to a file, update the maximum byte
C pointer.

      INTEGER SMAX,REABT
      INTEGER*4 BYTEPOINTER,MAXBYTEPOINTER
      INTEGER LOGUNIT

C Initialize the logical unit number.

      DATA LOGUNIT/10/

C Open the file.

      OPEN(LOGUNIT,FILE='MY-DATA-FILE',ACCESS='SPECIAL',...)

C Write some records containing data to the file.

      ...

C Get the value of the byte pointer for the file.

      CALL REABT(LOGUNIT,BYTEPOINTER)

C Check that all is well.

      IF(ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)

C Update the value of the maximum byte pointer for the file.

      MAXBYTEPOINTER=BYTEPOINTER-1
      CALL SMAX(LOGUNIT,MAXBYTEPOINTER)

C Check that all is well.

      IF(ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)
```

Scanned by Jonny Oddene for Sintran Data © 2011

SETBS - MON 76

Set the block size of a file to a specific value temporarily (until the file is closed), which will then be used for random read and write operations.

In the example below, the FACTOR= specifier of the OPEN statement is set to 4, which is the default value for operation on the ND-500. This program can be used on the ND-100 without any changes; but note that the default value is 2 for the ND-100. The FACTOR= specifier in the OPEN statement might be set to 1, which means that the monitor calls RFILE, WFILE, SETBS and MAGTP will use block size in bytes, rather than using the default word sizes relevant to either machine. Note that an even number of bytes should be used.

```
C Set the block size to 4096 bytes. The default block size when
C the file is opened is 512 bytes (256 words on the ND-100).

      INTEGER LOGUNIT, FAC500, TBSBYTES, TBSUNITS

C Initialize the logical unit number.

      DATA LOGUNIT/10/

C Initialize the temporary block size in bytes.

      DATA TBSBYTES/4096/

C Initialize factor value to the default for the ND-500.

      DATA FAC500/4/

C Open the file, use FACTOR= to set the default value for ND-500.

      OPEN(LOGUNIT,FACTOR=FAC500,ACCESS='SPECIAL',...)

C Change from the default block size to 4096 bytes. This block
C size will be used until the file is closed, or another SETBS
C call is made.

      TBSUNITS=TBSBYTES/FAC500
      CALL SETBS(LOGUNIT,TBSUNITS)

C Check that all is well.

      IF(ERRCODE .NE. 0) CALL ERMSG(ERRCODE)
```

Scanned by Jonny Oddene for Sintran Data © 2011

C.2.5 FILE OPERATIONS

RFILE - MON 117 and **WFILE - MON 120**

Access a file randomly and read or write a specified number of words to or from a file block.

```
C Read a block of data, 2048 bytes, randomly from one file and
C write it randomly to another file. Then read the next record
C from the input file. The FACTOR= parameter in the OPEN statement
C is set to the default value for the ND-100, i.e. 2 bytes.

      INTEGER INUNIT, OUTUNIT, FAC100, READNEXT
      INTEGER DATA(1024), LENGTH, BLOCKNUM
      INTEGER IOCOMPLETE

C Initialize the logical unit numbers.

      DATA INUNIT/10/, OUTUNIT/12/

C Initialize the default value for the ND-100 in OPEN statement.

      DATA FAC100/2/

C Set block number to "read next block".

      DATA READNEXT/-1/

C Set argument to wait until I/O operation is complete.

      DATA IOCOMPLETE/0/

C Initialize length of the data area for records, in words.

      DATA LENGTH/1024/

C Open the files. Use factor= setting for word size on ND-100.

      OPEN(INUNIT,...,FACTOR=FAC100,ACCESS='SPECIAL'...)
      OPEN(OUTUNIT,...,FACTOR=FAC100,ACCESS='SPECIAL'..)
```

Scanned by Jonny Oddene for Sintran Data © 2011

```
C Set the block sizes for input and output.

      CALL SETBS(INUNIT,LENGTH)
      CALL SETBS(OUTUNIT,LENGTH)

C Read a block randomly (fifth block) from the input file.

      BLOCKNUM=4
      CALL RFILE(INUNIT,IOCOMPLETE,DATA,BLOCKNUM,LENGTH)

C Check that all is well.

      IF(ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)

C Write the block randomly (second block) to the output file.

      BLOCKNUM=1
      CALL WFILE(OUTUNIT,IOCOMPLETE,DATA,BLOCKNUM,LENGTH)

C Check that all is well.

      IF(ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)

C Now do a "read next block" operation from the input file i.e.
C read the sixth block.

      CALL RFILE(INUNIT,IOCOMPLETE,DATA,READNEXT,LENGTH)

C Check that all is well.

      IF(ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)
```

Scanned by Jonny Oddene for Sintran Data © 2011

RSIO - MON 143

This program will output a message only if the program is in interactive mode. The monitor call used, RSIO, finds out the execution mode of the calling program, the user number, and the input and output file numbers.

```
C Output a message only if the program is in interactive mode.

      INTEGER TERMINAL,INTERACTIVE
      INTEGER EXMODE,INDEVICE,OUTDEVICE,INXUSER

C Initialize the logical unit number.

      DATA TERMINAL/1/

C Initialize the value for interactive mode.

      DATA INTERACTIVE/0/

C Request the execution mode, etc.

      CALL RSIO(EXMODE,INDEVICE,OUTDEVICE,INXUSER)

C Output a message if the program is in interactive mode.

      IF(EXMODE .EQ. INTERACTIVE) THEN
                WRITE(TERMINAL,*) 'Hello there user'
      ENDIF
```

Scanned by Jonny Oddene for Sintran Data © 2011

WAITF - MON 121

Check the state of a mass storage transfer, or whether or not a transfer initiated by RFILE or WFILE is complete.

```
C Wait until an I/O transfer is complete, before continuing
C processing.

      INTEGER LOGUNIT,IONOWAIT,IOCOMPLETE
      INTEGER IOSTATUS

C Initialize the logical unit number.

      DATA LOGUNIT/10/

C Set argument values for desired actions.

      DATAIONOWAIT/1/,IOCOMPLETE/0/

C Open the mass storage file.

      OPEN(LOGUNIT,ACCESS='SPECIAL',FILE=...)

C Read a record from the file.

      CALL RFILE(LOGUNIT,IONOWAIT,...)

C The program can continue processing here.

      ...

C Set program in a wait state until I/O has finished.

      IOSTATUS=WAITF(LOGUNIT,IOCOMPLETE)

C Check that all is well.

      IF(IOSTATUS .GT. 0)  CALL ERMSG(ERRCODE)

C If all is well, then the I/O transfer has finished.
```

Scanned by Jonny Oddene for Sintran Data © 2011

C.2.6 ERROR HANDLING

ERMSG - MON 64

Print the File System error message corresponding to the value in the argument. This is often used to print an error message to explain the value of ERRCODE which has been set by an earlier monitor call.

```
C Print the appropriate File System error message.
C Note that an ERRCODE value of zero usually means all is well.

