

ND—PASCAL User's Guide

ND—60.124.05

NOTICE

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

The information described in this document is protected by copyright. It may not be photocopied, reproduced or translated without the prior consent of Norsk Data A.S.

Copyright © 1984 by Norsk Data A.S

ND PASCAL User's Guide
Publ.No. ND-60.124.05
January 1984



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

New versions and revisions are announced in the ND Bulletin and can be ordered as described below.

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

These forms, together with all types of inquiry and requests for documentation should be sent to the local ND office or (in Norway) to:

Documentation Department
Norsk Data A.S
P.O. Box 4, Lindeberg gård
Oslo 10

Preface:

The Product

This manual describes version J of the Pascal compilers for the ND-100 and the ND-500. The ND-100 Pascal compiler is delivered in two versions, one for 32-bit and one for 48-bit floating point hardware. As the three compilers differ only in machine-dependent respects, they are described in the same manual.

The Reader

The reader is assumed to know the Pascal language, as this manual mainly describes only the extensions and differences between ND-Pascal and Standard Pascal as described in Jensen and Wirth: Pascal User manual and Report.

The reader is also expected to have sufficient experience with the SINTRAN operating system to be able to enter a program through an editor, and to load and execute the compiled program.

The Manual

The manual is organized as a reference manual, with the information ordered according to function. For the most part, only differences between Standard Pascal and ND-Pascal are described. For a complete example of a Pascal program, refer to chapter 9. Compiler error messages and run time error messages are listed in Appendices A and B.

The manual uses the term ND-Pascal to mean either of the compilers. Those parts of the manual which are relevant to only one of the computers, are marked as such in the chapter or section heading. Also, a part of the text may be marked with the comment N100 or N500 to signify that the text is relevant to the ND-100 or the ND-500 only.

The first of these is the fact that the system is not a simple one. It is a complex one, and it is one that is not easily understood. It is a system that is not easily understood, and it is one that is not easily understood.

The second of these is the fact that the system is not a simple one. It is a complex one, and it is one that is not easily understood. It is a system that is not easily understood, and it is one that is not easily understood.

The third of these is the fact that the system is not a simple one. It is a complex one, and it is one that is not easily understood. It is a system that is not easily understood, and it is one that is not easily understood.

The fourth of these is the fact that the system is not a simple one. It is a complex one, and it is one that is not easily understood. It is a system that is not easily understood, and it is one that is not easily understood.

TABLE OF CONTENTS

<u>Section</u>	<u>Page</u>
1 INTRODUCTION	1
1.1 The Pascal Compiler	1
1.2 The Main Implementation Dependent Characteristics	2
1.3 The Main Extensions	2
2 THE SOURCE PROGRAM	4
2.1 Special Symbols	4
2.2 Identifiers	5
2.3 Keywords	5
2.4 Standard Identifiers	5
2.5 Compiler Commands	6
2.5.1 Conditional compilation	6
2.5.2 Multiple source files	7
2.5.3 Intermixing definition sections	8
2.5.4 Options	8
2.5.5 Program listing	10
2.6 Implementation Dependent Features	11
2.6.1 Standard types	11
2.6.2 Structured types	12
2.6.3 Packed structures	12
2.6.4 Strings and character arrays	13
2.6.5 Procedure parameters	13
5.1 Conformant arrays	13
5.2 Formal procedures	15
2.6.6 ND-500 traps	15
2.7 Extensions in ND-Pascal	16
2.7.1 Variable initialization	16
2.7.2 External Pascal routines	16
2.7.3 External routines in other languages	17
2.7.4 Standard procedures and functions	18
2.7.5 External procedures and functions	18
2.7.6 Generic functions	23

<u>Section</u>	<u>Page</u>
2.7.7 Miscellaneous extensions	23
3 PROGRAM COMPILATION	25
3.1 HELP	25
3.2 COMPILE	25
3.3 RUN	27
3.4 CLEAR	28
3.5 OPTIONS	28
3.6 SET and RESET	28
3.7 EXIT	28
3.8 LINESPP	28
3.9 VALUE	28
3.10 SINTRAN Commands	29
3.11 Program Compilation Example	29
4 PROGRAM LOADING AND EXECUTION	31
4.1 Program Loading	31
4.1.1 ND-100 program loading	31
4.1.2 ND-500 program loading	32
4.2 Run-Time Errors	33
4.2.1 Trapping run-time errors	34
5 INPUT/OUTPUT	35
5.1 File Variables	35
5.1.1 The type TEXT	35
5.1.2 Standard files	36
5.1.3 Packed files	36
5.1.4 Non-TEXT files	37

Section	Page
5.2 Association to External Files	37
5.2.1 CONNECT	38
5.2.2 DISCONNECT	38
5.2.3 Scratch files	39
5.2.4 Program heading parameters	39
5.3 Terminal I/O	40
5.4 Random Access I/O	41
6 ND-100 REAL-TIME PROGRAMS	43
7 ND-100 OVERLAY PROGRAMS	45
7.1 Modules	45
7.2 Compilation of Modules	46
7.3 Loading Overlay Programs	49
7.4 Executing Overlay Programs	51
8 IMPLEMENTATION DESCRIPTION	53
8.1 ND-100 Implementation	53
8.1.1 Memory layout	53
1.1 One-bank programs	54
1.2 Two-bank programs	55
1.3 Forced allocation of stack and heap	55
8.1.2 Loader symbols	56
8.1.3 Procedure and function calls	56
8.1.4 Interface to FORTRAN and PLANC	58
8.1.5 Input/Output	58
8.2 ND-500 Implementation	60
8.2.1 Memory layout	60
1.1 Forced allocation of stack and heap	61
1.2 The size of the heap	61
8.2.2 Loader symbols	61
8.2.3 Procedure and function calls	62
8.2.4 Input/Output	63
9 SAMPLE Pascal PROGRAM	65

<u>Section</u>	<u>Page</u>
9.1 ND-100 Sample Program	65
9.2 ND-500 Sample Program	67
APPENDIX A Compile- Time Error Messages	69
APPENDIX B Run- Time Error Messages	73
Index	76

1 INTRODUCTION

The Pascal language was designed in 1971 by Niklaus Wirth. The language design had two principal aims. The first was to make available a language suitable to teach programming as a systematic discipline, the second was to develop implementations of this language which are both reliable and efficient on presently available computers.

The success of this language design proves that Pascal is not "yet another language". Today, Pascal has been implemented on almost all computers commonly in use, ranging from the very large computers to mini- and micro-computers.

This manual contains the information necessary to compile and execute Pascal programs on the ND-100 and the ND-500. It is assumed that the reader is familiar with the Pascal language. The uninitiated reader is referred to the Pascal Report or to an appropriate textbook.

The present chapter gives a general description of the ND-Pascal system. The specific information necessary for the compilation and execution of Pascal programs is found mainly in chapters 2 to 4. Most of chapters 5 to 8 describe features for the more advanced use of ND-Pascal.

ND-Pascal has been implemented according to the definition in "Niklaus Wirth: The Programming Language Pascal. Revised Report. (1973)". Also, the specifications in the ISO Pascal standard have been adhered to. Hereafter this language definition will be referred to as Standard Pascal.

ND-Pascal is a superset of Standard Pascal, and has several extensions in relation to it. Especially, extensions have been introduced to facilitate the compilation and execution of Pascal programs in a time-sharing environment. Explicit extensions of the Standard Pascal language will be noted as such in this manual. The extensions should be avoided if program exportation is planned or probable.

1.1 The Pascal Compiler

The ND-Pascal compiler was developed from the Pascal TRUNK compiler designed at ETH, Zurich. The compiler produces relocatable code, which can be loaded by the appropriate loader (ND-100 NRL or ND-500 Linkage Loader) and then executed. A program may refer to separately compiled procedures and functions written in Pascal, FORTRAN, PLANC, COBOL or assembly language.

The ND-Pascal compiler is itself written in Pascal. Also, parts of the run-time library are written in Pascal.

1.2 The Main Implementation Dependent Characteristics

The maximum set size is 256 elements. A variable of type set will occupy the minimum number of words necessary to represent the values in the set type. Sets of subranges of integer will contain 256 elements.

N100: A Pascal program may be run either as a one-bank or a two-bank program. As a one-bank program, all program and data reside within 128K bytes of memory. As a two-bank program, the program may occupy up to 128K bytes in the instruction bank, and the data occupy up to 128K bytes in the data bank. One- or two-bank execution may be selected at compile-time with the B option, or at load-time with the DEFINE NO8KS command.

A Pascal program may be run as a real-time program.

Large program systems may be overlaid using the standard NRL overlaying mechanism.

N500: The buddy instructions of the ND-500 hardware are utilized when a program administers the heap with the NEW and DISPOSE procedures.

1.3 The Main Extensions

Variables in the main program can be initialized. There is a convenient syntax for array initialization.

Variable conformant arrays, as specified in the ISO Pascal standard, are implemented. With this mechanism, a formal parameter will be compatible with actual array parameters of different sizes.

N100: The type LONGINT is a standard integer type with a precision of 32 bits.

N500: The type LONGREAL is a standard real type with a precision of approximately 16 digits.

The procedures CONNECT and DISCONNECT enable a program to associate a Pascal file variable with an external file at run-time. CONNECT has been implemented such that the actual name of the external file easily can be entered from the terminal running the program.

Random access I/O can be performed with the procedures GETRAND and PUTRAND.

Through the use of the FAULT procedure, a program may trap run-time errors.

2 THE SOURCE PROGRAM

A Pascal source file must contain either

- 1) A full Pascal program, or
- 2) One or more procedures or functions, or
- 3) N100: One or more procedures, functions, or modules.

The source language must be Standard Pascal, with the restrictions and possible extensions described in this manual.

A full Pascal program compiles into an executable object program, while procedures and functions compile into code that may be loaded together with a full program. A source file of the latter kind must be terminated with the character "." (period).

The source file character set must be ASCII, where the lines are separated by the Carriage Return character, and optionally, the Line Feed character. Files produced by QED, TED and PED are acceptable as input to the compiler.

The compiler recognizes the source file types :PASC and :SYMB by default, :PASC being the primary type. Any other file type must be specified explicitly.

A source input line must not exceed 96 characters. The Pascal compiler indicates a longer line as an error.

2.1 Special Symbols

Some of the special symbols in Standard Pascal have one or more alternate representations in ND-Pascal:

Standard Pascal	ND-Pascal
	or @
{	{ or (*
}	} or *)
[[or (.
]] or .)
<u>and</u>	<u>and</u> or &
<u>not</u>	<u>not</u> or ~

The ~ symbol has various external representations on different terminals and printers.

A comment opening with the character "{" must be closed with the character "}". Similarly, "(*" is matched only by "*)".

2.2 Identifiers

An identifier may be of any length, but only the first eight characters are significant. Within an identifier, lower case letters are converted to upper case, unless the U option is off.

2.3 Keywords

The following are Pascal keywords, and cannot be used as identifiers:

Standard Pascal keywords:

<u>and</u>	<u>array</u>	<u>begin</u>	<u>case</u>
<u>const</u>	<u>div</u>	<u>do</u>	<u>downto</u>
<u>else</u>	<u>end</u>	<u>file</u>	<u>for</u>
<u>function</u>	<u>goto</u>	<u>if</u>	<u>in</u>
<u>label</u>	<u>mod</u>	<u>nil</u>	<u>not</u>
<u>of</u>	<u>or</u>	<u>packed</u>	<u>procedure</u>
<u>program</u>	<u>record</u>	<u>repeat</u>	<u>set</u>
<u>then</u>	<u>to</u>	<u>type</u>	<u>until</u>
<u>var</u>	<u>while</u>	<u>with</u>	

Extra keywords in ND-Pascal:

<u>module</u>	<u>value</u>
---------------	--------------

Note: The keyword module is legal, but has no effect in ND-500 Pascal. It is retained in ND-500 Pascal to facilitate porting of programs between the ND-100 and the ND-500.

A keyword may be written with lower and/or upper case characters. However, within a keyword all lower case characters will be converted to upper case. Thus,

end END End

are all representations of the keyword end.

2.4 Standard Identifiers

Following is a list of the standard identifiers in ND-Pascal. A standard identifier may be considered as if it were defined in a block enclosing the program, and as such, may be redefined. Normally, such redefinition should be avoided, since it easily may lead to confusion.