      IF(ERRCODE .NE. 0)  CALL ERMSG(ERRCODE)
```

Scanned by Jonny Oddene for Sintran Data © 2011

C.3 ND-100 AND ND-500 MONITOR CALLS

NOTE:

In the following table, "integer" in the column "data type" means default integer type: integer*2 on the ND-100, and integer*4 on the ND-500.

Name	MON oct	Fun or Sub	Arguments/return: Number, purpose	Data type	Comments	ERRCODE set
LEAVE	0	S	none	-	no return	no
INCH	1	F	1. input unit	integer	FORTRAN unit number	yes
see			return value	integer	see note 1	
note 2						
OUTCH	2	F	1. output unit	integer	FORTRAN unit number	yes
see			2. output character	integer	right-adjusted	
note 2			return value	integer	see note 1	
ECHOM	3	S	1. device	integer	SINTRAN LDN	no
			2. strategy	integer		
			3. table	integer*2 array	8 elements, optional	
BRKM	4	S	1. device	integer	SINTRAN LDN	no
			2. strategy	integer		
			3. table	integer*2 array	8 elements, optional	
			4. number of characters	integer	optional	
TIME	11	F	return value	integer*4		no
SETCM	12	S	1. command string	character	see note 4	no
CIBUF	13	F	1. unit	integer	FORTRAN unit number	yes
			return value	integer	ERRCODE	
COBUF	14	F	1. unit	integer	FORTRAN unit number	yes
			return value	integer	ERRCODE	

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
MGTTY	16	F	1. unit	integer	FORTRAN unit number	yes
			2. terminal type	integer		
			return value	integer	ERRCODE	
MSTTY	17	F	1. unit	integer	FORTRAN unit number	yes
			2. terminal type	integer		
			return value	integer	ERRCODE	
LASTC	26	F	1. device	integer	SINTRAN LDN	no
			return value	integer	right-adjusted	
RTDSC	27	F	1. RT desc. address	integer		no
			2. RT desc. copy	integer array	26 elements	
			return value	integer		
GETRT	30	F	return value	integer		no
EXIOX	31	F	1. register content	integer	A-register (ND-100) I1-register (ND-500)	no
			2. dev. register address	integer		
			return value	integer		
MSG	32	S	1. message	character	see note 4	yes
ALTON	33	S	1. page table number	integer	must be used with APT COMMON	yes
ALTOF	34	S	none		must be used with APT COMMON	no
IOUT	35	S	1. radix	integer	2, 8, 10 or 16	no
			2. value	integer	see note 6	
NOWT	36	S	1. device	integer	SINTRAN LDN	no
			2. I/O flag	integer		
			3. wait flag	integer		
AIRDW	37	S	1. number of channels	integer	=N	no
			2. channel numbers	integer*2 array	N 16-bit elements	
			3. read values	integer*2 array	N 16-bit elements	
			4. error flag	integer		

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
SPCLO	40	F	1. unit	integer	FORTRAN unit number	yes
			2. text string	character	see note 4	
			3. number of copies	integer		
			4. print flag	integer		
			return value	integer	ERRCODE	
ROBJE	41	F	1. unit	integer	FORTRAN unit number	yes
			2. buffer	integer*2 array	32 elements	
			return value	integer	ERRCODE	
RUSER	44	F	1. user name	character	see note 4	yes
			2. buffer	integer*2 array	32 elements	
			return value	integer	ERRCODE	
TERMO	52	S	1. device	integer	SINTRAN LDN	no
			2. mode	integer		
MDLFI	54	F	1. file name	character	see note 4	yes
			return value	integer	ERRCODE	
PASET	56	S	1-5 parameters	integer		no
PAGET	57	S	1-5 parameters	integer		no
RMAX	62	F	1. unit	integer	FORTRAN unit number	yes
			2. number of bytes	integer*4		
			return value	integer	ERRCODE	
ERMSG	64	F	1. error number	integer	see note 7	no
QERMS	65	S	1. error number	integer	see note 7	no
ISIZE	66	F	1. unit	integer	FORTRAN unit number	yes
			return value	integer	see note 1	

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
OSIZE	67	F	1. unit	integer	FORTRAN unit number	yes
			return value	integer	see note 1	
COMND	70	F	1. command string	character	see note 4	yes
DESCF	71	S	1. device	integer	SINTRAN LDN	no
EESCF	72	S	1. device	integer	SINTRAN LDN	no
SMAX	73	F	1. unit	integer	FORTRAN unit number	yes
			2. byte count	integer*4		
			return value	integer	ERRCODE	
SETBT	74	F	1. unit	integer	FORTRAN unit number	yes
			2. byte pointer	integer*4	1st byte has number 0	
			return value	integer	ERRCODE	
REABT	75	F	1. unit	integer	FORTRAN unit number	yes
			2. byte pointer	integer*4	1st byte has number 0	
			return value	integer	ERRCODE	
SETBS	76	F	1. unit	integer	FORTRAN unit number	yes
			2. block size	integer	see note 5	
			return value	integer	ERRCODE	
SETBL	77	F	1. unit	integer	FORTRAN unit number	yes
			2. block number	integer	1st block has number 0	
			return value	integer	ERRCODE	
RT	100	S	1. RT program	external or integer		no
SET	101	S	1. RT program	external or integer		no
			2. number of time units	integer		
			3. basic unit	integer		

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
ABSET	102	S	1. RT program	external or integer		no
			2. seconds	integer		
			3. minutes	integer		
			4. hours	integer		
INTV	103	S	1. RT program	external or integer		no
			2. number of time units	integer		
			3. basic unit	integer		
HOLD	104	S	1. number of time units	integer		no
			2. basic units	integer		
ABORT	105	S	1. RT program	external or integer		no
CONCT	106	S	1. RT program	external or integer		no
			2. device	integer	SINTRAN LDN	
DSCNT	107	S	1. RT program	external or integer		no
PRIOR	110	F	1. RT program	external or integer		no
			2. priority	integer		
			return value	integer		
UPDAT	111	S	1-5 time	integer		no
CLADJ	112	S	1. number of time units	integer		no
			2. basic units	integer		
CLOCK	113	S	1. time	integer array	7 elements	no
TUSED	114	F	return value	integer*4		no
FIX	115	S	1. segment number	integer		no

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
WFILE	120	F	1. unit	integer	FORTRAN unit number	yes
			2. return flag	integer		
			3. memory address	array	any type except char.	
			4. block number	integer		
			5. length to be written	integer	count in units (see note 5)	
UNFIX	116	S	1. segment number	integer		no
RFILE	117	F	1. unit	integer	FORTRAN unit number	yes
			2. return flag	integer		
			3. memory address	array	any type except char.	
			4. block number	integer		
			5. length to be read	integer	count in units (see note 5)	
			return value	integer	ERRCODE	
WAITF	121	F	1. unit	integer	FORTRAN unit number	yes
			2. return flag	integer		
			return value	integer		
RESRV	122	F	1. device	integer	SINTRAN LDN	no
			2. I/O flag	integer		
			3. return flag	integer		
			return value	integer		
RELES	123	S	1. device	integer	SINTRAN LDN	no
			2. I/O flag	integer		
PRSRV	124	F	1. device	integer	SINTRAN LDN	no
			2. I/O flag	integer		
			3. RT program	external or integer		
			return value	integer		
PRLS	125	S	1. device	integer	SINTRAN LDN	no
			2. I/O flag	integer		

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
DSET	126	S	1. RT program	external or integer		no
			2. delay	integer*4		
DABST	127	S	1. RT program	external or integer		no
			2. time	integer*4		
DINTV	130	S	1. RT program	external or integer		no
			2. time interval	integer*4		
ABSTR	131	S	1. device	integer	SINTRAN LDN	no
			2. function	integer		
			3. memory address	integer*4	double integer	
			4. block address	integer		
			5. number of blocks	integer		
MEXIT	133	S	1. segment number	integer	right byte only see note 3	no
RTEXT	134	S	none	-	no return	
RTWT	135	S	none	-		no
RTON	136	S	1. RT program	external or integer		no
RTOFF	137	S	1. RT program	external or integer		no
WHDEV	140	F	1. device	integer	SINTRAN LDN	no
			2. I/O flag	integer		
			return value	integer		

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
IOSET	141	F	1. device	integer	SINTRAN LDN	no
			2. I/O flag	integer		
			3. RT desc. address	integer		
			4. control flag	integer		
			return value	integer		
ERMON	142	S	1. error number	integer	Hollerith string of two bytes	no
			2. suberror number	integer		
RSIO	143	S	1. execution mode	integer		no
			2. input device	integer	SINTRAN LDN	
			3. output device	integer	SINTRAN LDN	
			4. directory and user index	integer		
MAGTP	144	F	1. function	integer		yes
			2. memory address	array	any type except char.	
			3. unit	integer	FORTRAN unit number	
			4. parameter 1	integer	device dependent (optional)	
			5. parameter 2	integer	device dependent (optional)	
			return value	integer	ERRCODE	
ACM	145	F	1. device	integer	SINTRAN LDN	yes
			2. function	integer		
			3. memory address	array	any type except char.	
			4. destination	array	any type except char.	
			5. word count	integer	number of words	
			return value	integer	ERRCODE	
CAMAC	147	S	1. data	integer		no
			2. status	integer		
			3. crate	integer		
			4. station	integer		
			5. subaddress	integer		
			6. function	integer		
GL	150	S	1. value	integer		no
			2. crate	integer		

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
GRTDA	151	F	1. name	Hollerith	ends with apostrophe	no
			return value	integer		
IOXN	153	S	1. data	integer		no
			2. IOX code	integer		
ASSIG	154	S	1. device	integer	SINTRAN LDN	no
			2. graded LAM	integer		
			3. crate	integer		
PLOTT	155	F	1,2 X,Y co-ordinates	integer		no
GRAPHI	C		3. code	integer		
			4. device	integer	SINTRAN LDN	
			5. function	integer		
			return value	integer		
ENTSG	157	S	1. segment	integer		no
			2. page table	integer		
			3. interrupt level	integer		
			4. entry point	integer		
FIXC	160	F	1. segment number	integer		no
			2. page number	integer		
			return value	integer		
INSTR	161	F	1. unit	integer	FORTRAN unit number	yes
			2. text	character	starts on a word boundary	
			3. length	integer	length in bytes	
			4. end character	integer	right hand byte used	
			return value	integer		
OUTST	162	F	1. unit	integer	FORTRAN unit number	yes
			2. text	character	starts on a word boundary	
			3. length	integer	length in bytes (-1 = print all characters)	
			return value	integer		
WSEG	164	S	1. segment number	integer		no

Scanned by Jonny Oddene for Sintran Data © 2011

Name	MON oct	Fun or Sub	Arguments/return			ERRCODE set
			Number, purpose	Data type	Comments	
DIW	165	S	1. number of registers	integer	=N	no
			2. input registers	integer*2 array	N elements	
			3. input values	integer*2 array	N elements	
			4. error value	integer		
DOLW	166	S	1. number of registers	integer	=N	no
			2. registers	integer*2 array	N elements	
			3. output values	integer*2 array	N elements	
			4. masks	integer*2 array	N elements	
			5. error value	integer		
REENT	167	S	1. segment number	integer	see note 3	no
HDLC	201	F	1. SDCB/RDCB	integer		yes
			2. LDN	integer	SINTRAN LDN	
			3. DCB address	integer		
			4. DCB usize	integer		
			5. DCB msize	integer		
			return value	integer	ERRCODE	
EDTRM	206	S	1. EDFLA	integer	flag	no
			2. RTUSF	integer		
CPUST	262	F	1. ND number	integer	always zero	yes
			2. buffer	integer*2 array	system information	
			return value	integer	ERRCODE	

Scanned by Jonny Oddene for Sintran Data © 2011

NOTES:

1) If there was an error, the function returns the error code with the sign bit set.

2) The names of the monitor calls corresponding to these routines are:

   INCH - INBT
   OUTCH - OUTBT

3) Available on the ND-100 only.

4) Must start on a word boundary (ND-100 only) and end with an apostrophe.

5) The unit is defined by the FACTOR=fac specifier of the OPEN statement. The default is a 2-byte word on the ND-100, and a 4-byte word on the ND-500.

6) The radixes 2 and 16 are available on the ND-500 only.

7) The error number can be picked up from ERRCODE without change.

Scanned by Jonny Oddene for Sintran Data © 2011

A P P E N D I X D

LIBRARY UTILIITY FUNCTIONS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

LIBRARY UTILITY FUNCTIONS

D.1 TABLE OF LIBRARY UTILITY FUNCTIONS

N O T E

In the following table, "integer" in the column "data type" means default integer type; integer*2 on the ND-100 and integer*4 on the ND-500.

Name	*Fun or Sub*	*Arguments/return*		
		Number, purpose	*Data type*	*Comments*
EXCEPT	S	1. exception number	integer	
		2. function	integer	
		3. user routine	integer	
		4. number of messages	integer	
		5. number of traps	integer	
		6. exception flags	logical array	
		7. lower bound of 6.	integer	
		8. upper bound of 6.	integer	
EXCDEF	S	1. exception number	integer	
		2. exception flags	logical array	
		3. lower bound of 2.	integer	
		4. upper bound of 2.	integer	
EXCTERM	S	1. traceback print	integer	
		2. statistics print	integer	
		3. number of levels	integer	
		4. file number	integer	
GETMESS	F	1. exception number	integer	
		return value	character*50	
PRITRAC	S	1. trap	logical	
PRIMESS	S	1. exception number	integer	
RAN	F	1. seed value	integer*4	
		return value	real*4	real*6, 48-bit f.p.H/W

Scanned by Jonny Oddene for Sintran Data © 2011

Name	Fun or Sub	Arguments/return		
		Number, purpose	Data type	Comments
RDEFVAL	S	1. exception number	integer	
		2. number of messages	integer	
		3. number of exceptions	integer	
		4. enable flag	integer	
RCURVAL	S	1. exception number	integer	
		2. user routine	integer	
		3. number of messages	integer	
		4. number of exceptions	integer	
		5. exception count	integer	
		6. enable flag	logical	
REXTERM	S	1. traceback print	integer	current values
		2. statistics print	integer	
		3. number of levels	integer	
		4. file number	integer	
		5. traceback print	integer	default values
		6. statistics print	integer	
		7. number of levels	integer	
		8. file number	integer	

Scanned by Jonny Oddene for Sintran Data © 2011

D.2 LIBRARY SUBPROGRAM DESCRIPTIONS

This section contains a full description of each subprogram provided in the FORTRAN library, for general utility purposes. Since the topic of handling errors and exceptions is of a rather special nature, it is described separately, see Section D.3. All services provided by the SINTRAN operating system are described in Appendix C.

D.2.1 THE RAN FUNCTION

The RAN function is for generating random numbers, which are uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive.

The technique used for generating the random numbers is of the multiplicative congruential type.

The function returns a REAL*4 value, REAL*6 on a ND-100 with 48-bit floating point hardware.

The function may be invoked repeatedly, as follows:

RLVAR=RAN(INTVAR)

where

RLVAR is assigned the next random number generated.

INTVAR is an INTEGER*4 variable.

To get a series of random numbers, the first invocation of RAN must be made with the argument, here INTVAR, set to a large odd integer value prior to this invocation.

The RAN function stores a value in the argument on each invocation. This value will be used in the next invocation, to compute the next random number. This value is referred to as the **seed**.

There are no restrictions on the value which may be used for the **seed**. It should be initialized to a different value for successive runs if different series of random numbers are required. The series of random numbers are reproducible, if the same value of **seed** is used.

Scanned by Jonny Oddene for Sintran Data © 2011

The RAN function uses the following algorithm to compute the value of the **seed** to be used for the next invocation:

```
SEED=69069*SEED+1(MOD 2**32)
```

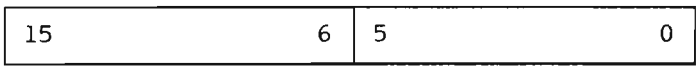
SEED is a 32-bit number whose high order 24 bits are converted to a floating-point value to be returned as the function value.

Scanned by Jonny Oddene for Sintran Data © 2011

D.3 ND-500 TRAPS AND EXCEPTION HANDLING
ND-100 EXCEPTION HANDLING

The term "exception" covers, in addition to all defined hardware traps, special situations and errors detected by software. An exception handler is a routine to be activated when an exception occurs, and which takes take appropriate recovery actions.

The exception number (16 bits) may be represented as shown below:



where

bits 15-6 = System software identification (SSI)

bits 5-0 = Specific status code (SC)

and any number fed into the exception handling system will be in this form.

For FORTRAN, the SSI may contain three different ranges of numbers. For ND-500 hardware traps the range will be of the form 76**xx**B, where **xx** specifies the trap, see the table in Section D.3.1. The range 04**xx**B is reserved for FORTRAN runtime errors, and the range 51xxB is used by the exception handling system itself. Status codes are numbers allocated to a particular system. For example, the list of FORTRAN Exceptions given in Section D.3.11, gives an explanation of numbers in the range 401B: 457B, where 4 is the FORTRAN SSI. The range prefixed by 777B is not used by ND system software, and any be freely utilized in user systems.

Scanned by Jonny Oddene for Sintran Data © 2011

A set of standard routines for exception handling for use with FORTRAN or PLANC has been developed. These are available in a standard library, and will be linked automatically if the user so wishes!!!

For each error condition, the user may determine:

- The number of times each error message is to be printed.
- The number of times an error may occur before the program is abnormally terminated.
- Whether a user-supplied exception handler is to be activated upon detection of an error.
- Whether traceback of routine stack frames is to be printed when the error occurs or when the program terminates. (In case of traps, this includes a register dump.)
- Printout of error statistics when the program terminates.

The library consists of the following routines:

EXCEPT - disable/enable handling of specified exception

EXCDEF - reset handling of exception to default

EXCTERM - define action to be taken upon program termination

PRITRAC - print traceback of routine instances (subroutines)

PRIMESS - print error message

GETMESS - return error text (FORTRAN)

PGETMESS - return error text (PLANC)

RDEFVAL - read default exception handling parameters values

RCURVAL - read current exception handling parameters values

REXTERM - reat exception terminating condition

In the following descriptions, the header of these routines is described, giving the number and types of the arguments. These routines are supplied with the standard ND FORTRAN library.

Scanned by Jonny Oddene for Sintran Data © 2011

• For the ND-500:

Traps and exceptions will be handled in the ND-500, providing they are locally enabled. There are default settings for all traps. If no local handling has been specified, or the trap has been disabled, then some traps may be handled as a system trap in the ND-100. The Monitor will then handle the trap in a standard manner, depending on the type of trap. System traps may also be disabled, but the user's right to modify trap handling may be restricted.

Handling of traps may be determined at load time or before execution through the commands LOCAL-TRAP-ENABLE, LOCAL-TRAP-DISABLE, SYSTEM-TRAP-ENABLE and SYSTEM-TRAP-DISABLE. These commands are available both in NLL and the Monitor, and set the default values to be used if no action is taken by the program. These commands are described in the ND-500 Loader/Monitor Manual, ND-60.136.

• For the ND-100:

Exceptions will be handled provided they are locally enabled. There are default settings for all exceptions.

Scanned by Jonny Oddene for Sintran Data © 2011

D.3.1 ND-500 TRAPS TABLE

The following is a list of defined hardware traps, their corresponding bit number in the status, OTE, MTE and TEMM registers, and the name of the trap. For a more detailed explanation, see the ND-500 CPU Reference Manual (ND-05.009).

Bit no.	*Exc. no.*	*Name*	*Mnemonic*	*D*	*msg*	*err*
9	7611B	OVERFLOW	O		10	unl
11	7613B	INVALID OPERATION	IVO	*	10	unl
12	7614B	DIVISION BY ZERO	DZ	*	10	unl
13	7615B	FLOATING UNDERFLOW	FU		10	unl
14	7616B	FLOATING OVERFLOW	FO	*	10	unl
15	7617B	BCD OVERFLOW	BO		10	unl
16	7620B	ILLEGAL OPERAND VALUE	IVO	*	10	unl
17	7621B	SINGLE INSTRUCTION TRAP	SIT		0	unl
18	7622B	BRANCH TRAP	BT		0	unl
19	7623B	CALL TRAP	CT		0	unl
20	7624B	BREAKPOINT INSTRUCTION TRAP	BPT		0	unl
21	7625B	ADDRESS TRAP FETCH	ATF		0	unl
22	7626B	ADDRESS TRAP READ	ATR		0	unl
23	7627B	ADDRESS TRAP WRITE	ATW		0	unl
24	7630B	ADDRESS ZERO ACCESS	AZ		10	unl
25	7631B	DESCRIPTION RANGE	DR		10	unl
26	7632B	ILLEGAL INDEX	IX	*	1	0
27	7633B	STACK OVERFLOW	STO	*	1	0
28	7634B	STACK UNDERFLOW	STU	*	0	0
29	7635B	PROGRAMMED TRAP	PRT	*	10	unl
30	7636B	DISABLE PROCESS SWITCH TIMEOUT	DT	*	1	0
31	7637B	DISABLE PROCESS SWITCH ERROR	DE	*	1	0
32	7640B	INDEX SCALING ERROR	XSE	*	1	0
33	7641B	ILLEGAL INSTRUCTION CODE	IIC	*	1	0
34	7642B	ILLEGAL OPERAND SPECIFIER	IOS	*	1	0
35	7643B	INSTRUCTION SEQUENCE ERROR	ISE	*	1	0
36	7644B	PROTECT VIOLATION	PV	*	1	0

Scanned by Jonny Oddene for Sintran Data © 2011

The **D** column refers to the default enabling of traps used by the standard exception handler library discussed in the next sections. The * indicates that the trap is enabled if the default trap library settings are used. **msg** = default maximum number of error messages.

err = default number of exceptions prior to abnormal termination.

unl = unlimited number

D.3.1.1 ND-100 SIMULATED TRAPS

In the list of the simulated traps listed below, the **D** refers to the default enabling used by the standard exception handler library discussed in the next sections. * indicates that the exception is enabled if the default settings are used. For an explanation of **msg** and **err** see Section D.3.1.

Exc. no.	*Name*	*D*	*msg*	*err*
7614B	DIVISION BY ZERO	*	10	unl
7633B	STACK OVERFLOW	*	1	0

D.3.2 THE EXCEPT ROUTINE

The EXCEPT routine is used to modify the current exception handling conditions.

FORTRAN Specification:

```
      SUBROUTINE EXCEPT (EXCINO, EXCFUN, EXCROUT, NOMSG, NOEXC,
     *EXCARR, EXCNOL, EXCNOH)
      INTEGER EXCNO, EXCFUN, EXCROUT, NOMSG, NOEXC, EXCNOH, EXCNOL
      LOGICAL EXCARR (EXCNOL:EXCNOH)

      <standard library routine>

      END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Parameter values:

EXCNO Exception number or exception number group for the ND-500:

Exception Number	*Meaning*
7600B	default group of traps to be set (see Section D.3.1)
5101B	LOGICAL array (EXCARR, EXCNOL and EXCNOH must be present, FORTRAN)
5102B	BITS (EXCARR must be present, PLANC)
7611B:7644B	specific trap number
400B	all FORTRAN errors (see Section D.3.11)
401B:457B	specific FORTRAN error
other	illegal

For the ND-100:

Exception Number	*Meaning*
7600B	default group of simulated traps to be set (see Section D.3.1.1)
5101B	LOGICAL array (EXCARR, EXCNOL and EXCNOH must be present, FORTRAN)
5102B	BITS (EXCARR must be present, PLANC)
7614B:7633B	specific simulated traps
400B	all FORTRAN errors (see Section D.3.11)
401B:457B	specific FORTRAN error
other	illegal

Scanned by Jonny Oddene for Sintran Data © 2011

EXCFUN Function:

Value of EXCFUN	*Meaning*
-1	disable exception(s) indicated by EXCNO and ignore all other exceptions. In Addition, the parameters EXROUT, NOMSG and NOEXC will be ignored.
0	enable exception(s) indicated by EXCNO as TRUE, set new handler/values, and disable all other exceptions which are indicated as FALSE. For EXCNO values 7611B:7644B on the ND-500 (or 7614B:7633B on the ND-100) or 401B:457B, only the single exception thus specified, is enabled.
1	enable exception(s), indicated by EXCNO, do not modify handler/values, and ignore all other exceptions.
other	illegal

EXCROUT User defined exception handler routine:

Value of EXCROUT	*Meaning*
not 0	routine address
0	no routine supplied

NOMSG Number of messages allowed before program is aborted:

Value of NOMSG	*Meaning*
-1	any number of messages allowed
≥0	number of messages allowed (<2**31-1)
other	illegal

Scanned by Jonny Oddene for Sintran Data © 2011

NOEXC Number of traps before program is aborted:

Value of NOEXC	*Meaning*
-1	any number of exceptions allowed
≥0	number of exceptions allowed (<2**31-1)
other	illegal

EXCARR LOGICAL array (FORTRAN) or BITS(PLANC) containing .TRUE. and .FALSE. for exceptions to be handled

EXCNOL (FORTRAN) Low limit of EXCARR

EXCNOH (FORTRAN) High limit of EXCARR

The handling of one or several exception conditions may be modified, selected through the EXCNO parameter. If this parameter is 5101B (FORTRAN) or 5102B (PLANC), the EXCARR parameter selects a set of exceptions to be handled. If the EXCFUN parameter is zero and EXCARR is present, the elements set to .TRUE. in this array will cause the corresponding exception to be enabled, while .FALSE. will cause it to be disabled. The array EXCARR must be declared as EXCARR (EXCARREXCNOH). For example, EXCARR(7611B:7644B) on the ND-500 or EXCARR (7614B:7633B) on the ND-100.

The EXCROUT parameter specifies the name of a user supplied routine to be executed when the exception occurs. The routine should conform to the following formal specification:

- In FORTRAN:

```
SUBROUTINE name (ierno)
INTEGER ierno
      .
<user written exception handler>
      .
END
```

Scanned by Jonny Oddene for Sintran Data © 2011

The parameter **ierno** will transfer the error number to the exception handler. If the EXCROUT parameter is zero, the standard exception handler routine from the library is used.

After an error has occured, the sequence of operations is as follows; the steps marked with an asterisk apply to traps on the ND-500 only:

NOTE: the details are slightly different in PLANC.

1) If the exception is a trap, the trap routine is activated.

2) A system provided exception handler is called.

3) This handler updates the occurrence counter for this type of exception and activates the user exception handler if one has been specified.

4) If the traceback condition (see note 1) is true, the system outputs:

   - register dump
   - traceback printout

5) If the message occurrence limit (NOMSG) has not been exceeded, or if the traceback condition (see note 1) is true, an error message is printed.

6) If the error count is less than or equal to the allowed number of occurrences for this exception type, control is returned to normal FORTRAN error handling.

   otherwise, the program is abnormally terminated with error statistics, if specified.

If the exception occurs during the execution of FORTRAN I/O statements (regardless of the type exception, SINTRAN, FORTRAN I/O, trigonometric error (430B:457B), or trap on the ND-500), the exception handler must not perform FORTRAN I/O. That is READ, WRITE, PRINT, OPEN, CLOSE, BACKSPACE, ENDFILE, or REWIND. Monitor Calls, however, may be called directly. Otherwise, FORTRAN I/O may be used, provided no new error situations are generated.

Scanned by Jonny Oddene for Sintran Data © 2011

N O T E

In FORTRAN, on the ND-500, the STACK UNDERFLOW trap condition is handled by special software mechanisms and must, in order to ensure correct termination of the I/O activities, always be default enabled.

N O T E

The traceback condition is evaluated by the following expression:

```
thiserror><'STACK UNDERFLOW' and
( ( TRACEBACK=2 and
    ( thiserror.NOMSG=unl or
      thiserror.numerrors in 0 : thiserror.NOMSG ) )
  or
  (TRACEBACK> 1 and
   (thiserror.NOEXC><UNL and
    NOT thiserror.numerrors in 0 : thiserror.NOEXC ) ) )
```

where
thiserror.numerrors is the current value of the number of exceptions of this type which have occurred.

EXAMPLES - FORTRAN

- Enable DIVISION BY ZERO detection using current exception values:

```
C DIVISION BY ZERO is trap number 12 on the ND-500
      CALL EXCEPT (7614B,1,0,0,0)
```

- For the ND-500 only: Enable OVERFLOW and allow a maximum of 2 error messages and 10 overflow errors before abnormal termination. Activate the user defined routine MYROUT each time the overflow trap occurs.

```
      CALL EXCEPT (7611B,0,MYROUT,2,10)
```

Scanned by Jonny Oddene for Sintran Data © 2011

- Disable error handling for exponential functions, FORTRAN error numbers 431B, 432B, 433B, 437B.

```
      LOGICAL ERRARRAY (431B:437B)
      DATA ERRARRAY/.FALSE.,.FALSE.,.FALSE.,.TRUE.,
     *               .TRUE.,.TRUE.,.FALSE./
      CALL EXCEPT (5101B,-1,0,0,0,ERRARRAY,431B,437B)
```

- Manipulation of some exception settings. Assume the following are the current table settings for exceptions:

Exc.no. (octal)	*EXCROUT setting*	*msg*	*err*	*Setting*
...				
431	A	10	unl	enabled
432	A	10	unl	enabled
433	A	10	unl	enabled
434	0	10	20	disabled
435	0	10	unl	enabled
436	0	10	unl	disabled
437	0	10	50	enabled
...				

If the following call were executed,

```
      CALL EXCEPT (5101B,0,MYROUT,5,-1,ERRARRAY,431B,437B)
C  ERRARRAY as declared in the previous example
```

then the table settings would be changed as follows:

Exc.no. (octal)	*EXCROUT setting*	*msg*	*err*	*Setting*
...				
431	A	10	unl	disabled
432	A	10	unl	disabled
433	A	10	unl	disabled
434	MYROUT	5	unl	enabled
435	MYROUT	5	unl	enabled
436	MYROUT	5	unl	enabled
437	0	10	50	disabled

Scanned by Jonny Oddene for Sintran Data © 2011

D.3.3 THE EXCDEF ROUTINE

EXCDEF is used to set the default exception handling values for a given set of exceptions. This is functionally equivalent to calling EXCEPT with the default parameter values for each of the exceptions specified, but is more convenient and relieves the programmer from knowing the defaults.

FORTRAN Specification:

```
SUBROUTINE EXCDEF [EXCNO, EXCARR, EXCNOL, EXCNOH]
INTEGER EXCNO, EXCNOL, EXCNOH
LOGICAL EXCARR [EXCNOL:EXCNOH]
<standard library routine>
END
```

Parameter values:

EXCNO Exception number or exception number group for the ND-500:

Value of EXCNO	*Meaning*
7600B	default setting (see Section D.3.1)
5101B	LOGICAL array (EXCARR, EXCNOL and EXCNOH present, FORTRAN)
5102B	BITS (EXCARR present PLANC)
7611B:7644B	default setting for specific trap number (see Section D.3.1)
400B	all FORTRAN errors
401B:457B	specific FORTRAN error
other	illegal

Scanned by Jonny Oddene for Sintran Data © 2011

For the ND-100:

Value of EXCNO	*Meaning*
7600B	default setting (see Section D.3.1)
5101B	LOGICAL array (EXCARR, EXCNOL and EXCNOH present, FORTRAN)
5102B	BITS (EXCARR present, PLANC)
7614B:7633B	default setting for specific simulated traps (see Section D.3.1)
400B	all FORTRAN errors
401B:457B	specific FORTRAN error
other	illegal

EXCARR LOGICAL array (FORTRAN) or BITS (PLANC) containing .TRUE. for exceptions to be handled, .FALSE. for those that should remain as they are

EXCNOL (FORTRAN) Low limit of EXCARR

EXCNOH (FORTRAN) High limit of EXCARR

The EXCARR parameter selects a set of exception conditions, like in the EXCEPT routine. Alternatively, one specific exception may be selected through the EXCNO parameter.

EXAMPLES - FORTRAN:

- Reset handling of all traps and FORTRAN errors to default:

```
C All traps (on ND-500), all simulated traps (ND-100)
C
      CALL EXCDEF (7600B)
C
C All fortran errors
C
      CALL EXCDEF (400B)
C
C Set default program termination conditions
C
      CALL EXCTERM (0, 1, 20, 0); % on the ND-500
      CALL EXCTERM (0, 0, 20, 0); % on the ND-100
```

Scanned by Jonny Oddene for Sintran Data © 2011

This setting is identical to the setting at the beginning of execution of a FORTRAN program.

- Reset special error handling for exponential functions, error numbers 431B, 432B, 433B and 437B, but keep possible handling of other exceptions:

```
LOGICAL ERRARRAY (431B:437B)
DATA ERRARRAY/.TRUE.,.TRUE.,.TRUE.,FALSE.,
* .FALSE.,.FALSE.,.TRUE./
CALL EXCDEF (5101B,ERRARRAY,431B,437B)
```

D.3.4 THE EXCTERM ROUTINE

EXCTERM may be called to determine how the printing of traceback and error statistics information is done. If it has been called more than once, the last call applies at program termination.

FORTRAN Specification:

```
SUBROUTINE EXCTERM (TRACEBACK,PRSTAT,NOLEV,FNUMB)
INTEGER TRACEBACK,PRSTAT,NOLEV,FNUMB
<standard library routine>
END
```

Parameter value:

TRACEBACK traceback print, for all errors:

Value of TRACEBACK	*Meaning*
0	no traceback (default)
1	traceback upon abnormal termination
2	traceback upon error
other	illegal

Scanned by Jonny Oddene for Sintran Data © 2011

PRSTAT error statistics print upon end of program, for all errors:

Value of PRSTAT	*Meaning*
0	no statistics output (default, ND-100)
1	print statistics (default, ND-500)
other	illegal

NOLEV maximum number of levels to process when a traceback is provided:

Value of NOLEV	*Meaning*
>0	maximum no. of stack levels to print, default 20
other	illegal

FNUMB

Value of FNUMB	*Meaning*
1-127	file number of an open file where all errors in information printout is to be directed (except MON64 type output). The file must be open with access type W.
0	reset to terminal (1) output (default)
other	illegal

Note the difference between a file with number 1 and terminal 1.

Scanned by Jonny Oddene for Sintran Data © 2011

D.3.5 THE PRITRAC ROUTINE

PRITRAC is a utility routine to print a traceback of routine instances (stack frames) after an exception. The routine is called from a user handler, or automatically upon abnormal termination of the job if traceback has been selected (in the EXCEPT call referring to the exception condition raised).