Standard identifiers in Standard Pascal:

ABS	ARCTAN	BOOLEAN	CHAR
CHR	COS	DISPOSE	EOLN
EOF	EXP	FALSE	GET
INPUT	INTEGER	LN	MAXINT
NEW	ODD	ORD	OUTPUT
PACK	PAGE	PRED	PUT
READ	READLN	REAL	RESET
REWRITE	ROUND	SIN	SQR
SQRT	SUCC	TEXT	TRUE
TRUNC	UNPACK	WRITE	WRITELN

Extra standard identifiers in ND-Pascal:

CONNECT	COSH	DISCONNECT	FIRST
GETRAND	HALT	LAST	LMAXINT
LONGINT	LONGREAL	LROUND	LTRUNC
MARK	MAXREAL	POWER	PUTRAND
RELEASE	SINH		

All standard identifiers are written in upper case letters.

2.5 Compiler Commands

The source program text may contain commands to the compiler. A command is signalled by the character "\$" in position one of a source line. The rest of such a line is treated as a command to the compiler, and no part of it will be included in the proper program text.

The available compiler commands are

\$SET

\$RESET

\$IFTRUE

\$IFFALSE

\$ENDIF

\$OPTIONS

\$INCLUDE

\$EOF

\$LINESPP

\$PAGE

A compiler command may be abbreviated to its shortest unambiguous form.

2.5.1 Conditional compilation

The ND-Pascal compiler may be instructed to skip specified parts of the source text. This may be useful in order to generate different versions of a program from the same source file.

The skipping of source text is steered by flags, which are Boolean variables. The flag identifiers are distinct from the program identifiers, therefore no name conflicts between flag and program identifiers can occur. A flag identifier can have up to eight significant characters. No distinction is made between upper and lower case characters.

A flag is given the value TRUE by the command

\$SET <flag>

A flag is given the value FALSE by the command

\$RESET <flag>

The skipping of source text is effected by the commands

\$IFTRUE, \$IFFALSE, and \$ENDIF

The command

\$IFTRUE <flag>

has the effect:

If <flag> has the value TRUE: No effect.

If <flag> has the value FALSE:

Skip source text up to an \$ENDIF <flag> with the same flag name.

The command

\$IFFALSE <flag>

has the effect:

If <flag> has the value TRUE:

Skip source text up to an \$ENDIF <flag> with the same flag name.

If <flag> has the value FALSE: No effect.

If an \$IFTRUE or \$IFFALSE command has a flag parameter that was not previously defined, it will become defined and given the value FALSE.

Note that when source text is skipped, compiler commands (such as \$SET, \$IFTRUE etc.) will also be skipped.

2.5.2 Multiple source files

The \$INCLUDE-command facilitates insertion of source text from an alternate file in the program being compiled. This is useful when a set of programs (within the same project, say) use a common set of type, variable, and procedure definitions. Also, "standard" data structures and procedures for handling problems within a specific problem area, can easily be incorporated in a program with the

\$INCLUDE-command.

The INCLUDE file may be divided into sections by the \$EOF-command.

The command

```
$INCLUDE <filename>
```

has the effect of switching the input stream from the present input file to <filename>. When end of file or \$EOF on <filename> is reached, the input stream will be switched back to the previous input file. The effect is to insert the text in <filename> at the place where the \$INCLUDE-command occurs.

The command

```
$INCLUDE
```

has the effect that the next section of the most recent INCLUDE file is inserted in the program.

\$INCLUDE-commands may be nested to a maximum depth of four.

2.5.3 Intermixing definition sections.

In the standard mode, when the N option is off, ND-Pascal requires that the label, const, type, var, value, and procedure/function sections of a block appear in this order. When the N option is on, these sections may appear in any order, and each section kind may appear more than once. However, a main program may not contain another var section after a value section, or after the first procedure or function declaration.

2.5.4 Options

There is a set of options that affect the output produced by the Pascal compiler. Each option has a one-letter name.

Some of the options are associated with counters. A counter value greater than zero means that the option is on, a value equal to or less than zero means that the option is off. The remaining options are associated with specific values.

A counter option is increased or decreased by one by writing the option name followed by "+" or "-", respectively.

The available options are (counter options are indicated by the character "*"):

- Bn N100: Specify n-bank execution of program (n=1 or n=2). When n=2 the compiler will produce two-bank BRF code. Default value is n=1.
- C* When on, the value range of CHAR is extended to 256 values (internal values 0 to 255). Also, on TEXT files parity will not be removed on input, nor generated on output, and both values 15 octal and 215 octal will give EOLN = TRUE. The default value is 0 (off), which implies that CHAR is the ASCII set (128 values), and that parity is removed (generated) on input (output) from/to TEXT files.
- Ic Allow c as a legal character in an identifier. c must be in the set ['!', ' ', '#', 'Z', '?', '_', '|', '\']. The character "#" should in general be avoided, since it is used in entry point names in the Pascal library.
- L* Generate listing. Default value is 1 (on).
- M* List generated object code in symbolic form. Default value is 0 (off).
- N* This option (Non-standard) must be on to allow the following extensions to be used:
- a) Intermixing definition sections.
 - b) Use of the FAULT procedure for error trapping.
- Default value is 0 (off).
- P* Program code dump. Default value is 0 (off). This option produces listing output which enables a closer inspection of the code generated by the compiler. This is very useful when tracing a possible error in the Pascal system. Therefore, whenever there is reason to believe that a failure is caused by erroneous object code, the user is requested to submit a listing of a P dump compilation together with the error report.
- Rn N100: Specify n-word real (n=2 or n=3). Default value is 2 on ND-100s with 32-bit floating point hardware, and 3 on ND-100s with 48-bit hardware. A program that is to be cross-compiled must not contain real constants.
- T* Generate code to check array indices, subrange assignments, pointer values and arithmetic overflow. Turning this option off will make the object program smaller and faster, but also unsafe. Default value is 1 (on).

The T option may be switched on and off at any point in the program, in order to perform run time checks in selected parts of the program.

N100: The ND-100 hardware does not facilitate the checking of overflow on floating point arithmetic operations. Therefore, ND-Pascal can only detect overflow on integer operations. As a special case, attempted floating division by zero is detected.

N500: In the ND-500, overflow is trapped by hardware, and not by explicit code checking for overflow. This implies that check for overflow will not be turned off by turning the T option off. However, a program may use the SETE and CLTE procedures (cfr. section 2.7.5) to dynamically turn any hardware trap on or off.

U* Convert lower case characters outside strings to upper case. Default is 1 (on).

V* For each procedure, list local variables in alphabetical order, with their respective relative addresses and the number of times each variable is referenced. Default value is 0 (off).

X* When on, the loader symbols generated as entry point names for procedures/functions on the outermost level of a main program or a separately compiled file will be the names given by the programmer. If the option is off, anonymous entry point names will be generated for these routines (cfr. chapter 8). Default value is 0 (off).

Z* Initialize all variables to zero: At load-time, initialize all main program variables to zero before the value section is loaded. At run-time, every time a procedure is called, or an object generated by NEW, all variables local to that procedure or object will be initialized to zero. Default value is 0 (off).

Options may be set within a comment in the source program. The first character within the comment must be "\$". Thereafter, option settings separated by "," may follow. Options may also be set following the \$OPTIONS compiler command.

Examples:

{SM+,I_,T-} means:

M+ List object code.

I_ Allow "_" as a legal character in an identifier.

T- Do not generate testing instructions.

\$OPT Z+,U- means:

Z+ Initialize all variables to zero.

U- Do not convert lower case characters to upper case.

2.5.5 Program listing

The command

\$LINESPP n

orders the Pascal compiler to print the program listing with n lines per page. The default value for n is 60. (This default may be set to some other value when the ND-Pascal system is installed.)

The command

\$PAGE

gives new page in the program listing.

2.6 Implementation Dependent Features

2.6.1 Standard types

Standard Pascal has the following standard types:

BOOLEAN, CHAR, INTEGER, REAL, TEXT

ND-Pascal in addition has the following standard types:

LONGINT, LONGREAL

Actually, LONGINT is an extension only in ND-100 Pascal, while LONGREAL is an extension only in ND-500 Pascal. In ND-100 Pascal LONGREAL is equivalent to REAL. In ND-500 Pascal LONGINT is equivalent to INTEGER.

The following table gives the memory space, in bytes, occupied by variables of the standard types (provided they do not occur within packed structures):

	<u>N100</u> 32-bit	<u>N100</u> 48-bit	<u>N500</u>
BOOLEAN	2	2	1
CHAR	2	2	1
INTEGER	2	2	4
LONGINT	4	4	4
REAL	4	6	4
LONGREAL	4	6	8
TEXT	16	16	28

The maximum values and accuracy of the arithmetic types are given in the following table:

	Maximum value	Precision
2-byte INTEGER	32,767	-
4-byte INTEGER	2,147,483,647	-
4-byte REAL	10 ⁷⁶	7 digits
6-byte REAL	10 ⁴⁹³⁰	10 digits
8-byte REAL	10 ⁷⁶	16 digits

An integer constant which exceeds the 16-bit integer maximum value will get the type LONGINT. Also, an integer constant may be suffixed with the letter L to force it to become a LONGINT constant.

The standard functions LROUND and LTRUNC are available to round or truncate, respectively, reals to LONGINT.

In an array declaration, the indices may not be of type LONGINT.

LMAXINT is a standard constant with a value equal to the maximum LONGINT value.

A real constant with 10 or more digits is given the type LONGREAL. Also, the type of a real constant will be LONGREAL if the exponent character D is used instead of E.

When necessary, ND-Pascal automatically converts from INTEGER to REAL or LONGREAL, and between REAL and LONGREAL.

MAXREAL is a standard constant with a value equal to the maximum floating point value.

2.6.2 Structured types

Variables of structured types (records and arrays) may be assigned to and compared for equality or inequality, provided the variable type is not packed nor contains packed variables. Variables of type packed array [...] of CHAR may be assigned to and compared using all the relational operators (=, <>, <, <=, >=, >).

Note that there is no syntax for the specification of a structured constant.

2.6.3 Packed structures

Record and array types may be specified as packed. Each single variable will then occupy a minimum number of bits, and several single variables may be packed into one computer byte or word. A record or an array will always start at a word (N100) or byte (N500) boundary.

The use of packed structures saves data space, but may increase execution time significantly.

A variable within a packed structure cannot be used as a var parameter to a procedure. However, the standard procedure READ may have an element of a packed array ... of CHAR as a parameter.

See chapter 5 for information on packed files.

2.6.4 Strings and character arrays

A string constant is padded with blanks to the required length. The string may occur in a value section, in an assignment statement, as an actual parameter, or in a Boolean expression. This is an extension to Standard Pascal.

In Standard Pascal, a string constant with n characters is of the type packed array [1..n] of CHAR. This inhibits assignment of, or parameter substitution with, a string to a variable or formal of type packed array [...] of CHAR where the lower bound is different from 1. In ND-Pascal such assignment or substitution is legal, provided the length of the string is equal to the length of the array.

2.6.5 Procedure parameters

2.6.5.1 Conformant arrays

Variable conformant arrays, as specified in the ISO Pascal standard, are implemented in ND-Pascal. (Conformant arrays by value is not implemented.)

A variable (non-value) formal parameter may be specified as a conformant array. It is then possible to transmit array parameters of different sizes through this formal parameter. The index bounds of the actual parameter are implicitly available to the body of the called procedure.

A conformant array parameter is specified as such in the procedure heading by the following syntax:

```
<variable-parameter-specification> ::=  
  "var" <identifier-list> ":"  
  ( <type-identifier> | <conformant-array-schema> )  
  
<conformant-array-schema> ::=  
  "array" "[" <index-type-specification>  
  { ";" <index-type-specification> } "]" "of"
```

(<type-identifier> | <conformant-array-schema>)

<index-type-specification> ::=
 <bound-identifier> ".." <bound-identifier> ":"
 <ordinal-type-identifier>

<bound-identifier> ::=
 <identifier>

Example:

```
procedure matmult(var x, y, z: array [l1..h1: INTEGER] of
                    array [l2..h2: INTEGER] of REAL);
```

If the component of a conformant-array-schema is itself a conformant-array-schema, then an abbreviated form of definition, equivalent to the abbreviated form of multiple-dimension array definition, may be used.