FORTRAN Specification:

```
SUBROUTINE PRITRAC (TRAP)
LOGICAL TRAP
<standard library routine>
END
```

Parameter value(which is ignored in the ND-100 version):

TRAP .TRUE. if called while a trap is being handled.

.FALSE. should be set for any other condition.

Note that the default maximum number of stack levels to be printed is 20.

D.3.6 THE PRIMESS ROUTINE

The PRIMESS routine will print the error message, corresponding to the parameter value, on the standard output device (unit 1).

FORTRAN Specification:

```
SUBROUTINE PRIMESS (EXCNO)
INTEGER EXCNO
<standard library routine>
END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Parameter values:

EXCNO Exception number (for the ND-500)

The parameter (EXCNO) must be in the range 7611B:7644B (traps) or 401B:457B (FORTRAN errors).

Exception number (for the ND-100)

The parameter (EXCNO) must be in the range 7614B:7633B (simulated traps) or 401B:457B (FORTRAN errors).

D.3.7 THE GETMESS/PGETMESS ROUTINE

GETMESS/PGETMESS will return the error text corresponding to the specified exception number.

FORTRAN Specification:

```
      FUNCTION GETMESS (EXCNO)
C This function must be declared to be of type character in the
C calling program
      INTEGER EXCNO
      CHARACTER*50 GETMESS
      <standard library routine>
      END
```

Parameter values:

EXCNO The number of an exception condition (for the ND-500)

EXCNO must (for the ND-500) be the number of a defined exception condition, in the range 7611B:7644B (traps) or 401B:457B (FORTRAN error).

EXCNO must (for the ND-100) be the number of a defined exception condition, in the range 7614B:7633B (simulated trap) or 401B:457B (FORTRAN error).

Scanned by Jonny Oddene for Sintran Data © 2011

D.3.8 THE RDEFVAL ROUTINE

RDEFVAL may be called to read the default values of the exception parameters corresponding to a given exception number (EXCNO).

FORTRAN Specification:

```
SUBROUTINE RDEFVAL (EXCNO, NOMSG, NOEXC, ENABL)
INTEGER EXCNO, NOMSG, NOEXC,
LOGICAL ENABL
<standard library routine>
END
```

Parameter values:

EXCNO Exception number

NOMSG Default number of messages allowed

NOEXC Default number of exceptions allowed

ENABL Logical parameter .TRUE. if exception is enabled as default.

D.3.9 THE RCURVAL ROUTINE

RCURVAL may be called to read the current values of the exception parameters corresponding to a given exception number (EXCNO).

FORTRAN Specification:

```
SUBROUTINE RCURVAL (EXCNO, EXCROUT, NOMSG, NOEXC,EXCCOUNT,
*ENABL)
INTEGER EXCNO, EXCROUT, NOMSG, NOEXC, EXCCOUT
LOGICAL ENABL
<standard library routine>
END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Parameter values:

EXCNO Exception number

EXCROUT Address of current user exception handler or zero

NOMSG Current number of messages allowed before termination

NOEXC Current number of exceptions allowed before termination

EXCCOUNT Current exception count

ENABL Logical parameter .TRUE. if exception is enabled at the moment

D.3.10 THE REXTERM ROUTINE

REXTERM is used to read the exception terminating condition.

FORTRAN Specification:

```
      SUBROUTINE EXCTERM (TRACEBACK, PRSTAT, NOLEV,FNUMB,
     *                    DTRACEBACK, OPRSTAT, DNOLEV, DFNUMB)
      INTEGER TRACEBACK, TRSTAT, NOLEV, FNUMB,DTRACEBACK, DPRSTAT,
     *DNOLEV, DFNUMB
     <standard library routine>
     END
```

The first parameters will read the current value of the variables represented by the parameters in the EXCTERM routine. The last four read the default values of the corresponding variables. See the EXCTERM routine for the parameter descriptions.

Scanned by Jonny Oddene for Sintran Data © 2011

D.3.11 FORTRAN EXCEPTIONS

Dec	*Oct*	*Name*	*msg*	*err*	
257	401	FATAL FORMATTING SYSTEM ERROR	1	0	
258	402	TOO LOW PARENTHESES LEVEL IN FORMAT	1	0	
259	403	ILLEGAL CHARACTER IN FORMAT	1	0	
260	404	ILLEGAL TERMINATION OF FORMAT	1	0	
261	405	OUTPUT RECORD SIZE EXCEEDED	10	unl	
262	406	FORMAT REQUIRES GREATER INPUT RECORD	10	unl	
263	407	INTEGER OVERFLOW ON INPUT	10	unl	
264	410	INPUT RECORD SIZE EXCEEDED	10	unl	
265	411	BACKSPACE ILLEGAL	10	unl	
266	412	BAD CHARACTER ON INPUT	10	unl	
267	413	REAL OVERFLOW ON INPUT	10	unl	
268	414	REAL UNDERFLOW ON INPUT	10	unl	
269	415	STRING DOES NOT START ON A WORD BOUNDARY	10	unl	
270	416	REAL OVERFLOW ON OUTPUT	10	unl	
271	417	FORMAT SPECIFICATION DOES NOT APPLY	1	0	
272	420	OVERFLOW IN EXPONENT ON INPUT	10	unl	
273	421	WRONG NUMBER OF PARAMETERS IN CALL	1	0	
274	422	TOO MANY FILES OPENED	1	0	ND-100 only
276	424	MIXING OF BINARY/ASCII ILLEGAL	1	0	
277	425	NO MORE BUFFERS AVAILABLE	1	0	
278	426	NON-FATAL ERROR (CHARACTER)	10	unl	ND-500 only
279	427	FATAL ERROR (I/O)	1	0	
280*	430	I/O ERROR WITHOUT SPECIAL HANDLING	0	0	
281	431	ZERO BASE AND NEGATIVE EXPONENT	10	unl	
282	432	BASE LESS THAN ZERO IN EXPONENTIATION	10	unl	
283	433	OVERFLOW IN EXPONENTIATION	10	unl	
284	434	NEG. ARG. IN SQUARE ROOT	10	unl	
285	435	TOO LARGE ARG. IN SINE	10	unl	
286	436	TOO LARGE ARG. IN COSINE	10	unl	
287	437	TOO LARGE ARG. IN EXP-FUNCTION	10	unl	
288	440	ZERO OR NEG. ARG. IN LOGARITHM	10	unl	
289	441	BOTH ARGS. ZERO IN ARC-TAN	10	unl	
294	446	TOO LARGE ARG. IN HYPERB. SINE	10	unl	
295	447	TOO LARGE ARG. IN HYPERB.COSINE	10	unl	
296	450	TOO LARGE ARG. IN COMPLEX ABS OR SQUARE ROOT	10	unl	
302	456	ILLEGAL ARG. IN ARC-SINE/COSINE	10	unl	
303	457	ILLEGAL ARG. IN TAN	10	unl	

Scanned by Jonny Oddene for Sintran Data © 2011

* = must be enabled

msg = default maximum number of error messages

err = default number of exceptions prior to abnormal termination

unl = unlimited numbe

Numbers not listed are currently not used. All FORTRAN errors except 430B are default enabled.

All languages:

The hardware traps are listed in Section D.3.1.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

APPENDIX E

STORAGE MAPPING

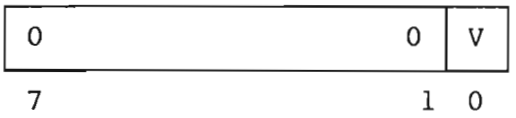
Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

STORAGE MAPPING

ND FORTRAN data types are stored in the following way:

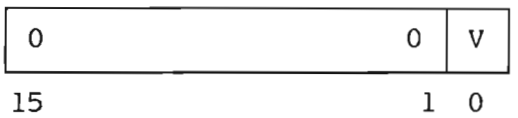
LOGICAL*1



Bits 7-1 : set to 0

Bit 0(V) : 0 = .FALSE.
1 = .TRUE.

LOGICAL*2



Bits 15-1 : set to 0

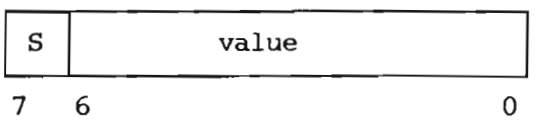
Bit 0(V) : 0 = .FALSE.
1 = .TRUE.

Scanned by Jonny Oddene for Sintran Data © 2011

LOGICAL*4

Bits 31-1 : set to 0
Bit 0(V) : 0 = .FALSE.
1 = .TRUE.

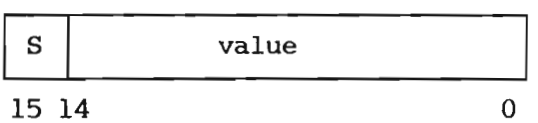
INTEGER*1



Bit 7 : 0 = greater than or equal to zero
1 = negative

Bits 6-0 : value held in twos-complement form.

INTEGER*2

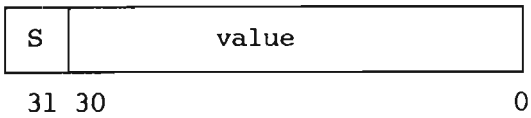


Bit 15 : 0 = greater than or equal to zero
1 = negative

Bits 14-0 : value held in twos-complement form.

Scanned by Jonny Oddene for Sintran Data © 2011

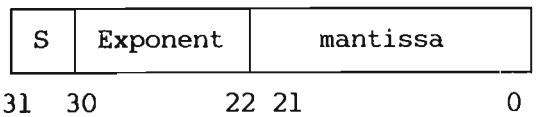
INTEGER*4



Bit 31 : 0 = greater than or equal to zero
1 = negative

Bits 30-0 : value held in twos-complement form.

REAL*4 (ND-500 or NORD-10/ND-100 with 32-bit floating-point hardware option.)



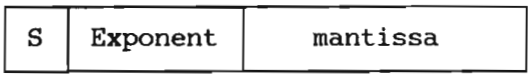
Bit 31 : 0 = greater than or equal to zero
1 = negative

Bits 30-22 : Binary exponent
Stored with a bias of 256 (400 octal). This is the power of 2 that the mantissa must be multiplied by. A value of 256 means that the mantissa is the value. If the exponent is 0, the whole value is zero.

Bits 21-0 : Mantissa
Stored without the 0.5 (0.1 binary) excess, unless the value is zero. The binary point is one place to the left of the mantissa. The mantissa is normalised so that $0.5 \leq \text{mantissa} < 1.0$

Scanned by Jonny Oddene for Sintran Data © 2011

REAL*6 (NORD-10/ND-100 with 48-bit floating-point hardware option)



Bit 47 : 0 = greater than or equal to zero
1 = negative

Bits 46-32 : Binary exponent. Stored with a bias of 40000 octal.
Zero exponent means that the whole value is zero.

Bits 31-0 : Mantissa.
Stored with all bits included. Binary point is immediately to the left of bit 31.

REAL*8 (all machines)

Bit 63 : 0 = greater than or equal to zero
1 = negative

Bits 62-54 : Binary exponent
Stored with a bias of 256 (= 400 octal). A zero exponent means the whole value is zero.

Bits 53-0 : Mantissa.
Stored without the 0.5 (= 0.1 binary) excess unless the value is zero. The mantissa is normalised so the $0.5 \leq$ mantissa < 1.0. The binary point is one place to the left of bit 53.

Scanned by Jonny Oddene for Sintran Data © 2011

COMPLEX*8

2 consecutive REAL*4 values.

COMPLEX*12

2 consecutive REAL*6 values.

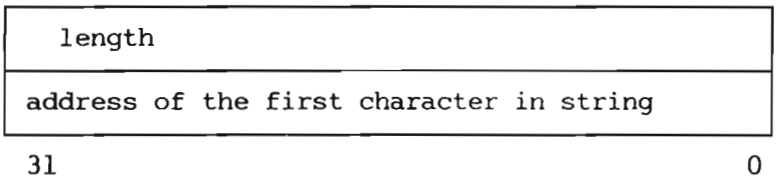
COMPLEX*16

2 consecutive REAL*8 values.

CHARACTER*N

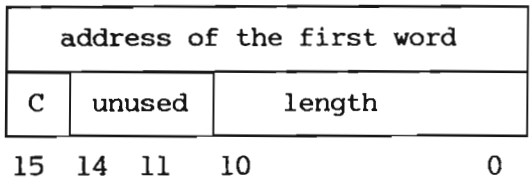
N consecutive bytes.
A character is addressed via a descriptor.

On the ND-500, the descriptor is 2 words:



Scanned by Jonny Oddene for Sintran Data © 2011

On the NORD-10/ND-100, the descriptor is 2 words:



If C (bit 15 of 2nd. word) = 0, the string starts in the high-order byte of the first word; if 1, it starts in the low-order byte.

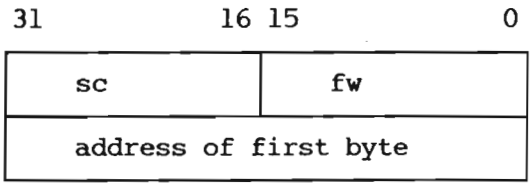
The following tables give the size in BYTES of each data type for the various machines.

NUMERIC (fw,sc)

Only ND-500:

(fw / 2) +1 consecutive bytes

A packed decimal operand is addressed via a descriptor. The descriptor takes two words:



Scanned by Jonny Oddene for Sintran Data © 2011

TABLE 1

NORD-10 OR ND-100 WITH 48-BIT FLOATING-POINT HARDWARE

Data Type		Length in Bytes	Alignment (Note 1)
CHARACTER*1	(Note 2)	1	Byte
LOGICAL	(Note 3)	2	Word
LOGICAL*2		2	Word
LOGICAL*4		4	Word
INTEGER	(Note 3)	2	Word
INTEGER*2		2	Word
INTEGER*4		4	Word
DOUBLE INTEGER		4	Word
REAL	(Note 3)	6	Word
REAL*4	(Note 4)	6	Word
REAL*6	(Note 4)	6	Word
REAL*8		8	Word
DOUBLE PRECISION	(Note 5)	8	Word
COMPLEX		12	Word
COMPLEX*8	(Note 4)	12	Word
COMPLEX*12	(Note 4)	12	Word
COMPLEX*16	(Note 5)	16	Word
DOUBLE COMPLEX	(Note 5)	16	Word

TABLE 2

NORD-10 OR ND-100 WITH 32-BIT FLOATING-POINT HARDWARE

The CHARACTER, LOGICAL, and INTEGER types are as for the 48-bit table above. The other data types are listed below.

Data Type		Length in Bytes	Alignment (Note 1)
REAL	(Note 3)	4	Word
REAL*4	(Note 4)	4	Word
REAL*6	(Note 4)	4	Word
REAL*8	(Note 5)	8	Word
DOUBLE PRECISION	(Note 5)	8	Word

Scanned by Jonny Oddene for Sintran Data © 2011

TABLE 3

ND-500

Data Type		Length in Bytes	Alignment (Note 1)
CHARACTER*1	(Note 2)	1	Byte
LOGICAL	(Note 3)	4	Word
LOGICAL*1		1	Byte
LOGICAL*2		2	Half-Word
LOGICAL*4		4	Word
INTEGER	(Note 3)	4	Word
INTEGER*1		1	Byte
INTEGER*2		2	Half-Word
INTEGER*4		4	Word
DOUBLE INTEGER		4	Word
REAL	(Note 3)	4	Word
REAL*4	(Note 4)	4	Word
REAL*6	(Note 4)	4	Word
REAL*8	(Note 5)	8	Word
DOUBLE PRECISION	(Note 5)	8	Word
COMPLEX		8	Word
COMPLEX*8	(Note 4)	8	Word
COMPLEX*12	(Note 4)	8	Word
COMPLEX*16	(Note 5)	16	Word
DOUBLE COMPLEX	(Note 5)	16	Word
NUMERIC (fw,sc)		(fw/2)+1	Byte

Scanned by Jonny Oddene for Sintran Data © 2011

NOTES

1. For the NORD-10 and ND-100, a word is 16 bits = 2 bytes. For the ND-500, a word is 32 bits = 4 bytes.

2. For CHARACTER strings of length N, the length is N bytes.

3. These are default values. The meaning of these attributes can be modified by the DEFAULT command.

4. REAL*6 and REAL*4 both mean single precision irrespective of the machine the program is executed on. The alternatives are provided for completeness and comments since the 48-bit format uses 6 bytes for a REAL value. Similar remarks apply to COMPLEX*8 and COMPLEX*12.

5. The DOUBLE PRECISION forms are identical on all machines. For the NORD-10 and ND-100, the implementation uses software routines and is relatively slow.

COMMON MAPPING

To allocate addresses within a common block, the following algorithm is used:

1. Place the first variable on a word boundary.

2. Place each subsequent variable on the first available alignment boundary at, or following the end of the previous variable.

It is the user's responsibility to ensure that the COMMON blocks are correctly defined. Particular care should be taken over the COMMON blocks shared between unlike processors (i.e., ND-100 and ND-500).

Scanned by Jonny Oddene for Sintran Data © 2011

RESTRICTION

ANSI FORTRAN 77 implies that an INTEGER and a REAL data item occupy the same amount of storage. For the ND-500 and the ND-100 with 32-bit floating-point hardware with DEFAULT INTEGER*4 specified, this condition is met.

However, for the ND-100 with 48-bit floating-point, hardware programs cannot be accepted as ANSI FORTRAN 77 standard, if equivalent storage occupation for INTEGER and REAL is required by either different COMMON definitions or by EQUIVALENCE statements.

CODE AND DATA SIZES

At the end of a compilation, the compiler indicates the total storage requirements of all the program units compiled by the last COMPILE command.

All the numbers given are in decimal, representing for the ND-100 the number of words, and for the ND-500, the number of bytes.

The values given are:

1) PROG SIZE is the size of the program code. On the ND-100 in single-bank operation (i.e., SEPARATE-DATA OFF), this figure also includes the data areas of the program, since they are not separated.

2) DATA SIZE is the size of the data areas used by the program, but excluding any COMMON blocks. This size is placed in the data bank in ND-500 programs and if SEPARATE-DATA ON is used in ND-100 programs. (The figure is omitted when using SEPARATE-DATA OFF in ND-100 programs.)

3) COMMON SIZE is the total of all the maximum sizes of the COMMON blocks found in the last compilation. It is placed in the data bank where applicable, or at the high address if SEPARATE-DATA OFF is used. (See the NRL and RT-Loader manuals for alternative methods of placing COMMON.)

Scanned by Jonny Oddene for Sintran Data © 2011

4) STACK SIZE is the sum of all the local storage requirements of all the units compiled in the last COMPILE command. These areas are obtained dynamically on entry to each program unit, and released on exit. If only one unit was compiled, it accurately reflects that unit's stack requirement (but not the routines it calls). The actual stack size should include enough for the longest nest of CALL's or function references, including library calls. The figure is omitted on the ND-100 with REENTRANT OFF, since there is no stack in this case.

Note that if LIBRARY-MODE is ON, these figures represent the total if all units are incorporated in a program. If only a selection is used, the figures are accordingly reduced.

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

A P P E N D I X F

INTERFACES TO OTHER LANGUAGE PROGRAMS

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

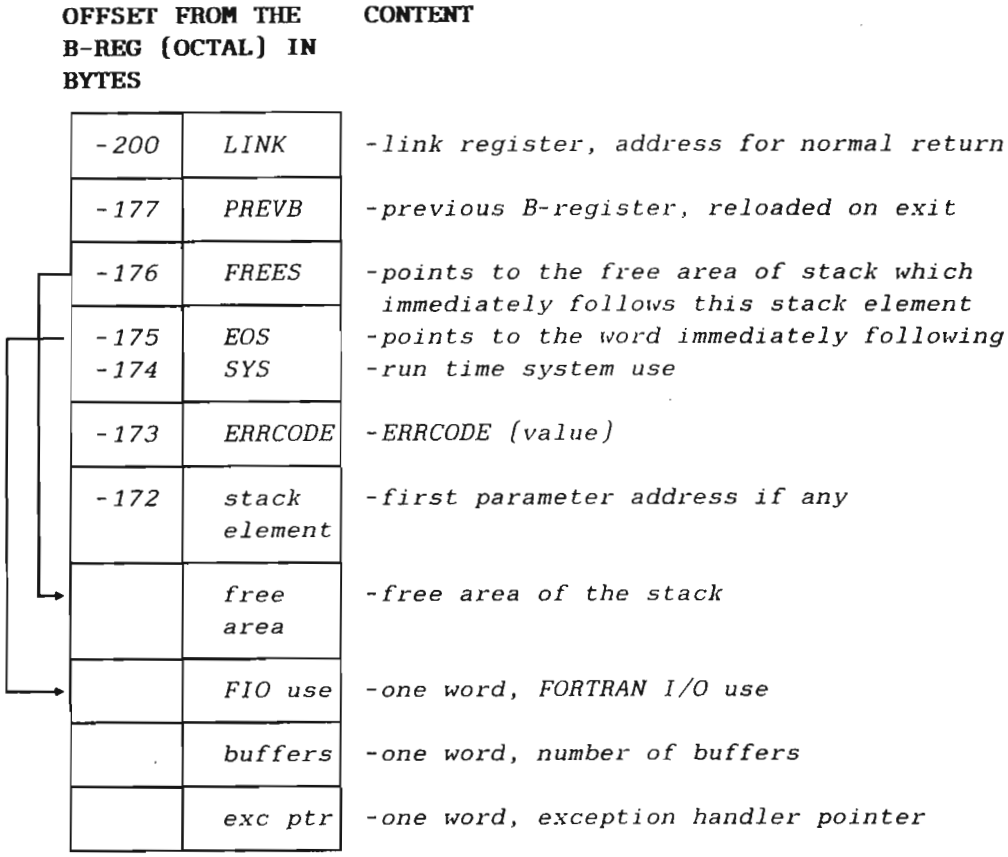
INTERFACES TO OTHER LANGUAGE PROGRAMS

ND FORTRAN has a standard calling sequence for its subroutine and function invocations. This will make it easier to interface programs and subprograms written in other languages with those written in FORTRAN. This interface is described in detail first, followed by examples showing how to use it to interface to other languages on both the ND-100 and ND-500.

F.1 FORTRAN INTERFACES ON THE ND-100

Each FORTRAN subprogram holds its local variables in a local data area. If a program, comprising a number of subprograms, is compiled as non-reentrant, then each local data area will be in a separate stack for each subprogram. If such a program is compiled as reentrant, then the local data area for each subprogram will be dynamically allocated from a single stack. The B-register must always address the appropriate stack element during execution of a FORTRAN subprogram.

Scanned by Jonny Oddene for Sintran Data © 2011



In FORTRAN, there are always three words following the stack. If the FORTRAN I/O system is to be used, and the program is non-reentrant, these should be initialized. The first word points to a special FORTRAN I/O area whose name is 5FIO-BL, and the second word should be zero. The third word, which is used by the EXCEPTION HANDLER, is called 5EXCINF. If the program is reentrant, these 3 words are initialized at run time, to zero, on entry to the FORTRAN main program.

The free area following the current stack element should always be large enough to contain the work areas for the FORTRAN run time routines (except the I/O routines in non-reentrant execution).

Scanned by Jonny Oddene for Sintran Data © 2011

When FORTRAN calls an EXTERNAL entry point, the registers are used as follows:

L = return address

B = current stack element; must be restored on return

T = number of parameters

A = parameter list address

D = address of descriptor for the return value if the call is to a function which returns a character string

X = unused

P = entry point of called routine

On return from a function, the value of the function is returned as follows:

LOGICAL*2, INTEGER*2	A-register
LOGICAL*4, INTEGER*4	AD-register
REAL*4 (32-bit floating-point hardware)	AD-register
REAL*6 (48-bit floating-point hardware)	TAD-register
REAL*8, COMPLEX, COMPLEX*16	A-register points to the result
CHARACTER	result resides in storage described by descriptor which D-reg pointed at on entry

For character functions, a memory area of the required size has been allocated by the calling routine before invoking of the function, and the D-register points to a descriptor upon entry to the function.

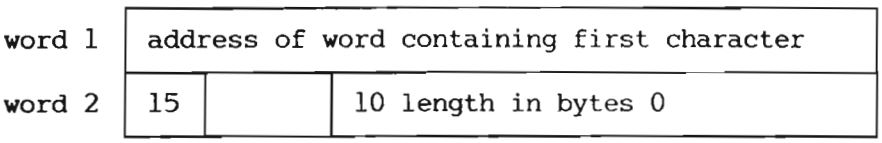
Reentrant FORTRAN routines assume that the parameter addresses are already in position (-172B from the B-register and onwards) at entry to the routine. It is the responsibility of the calling routine to place them there. From the calling viewpoint, they lie at +6 from the free area and onwards. Thus the addresses are not copied.

Scanned by Jonny Oddene for Sintran Data © 2011

If a subroutine has alternate returns specified in its dummy argument list, these are not included in the parameter list. Instead, the alternate return value (0 for normal return) is set in the ERRCODE position in the caller's stack element. This value may then be used in a COMPUTED GO TO after return has been made to the caller.

The parameter list consists of a sequence of words, one for each dummy argument. For arithmetic variables or expressions and logical variables or expressions, the corresponding word contains the address of the variable. For arrays of arithmetic or logical types, the word contains the address of the first element of the array.

For character variables or expressions, the word contains the address of a descriptor consisting of two words.



Bit 15 of word two is 0 if the string starts in the left-hand (high-order) byte of the word, and is 1 if it starts in the right-hand byte.

Bits 14 - 11 are used by the commercial instruction set and should normally be zero.

For character arrays, the parameter word contains the address of a descriptor for the first element of the array (i.e., one whose address part is for the start of the array, and whose length is that of a single element of the array).

For two-bank programs, all parameter values and their descriptors, if character, must be in the data bank.

The ASSEMBLY statement modifies the calling sequence to EXTERNAL program units. It can be used where the external routine is written in MAC, NPL; or PLANC with the SPECIAL option.

Scanned by Jonny Oddene for Sintran Data © 2011

The calling sequence is modified as follows:

- Only integer parameters or array names may occur in direct calls.

- The arguments are passed in registers. Integers values are contained in the register; array names are passed as the address of their first word. The arguments 1 to 4 are in T, A, D, and X-registers respectively. It is not possible to modify the FORTRAN arguments in the called routine, unless they are arrays.

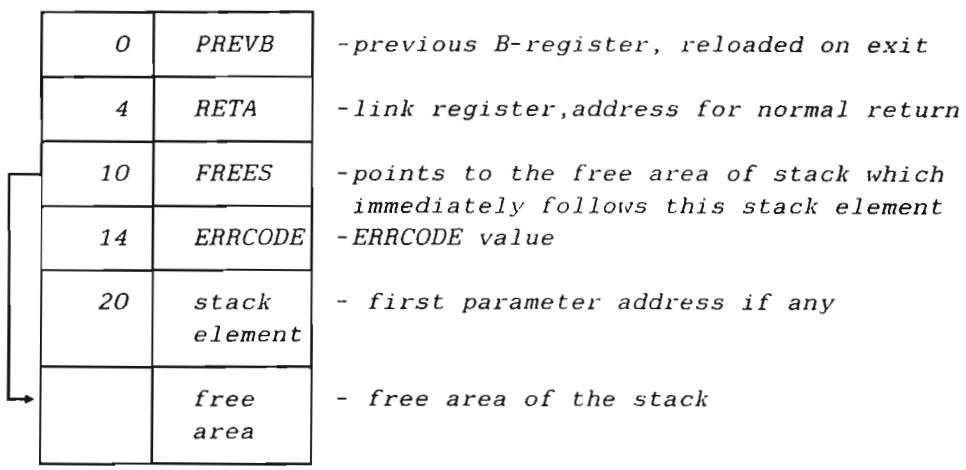
- The return address is one word beyond the contents of the L-register at entry to the routine.