Example:

```
array [l1..h1: T1] of array [l2..h2: T2] of T3
```

is equivalent to

```
array [l1..h1: T1; l2..h2: T2] of T3
```

When transmitting an array as a parameter through a formal conformant array parameter, the actual parameter must be conformable with the conformant-array-schema. The term conformable is defined as follows:

If T1 is an array-type, and T2 is the type denoted by the ordinal-type-identifier of the index-type-specification of a conformant-array-schema, then T1 is conformable with the conformant-array-schema if all the following four statements are true.

- (a) The index-type of T1 is compatible with T2.
- (b) The smallest and largest value of the index-type of T1 lie within the closed interval defined by values of T2.
- (c) The component type of T1 is the same as the component type of the conformant-array-schema, or is conformable to the component conformant-array-schema.
- (d) If T1 is designated packed then T2 must be declared as packed.

It is an error if the smallest or largest value of the index-type of T1 lies outside the closed interval defined by the values of T2.

The bound-identifiers denote the smallest and largest values, respectively, of the index-type of the actual parameters. These values are implicitly transmitted to the called procedure. The procedure may not change the values of the bound-identifiers.

Example:

```
var x, y, z: array [1..10] of REAL;
    p, q, r: array [0..100] of REAL;

procedure product(var a, b, c: array [low..high: INTEGER] of REAL);
var i: INTEGER;
begin
    for i := low to high do
        c[i] := a[i]*b[i]
    end (*product*);

    product(x,y,z);
    product(p,q,r);
```

2.6.5.2 Formal procedures

A procedure which appears as an actual procedure parameter, may itself only have value parameters. On entry to a formal procedure, ND-Pascal checks the actual parameters only to see if they occupy the same number of words as the formal parameters. The user is warned that the use of formal procedures with pointer parameters is unsafe.

2.6.6 ND-500 traps

When an ND-500 Pascal programs is started, the following traps are set in the OTE register:

bit 9	overflow
bit 11	invalid operation
bit 12	divide by zero
bit 14	floating overflow
bit 16	illegal operand value
bit 24	address zero access
bit 25	descriptor range
bit 26	illegal index
bit 27	stack overflow

When a routine defined as STANDARD is entered, all trap bits are switched off. The trap bits are restored when return to Pascal is made.

The reader is referred to the ND-500 Reference Manual (ND-05.009) for further details on hardware traps.

2.7 Extensions in ND-Pascal

This section describes most extensions in ND-Pascal. Refer to chapter 5 for I/O extensions. Real-time programs are described in chapter 6, and overlays are described in chapter 7.

2.7.1 Variable initialization

Scalar and array variables in the main program may be initialized. Initialization is signalled by the keyword value. A value section must appear after the var-declarations and before the first procedure or function declaration, or main program begin.

Packed arrays, except for packed array ... of CHAR, records, sets and pointers may not be initialized.

The syntax for initialization is:

```

<variableinit> ::=      "value" <initialization>
                        { <initialization> }
<initialization> ::=    <variable> "=" <val> ";"
<val> ::=                <constant> | "(" <valuelist> ")"
<valuelist> ::=          <aval> { "," <aval> }
<aval> ::=               <constant> | <count> "*" <constant>
<count> ::=              <integer constant>

```

Examples:

value

```

X = 2.55;
I = 19;
TABLE = (1,3,2*7,-1,11*0);
NAME = ('PASCAL ');

```

Since a string has the type packed array [1..n] of CHAR, a string constant must be enclosed in parentheses as shown in the last example.

2.7.2 External Pascal routines

The compiler accepts a source file containing procedure and function declarations only. The file must be terminated with a period.

The generated relocatable file may be loaded with any Pascal main program which contains external declarations of one or more of the Pascal routines. Only those routines which are actually referred, are loaded (each external Pascal routine contains a LI8 <entrypoint> loader directive). An external declaration is a procedure or function heading followed by a body consisting of the word "EXTERN". Example:

function f(x: REAL): INTEGER; EXTERN;

External routines may use external declarations to get access to routines on the outermost level of the main program, provided the main program was compiled with the X option on.

There is no check of the correspondence between the parameter list of the external declaration and of the separately compiled procedure.

A file of Pascal routines may be headed by constant, type and variable definitions. The variable definitions, if present, will overlap the variables of the main program. These definitions may be used in parameter specifications, or within the routines. The user is warned that ND-Pascal does not check that the definitions are consistent with corresponding definitions in the main program. It is therefore strongly recommended that the \$INCLUDE facility be used to incorporate global definitions in an external program module.

2.7.3 External routines in other languages

Separately compiled FORTRAN, PLANC or COBOL subroutines may be called from an ND-Pascal program. Such a routine must be declared in the Pascal program with a procedure or function heading, and a body consisting of the word "STANDARD". Example:

procedure ext(var x, y: REAL); STANDARD;

Parameters of any type and kind, except Pascal procedure or function names, may be transmitted to the external routine; however, no check is made that the parameters are consistent with the formal arguments of that routine.

N100: In order to interface to the old version of ND-100 FORTRAN, the routine must be specified as "FORTRAN".

N500: All hardware traps are switched off when entering a STANDARD routine. The original traps are restored when returning to Pascal.

Pointers to the actual arguments are transferred to the external routine. A value (non-var) parameter will be copied to a scratch area, and a pointer to this copy transferred.

Be aware that many library utility routines in other languages may get the parameters transferred in a non-standard way, and thus may not be called directly from a Pascal program.

When loading modules for a mix of Pascal and routines in other languages, the following order must be observed:

- 1) Pascal main program

- 2) Pascal and other external routines
- 3) Other language libraries as necessary
- 4) Pascal library

2.7.4 Standard procedures and functions

In addition to the standard procedures and functions in Standard Pascal, the following are standard in ND-Pascal.

SINH and COSH

These real functions calculate the arithmetic functions sinh and cosh respectively.

POWER

POWER is a real function with two parameters x and y which calculates the function $x|y$. When y is real, $x|y$ is calculated by the formula $x|y = e^{(y \ln(x))}$. Thus, POWER(-1.0,2.0) will give a runtime error, while POWER(-1.0,2) will give the correct result 1.0.

HALT

HALT is a procedure which takes an optional string parameter. HALT writes the string (if any) to the terminal, and aborts the program.

MARK and RELEASE

MARK and RELEASE provide an alternative to DISPOSE for the deallocation of heap space. In applications where heap space is allocated and deallocated in a stack fashion, the use of MARK and RELEASE is more efficient, and may be more convenient, than the use of DISPOSE.

Both procedures take a pointer variable as a parameter. The call MARK(<ptr>) assigns the address of the current heap top to <ptr>. The call RELEASE(<ptr>) deallocates all variables on the heap beyond the value of <ptr>.

A program which calls DISPOSE may not call MARK or RELEASE.

2.7.5 External procedures and functions

The Pascal library contains a set of external procedures and functions. To use one of these, the procedure or function must be declared as external within the program.

An installation may choose to have a system file containing external declarations for these external procedures and functions. This file may then be included in a program with the \$INCLUDE compiler command.

TUSED

External declaration:

function TUSED: REAL; EXTERN;

TUSED gives the elapsed CPU time in seconds.

TIME and DATE

External declarations:

procedure TIME(var hour, min, sec: INTEGER); EXTERN;

procedure DATE(var year, month, day: INTEGER); EXTERN;

TIME and DATE give the current time and date, respectively.

ECHOM

External declaration:

procedure ECHOM(echomode: INTEGER); EXTERN;

Executes MON ECHOM with echomode as parameter. This will define the echo mode for the terminal as specified in the SINTRAN manual.

Note: The file CONNECTed to the terminal must have logical unit number 1.

BRKM

External declaration:

procedure BRKM(breakmode: INTEGER); EXTERN;

Executes MON BRKM with breakmode as parameter. This will define the break mode for the terminal as specified in the SINTRAN manual.

Note: The file CONNECTed to the terminal must have logical unit number 1.

ERMSG

External declaration:

procedure ERMSG(errorno: INTEGER); EXTERN;

Executes MON ERMSG with errorno as parameter. This will write the SINTRAN error message corresponding to the given error number to the terminal.

HOLD

External declaration:

procedure HOLD(time: REAL); EXTERN;

Suspends execution of the program in <time> seconds. <time> is accurate to 20 milliseconds.

VERSN

External declaration:

procedure VERSN(var year, month, day: INTEGER); EXTERN;

Gives the date when the executing program was compiled.

RUNMODE

External declaration:

function RUNMODE: INTEGER; EXTERN;

Gives the execution mode of the running program:

- 0 - interactive
- 1 - batch
- 2 - mode
- 3 - real-time

FREEMEM

External declaration:

function FREEMEM: LONGINT; EXTERN;

Gives the size of the present free memory, that is, the size of the area between stack top and heap top, in number of bytes.

LUNIT

External declaration:

function LUNIT(var f: <filetype>): INTEGER; EXTERN;

Gives the logical unit number of the (open) file f.

ISIZE

External declaration:

function ISIZE(lun: INTEGER): INTEGER; EXTERN;

Gives the result of a MON ISIZE on the given logical unit.

OSIZE

External declaration:

```
function OSIZE(lun: INTEGER): INTEGER; EXTERN;
```

Gives the result of a MON OSIZE on the given logical unit.

ROBJENT

External declaration:

```
procedure ROBJENT(lun: INTEGER; var b: BUFFER;  
                  var status: INTEGER); EXTERN;
```

Reads the object entry of the file with logical unit lun into the buffer b. BUFFER may be any type with a length of at least 64 bytes. The SINTRAN status of the operation is left in the status parameter.

COMMAND

External declaration:

```
procedure COMMAND(str: STRING); EXTERN;
```

Performs MON COMND with str as parameter. The type STRING must be defined as packed array ... of CHAR. The value str must be terminated by the character "" (written "" within a string constant).

N100: In ND-100 Pascal the type STRING must have a length greater than 16.

N500: The ND-500 monitor allows only a subset of the SINTRAN commands to be executed by the COMND monitor call.

MDLFI

External declaration:

```
procedure MDLFI(var str: STRING); EXTERN;
```

Deletes the file with the name found in str.

REABT

External declaration:

```
procedure REABT(lunit: INTEGER; var ibyte: LONGINT); EXTERN;
```

Executes the REABT monitor call.

SETBT

External declaration:

```
procedure SETBT(lunit: INTEGER; ibyte: LONGINT); EXTERN;
```

Executes the SETBT monitor call.

RMAX

External declaration:

```
procedure RMAX(lunit: INTEGER; var ibyte: LONGINT); EXTERN;
```

Executes the RMAX monitor call:

SMAX

External declaration:

```
procedure SMAX(lunit: INTEGER; ibyte: LONGINT); EXTERN;
```

Executes the SMAX monitor call.

RANDOM

External declaration:

```
function RANDOM(var x: REAL): REAL; EXTERN;
```

This function produces a uniformly distributed pseudo random number in the open interval $\langle 0,1 \rangle$. Each new value is calculated from the value of the parameter. The new value is also assigned to the parameter variable. Thus, successive calls on RANDOM with the same variable as a parameter, produces a uniformly distributed pseudo random number stream.

N500: SETE

External declaration:

```
procedure SETE(bitno: INTEGER); EXTERN;
```

Sets the given bit in own trap enable register.

CLTE

External declaration:

```
procedure CLTE(bitno: INTEGER); EXTERN;
```

Clears the given bit in own trap enable register.

2.7.6 Generic functions

For each scalar type T there is a function $T(n)$ which converts the integer n to the value of type T with ordinal number n .

Example:

```
type
  Season = (Winter, Spring, Summer, Autumn);
var
  s: Season;
  . . .
  s := Season(2);
```

s now has the value Summer.

The functions $FIRST(T)$ and $LAST(T)$, where T is an ordinal type identifier, gives the value of type T which is the smallest and greatest value, respectively, within the type T .

Example:

$LAST(Season)$ is equal to Autumn.

2.7.7 Miscellaneous extensions

The compiler accepts octal and hexadecimal integer constants. The syntax is as follows:

```
<octal constant> ::= <sign> <octdig> { <octdig> } <size> "B"
<hex constant> ::= <sign> <digit> { <hexdig> } <size> "H"
<sign> ::= <empty> | "+" | "-"
<octdig> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
<hexdig> ::= <digit> | "A" | "B" | "C" | "D" | "E" | "F"
<size> ::= <empty> | "L"
```


3 PROGRAM COMPILATION

The ND-Pascal compiler is invoked by the command

N100: @PASCAL N500: @ND-500-MONITOR PASCAL

Initially, the compiler enters into a command processing mode, to enable the user to specify source, list and code files, options etc. The command processor prompts the user to give a new command with the character "\$".