Note that functions returning DOUBLE PRECISION and COMPLEX values do so in a manner incompatible with the 2090 series of FORTRAN compilers.

Scanned by Jonny Oddene for Sintran Data © 2011

F.2 FORTRAN INTEFACES ON THE ND-500

Each FORTRAN subprogram holds its local variables in a local data area. A program, comprising a number of subprograms, will result in each local data area being in a separate stack for each subprogram. The B-register must always address the appropriate stack element during execution of a FORTRAN subprogram.



Scanned by Jonny Oddene for Sintran Data © 2011

On return from a function, the value of the function is as follows:

LOGICAL*1, LOGICAL*2, INTEGER*1, INTEGER*4	I1-register INTEGER*2,LOGICAL*4,
REAL*4, (32-bit floating-point)	A1-register
REAL*8, (48-bit floating-point)	D1-register
COMPLEX*8	A1-register, A2-register
COMPLEX*16	D1-register, D2-register
CHARACTER, NUMERIC (fw,sc)	result resides in storage described by descriptor which R-reg pointed at on entry.

For character functions, a memory area of the required size has been allocated by the calling routine before invocation of the function, and the R-register points to a descriptor upon entry to the function.

If a subroutine has alternate returns specified in its dummy argument list, these are not included in the parameter list. Instead, the alternate return value (0 for normal return) is set in the ERRCODE position in the caller's stack element. This value may then be used in a COMPUTED GO TO after return has been made to the caller.

The parameter list consists of a sequence of words, one for each dummy argument. For arithmetic variables or expressions and logical variables or expressions, the corresponding word contains the address of the variable. For arrays of arithmetic or logical types, the word contains the address of the first element of the array.

Scanned by Jonny Oddene for Sintran Data © 2011

For character variables or expressions, the word contains the address of a descriptor consisting of two words:

word 1	*length in bytes*
word 2	*address of word containing first character*

For character arrays, the parameter word contains the address of a descriptor for the first element of the array (i.e., one whose address part is for the start of the array, and whose length is that of a single element of the array).

Scanned by Jonny Oddene for Sintran Data © 2011

F.3 INVOKING PLANC FROM FORTRAN

All PLANC routines called from FORTRAN should be 'STANDARD'. Any PLANC routine called from FORTRAN must contain an INISTACK invocation unless the FORTRAN program is compiled using the REENTRANT-MODE command on ND-100 or FIXED-DATA-AREA OFF on ND-500.

Example 1 - simple subroutine call

To call a subroutine with no complex arithmetic actual arguments, the following can be written in FORTRAN:

```
      EXTERNAL PLSUBR
      INTEGER I
      REAL R
C Call a subroutine written in PLANC
      CALL PLSUBR (I,R)
```

and the corresponding PLANC code is:

```
    MODULE msubr
    EXPORT plsubr
    INTEGER ARRAY : stack (1:1000)
    ROUTINE STANDARD VOID,VOID (INTEGER,REAL):plsubr (int,rl)
    INISTACK stack
% body of routine
    ENDROUTINE
    ENDMODULE
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example 2 - simple function call

To invoke a function which returns a non-complex arithmetic result.

- **In FORTRAN:**

```
      EXTERNAL PLFUNC
      REAL X,Y,PLFUNC
      DOUBLE PRECISION D
C Invoke a function written in PLANC
      Y=PLFUNC (X,D)
```

- **In PLANC:**

```
    ROUTINE STANDARD VOID,REAL (REAL,REAL8) : plfunc (rl,db)
    INISTACK stack
% PLANC REAL8 is the same as FORTRAN DOUBLE PRECISION
    ... RETURN
    ENDROUTINE
```

Example 3 - use of logical arguments

On the ND-100:

FORTRAN LOGICAL*2 corresponds to PLANC BOOLEAN. FORTRAN LOGICAL*4 is the following PLANC data type:

```
TYPE boolean4 = RECORD
                  BOOLEAN : unused % first word always zero
                  BOOLEAN : value  % contains actual value
                ENDRECORD
```

LOGICAL*4 cannot be returned from a PLANC STANDARD function.

Scanned by Jonny Oddene for Sintran Data © 2011

- **In FORTRAN:**

```
EXTERNAL PLBOOL
LOGICAL PLBOOL,V
LOGICAL*4 M4
V=PLBOOL (V,M4)
```

- **In PLANC:**

```
ROUTINE STANDARD VOID,BOOLEAN (BOOLEAN,boolean4) : plbool &
                                                   (m,m4)
INISTACK stack
IF m4.value THEN
  m RETURN
ENDIF
NOT m RETURN
ENDROUTINE
```

On the ND-500:

FORTRAN LOGICAL*4 corresponds to PLANC BOOLEAN. The FORTRAN LOGICAL*2 data type has no direct equivalent in PLANC. FORTRAN LOGICAL*2 can be handled in PLANC in the following way:

- **In FORTRAN:**

```
EXTERNAL PLBOOL
LOGICAL PLBOOL,V
LOGICAL*2 M2
V=PLBOOL (V,M2)
```

Scanned by Jonny Oddene for Sintran Data © 2011

- **In PLANC:**

```
    ROUTINE STANDARD VOID,BOOLEAN (BOOLEAN,INTEGER2) :&
                     plbool[m,m2]
    INISTACK stack
% the 2 integers must be contiguous in memory
    INTEGER2 : int1,int2
    BOOLEAN : bool1=int1
    m2=:int2
    0=:int1
    IF bool1 THEN
      m RETURN
    ENDIF
    NOT m RETURN
    ENDROUTINE
```

Example 4 - complex arguments and functions

FORTRAN COMPLEX has no direct corresponding data type in PLANC. It may be defined as follows:

```
    TYPE complex = RECORD
                      REAL : re    % real part
                      REAL : im    % imaginary part
                   ENDRECORD
```

Similarly the equivalent of FORTRAN DOUBLE COMPLEX is:

```
    TYPE complex = RECORD
                      REAL8 : dre
                      REAL8 : dim
                   ENDRECORD
```

These types, once defined, may be used just like other record data types.

Scanned by Jonny Oddene for Sintran Data © 2011

On the ND-100:

- **In FORTRAN:**

```
      COMPLEX C,CFUNC
      EXTERNAL CFUNC
      REAL R
C Invoke a PLANC function which returns a complex result
      C=CFUNC (R)
```

- **In PLANC:**

```
    ROUTINE STANDARD VOID,complex (REAL) : cfunc (r)
    INISTACK stack
    complex : c
      r=:c.im=:c.re     % store value in two identifiers
      c RETURN
    ENDROUTINE
```

On the ND-500:

- **In FORTRAN:**

```
      COMPLEX C,CFUNC
      EXTERNAL CFUNC
      REAL R
C Invoke a PLANC function which returns a complex result
      C=CFUNC (R)
```

- **In PLANC:**

```
    ROUTINE STANDARD VOID,VOID (REAL) : cfunc (r)
    INISTACK stack
    complex : c
      r=:c.im=:c.re     % store value in two identifiers
% set up values to be returned
     $* A1:=c.re; A2:=c.im
    RETURN
    ENDROUTINE
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example 5 - character string arguments

Since FORTRAN passes character strings through a descriptor, PLANC routines must accept these as records. It is often most convenient to recast the FORTRAN string descriptor as a PLANC bytes pointer. Thus:

On the ND-100:

```
TYPE ftnstring = RECORD    % a blank must precede the -1
                  BYTES : ftnchars (0: -1) % character data
                 ENDRECORD

TYPE ftndesc = RECORD PACKED
 ftnstring POINTER        :cstring  % address of string
 INTEGER RANGE (0:1B)     :coddbyte % left/right byte start
 INTEGER RANGE (0:17B)    :cunused  % unused
 INTEGER RANGE (0:3777B) :clength  % length of string
         ENDRECORD
```

Then in FORTRAN:

```
      CHARACTER H*20
      INTEGER I,J
      EXTERNAL HSUB
      CALL HSUB ( H(I:J) )
```

- which can be picked up in PLANC by:

```
    ROUTINE STANDARD VOID,VOID (ftndesc) : hsub (hij)
    INISTACK stack
    BYTES POINTER : bp
      ADDR (hij.cstring.ftnchars   &
           (hij.coddbyte: hij.clength-1+hij.coddbyte)) =:bp
% bp now contains the address of the FORTRAN character string
      ENDROUTINE
```

Scanned by Jonny Oddene for Sintran Data © 2011

On the ND-500:

```
TYPE ftnstring = RECORD
                   BYTES : ftnchars (0: -1)    &
                             % character data
                  ENDRECORD  % a blank must precede &
                             the -1

TYPE ftndesc = RECORD
                 INTEGER RANGE (0:777777777B)  : clength
                 ftnstring POINTER             : cstring
               ENDRECORD
```

- Then in FORTRAN:

```
      CHARACTER H*20
      INTEGER I,J
      EXTERNAL HSUB
      CALL HSUB (H(I:J))
```

- which can be picked up in PLANC by:

```
    ROUTINE STANDARD VOID,VOID (ftndesc) : hsub (hij)
    INISTACK stack
    BYTES POINTER : bp
      ADDR (hij.cstring.ftnchars (0:hij.clength-1))=:bp
% bp now contains the address of the FORTRAN character string
      ENDROUTINE
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example 6 - functions returning a character value

The definition of character data types must be made as in example 5. But in this case there can be no true return value for the function, so the PLANC code must simulate the return.

On the ND-100:

- **In FORTRAN:**

```
CHARACTER H*20,HFUNC*10
EXTERNAL HFUNC
H(1:10) = HFUNC (...)
```

- **In PLANC:**

```
ROUTINE STANDARD VOID,VOID : hfunc
INISTACK stack
BYTES POINTER : bp
ftndesc POINTER : dreg
 $* COPY SD DA; STA dreg    % return value descriptor
  ADDR (dreg.cstring.ftnchars       &
    (dreg.coddbyte : dreg.clength-1+dreg.coddbyte))=:bp
  '0123456789'=:IND (bp)    % set 'return value'
  ENDROUTINE
```

On the ND-500:

- **In FORTRAN:**

```
CHARACTER H*20,HFUNC*10
EXTERNAL HFUNC
H(1:10) = HFUNC (...)
```

Scanned by Jonny Oddene for Sintran Data © 2011

- **In PLANC:**

```
ROUTINE STANDARD VOID,VOID : hfunc
INISTACK stack
BYTES POINTER : bp
ftndesc POINTER : rreg
  $* R=:B.rreg              % return value descriptor
  ADDR (rreg.cstring.ftnchars (0 : rreg.clength-1))=:bp
  '0123456789'=:IND (bp)    % set 'return value'
  ENDROUTINE
```

F.4 INVOKING FORTRAN FROM PLANC

All FORTRAN subprograms invoked from PLANC must be IMPORT'ed as STANDARD routines. FORTRAN functions have out-values, but no FORTRAN routines have in-values.

Example 1 - a simple subroutine call

Call a FORTRAN subroutine with non-complex arithmetic dummy arguments.

- **In PLANC:**

```
IMPORT (ROUTINE STANDARD VOID,VOID (REAL,REAL8) : fsubr)
REAL : r
REAL8 : d
... fsubr (r,d)      % call the FORTRAN subroutine
```

- **In FORTRAN:**

```
  SUBROUTINE FSUBR (R,D)
  REAL R
  DOUBLE PRECISION D
  END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example 2 - a simple function

To invoke a function, returning a non-complex arithmetic result.

- **In PLANC:**

```
IMPORT (ROUTINE STANDARD VOID,VOID,INTEGER (INTEGER4) :ifunc)
INTEGER : k
INTEGER4 : kd
ifunc (kd)=:k     % invoke the FORTRAN function
```

- **In FORTRAN:**

```
      INTEGER FUNCTION IFUNC (KD)
      INTEGER*4 KD
      IFUNC=...
      RETURN
      END
```

Example 3 - use of logical arguments

PLANC BOOLEAN is the same as LOGICAL in FORTRAN, LOGICAL*2 on the ND-100 and LOGICAL*4 on the ND-500. LOGICAL*4 on the ND-100 or LOGICAL*2 on the ND-500 may be simulated as in example 3 in the previous section.

On the ND-100:

- **In PLANC:**

```
IMPORT (ROUTINE STANDARD VOID,BOOLEAN (boolean4):lfunc)
boolean4 : m4
  IF lfunc (m4) THEN...
```

Scanned by Jonny Oddene for Sintran Data © 2011

- **In FORTRAN:**

```
      LOGICAL FUNCTION LFUNC (M4)
      LOGICAL*4 M4
      LFUNC=...
      RETURN
      END
```

On the ND-500:

- **In PLANC:**

```
    IMPORT (ROUTINE STANDARD VOID,BOOLEAN (INTEGER2) :lfunc)
% the 2 integers must be contiguous in memory
     INTEGER2 : int1,int2
     BOOLEAN : bool1=int1
% put a value in the boolean data-element
     ...=:bool1
      IF lfunc (int2) THEN ...
```

- **In FORTRAN:**

```
      LOGICAL FUNCTION LFUNC (M2)
      LOGICAL*2 M2
      LFUNC=...
      RETURN
      END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example 4 - complex arguments and out-values

FORTRAN COMPLEX can be simulated in PLANC by the type declarations of example 4 in the previous section.

- **In PLANC:**

```
   IMPORT (ROUTINE VOID,complex (REAL) :cfunc)
   complex : c
   REAL : r
% on the ND-100 invoke the FORTRAN function normally
       cfunc(r)=:c
% on the ND-500 invoke the FORTRAN function normally,
% but assembler is required to get the returned function value
       cfunc (r)
      $* A1=:c.re; A2=:c.im
```

- **In FORTRAN:**

```
      COMPLEX FUNCTION CFUNC (R)
      REAL R
      CFUNC=CMPLX (R,R)
      RETURN
      END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example 5 - character string arguments

FORTRAN handles character strings by means of descriptors, which can be declared in PLANC as in example 5 in the previous section. These descriptors must be created in PLANC before invocation of the FORTRAN subprogram takes place.