N100: If the compiler has been aborted by typing the ESC key, it may be resumed with the @CONTINUE command. In this case the previous flag and option settings are retained. However, files have been closed and their names are no longer known to the compiler.

The available commands are:

HELP
COMPILE
RUN
CLEAR
OPTIONS
SET
RESET
VALUE
LINESPP
EXIT

A command may be abbreviated to its shortest unambiguous form.

Note that the SET, RESET, LINESPP, and OPTIONS commands also are available as compiler commands (cfr. section 2.5).

3.1 HELP

The HELP command lists the available commands on the user's terminal (or batch output file). The list includes both the command processor commands and the compiler commands.

3.2 COMPILE

The COMPILE command instructs ND-Pascal to compile the specified source file. The present setting of flags and options will be used during the compilation.

The syntax of the COMPILE command is

COMPILE <source file>, <list file>, <code file>

The entire parameter list may be omitted, in which case the command processor prompts the user to specify the files one by one. If only one or two parameters are specified, defaults are assumed for the remaining parameters.

The parameters to COMPILE may either be the actual file names, or the logical units (octal) of open files.

<source file> contains the program to be compiled. The default file types are :PASC and :SYMB, :PASC being the primary type.

<list file> is the file to which the listing of the compiled program is written. The <list file> parameter may be omitted, in which case no listing is generated.

The listing contains:

in column 1: The character "*" if the line contains one or more language features not in Standard Pascal. Otherwise the character " ".

in column 2: Program (source) line number.

in column 3: Source file line number and nesting level for INCLUDED files.

in column 4: Relative program and variable addresses (octal).

in column 5: A numbering of the begin-end, repeat-until, case-end, and if-else pairs in the program, to indicate the nesting structure of the program. Also, the declaration level for each procedure and function is indicated.

in column 6: The source program.

Columns 4 and 5 are suppressed if the listing file is the terminal.

The listing is divided into pages with a heading on each page containing: version of compiler, date and time of compilation, and page number.

The listing indicates a language syntax error at the exact spot where it was discovered, together with an error number. If a part of the source text was skipped as a result of the error, the part that was skipped is indicated by a line containing the text **SKIP* at the left, and hyphens under the skipped text. Lines containing syntax errors are also written to the terminal.

At the end of the listing a list of the error numbers and an explanatory text for each error will appear.

A list of all compiler error messages is found in appendix A.

<code file> is the file on which the relocatable output will be written. The <code file> parameter may be omitted, in which case no object code is generated. Be aware that the ND-500 Linkage Loader does not accept a file number as an NRF input file.

In a second or following COMPILE command, only <source file> need be specified. The previous <list file> and <code file> are used if they were specified in a previous COMPILE command. If a new <list file> or <code file> is specified, the previous file is closed, and the new file opened.

Be aware that option and flag values may be affected by a compilation, and thus may influence the result of a succeeding compilation. Use the CLEAR command to bring the processor back to its initial state.

3.3 RUN

The RUN command may be used to compile and execute a program, or to load and execute a previously compiled program.

The syntax of the RUN command is

RUN <filename>

where the <filename> parameter is optional. If not present, the most recently produced relocatable file is loaded and executed.

If <filename> is given, the following actions are taken:

Pascal attempts to open <filename>:PASC (or <filename>:SYMB) and <filename>:BRF (N100) or <filename>:NRF (N500):

- a) If only the :PASC (:SYMB) file exists, the program is compiled to a scratch file, and then loaded and executed.
- b) If only the file containing the relocatable code exists, then this program is loaded and executed.
- c) If both exist, a compilation to the relocatable file is done if the :PASC (:SYMB) file is more recent than the relocatable file. Then the relocatable file is loaded and executed.

Note: After a program has finished a RUN execution, the SINTRAN "a" prompt character will not appear. The user therefore must type ESC to get back to SINTRAN command mode.

3.4 CLEAR

The CLEAR command brings the command processor back to its initial state. The following actions are taken by CLEAR:

Set all options to their default values.

Delete all flags.

Close <list file> and <code file>.

3.5 OPTIONS

The OPTIONS command is used to set compiler options. The command and the options are described in section 2.5.4.

3.6 SET and RESET

The SET and RESET commands set a flag to TRUE and FALSE, respectively. These commands, and the use and effect of flags are described in section 2.5.1.

3.7 EXIT

The EXIT command closes all files and returns control to the operating system.

3.8 LINESPP

The LINESPP command is described in section 2.5.5.

3.9 VALUE

The command

\$VALUE OPTIONS

lists the current value of all options.

The command

\$VALUE FLAGS

lists the current value of all flags.

3.10 SINTRAN Commands

SINTRAN commands may be executed by starting a command line with the character "@". Pascal will then pass the rest of the line to SINTRAN for interpretation and execution.

N500: The ND-500 monitor allows only a subset of the SINTRAN commands to be executed. When attempting to execute a SINTRAN command outside this subset, the SINTRAN error message is written to the terminal.

3.11 Program Compilation Example

Following is an example of a program compilation. User input is underlined.

Terminal input/output	Comments
<u>@PASCAL</u>	Call Pascal compiler
or	
<u>@ND-500-MONITOR PASCAL</u>	Call Pascal compiler
PASCAL/ND-xxx VERSION J 83-xx-xx	Identifying text
<u>\$OPTION T-,M+</u>	Suppress generation of test instructions and list generated object code.
<u>\$SET PARIS</u>	Generate "PARIS" version of program. (Assumes source file contains \$IFTRUE and \$IFFALSE tests on flag with name PARIS.)
<u>\$COMPILE</u>	Compile
Source file=MYPROG	Source is MYPROG
List file=LINE-PRINTER	Listing to line printer
Code file=MYPROGCODE	Relocatable code to MYPROGCODE
NO ERRORS	Messages from compiler
LENGTH OF PROGRAM: 01077B WORDS/BYTES	
LENGTH OF FIXED DATA: 00203B WORDS/BYTES	
6 USES OF NON-STANDARD FEATURES	
24.32 SECONDS COMPILATION TIME	
<u>\$EXIT</u>	Exit
@	Control to SINTRAN

Cfr. chapter 9 for a complete example of a program compilation and execution.

4 PROGRAM LOADING AND EXECUTION

4.1 Program Loading

This chapter gives examples of the loading and execution of ND-Pascal programs. Further information on memory allocation, absolute programs etc. is found in chapter 8. Cfr. chapter 9 for a complete example of a program compilation and execution.

4.1.1 ND-100 program loading

A compiled ND-100 Pascal program must be loaded by the NRL loader before it can be executed. Also, the Pascal library must be loaded together with the object program. The library comes in two versions: PASCAL-LIB:BRF for one-bank code, and PASCAL-2LIB:BRF for two-bank code. The reader should consult the NRL manual (ND-60.066) for details concerning the loader and the loading process.

Example (one-bank program):

Terminal input/output	Comments
<u>@NRL</u>	Call loader
RELOCATING LOADER LDR-1935x	Identifying text
<u>*LOAD MYPROGCODE PASCAL-LIB</u>	Load code file and Pascal library
FREE:027433-162504	Free memory area
<u>*RUN</u>	Execute program
@	Execution finished

A two-bank program may be generated in one of two ways:

1. @CC Compile program, producing one-bank code
@NRL
RELOCATING LOADER . . .
*DEFINE NOBKS 2
*LOAD MYPROGCODE PASCAL-LIB
*RUN
. . .
2. @CC Compile program with option B2, producing two-bank code
@NRL
RELOCATING LOADER . . .
*PROG-FILE MYPROG
*LOAD MYPROGCODE PASCAL-2LIB
*EXIT
@RECOVER MYPROG

Method number 2 will save space in the instruction bank, however, method number 1 must be used with SINTRAN version H or earlier if the program is to be dumped as a re-entrant subsystem. The reader should consult section 8.1.1 for further details on two-bank programs.

When loading files for a Pascal execution, the main program must always be loaded first, and the Pascal library last. This means that all external Pascal, FORTRAN or assembly routines and other libraries (i.e. FTNLIBR) must be loaded between the main program and the Pascal library.

Take note of the fact that NRL uses entry point names with seven letters, and gives no warning when an already defined name is redefined. This may lead to undetected name conflicts when loading a Pascal program which was compiled with the X option on, or when loading separately compiled Pascal procedures. Under these circumstances, procedure and function names therefore should be distinct within the first seven letteres. Cfr. section 8.1.2 for further details on entry point names.

Instead of direct execution with the RUN-command, as shown in the above example, a program may be dumped on a :PROG file and subsequently executed any number of times. Also, by generating a :BPUN file, a Pascal program may be dumped as a re-entrant subsystem. The entry point name of the start address of the program is the name which appears in the program statement. This name and the corresponding program address are found in the loader map.

If the label 0 (zero) appears in the main program, its address will appear in the loader map with the name CONTINU. This address may be used as a restart address for the program. It is the programmer's responsibility that necessary reinitialization is done after a restart. For example, files which might have been open when the program was aborted, should be DISCONNECTed in order to deallocate I/O buffers.

The NRL command PROG-FILE should be used with great care due to limitations in the SINTRAN RECOVER command. Unless special precautions are taken, a "hole" may remain in the area between code and data. If there are pages that have never been loaded to (and therefore never assigned to the file), a SINTRAN error message: NO SUCH PAGE will be returned when the program is executed.

4.1.2 ND-500 program loading

A compiled ND-500 Pascal program must be loaded by the ND-500 Linkage Loader before it can be executed. Also, the Pascal library must be loaded together with the object program. The library is found on the file PASCAL-LIB:NRF. The reader should consult the ND-500 Linkage Loader manual (ND-60.136) for details concerning the loader and the loading process.

Example:

Terminal input/output	Comments
<u>@ND-500-MONITOR LINKAGE-LOADER</u>	Call loader
ND-Linkage-Loader - x	Identifying text
NLL: <u>SET-DOM SCRATCH-DOMAIN</u>	
NLL: <u>LOAD-SEG MYPROGCODE</u>	Load code file
Program:....xxxxx P01 Data:.....xxxxx D01	
NLL: <u>EXIT</u>	
SEGMENT NO.....xx IS LINKED	Pascal library is auto-linked
<u>@ND-500-MONITOR SCRATCH-DOMAIN</u>	Execute program
@	Execution finished

4.2 Run-Time Errors

If a program attempts to do an illegal operation, the program is aborted with an appropriate error message. If the error was an illegal I/O operation, the name of the file variable involved will be part of the message. A list of all run-time error messages is found in appendix B.

The error message indicates at which absolute address (octal) the error occurred, and, if the T option was on during compilation, which line number in the source program this address corresponds to.

Be aware of the following pitfalls regarding the source program line number:

- 1) If the T option was turned off and on one or more times during the compilation, the source line number may be wrong.
- 2) If the program calls separately compiled procedures, the source line number may be that of an external procedure, if that procedure was compiled with the T option on.
- 3) If an error occurs within an external routine in another language, the Pascal system will not be able to give any information about the error.

If there is any doubt regarding the source line number given in cases 1) and 2) above, you should correlate the octal address in the error message with the octal program addresses in the listing by the help of a loader map. The loader map can be acquired by the NRL ENTRIES-DEFINED command or the Linkage Loader LIST-MAP command.

4.2.1 Trapping run-time errors

A Pascal main program may contain the declaration of a procedure

```
procedure FAULT(erno, lino, objad, status: INTEGER);  
  . . .  
begin  
  . . .  
end;
```

The effect is that when a run-time error occurs, FAULT will be called.

Note: The N option must be on in order to make FAULT have this effect.

The parameters are

erno The error number. The meaning of these is found in appendix 8.

lino The source program line number at which the error occurred.

objad The object code address at which the error occurred.

status The SINTRAN error status in case of a file system or I/O error (error numbers 17, 33, and 37).

The procedure may contain any legal Pascal code - for example, if the error is considered non-fatal, a jump to a main program label. If the procedure exits through its end, the normal error processing is done.

It is the programmer's responsibility that the declaration of FAULT follows the rules above, and that a program does not continue execution after a fatal error has occurred. In particular, be aware of the possibility that FAULT will be called recursively if an error occurs within the FAULT routine itself.

5 INPUT/OUTPUT

Input/output is that part of a programming language which is most operating system dependent. Several design and implementation decisions therefore must be taken by any implementor of Pascal. The reader is warned that some of the features described in this chapter may not be implemented, or may work differently, in other Pascal implementations.

5.1 File Variables

File types may be used as any other type in a Pascal program, with the following limitations:

- 1) file of T where T is or contains a file type is not allowed.
- 2) File variables, or structures containing file variables may not be generated with the NEW constructor. A file variable may not occur in a variant of a record.
- 3) Assignment to a file variable f (not to be confused with the file buffer f[]) is not possible, nor is the use of a file variable in an expression.

5.1.1 The type TEXT

There is a standard file type TEXT. A file of type TEXT is assumed to contain a sequential text, subdivided into lines. A line may contain any number of characters.

Note: The type TEXT is not equivalent to the type packed file of CHAR. The latter type will be interpreted as a sequence of characters where no line subdivision is visible.

The following procedures and functions may be used on files of type TEXT:

EOLN READ READLN WRITE WRITELN

On input, the CR character (value 15 octal) is taken as a line separator. An LF character (value 12 octal) following CR is ignored. According to Standard Pascal, EOLN(<file>) becomes TRUE when a READ(<file>,c) reads the last character before the CR. When EOLN(<file>) is TRUE, the next READ(<file>,c) delivers the space character (value 40 octal). On input, character parity is removed.

On output, WRITELN writes the two characters CR and LF. Characters output will have even parity.

The characters in a TEXT file are assumed to be ASCII characters with internal values in the range 0..127. When the C option is on, however, the internal values can be in the range 0..255. In this case, the parity bit is neither removed on input, nor generated on output.

The editing specifications in READ and WRITE are extended to enable I/O of non-decimal representation of integers. In READ, an integer parameter may be followed by a :<radix> specification, while in WRITE, an integer parameter may have a :<radix> specification after the :<field width> specification. In this case, the <radix>-base representation of the integer is read or written. <radix> must be in the range 2 to 36. Digits in the range 10..35 are represented by the uppercase letters A..Z.

The following table gives the number of character positions used in the output file when a value needing a minimum of p characters for its representation is written. In the table, w is the value of <field width>.

	default		0 < w < p	p <= w (1)
	N100	N500		
CHAR	1	1	-	w
BOOLEAN	6	6	w (3)	w
INTEGER, decimal	6	12	p	w
LONGINT, decimal	12	12	p	w
INTEGER, non-decimal	-	-	w (4)	w
LONGINT, non-decimal	-	-	w (4)	w
REAL, floating point	12 (5)	12	w (2)	w
REAL, fixed point	-	-	p	w
string	p	p	w (3)	w

(1) Blank fill to the left

(2) Minimum 10 for 48-bit reals, minimum 8 for 32- and 64-bit reals

(3) The initial w characters of the string
('FALSE ' and 'TRUE ' when Boolean)

(4) The w least significant digits

(5) 12 for 32-bit reals, 16 for 48-bit reals

5.1.2 Standard files

There are two standard files, INPUT and OUTPUT, both of type TEXT. These files may therefore be used without declaration.

5.1.3 Packed files

In a GET or PUT-operation, an integral number of 8-bit bytes will always be transferred. If the file is not designated packed, this number may be deduced from the table in section 2.6.1.

In the declaration

packed file of T,

the keyword packed has an effect only if the values of type T occupy no more than eight bits (N100) or 16 bits (N500). In these cases, PUT and GET will read or write the minimum number of bytes necessary to represent the value.

Since the internal representation of values may use a different number of bytes on the ND-100 and the ND-500, non-packed files may be incompatible as seen from ND-100 Pascal and ND-500 Pascal. That is, a file of T generated on one computer may not be readable on the other computer, using the same file declaration.

If a file type is designated packed, however, the external file structures assumed by ND-100 Pascal and ND-500 Pascal will usually be identical. Especially, if the type T occupies eight bits or less, then packed file of T will always correspond to the same file structure on the two computers.

5.1.4 Non-TEXT files

When f is not of type TEXT, then

READ(f,x); is equivalent to

begin x := f[; get(f) end;

WRITE(f,x); is equivalent to

begin f[:= x; put(f) end;

READ(f,x1,x2,...,xn); is equivalent to

READ(f,x1); READ(f,x2); ... READ(f,xn);

WRITE(f,x1,x2,...,xn); is equivalent to

WRITE(f,x1); WRITE(f,x2); ... WRITE(f,xn);

5.2 Association to External Files

The procedures CONNECT and DISCONNECT have been implemented in ND-Pascal to enable run-time association between a file variable and an external file.

5.2.1 CONNECT

The CONNECT procedure can have up to five parameters:

```
CONNECT(<file>,<filename>,<type>,<access>,<status>)
```

<file> is the variable name of the file.

<filename> is either an integer giving the logical unit number of an open file, or a string (or a packed array of CHAR) containing the external name of the file.

<type> is a string giving the default file type.

<access> is a string giving the file access mode (W, R, WX, RX, RW, WA, WC or RC, or the reverse of one of these strings).

<status> is an integer variable where status for the CONNECT operation is left. If the CONNECT was successful, <status> will be equal to zero; if an error occurred, <status> will be equal to the SINTRAN error number.

The <file> parameter is mandatory. One or more of the remaining parameters may be omitted, either by leaving the parameter position empty, or by prematurely closing the parameter list with the right parenthesis.

The effect of omitting one of the parameters <filename>, <type> and <access> is that Pascal will enquire the user to supply the value from the terminal. When CONNECTing a logical unit the <type> parameter may be omitted, and in that case will not be enquired for.

The effect of omitting the <status> parameter is: If the CONNECT operation failed, then write the error message to the terminal. Repeat the CONNECT operation if the file name was specified from the terminal and the job is interactive, otherwise abort the program.

Remember that RESET or REWRITE must be called before sequential I/O on the file can be performed.

Example:

```
CONNECT(infile,,'SYMB','R'); RESET(infile);
```

5.2.2 DISCONNECT

The DISCONNECT procedure has one parameter:

```
DISCONNECT(<file>)
```

The external file will be disassociated from the <file> variable. If a file name was given when <file> was opened, the external file will be closed. A <file> opened with a logical unit number will not be closed. A later CONNECT may associate <file> with another external file.

When ND-Pascal goes through a block end, all files local to that block will implicitly be DISCONNECTed.

5.2.3 Scratch files

If a REWRITE(<file>) is done on an un-CONNECTed file, Pascal will create, if necessary, a scratch file with the name <file>-cc:TEMP, and open it. cc are two characters generated to make filenames distinct.

If a scratch file is DISCONNECTed, or the program terminates normally, the file will be deleted.

5.2.4 Program heading parameters

The program heading may have file variable names as parameters. For each of these file variables the compiler automatically generates some code in the beginning of the main program:

For the file INPUT:

```
CONNECT(INPUT,0,'R'); RESET(INPUT);
```

For the file OUTPUT:

```
CONNECT(OUTPUT,1,'W'); REWRITE(OUTPUT);
```

For other file variables F:

```
CONNECT(F);
```

The effect is that for every user-defined file variable in the program heading, the user is enquired to supply the actual file name, type and access mode. The files INPUT and OUTPUT are associated with the standard input and output files, i.e. the terminal for interactive jobs, and the appropriate disk or terminal files for mode and batch jobs. Files other than INPUT and OUTPUT must be declared in the main program var section.

For all file names in the program heading, except INPUT and OUTPUT, the call on RESET or REWRITE must be programmed.

Since CONNECT and DISCONNECT are not part of Standard Pascal, file variables in programs that are to be ported should appear in the program heading, instead of being explicitly opened by calls on CONNECT.

5.3 Terminal I/O

When the actual external file is the terminal running the program, certain special actions are taken by the I/O system.

On input, a RESET will not read the first character into the file window, as specified in Standard Pascal. Instead, RESET will put the space character into the window. Thus, in the input from the terminal, an extra initial space will appear. The reason for this modification is to permit output to the terminal prior to the first input without program hang-up.

On the first file which a program CONNECTs for input from the terminal (as for instance the default connection of INPUT), EOLN will be TRUE initially if no text followed the program name in the program call command line.

An input TEXT file associated with "TERMINAL" is given logical unit number zero. This enables editing of the terminal input with CTRL A and CTRL Q. Be aware that SINTRAN converts lower case characters to upper case on input from unit zero. A file may be CONNECTed for input from logical unit one, where this conversion is not done. Editing with CTRL A and CTRL Q is not possible on unit one.

In some applications (e.g. screen-handling) it is necessary to read from the keyboard on a character-by-character basis, and with no echo. To do this from a Pascal program, use the method illustrated by the following example:

```

var keyboard: TEXT;
. . .
procedure ECHOM(mode: INTEGER); EXTERN;
procedure BRKM(mode: INTEGER); EXTERN;
. . .
begin (* Main program *)
  CONNECT(keyboard,1,, 'R'); RESET(keyboard);
  ECHOM(-1); BRKM(0);
  . . .
  repeat
    GET(keyboard);
    (* Do action on character now in keyboard *)
    (* Program must do its own echoing *)
  until . . .
. . .
end.
```

In a READ operation from the terminal, a number syntax error does not result in the program being aborted (provided the program is run interactively). Instead, the message

ILLEGAL NUMBER SYNTAX

is written to the terminal, and the READ performed anew such that the correct number can be retyped.

5.4 Random Access I/O

A file variable may be associated with an external random access file. Random access I/O may be done on that file with the procedures PUTRAND and GETRAND:

```
PUTRAND(<f>: <filetype>; <block number>: INTEGER;  
      var <status>: INTEGER);
```

```
GETRAND(<f>: <filetype>; <block number>: INTEGER;  
      var <status>: INTEGER);
```

PUTRAND writes the current content of the file window to the given <block number> on the file. GETRAND reads the block in <block number> on the file into the file window.

The <status> parameter is optional. If present, the SINTRAN status of the I/O operation is left in this variable. If not present, the program aborts in case PUTRAND or GETRAND fails.

The block size is equal to the number of bytes occupied by the file component type. This block size is determined when the file is opened by a call on CONNECT. Note that the smallest block size that SINTRAN accepts is two; therefore it is not possible to randomly access single bytes of a file.

RESET and REWRITE have no effect on random access files.

A random access file cannot be packed, but may contain packed elements.

THESE DOCUMENTS SONT
DEPOSES EN 1974

LES DOCUMENTS SONT
DEPOSES EN 1974

LES DOCUMENTS SONT
DEPOSES EN 1974

LES DOCUMENTS SONT
DEPOSES EN 1974

LES DOCUMENTS SONT
DEPOSES EN 1974

6 ND-100 REAL-TIME PROGRAMS

Any ND-Pascal program may be run as a real-time program. This requires no changes to the BRF code generated by the compiler. Thus, the same code may be used for both regular and real-time execution.

To load a program for real-time execution, enter the command

*REFER-SYMBOL 5RTPM

before the Pascal library is loaded. This will have the effect of selecting library routines adapted to real-time execution. In particular, the following effects should be noted:

1. When a run-time error occurs, the following statements will be executed:
 ERMON(50,<Pascal error number>); (* Cfr. appendix B *)
 ERMON(51,<source line number>);
 RTEXT;
2. No terminal will be connected to the program. Thus, to execute a CONNECT operation where one or more parameters are missing, unit 1 must be reserved prior to the CONNECT.

The Pascal library is not completely re-entrant. However, several real-time programs may share the same (re-entrant) segment containing external procedures and/or the Pascal library, provided the real-time programs have the same COMMON start address.

The STACK-HEAP area will by default be allocated as for background programs (cfr. section 8.1). The placement and size of this area may be determined by the user if some other allocation is desired (cfr. section 8.1).

For a real-time program, RUNMODE is equal to 3 (cfr. section 2.7.5).

FORTTRAN routines compiled with the old FORTRAN compiler in re-entrant mode may not be called from a Pascal program.

7 ND-100 OVERLAY PROGRAMS

Large program systems written in ND-Pascal may be run as a set of overlaid programs. The Pascal overlay system is adapted to the NRL overlay generation facility. The reader is referred to the NRL manual (version 6 or later) for details concerning the overlaying of programs.

7.1 Modules

A Pascal program system which is to be run in overlay mode will consist of a set of modules. A Pascal main program is the base, or root, module. All other modules will be procedures or functions. A procedure or function will become an overlay module when the key-word module precedes the procedure/function declaration.

Example: module procedure OSLO(var n: NORWEGIAN); . . .

Modules may be nested. The maximum number of overlay levels is ten.