- **In PLANC:**

```
    IMPORT (ROUTINE STANDARD VOID,VOID (ftndesc) : hsub)
    ftndesc : ed
    BYTES : arg (1:100)      % begins in left byte of word
    INTEGER : i,j
% now transfer arg (i:j) to FORTRAN
    ADDR(arg (i) ) FORCE ftnstring POINTER=:fd.cstring
% first byte - the following 2 lines are for the ND-100 only
  1-(i MOD 2) =:fd.coddbyte %left/right byte
  0=:fd.cunused
%
  j-i+1=:fd.clength        % length of string
  hsub (fd)                % invoke FORTRAN subprogram
```

- **In FORTRAN:**

```
      SUBROUTINE HSUB (FD)
      CHARACTER FD* (*)
C
      END
```

Scanned by Jonny Oddene for Sintran Data © 2011

Example 6 - character functions

Characters cannot be returned by FORTRAN to PLANC as out-values. The memory area for the returned string must be allocated before invoking the function and a special calling sequence is required.

- **In PLANC:**

```
    IMPORT (ROUTINE STANDARD VOID,VOID : hfunc)
    ftndesc : fd
    BYTES : val (0:19)              % value returned here
    ftndesc POINTER : fdp
%
    ADDR(val (0) ) FORCE ftnstring POINTER=:fd.cstring
% first byte - the following 2 lines are required for the
% ND-100 only
    0 =:fd.coddbyte
    0 =:fd.cunused
%
   MAXINDEX (val,1)-MININDEX (val,1)+1=:fd.clength
   ADDR (fd) =:fdp
% on the ND-100 use:
     $* LDA fdp; COPY SA DD % return descriptor address

% on the ND-500 use:
     $* R:=fdp                  % return descriptor address
%
  hfunc                         % put result in 'val'
```

- **In FORTRAN:**

```
CHARACTER * (*) FUNCTION HFUNC
HFUNC
RETURN
END
```

Scanned by Jonny Oddene for Sintran Data © 2011

F.5 CALLING COBOL FROM FORTRAN

On both the ND-100 and the ND-500, a FORTRAN program may call a subprogram written in COBOL. Parameters are transferred by reference between FORTRAN and COBOL. The data types which correspond in FORTRAN and COBOL are as follows:

FORTRAN	*COBOL*
*INTEGER*2, 16-bits*	*PIC S9(n) COMPUTATIONAL where 1 ≤ n ≤ 4*
*INTEGER*4, 32-bits*	*PIC S9(n) COMPUTATIONAL where 5 ≤ n ≤ 10*
REAL	*COMPUTATIONAL-2*
HOLLERITH strings	*PIC X(n) where n is the number of bytes*

COMPUTATIONAL-2 variables may only be used as a parameter in a subroutine call to or from COBOL, or to convert to/from COMPUTATIONAL-3 variables.

For example:

- **In FORTRAN:**

```
      INTEGER*2 INT2
      REAL RL
      INTEGER*4 INT4
      INT2=56
      RL=54.12345
      INT4=123456
C Call a COBOL subroutine
      CALL CBSUB (INT2,RL,INT4,'HOLL')
```

Scanned by Jonny Oddene for Sintran Data © 2011

- **In COBOL:**

```
PROGRAM-ID. CBSUB.
DATA DIVISION
WORKING-STORAGE SECTION.
01  CB-REAL                PIC S9(3)V9(6) COMP-3.
LINKAGE SECTION.
01 FTN-INT2                 PIC S9(4) COMP.
01 FTN-INT4                 PIC S9(6) COMP.
01 FTN-REAL                 COMP-2.
01 FTN-HOLLERITH            PIC X(4).
PROCEDURE DIVISION USING    FTN-INT2
                            FTN-REAL
                            FTN-INT4
                            FTN-HOLLERITH.
  PARA-1.
* CONVERT THE FORTRAN REAL VALUE TO THE INTERNAL
* COBOL FORM

  MOVE FTN-REAL             TO CB-REAL.
```

On ND-500 it is possible to transfere parameters of type CHARACTER and NUMERIC between FORTRAN and COBOL routines. The routine that calls the COBOL routine must be compiled with the command:

```
COBOL-INTERFACE <routine-name>
```

- **In FORTRAN:**

```
NUMERIC (5,3) N
CHARACTER*4  CH
CALL COBROU (CH,N)
```

- **In COBOL:**

```
PROGRAM-ID. COBROU.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  FTN-STRING     PIC X(4).
01  FTN-NUMERIC    PIC S9(3)V9(2) PACKED DECIMAL.
PROCEDURE DIVISION USING   FTN-STRING
                           FTN-NUMERIC.
```

Scanned by Jonny Oddene for Sintran Data © 2011

F.6 CALLING FORTRAN FROM COBOL

On both the ND-100 and the ND-500, a COBOL program may call a subprogram written in FORTRAN. Parameters are transferred by reference between FORTRAN and COBOL. The data types which correspond in FORTRAN and COBOL are as follows:

FORTRAN	*COBOL*
*INTEGER*2, 16-bits*	*PIC S9(n) COMPUTATIONAL where 1 ≤ n ≤ 4*
*INTEGER*4, 32-bits*	*PIC S9(n) COMPUTATIONAL where 5 ≤ n ≤ 10*
REAL	*COMPUTATIONAL-2*
HOLLERITH strings	*PIC X(n) where n is the number of bytes*

COMPUTATIONAL-2 variables may only be used as a parameter in a subroutine call to or from COBOL, or to convert to/from COMPUTATIONAL-3 variables.

Parameters from COBOL must start on a word boundary, ND-100 only.

Scanned by Jonny Oddene for Sintran Data © 2011

For example:

- **In COBOL:**

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 FTN-INT2          PIC S9(4) COMP          VALUE 123.
01 FTN-INT4          PIC S9(6) COMP          VALUE 123456.
01 CB-REAL           PIC S9(3)V9(6) COMP-3 VALUE -2. 71.
01 FTN-REAL          COMP-2.
01 FTN-HOLLERITH     PIC X(10)               VALUE 'A123456789'.
01 FTN-HLENGTH-WDS PIC S9(4) COMP            VALUE 5.

* NUMBER OF CHARACTERS PER WORD IS DIFFERENT ON THE
* ND-500


PROCEDURE DIVISION.
  PARA-1.
* CONVERT THE INTERNAL COBOL FROM THE FORTRAN REAL FORM
  MOVE CB-REAL                          TO FTN-REAL.
* CALL A FORTRAN SUBROUTINE
  CALL "FTNSUB" USING                       FTN-INT2
                                            FTN-REAL
                                            FTN-INT4
                                            FTN-HOLLERITH
                                            FTN-HLENGTH-WDS.
```

- **In FORTRAN:**

```
      SUBROUTINE FTNSUB (INT2,RL,INT4,HSTRING,HLENGTH)
      INTEGER*2 INT2,HLENGTH,HSTRING (HLENGTH)
      INTEGER*4 INT4
C May now access values passed from COBOL and return values to
C COBOL in the normal manner
       RETURN
       END
```

Scanned by Jonny Oddene for Sintran Data © 2011

On ND-500 it is also possible to transfer parameters of type CHARACTER and NUMERIC between FORTRAN and COBOL routines. The FORTRAN routine that is called from COBOL must be compiled with the command:

```
COBOL-INTERFACE <routine-name>
```

- **In COBOL:**

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01   FTN-STRING  PIC X(4) VALUE 'TEST'.
01   FTN-NUMERIC PIC S9(3)V9(2) PACKED-DECIMAL VALUE 345.67.
PROCEDURE DIVISION.
CALL "FTNSUB" USING FTN-STRING, FTN-NUMERIC.
```

- **In FORTRAN:**

```
SUBROUTINE FTNSUB (CH,N)
CHARACTER*4  CH
NUMERIC (5,2) N
END
```

Scanned by Jonny Oddene for Sintran Data © 2011

F.7 MAC SUBROUTINES (ND-100 ONLY)

When writing subroutines or functions to be called from FORTRAN, the user should clearly understand the format of the run-time stack, and the use of registers in the calling sequence, see Section F.1.

There is a marked difference between reentrant and non-reentrant routines with regard to the available methods for acquiring local workspace.

F.7.1 NON-REENTRANT ROUTINES

In this case, there is no space available in the FORTRAN routine's local area (addressed by the B-register) which can be used by a called subroutine. It is the called subroutine's responsibility to acquire the local areas it needs on its own behalf.

An example of how to address parameters from a MAC routine is as follows:

```
)9BEG
)9ENT SUBR
SUBR,                          % if called as CALL SUBR (I,R)
              SWAP   SA  DB
              STA    SAVB      % save FORTRAN's B-reg.
              LDA    I 0,B     % first parameter (I)
              ....   .....
              LDF    I 1,B     % second parameter (R) 1
              ....   .....
              LDA    SAVB
              COPY   SA  DB    % restore FORTRAN's B-reg.
              EXIT
SAVB,0
)9END
```

Scanned by Jonny Oddene for Sintran Data © 2011

In order to mix MAC routines with FORTRAN, it is recommended that the following calling sequence be used (see Sections F.7.4 and F.7.5.):

```
)9BEG
)9ENT          SUBR
SUBR,          COPY   SL  DX       % return address
               JPL    I   (5INIT   % create a stack unit
               FRAME              % size (in words) of the local
                                  % frame
               STACK              % address of stack space
               STSZ               % total size of stack
               0                  % or 1 if two-bank
                                  % operation
               0                  % for debug use
% routine starts here
.............. .....
)9END          JPL    I   (5LEAV   % return to caller
```

This will also aid the Symbolic Debugger to identify the FORTRAN routines and trace the stack frames correctly.

Scanned by Jonny Oddene for Sintran Data © 2011

F.7.2 REENTRANT ROUTINES

Routines which can be shared among several programs can be called only from reentrant FORTRAN routines (see REENTRANT command, Section 12.8). They can use the standard FORTRAN stack, which has been initialised by a FORTRAN program. The MAC subroutines must not alter the length of the stack, nor interfere with the two words which follow it. The acquisition of the local stack area and return, can be done as follows:

```
)9BEG
)9ENT SUBR
SUBR,          COPY  SL  DX      % save routine return
               JPL    I  (5ENTR  % acquire next frame
               SIZE              % size of frame (in   words)
               COPY  SA  DX      % B-reg addresses stack
               ..... .....
                                 % frame
               JPL   I   (5LEAV  % return up stack
)FILL
)9END
```

F.7.3 ALTERNATE RETURNS

An alternate return is handled by setting the appropriate value (from 1 upwards) corresponding to the number of the asterisks in the parameter list of the CALL statement, into the return code slot of the calling routine. Zero must be set if there is no alternate return taken, but one was expected by the CALL.

The address of the caller's stack frame can be obtained by:

```
LDX   SAVB          % non-reentrant case
or
LDX   -177,B        % reentrant case
```

and then the return value is set by:

```
LDA   RETNV         % value for return
STA   -173,X        % store in caller's error code
```

Scanned by Jonny Oddene for Sintran Data © 2011

F.7.4 CALLING FORTRAN SUBROUTINES

For non-reentrant FORTRAN routines, the parameter list must be built at its correct place in the called routine's stack frame. For example:

```
            LDA   (PADR       % parameter list address
            JPL I (FSUB       % call FORTRAN subroutine
            ..... .....
PADR,    I                    % first parameter address
         R                    % second parameter address
            ..... .....
I,54                          % first parameter value
R, (3.141593                  % second parameter value


            LDX   - 176,B     % free stack space
            LDA   (I           % first parameter address
            STA   6,X          % first parameter position
            LDA   (R
            STA   7,X          % second parameter position
            ..... .....
            JPL  I(FSUB
```

F.7.5 INVOKING FORTRAN INTRINSIC FUNCTIONS

All FORTRAN library routines must be treated like reentrant FORTRAN subroutines, and space provided on the local stack for the library workspace. To set up a local stack (and/or stack frame) see Sections F.7.1 and F.7.2. The amount of space required by the library for its stack frame can generally be determined by the data type of the returned value. These values are subject to small changes without notice, therefore a certain margin should be allowed.

DATA TYPE	STACK SIZE
INTEGER*2	10
INTEGER*4	40
REAL	65
DOUBLE PRECISION	160

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

APPENDIX G

HOLLERITH

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

HOLLERITH

G.1 HOLLERITH CONSTANTS

The ANSI FORTRAN 77 Standard does not accept Hollerith constants. It merely gives recommendations as to their use in an appendix to retain some compatibility with previously existing programs.

ND FORTRAN implements these recommendations, with a few minor additions and changes, to retain compatibility with programs which run on the NORD-10 FORTRAN compiler. Details are as follows:

- **CONSTANTS**

Hollerith constants may have one of two forms. The first is:

$$nHh_1 h_2 h_3 \ldots . hn$$

where

n is a non-negative number.

h_1 ... hn are the **n** characters in the source program which immediately follow the **H.**

Blanks are significant among the **hi**, but the **hi** cannot contain a carriage return, line feed, or tab characters. The internal representation of a Hollerith constant is the sequence of **hi**, as ASCII characters with the parity bit set to zero.

Scanned by Jonny Oddene for Sintran Data © 2011

The second form is:

```
"h.....h"
```

The double-quote characters ", octal 42, is the delimiter for the Hollerith string. The **h** characters inside the double-quotes may be replaced by any character except carriage return, line feed or tab characters. If the string is to contain a double-quote character, two of these should be written.