Modules may appear either

- 1) within a main program, or
- 2) in a separately compiled file containing external modules, procedures and functions.

The modules for a program system may be generated in either way, or by using a combination of the two.

A module which calls an external, separately compiled module, must contain an external declaration of the latter module.

Example: module procedure MADRID(x,y: SPANIARD); EXTERN;

A module may not be forward declared.

A file containing module declarations may be headed by a copy of the main program const, type and var definitions. This feature allows for easy communication between modules through main program variables. In a similar manner, nested modules may be used to allow child modules to communicate through the local variables of the mother module.

If an external module, procedure or function refers to procedures or functions in the main program, the main program must be compiled with the X option on (cfr. section 8.1.2).

7.2 Compilation of Modules

The code for each module must be written on a separate BRF file. The compiler will prompt the user to specify the BRF file when a module declaration is encountered in the source file. This means that when compiling a file of modules only, no code file should be specified in the \$COMPILE command.

Example

The following example consists of a main program with modules, and one external module which the main program calls.

Main program:

```
program EXAMPLE(OUTPUT);

const SIZE = 10;

type INDEX = 1..SIZE;

var A, B, C: array [INDEX,INDEX] of REAL;
    I: INTEGER;

procedure RESULT;
var I, J: INDEX;
begin
    for I := 1 to SIZE do
        begin
            for J := 1 to SIZE do WRITE(C[I,J]:10);
            WRITELN
        end
    end (*RESULT*);

module procedure INIT;
var I, J: INDEX;
begin
    for I := 1 to SIZE do
        for J := 1 to SIZE do
            begin A[I,J] := SQR(I)*J;
                B[I,J] := LN(I)*SQR(J);
                C[I,J] := 0
            end;
        end
    end
    RESULT
end (*INIT*);

module function FACTOR(I: INTEGER): INTEGER;
begin
    if I <= 1 then FACTOR := 1
    else FACTOR := I*FACTOR(I-1)
end (*FACTOR*);

module procedure ACCUM; EXTERN;

begin (*MAIN PROGRAM*)
    . INIT;
    for I := 1 to 7 do WRITELN(FACTOR(I):10);
    ACCUM
end.
```

External module:

```

const SIZE = 10;

type INDEX = 1..SIZE;

var A, B, C: array [INDEX,INDEX] of REAL;
    I: INTEGER;

module procedure ACCUM;
var I, STATUS: INTEGER;

    procedure RESULT; EXTERN;

    procedure ROW(J: INDEX);
    var K: INDEX;
        SUM: REAL;
    begin SUM := 0.0;
        for K := 1 to SIZE do SUM := SUM+A[I,K]*B[K,J];
        if SUM > 1.0E6 then STATUS := 1;
        C[I,J] := SUM
    end (*ROW*);

    module procedure COLUMN(I: INTEGER);
    var J: INDEX;
    begin STATUS := 0;
        for J := 1 to SIZE do ROW(J)
    end (*COLUMN*);

    module procedure WRITCOL(I: INTEGER);
    var J: INDEX;
    begin
        for J := 1 to SIZE do WRITE(C[I,J]:12);
        WRITELN
    end (*WRITCOL*);

begin (*ACCUM*)
    for I := 1 to SIZE do
    begin COLUMN(I);
        if STATUS = 0 then WRITCOL(I)
        else WRITELN('COLUMN',I:3,' IN ERROR')
    end;
    RESULT
end (*ACCUM*);

```

This program contains examples of the following:

- Child modules communicate through variables of the mother module (STATUS)
- Child modules use a procedure within the mother module (ROW)

- A module may be called recursively - in this case the call is executed as a normal procedure or function call (FACTOR)
- Compilation of the example programs:

```
@PASCAL
PASCAL/ND-100 VERSION J 83-xx-xx
$OPT X+
$COMPILE EXAMPLE LINE-PRINTER "EXAMPLE"
Codefile for module INIT      : "INIT"
Codefile for module FACTOR    : "FACTOR"
```

```
NO ERRORS
LENGTH OF PROGRAM: . 000363B WORDS
LENGTH OF FIXED DATA: 002146B WORDS
  7 USES OF NON-STANDARD FEATURES
  1.34 SECONDS COMPILATION TIME
```

```
$COMPILE ACCUM LINE-PRINTER
Codefile for module ACCUM     : "ACCUM"
Codefile for module COLUMN    : "COLUMN"
Codefile for module WRITCOL   : "WRITCOL"
```

```
NO ERRORS
LENGTH OF PROGRAM: . 000403B WORDS
LENGTH OF FIXED DATA: 000010B WORDS
  7 USES OF NON-STANDARD FEATURES
  1.20 SECONDS COMPILATION TIME
```

\$EXIT

7.3 Loading Overlay Programs

When loading modules to create a system of overlaid programs, the following points must be noted:

- The user must allocate the STACK-HEAP area with the *DEFINE STACK xxxxx and *DEFINE HEAP xxxxx commands (cfr. section 8.1). It may be necessary to do a trial load of the system in order to determine the optimum setting of STACK and HEAP.
- The Pascal library must be loaded together with the main program, and with any module which refers routines in the library not referred to in the main program. To be safe, the library may be loaded with every module (only those routines not already present will actually be loaded).
- When loading two-bank code, one must enter the command

SET-MODE DATA

before the first OVERLAY-GENERATION command.

- The modules must be loaded in an order which corresponds to the overlay tree structure, that is:

1. The main program. Call this the current module.
2. The next module within the current module. This module becomes the current module. Apply rule 2 recursively.

Be aware that when specifying entry point names to the loader, NRL reads the last 7 characters, whereas Pascal uses the 7 first. Therefore, to avoid problems, never specify entry point names longer than 7 characters.

A file containing an overlay program (:PROG file) should not be renamed with the SINTRAN RENAME-FILE command, as the absolute program must contain a record of the file name where the overlay segments are found. This record is not updated with the RENAME-FILE command.

The file name is recorded exactly as specified in the DUMP command. Therefore, to avoid ambiguity with file names created at a later time, it is recommended that the file name is not abbreviated. If the user name is specified, the :PROG file cannot be copied to other users and executed. (If the receiving user has access to the original owner's file, the root segment will be taken from the receiver and the overlay segments from the original owner. This is, at best, hazardous.)

Example

Loading of the program example in section 7.2:

```
@NRL
RELOCATING LOADER LDR-1935x
*IMAGE-FILE 100
*OVERLAY-GENERATION 10
*DEFINE STACK 0
*DEFINE HEAP 150000
*DEFINE NOBKS 2
*LOAD EXAMPLE PASCAL-LIB
FREE: 012774-175473
*OVERLAY-ENTRY (1) INIT
*LOAD INIT
OVERLAY 1 LEVEL 1 COMPLETED. AREA: 012774-013115
    SLDAT=012774    INIT=012774    HEAP=150000
*OVERLAY-ENTRY (1) FACTOR
*LOAD FACTOR
OVERLAY 2 LEVEL 1 COMPLETED. AREA: 012774-013033
    FACTOR=012774
*OVERLAY-ENTRY (1) ACCUM
*LOAD ACCUM
OVERLAY 3 LEVEL 1 COMPLETED. AREA: 012774-013263
    ROWFS*=012774    ACCUM=013140    ACCUFQ&/175463
*OVERLAY-ENTRY (2) COLUMN
*LOAD COLUMN PASCAL-LIB
OVERLAY 4 LEVEL 2 COMPLETED. AREA: 013264-013320
    COLUMN=013264
*OVERLAY-ENTRY (2) WRITCOL
*LOAD WRITCOL
OVERLAY 5 LEVEL 2 COMPLETED. AREA: 013264-013341
    WRITCOL=013264
*DUMP "EXAMPLE"
*EXIT
```

7.4 Executing Overlay Programs

An overlaid program is activated by calling the root module, i.e.

@EXAMPLE

Note: If an overlaid program is interrupted by ESCAPE, it may not be continued with the @CONTINUE command.

8 IMPLEMENTATION DESCRIPTION

This chapter gives some information on how the ND-Pascal system works internally, to enable more advanced use of the system. Be aware that most of the features described in this chapter are machine and SINTRAN dependent. Therefore, the reader should not assume that other Pascal implementations work in the same or a similar manner. Also, the reader is warned that implementation details may change in future versions of ND-Pascal.

8.1 ND-100 Implementation

8.1.1 Memory layout

The following figures show how memory is utilized by a running ND-100 Pascal program (including the Pascal compiler itself).

One-bank program

Two-bank program
(one-bank library)

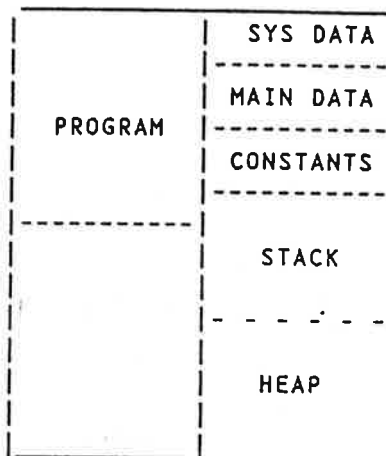
address

0

(LOADER)
PROGRAM
STACK
HEAP
CONSTANTS
MAIN DATA
SYS DATA

177777

(LOADER)	STACK
PROGRAM	
	HEAP
(constants)	CONSTANTS
(main data)	MAIN DATA
(sys data)	SYS DATA

Two-bank program
(two-bank library)address
0

177777

- PROGRAM** The Pascal program together with the necessary library routines.
- STACK** The memory used by procedures and functions that the program calls. The stack grows from low towards high addresses.
- HEAP** The memory used by data allocated with the NEW constructor. The heap grows from high towards low addresses.
- CONSTANTS** The constants referred to by procedures. For each procedure, a common block containing such data is allocated within the CONSTANTS area.
- MAIN DATA** All variables declared in the main program. This area is a common block named C.MAIN.
- SYS DATA** The variables and constants used by the Pascal library routines. This area consists of two common blocks named SCRTL and SCRTD.

8.1.1.1 One-bank programs

In a one-bank execution, Pascal places the stack and heap in the largest of the two areas

- a) address zero to first PROGRAM location
- b) last PROGRAM location to first CONSTANTS location

To make maximum space for the stack and heap, one may either do an image load, or use the NRL SET-LOAD-ADDRESS command to minimize area b).

Be aware that the area between the last PROGRAM location and the first CONSTANTS location will occupy space on the :PROG file. If the default load address is used, the size of the :PROG file will be in excess of 50 pages. To make a minimal absolute version of a program, use the SET-LOAD-ADDRESS command to minimize area b).

8.1.1.2 Two-bank programs

A two-bank program may be generated in one of two ways, as described in section 4.1.1.

Method 1

Compile the program, producing one-bank code. Before loading, enter the command

```
*DEFINE NOBKS 2
```

Then load the program together with PASCAL-LIB. The program is loaded exactly as a one-bank program. Before execution starts, the CONSTANTS, MAIN DATA, and SYS DATA areas will be copied to the data bank. The data will be located at the same addresses as they had in the instruction bank.

To make a minimal absolute version of a the program, use the SET-LOAD-ADDRESS command to minimize the space between the PROGRAM and CONSTANTS areas. The absolute program may be dumped to a :PROG file, or dumped as a re-entrant subsystem.

This method uses more space in the instruction bank than method 2, but must be used if the program is to be dumped as a re-entrant subsystem under SINTRAN version H or earlier.

Method 2

Compile the program with option B2 set, thereby producing two-bank code. Then load the program with PASCAL-2LIB. The absolute program may be dumped to a :PROG file, or dumped as a re-entrant subsystem under SINTRAN version I or later.

It is not possible to force a one-bank execution from a program compiled in two-bank mode.

8.1.1.3 Forced allocation of stack and heap

The user may determine where to allocate the stack and heap. This can be done at load-time by entering the following commands before the Pascal library is loaded:

*DEFINE STACK <value>
*DEFINE HEAP <value>

The starting addresses for the stack and heap will then be the given values. It is the user's responsibility that the definitions are consistent, and that no part of the stack-heap area overlaps the program or common areas. The result of doing one of the definitions and omitting the other is undefined.

8.1.2 Loader symbols

The compiler generates 7-letter entry point names. The names found in the loader map are constructed as follows:

Main entry point: The first 7 letters of the name given by the programmer in the PROGRAM statement.