They can be used as constants only in DATA statements, as actual arguments in subroutine or function invocations, as the right-hand side of an arithmetic assignment statement, or as the value given to a symbolic constant.

They may not appear in any other context; in particular, as values for output, or in expressions.

- **IN ASSIGNMENT STATEMENTS**

A Hollerith constant as the right-hand side of an assignment statement may be moved to an arithmetic or logical variable or array element name without any form of conversion. The ASCII characters are assigned byte-by-byte to the storage of the left-hand side, starting at the leftmost byte. Padding with blanks or truncation occurs on the right to the length of the storage for the target. See Appendix E for the sizes of the variables. Character variables cannot receive Hollerith constants.

- **IN DATA STATEMENTS**

An arithmetic or logical variable may take a Hollerith constant from the constant list as its initial value in a DATA statement. The assignment is as for the assignment statement. The correspondence of data list and constant list is preserved. Character variables cannot receive Hollerith values.

Scanned by Jonny Oddene for Sintran Data © 2011

- **AS ACTUAL ARGUMENTS**

Hollerith constants used as actual arguments must match their corresponding dummy arguments for storage length. No padding will occur; but if the actual argument is longer, only the first characters are used in the dummy argument. The dummy argument cannot be of type CHARACTER.

- **AS A SYMBOLIC CONSTANT VALUE**

The Hollerith constant is assigned to the symbolic constant as if it were being assigned to a variable of the type of the symbolic constant on the target machine of the compilation. The resulting arithmetic value is then the value of the symbolic constant. The allowable data types are only INTEGER*2 and INTEGER*4.

- **IN A RELATIONAL COMPARISON**

Variables can be compared with Hollerith data in an IF statement, or general logical expression. The Hollerith data is treated as though it were assigned to a variable of the same data type as the other operand of the comparison, and the comparison is performed as for that other data type. Character strings cannot be compared with Hollerith data.

- **A-FORMAT FOR HOLLERITH DATA**

If the format **Aw** is used when the corresponding I/O list item is arithmetic or logical, then the data transfer is done without conversion, except for parity bits being cleared on input (unless the parity option has been coded on the OPEN statement for the file, see Section 9.3.1).

On input, the **w** input characters are treated like a Hollerith constant and assigned as in the assignment statement. On output, w output characters are written from the storage of the arithmetic or logical item.

Scanned by Jonny Oddene for Sintran Data © 2011

- **RESTRICTIONS**

If a logical variable has been assigned a Hollerith value, then its use as a logical value will be unpredictable.

Real, double precision and complex variables containing Hollerith values may be moved, but any form of arithmetic operation may give unpredictable results due to hidden optimisations or conversions.

It is recommended that Hollerith constants be avoided wherever possible. Character variables may be equivalenced as an alternative. If Holleriths constants must be used, it is recommended that the exact length be specified to prevent the implied padding and truncation. This should ease the transport and maintenance of these non-standard features.

Scanned by Jonny Oddene for Sintran Data © 2011

INDEX LIST

Index term	Reference
A format descriptor	227
actual	
arguments	262
declarator	40
adjustable	
arrays	39
dimensions	46
alternate returns	436
arithmetic	
array expression	109
constant	29
expression	85
operand	85
operator	85
Arithmetic IF statement	137
array	
adjustable	39
assumed-size	40
declarator	40
definition	37
element name	38
expressions	109
size of	37
storage, order of elements in	39
subscript	38
ASCII Character Set	293
ASSEMBLY statement	74
Assigned GO TO statement	135
assignment statements	
arithmetic	123
character	127
conversion in	124
logical	125
statement label	126
asterisk	
as a dummy argument	243

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
BACKSPACE statement	195
Blank COMMON	
difference between named COMMON and	54
storage sequence of	53
Block COMMON	
definition	52
storage sequence of	53
BLOCK DATA statement	279
BLOCK DATA subprogram restrictions	279
Block IF statement	139
BN and BZ format descriptors	223
character	
alphanumeric	4
constant	34
data type	27
expression	93
operands	93
operator	93
special	4
substrings	41
Type statement	61
CHARACTER Alignment	287
CHARACTER and Hollerith	286
character array	
expression	113
operands	113
operator	113
character set,FORTRAN	4
CLOSE statement	194
COBOL programs	
invoking from FORTRAN	429
which invoke FORTRAN	431
code and data sizes	402
collating sequence	5
columns	7
comment line	7
COMMON block storage sequence	53
common mapping	401
COMMON statement	52

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
compiler	
sample program	13
COMPLEX	
constant	33
data type	27
expression	85
Type statement	58
Computed GO TO statement	133
constant expression	
arithmetic	89
character	95
logical	103
constants	
arithmetic	29
character	34
complex	33
double precision	32
integer	29
logical	34
real	31
constants, Hollerith	441
CONTINUE statement	151
control statements	131
DATA statement	77
data types	27
Device handling	340
digit,definition	4
dimension	
bounds	45
declarator	38
DIMENSION statement	45
DO	
FOR-ENDDO statements	148
loop	144
loop, range of	144
statement	144
statement, execution	146
WHILE-ENDDO statements	149

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
DOUBLE PRECISION	
constant	32
data type	27
expression	85
Type statement	58
dummy	
argument	234
array declarator	40
procedure	241
E and D format descriptors	220
editing, use of format descriptors for	214
ELSE statement	140
ELSEIF statement	139
END statement	154
ENDFILE statement	196
ENDIF statement	141
End-of-File Specifier	164
ENTRY statement	266
EQUIVALENCE statement	48
error handling	351
Error Messages	301
Error specifier	165
EXCDEF routine	380
EXCEPT routine	373
exception handling	369
exceptions	
FORTRAN	388
EXCTERM routine	382
executable statement	9
exponent	
double precision	32
real	32
expression	
arithmetic	85
arithmetic array	109
array	109
character	93
character array	113
constant	103
definition of	85

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
expression	
evaluation	103
logical	99
relational	96
subscript	38
substring	41
external	
functions	261
procedure	10
statement	70
F format descriptor	217
file	
definition	158
File Accessing	287
file operations	347
Format	
descriptors	210
specifications	209
specifier and identifier	163
formatted	
data transfer	170
records, printing of	178
FORTRAN	
character set	4
statement	9
FORTRAN exceptions	388
FORTRAN interfaces on the ND-100	407
FORTRAN interfaces on the ND-500	412
FORTRAN intrinsic functions	437
FORTRAN subroutines	
invoking from MAC	437
function	
RAN	367
functions	
definition	233
functions, library utility	365
G format descriptor	222

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
GETMESS and PGETMESS routines	385
global item	6
GO TO statement	
Assigned	135
Computed	133
Unconditional	132
H format descriptor	224
Hollerith constants	441
I and J format descriptors	215
IF statement	
Arithmetic	137
Logical	138
IMPLICIT	
statement	66
implied	
DO lists	169
type rules for identifiers	28
INPUT statement	181
input, list-directed	171
Input-Output	
Buffer Allocation	288
file access	160
file format	159
list-directed	170
lists	168
statements	157
status specifier	166
terms and concepts	157
INQUIRE statement	198
INTEGER	
constant	29
data type	27
expression	85
Type statement	58

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
interfaces	
on the ND-100	407
on the ND-500	412
to other language programs	407
with COBOL programs	429, 431
with MAC	437
with MAC subroutines	434
with non-reentrant MAC routines	434
with PLANC programs	415, 423
with reentrant MAC routines	436
INTRINSIC	
statement	71
INTRINSIC functions	244, 437
invoking COBOL from FORTRAN	429
invoking FORTRAN from COBOL	431
invoking FORTRAN from PLANC	423
invoking FORTRAN intrinsic functions	437
invoking FORTRAN subroutines from MAC	437
invoking MAC subroutines from FORTRAN	434
invoking non-reentrant MAC routines	434
invoking PLANC from FORTRAN	415
invoking reentrant MAC routines	436
keyword	6
L format descriptor	226
languages	
interfaces to programs written in other	407
letter,definition	4
library subprogram descriptions	367
library utility functions	365
table	365
line	
comment	7
continuation	7
initial	7
Loader error messages	316
local item	6

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
LOGICAL	
array expressions	117
constant	34
data type	27
expression	99
operand	99
operator	99
Type statement	58
Logical IF statement	138
loop	
control variable	284
definition	283
MAC subroutines	
invoking from FORTRAN	434
main program	275
mapping	
common	401
storage	393
Monitor calls	
commonly used	329
introduction	325
ND-100 and ND-500 monitor calls	352
ND-100 exception handling	369
ND-100 simulated traps	373
ND-500 traps and exception handling	369
ND-500 traps table	372
nonexecutable statement	9
non-reentrant MAC routines	
invoking from FORTRAN	434
numeric editing	214
O format descriptor	228
octal values	29
OPEN statement	182

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
operands	
arithmetic	86
character	93
character array	113
logical	99
operators	
arithmetic	86
character	93
character array	113
logical	99
precedence order of	99
relational	96
other languages	
interfaces to programs written in	407
OUTPUT statement	181
output, list-directed	173
P format descriptor	218
PARAMETER	
statement	68
parentheses	103
PAUSE statement	153
PGETMESS and GETMESS routines	385
PLANC programs	
invoking from FORTRAN	415
which invoke FORTRAN	423
precedence of arithmetic operators	86
PRIMESS routine	384
PRINT statement	180
PRITRAC routine	384
procedure	
definition	10
external	10
main	10
program	
unit	10
program administrator	330
PROGRAM statement	275
Programming Techniques	283

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
programs, COBOL	
invoking from FORTRAN	429
which invoke FORTRAN	431
programs, PLANC	
invoking from FORTRAN	415
which invoke FORTRAN	423
RAN function	367
RCURVAL routine	386
RDEFVAL routine	386
READ statement	174
REAL	
constant	31
data type	27
expression	85
Type statement	58
record	
definition	157
record specifier	167
reentrant MAC routines	
invoking from FORTRAN	436
relational	
array expressions	115
expression	96
operand	96
operator	96
restriction on accepting programs as ANSI FORTRAN 77 standard	402
results for arithmetic array expressions	111
retrieving, changing	
device information	337
file system information	345
RETURN statement	270
returns	
alternate	436
REWIND statement	197
REXTERM routine	387
routine	
EXCDEF	380
EXCEPT	373
EXCTERM	382

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
routine	
GETMESS and PGETMESS	385
PRIMESS	384
PRITRAC	384
RCURVAL	386
RDEFVAL	386
REXTERM	387
routines, non-reentrant MAC	
invoking from FORTRAN	434
routines, reentrant MAC	
invoking from FORTRAN	436
Runtime error diagnostics	316
S, SP and SS format descriptors	223
SAVE statement	72
simulated traps, ND-100	373
size in bytes of data types	
ND-500	400
NORD-10 or 100, 32-bit floating point	399
NORD-10 or 100, 48-bit floating point	399
sizes, code and data	402
slash format descriptor	226
special characters	4
special names in INTRINSIC functions	244
statement	
executable	9
FORTRAN	9
functions	258
label	9
nonexecutable	9
statement label	9
STOP statement	152
storage mapping	393
subprogram descriptions, library	367
subroutine	
definition	264
referencing to a	265
subprogram restrictions	266

Scanned by Jonny Oddene for Sintran Data © 2011

Index term	Reference
subroutines, FORTRAN	
invoking from MAC	437
subroutines, MAC	
invoking from FORTRAN	434
subscript	
array	38
expression	38
substring	
character	41
expression	41
symbolic name	6
syntactic item	6
T, TL, TR and rX format descriptors	225
tab positions	8
text format descriptor	224
traps	
ND-100 simulated	373
ND-500 and exception handling	369
ND-500, table	372
Type statements	58
Unconditional GO TO statement	132
unformatted	
data transfer	170
Units	162
utility functions, library	365
variable	
as dummy argument	238
variables	
definition	36
WRITE statement	176

Scanned by Jonny Oddene for Sintran Data © 2011

Index term Reference

Z format descriptor 229

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

SEND US YOUR COMMENTS!

Are you frustrated because of unclear information in our manuals? Do you have trouble finding things?

Please let us know if you:
- find errors
- cannot understand information
- cannot find information
- find needless information.

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

Send to:
Norsk Data A.S
Documentation Department
P.O. Box 25 BOGERUD
N - 0621 OSLO 6 - Norway

NOTE!

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

Manual Name: _____ Manual number: _____

Which version of the product are you using? _____

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

_____

_____ _____

_____

_____

_____

Do you have suggestions for improving this manual? _____

_____

_____

_____

_____

_____

_____

Your name: _____ Date: _____

Company: _____ Position: _____

Address: _____

_____

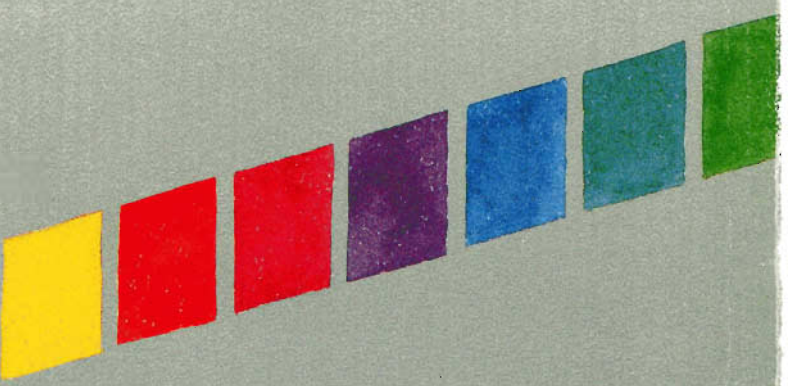
What are you using this manual for? _____

_____

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011

Scanned by Jonny Oddene for Sintran Data © 2011



Scanned by Jonny Oddene for Sintran Data © 2011