Modules regardless of declaration level; procedures and functions on the outermost level of a separately compiled file; procedures and functions on the outermost level of a main program when the X option is on: The name given by the programmer. Note that the loader uses 7-letter names, so that these identifiers must be distinct within the 7 first letters.

Procedures and functions local to other routines or modules; all procedures and functions when the X option is off: These have the form nnnndd* where nnnn are the first four characters of the procedure or function name. dd are two characters generated to make entry point names distinct.

Non-local labels: These have the form nnnndd+ where nnnn are the first four characters of the name of the procedure or function within which the label occurs. dd are generated characters.

External procedures and functions: The name given by the programmer.

Labelled common areas: These have the form nnnndd& where nnnn are the first four characters of the name of the procedure or function with which this common area is associated. dd are generated characters.

8.1.3 Procedure and function calls

The following information on how procedure and function calls are handled by ND-Pascal should enable a user to write simple external routines in MAC or NPL.

For each procedure or function call, Pascal generates an object on top of the stack to hold system data, parameters, and data local to the routine. At the time of entry to the routine, the registers and stack contain the following data:

X Static Link
A Top of new procedure object relative to B
B Dynamic Link (calling procedure object)
L Return Address

Stack:

(A)+(B) -> system location (0)
 system location (1)
 system location (2)
 system location (3)
 system location (4)
 system location (5)
 function value
 parameter (1)
 parameter (2)
 .
 .
 parameter (n)

In a proper Pascal procedure system location (0) contains Return Address, system location (1) contains Dynamic Link, and system location (4) contains Static Link. The other system locations are not used by Pascal.

The function value occupies zero words if the object is a procedure; one, two, or three words if the object is a function.

parameter (i) can have the following form:

when <u>var</u> parameter	reference to actual
when value parameter	k-word value if k <= 8 or value is a set, otherwise reference to actual

The routine may use 200 octal stack locations without causing stack-heap overflow.

On exit from a procedure or function, the following conditions must be satisfied:

- 1) The B-register must hold the same value as it had on entry.
- 2) For a function, the A-, AD-, or TAD-register must hold the function value.
- 3) The exit must be to Return Address (= contents of L-register on entry).

Example:

The Pascal program contains

```
function mgngre(a, b: INTEGER): BOOLEAN; EXTERN;
```

This is an assembly routine which returns the value TRUE if the magnitude of a is greater than or equal to the magnitude of b.

Assembly routine:

```

)9BEG
)9LIB      MGNGRE
)9ENT      MGNGRE

FVAL=      6                % FUNCTION VALUE
AA=        7                % ARGUMENT A
AB=        10               % ARGUMENT B

MGNGRE=    *
           COPY      SA DX
           LOD      AA,X,8
           RCLR      DT      % 0 = FALSE
           SKP IF DA MLST SD
           RINC      DT      % 1 = TRUE
           COPY      ST DA
           EXIT
)9END
```

8.1.4 Interface to FORTRAN and PLANC

The routine to be called has to be defined with the body STANDARD. PLANC routines with or without INISTACK may be called. There is no check for stack overflow in the PLANC routines, therefore, HEAP data in the Pascal program may be destroyed.

FORTRAN routines with or without REENTRANT-MODE set may be called. To interface to the old FTN, use the body FORTRAN. With FTN, REENTRANT-MODE may not be used.

8.1.5 Input/Output

To save I/O execution time, ND-Pascal buffers access to sequential files. This is handled automatically by Pascal, and requires no intervention by the user. Pascal allocates n buffers of 256 words for the buffering. Up to n disk files which the program has CONNECTed for sequential I/O will then be accessed via buffers.

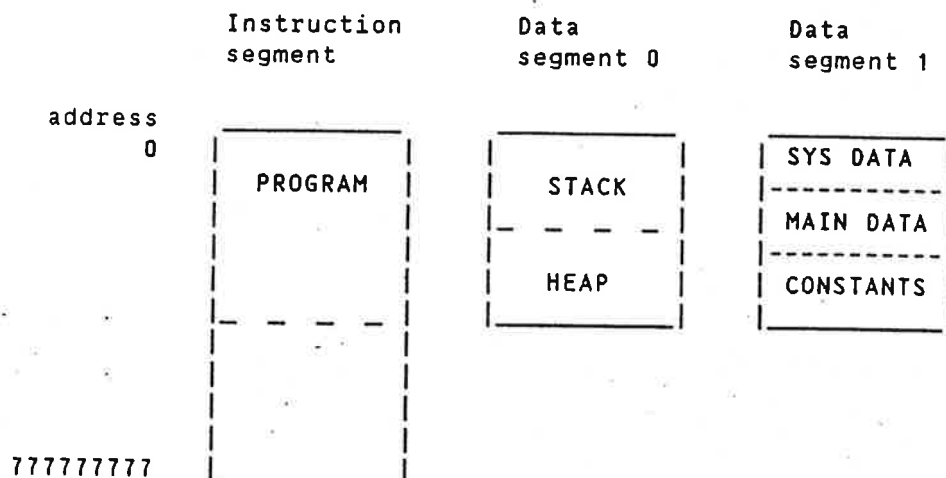
By default the number of buffers, n, is equal to three. To redefine this number, either to save space, or to simultaneously access more than three files via buffers, enter the command

*DEFINE NOBUF n

before loading the program. The maximum legal value for n is 10.

8.2 ND-500 Implementation8.2.1 Memory layout

The following figure shows how memory is utilized by a running ND-500 Pascal program (including the Pascal compiler itself).



PROGRAM The Pascal program together with the necessary library routines.

STACK The memory used by procedures and functions that the program calls. The stack grows from low towards high addresses.

HEAP The memory used by data allocated with the NEW constructor. When deallocation is done with the use of MARK and RELEASE, the heap grows from high towards low addresses. When DISPOSE is used, the HEAP area has a fixed size which may be defined at load-time (see below).

CONSTANTS The constants referred to by procedures. For each procedure, a common block containing such data is allocated within the CONSTANTS area.

MAIN DATA All variables declared in the main program.

SYS DATA The variables and constants used by the Pascal library routines. SYS DATA and MAIN DATA lie in a common block named C.MAIN.

8.2.1.1 Forced allocation of stack and heap

The default size of the STACK-HEAP area is 400000 octal (= 131,072 decimal) bytes. The area is allocated by the GSWSP monitor call. This allocation may be redefined at load-time by entering the following command before the main program is loaded:

DEFINE-ENTRY STHPSIZE <value> D

The minimum size of the STACK-HEAP area is determined by the number of I/O buffers (cfr. section 8.2.4). If the area is too small, the program will be aborted at the outset with the error message STACK-HEAP OVERFLOW.

8.2.1.2 The size of the heap

When deallocation of dynamic data is done with DISPOSE, Pascal uses the ND-500 buddy allocator. In this case the heap area has a fixed size. The default size is 200000 octal (= 65,536 decimal) bytes. (200000 octal bytes are then left for the stack.) This size may be redefined at load-time by entering the command

DEFINE-ENTRY HEAPSIZE <value> D

before the main program is loaded. Take care that a definition of HEAPSIZE is consistent with the definition of STHPSIZE.

Note: ND-Pascal does not combine non-used neighbour buddies.

8.2.2 Loader symbols

The compiler generates entry point names with maximum 10 letters. The names found in the loader map are constructed as follows:

Main entry point: The name given by the programmer in the PROGRAM statement.

Procedures and functions on the outermost level of a separately compiled file; procedures and functions on the outermost level of a main program when the X option is on: The name given by the programmer.

Procedures and functions local to other routines; all procedures and functions when the X option is off: These have the form <name>c* where <name> is the procedure or function name. c is a character generated to make entry point names distinct.

Non-local labels: These have the form <name>c+ where <name> is the name of the procedure or function within which the label is declared, and c is a generated character.

External procedures and functions: The name given by the programmer.

Labelled common areas: These have the form <name>c& where <name> is the name of the procedure or function with which this common area is associated. c is a generated character.

8.2.3 Procedure and function calls

The following information on how procedure and function calls are handled by ND-Pascal should enable a user to write simple external routines in assembly code.

When Pascal calls a procedure or function, it will first place the parameters on the stack beyond the locations needed for system information. Pascal then executes a CALL instruction to the routine. When entering the routine, the situation is as follows:

```
(B) -> PREVB
        RETA
        SP
        AUX
        NARG
        function value
        parameter (1)
        parameter (2)
        ...
        parameter (n)
```

An assembly routine with parameters therefore must be entered by an appropriate ENTS instruction.

The function value and each parameter are located at word addresses relative to B. If a value occupies less than four bytes, it will lie left justified within a word. One to three trailing bytes may be unused if the needed space (in bytes) is not a multiple of four.

The function value occupies zero bytes if the routine is a procedure, and from one to eight bytes, depending on the type of the result, if the routine is a function.

A value parameter occupies the minimum number of bytes necessary to represent values of the given type. A var parameter is a pointer to the actual parameter, and occupies 4 bytes. A procedure or function parameter occupies 3 words:

1. length of parameter area in bytes
2. address of routine
3. static link of routine

A function result must be left in the W1 or F1 (D1) register before exit from the function, which should be done with a RET instruction.

An assembly routine may use all registers except the R register, which must have the same value on exit as it had on entry.

Example:

The Pascal program contains

```
function mgngre(a, b: INTEGER): BOOLEAN; EXTERN;
```

This is an assembly routine which returns the value TRUE if the magnitude of a is greater than or equal to the magnitude of b.

Assembly routine:

```
MODULE      MAGNITUDE

EXPORT      MGNGRE
LIB         MGNGRE

STACK
FVAL:  W BLOCK 1      Z FUNCTION VALUE
AA:    W BLOCK 1      Z ARGUMENT A
AB:    W BLOCK 1      Z ARGUMENT B
ENDSTACK

ROUTINE     MGNGRE

MGNGRE:
  ENTS      #SCLC
  W1 CLR                    Z 0 = FALSE
  W COMP2 B.AA,B.AB
  IF << GO FALSE
  W SET1 R1                  Z 1 = TRUE
FALSE: RET

  ENDRoutine

ENDMODULE
```

8.2.4 Input/Output

To save I/O execution time, ND-Pascal buffers access to sequential files. This is handled automatically by Pascal, and requires no intervention by the user. Pascal allocates n buffers of 2048 bytes for the buffering. Up to n disk files which the program has CONNECTed for sequential I/O will then be accessed via buffers.

By default, the number of buffers, n, is equal to four. To redefine this number, either to save space, or to simultaneously access more than four files via buffers, enter the command

DEFINE-ENTRY NOBUF <value> 0

before loading the program. .

9 SAMPLE Pascal PROGRAM

9.1 ND-100 Sample Program

@PASCAL

PASCAL/ND-100 VERSION J 83-xx-xx
\$COMPILE PASSCAN, TERMINAL, "PASSCAN"

PASCAL/ND-100 VERSION J 83-xx-xx

```
1      PROGRAM PASSCAN (OUTPUT);
2      (* TIMES THE AVERAGE OF N X N ACCESSES *)
3      CONST MAXARRAY = 1000;
4          CHUNK      = 200;
5      VAR  X,Y,K      : INTEGER;
6          Z           : REAL;
7          STIME, ETIME : REAL;
8          TABLE      : ARRAY [1..MAXARRAY] OF REAL;
9
* 10     FUNCTION TUSED : REAL; EXTERN;
11
12     BEGIN K := CHUNK;
13         REPEAT
14             FOR X := 1 TO K DO BEGIN
15                 STIME := TUSED;
16                 FOR Y := 1 TO K DO Z := TABLE[Y];
17                 ETIME := TUSED;
18                 TABLE[X] := ETIME - STIME
19             END ;
20             Z := 0;
21             FOR X := 1 TO K DO Z := Z + TABLE[X];
22             Z := Z / K;
23             WRITELN (' AVERAGE TUSED TO ACCESS ', K,
24                     ' X ', K, ' ELEMENTS = ', Z:8:4);
25             K := K + CHUNK
26         UNTIL K > MAXARRAY
27     END.
```

NO ERRORS

LENGTH OF PROGRAM: 0002648 WORDS

LENGTH OF FIXED DATA: 0062728 WORDS

1 USES OF NON-STANDARD FEATURES

1.46 SECONDS COMPILATION TIME

\$EXIT

<continued on next page>

@NRL
RELOCATING LOADER LDR-1935x
*SET-LOAD-ADDRESS 160000
*LOAD PASSCAN PAS-LIB
FREE: 170270-171332
*DUMP "PASSCAN"
*EXIT
@PASSCAN

AVERAGE TUSED TO ACCESS	200 X	200 ELEMENTS =	0.0072
AVERAGE TUSED TO ACCESS	400 X	400 ELEMENTS =	0.0140
AVERAGE TUSED TO ACCESS	600 X	600 ELEMENTS =	0.0211
AVERAGE TUSED TO ACCESS	800 X	800 ELEMENTS =	0.0287
AVERAGE TUSED TO ACCESS	1000 X	1000 ELEMENTS =	0.0356

@

9.2 ND-500 Sample Program

@ND-500-MONITOR PASCAL

PASCAL/ND-500 VERSION J 83-xx-xx
\$COMPILE PASSCAN, TERMINAL, "PASSCAN"

PASCAL/ND-500 VERSION J 83-xx-xx

```
1      PROGRAM PASSCAN (OUTPUT);
2      (* TIMES THE AVERAGE OF N X N ACCESSES *)
3      CONST MAXARRAY = 1000;
4          CHUNK      = 200;
5      VAR  X,Y,K      : INTEGER;
6          Z           : REAL;
7          STIME, ETIME : REAL;
8          TABLE      : ARRAY [1..MAXARRAY] OF REAL;
9
* 10     FUNCTION TUSED : REAL; EXTERN;
11
12     BEGIN K := CHUNK;
13         REPEAT
14             FOR X := 1 TO K DO BEGIN
15                 STIME := TUSED;
16                 FOR Y := 1 TO K DO Z := TABLE[Y];
17                 ETIME := TUSED;
18                 TABLE[X] := ETIME - STIME
19             END ;
20             Z := 0;
21             FOR X := 1 TO K DO Z := Z + TABLE[X];
22             Z := Z / K;
23             WRITELN (' AVERAGE TUSED TO ACCESS ', K,
24                 ' X ', K, ' ELEMENTS = ', Z:8:4);
25             K := K + CHUNK
26         UNTIL K > MAXARRAY
27     END.
```

NO ERRORS

LENGTH OF PROGRAM: 00000711B BYTES

LENGTH OF FIXED DATA: 00010233B BYTES

1 USES OF NON-STANDARD FEATURES

0.60 SECONDS COMPILATION TIME

\$EXIT

<continued on next page>

@ND-500-MONITOR LINKAGE-LOADER.

ND-Linkage-Loader - x

NLL: SET-DOMAIN "PASSCAN"

NLL: LOAD-SEGMENT PASSCAN

Program:.....711 P01 Data:.....10233 D01

NLL: EXIT

SEGMENT NO.....15 IS LINKED

@ND-500-MONITOR PASSCAN

AVERAGE TUSED TO ACCESS	200 X	200 ELEMENTS =	0.0023
AVERAGE TUSED TO ACCESS	400 X	400 ELEMENTS =	0.0038
AVERAGE TUSED TO ACCESS	600 X	600 ELEMENTS =	0.0050
AVERAGE TUSED TO ACCESS	800 X	800 ELEMENTS =	0.0065
AVERAGE TUSED TO ACCESS	1000 X	1000 ELEMENTS =	0.0082

@

APPENDIX A Compile-Time Error Messages

Compile-Time Error Messages

- 1: Error in simple type
- 2: Identifier expected
- 3: 'PROGRAM' expected
- 4: ')' expected
- 5: ':' expected
- 6: Illegal symbol
- 7: Error in parameter list
- 8: 'OF' expected
- 9: '(' expected
- 10: Error in type
- 11: '[' expected
- 12: ']' expected
- 13: 'END' expected
- 14: ';' expected
- 15: Integer constant expected
- 16: '=' expected
- 17: 'BEGIN' expected
- 18: Error in declaration part
- 19: Error in field-list
- 20: ',' expected
- 21: '*' expected
- 22: Illegal character
- 50: Error in constant
- 51: ':=' expected
- 52: 'THEN' expected
- 53: 'UNTIL' expected
- 54: 'DO' expected
- 55: 'TO'/'DOWNTO' expected
- 56: 'IF' expected
- 57: 'FILE' expected
- 58: Error in factor
- 59: Error in variable
- 60: '...' expected
- 101: Identifier declared twice in same block
- 102: Lowbound exceeds highbound
- 103: Identifier is not of appropriate class
- 104: Identifier not declared
- 105: Sign not allowed here
- 106: Number expected
- 107: Incompatible subrange types
- 108: File not allowed here
- 109: Type must not be real or longint
- 110: Tagfield type must be ordinal
- 111: Incompatible with tagfield type
- 112: Index type must not be real or longint
- 113: Index type must be ordinal

- 114: Base type must not be real or longint
- 115: Base type must be ordinal.
- 116: Error in type of standard procedure parameter
- 117: Unsatisfied forward reference
- 118: Forward reference type identifier in variable declaration
- 119: Forward declared - repetition of parameter list not allowed
- 120: Function result type must be simple or pointer
- 121: File value parameter not allowed
- 122: Forward declared function - repetition of result type not allowed
- 123: Missing result type in function declaration
- 124: Second format specifier only allowed for real and integer
- 125: Error in type of standard function parameter
- 126: Number of parameters does not agree with declaration
- 127: Illegal parameter substitution
- 128: Result type of function parameter does not agree with declaration
- 129: Types of operands conflict
- 130: Expression is not of set type
- 131: Only tests on equality allowed
- 132: Strict inclusion not allowed
- 133: File comparison not allowed
- 134: Illegal type of operand(s)
- 135: Type of operand must be Boolean
- 136: Set element type must be ordinal
- 137: Set element types not compatible
- 138: Type of variable is not array
- 139: Index type is not compatible with declaration
- 140: Type of variable is not record
- 141: Type of variable must be file or pointer
- 142: Illegal parameter substitution
- 143: Illegal type of loop control variable
- 144: Illegal type of expression
- 145: Type conflict
- 146: Assignment of files not allowed
- 147: Label type incompatible with selecting expression
- 148: Subrange bounds must be ordinal
- 149: Index type must not be integer
- 150: Assignment to standard function is not allowed
- 151: Assignment to function formal parameter is not allowed
- 152: No such field in this record
- 153: Type error in read
- 154: Actual parameter must be a variable
- 155: Loop control variable must be local
- 156: Multidefined case label
- 157: Too many cases in case statement
- 158: Corresponding variant declaration is missing
- 159: Real and string tagfields not allowed
- 160: Previous declaration was not forward
- 161: Multiple forward declarations
- 162: Parameter size must be constant
- 163: Missing variant in declaration
- 164: Substitution of standard procedure/function not allowed
- 165: Multidefined label
- 166: Multideclared label
- 167: Undeclared label
- 168: Undefined label
- 169: Error in base set
- 170: Value parameter expected

- 171: Standard file redeclared
- 172: Undeclared external file
- 173: Fortran procedure or function expected
- 174: Pascal procedure or function expected
- 175: File 'INPUT' missing from program heading
- 176: File 'OUTPUT' missing from program heading
- 177: Illegal assignment to control variable
- 178: Variable used as control variable in outer loop
- 179: Read into control variable not allowed
- 180: Source line too long
- 181: Value of tagfield out of range
- 182: Illegal assignment to function name
- 183: Forward declared procedure/function not defined
- 184: Illegal jump to label
- 185: Variant already defined
- 186: Assignment of conformant array not allowed
- 187: Illegal assignment to conformant array bound
- 188: Variant selector not in range of tagfield type

- 190: Type must be ordinal or array
- 191: Value list too long

- 193: Modules cannot be forward declared

- 200: Illegal label value
- 201: Error in real constant - digit expected
- 202: String constant must not cross line boundary
- 203: Integer constant exceeds range
- 204: 8 or 9 or hex-digit in octal number
- 205: Real number overflow
- 206: Real number underflow
- 207: Too many decimal places
- 208: String of length zero not allowed
- 209: Hex-digit in decimal number

- 250: Too many nested scopes of identifiers
- 251: Too many nested procedures/functions
- 252: Too many forward references to procedure/function entries
- 253: Procedure/function too long
- 254: Too many long constants in procedure/function
- 255: Too many errors in this source line
- 256: Too many external references
- 257: Too many external files
- 258: Too many local files
- 259: Expression too complicated
- 260: Too many local variables
- 261: Too many nested scopes of overlays
- 262: No assignment to function name

- 300: Division by zero
- 301: No case provided for this value
- 302: Index expression out of bounds
- 303: Value to be assigned is out of bounds
- 304: Element expression out of range
- 305: Second operand to mod operator must be > 0

- 320: Internal error (reference out of range)

322: Internal error (GETOPR)
331: Internal error (LOADAD - packed address)
332: Internal error (LOADAD - condition address)
333: Internal error (MAKEMREG)
340: Internal error (SELECTREG)
380: Illegal compiler command
381: Unknown compiler command
382: Ambiguous compiler command
383: Too many flags
384: Too deep nesting of INCLUDE files
385: INCLUDE open error
386: Missing file name in INCLUDE
387: Code file open error
390: EOF encountered on source file
398: Implementation restriction
399: Variable-dimension arrays not implemented
400: Internal error (MOVATTR, RESETGATTRP)

APPENDIX B Run-Time Error Messages

Run-Time Error Messages

- 19 ARGUMENT TO EXP TOO BIG
The argument to EXP will cause arithmetic overflow.
- 20 ARGUMENT TO LN WAS ≤ 0
The logarithm of a negative number or zero is not defined.
- 23 ARGUMENT TO SIN OR COS TOO BIG
Lost accuracy makes the function result meaningless.
- 25 ARGUMENT TO SINH OR COSH TOO BIG
The argument will cause arithmetic overflow in the result.
- 21 ARGUMENT TO SQRT WAS < 0
The square root of a negative number is not defined.
- 7 ARITHMETIC OVERFLOW
Overflow caused by
 - a) arithmetic operations,
 - b) division by zero, or
 - c) conversion of REAL to INTEGER, or
 - d) conversion of LONGINT to INTEGER.
- 22 BAD ARGUMENT TO ARCTAN
Lost accuracy makes the function result meaningless.
- 33 BLOCK DOES NOT EXIST
Program tried to read non-existing block on a random file.
- 17 CONNECT ERROR
Failure in an attempt to CONNECT a file. The SINTRAN error message will indicate the cause.
- 12 EOF ON INPUT
Program tried to read past end-of-file on an input file.
- 15 FILE ALREADY CONNECTED
Program tried to CONNECT an already connected file.
- 16 FILE NOT CONNECTED
Program tried to access a non-connected file.
- 32 FILE NOT RANDOM
Program tried random access to a sequential file.

- 31 FILE NOT SEQUENTIAL
Program tried sequential access to a random file.
- 24 ILLEGAL ARGUMENT(S) TO POWER
Either attempt to raise negative number to a real power, or the arguments will cause arithmetic overflow.
- 38 ILLEGAL CALL OF MARK OR RELEASE
MARK or RELEASE was called from a program which also uses DISPOSE.
- 4 ILLEGAL CASE INDEX
The case label corresponding to the value of the case variable is not defined.
- 34 ILLEGAL FORTRAN CALL
A FORTRAN routine was called from a two-bank Pascal program.
- 42 ILLEGAL MOD OPERATION
Attempted mod operation with second operand zero or negative.
- 13 ILLEGAL NUMBER SYNTAX
The number being read did not have the correct syntax.
- 6 ILLEGAL PARAMETER(S) TO FORMAL PROCEDURE OR FUNCTION
The actual parameters to a formal procedure or function did not correspond in number or type to the formal parameters.
- 3 ILLEGAL SUBRANGE ASSIGNMENT
Attempted assignment of a value outside the subrange, or the controlled variable in a for-loop was of a subrange type and lower or upper bound of the loop was outside the subrange.
- 26 INTERNAL PASCAL ERROR
Error within the Pascal system. Contact a systems expert.
- 37 I/O ERROR
An I/O operation failed. The SINTRAN error message will indicate the cause.
- 40 INVALID OPERAND
Illegal argument to POWER or SQRT.
- 39 INVALID OPERATION
Error within the Pascal system. Contact a systems expert.
- 29 NO RESET
Program tried to read from a file without a previous RESET.
- 30 NO REWRITE
Program tried to write to a file without a previous REWRITE.
- 14 NUMBER TOO BIG
The number being read was too big to be represented in the computer.

- 0 POINTER IS NIL
Attempted access to data via a pointer with the value nil, or call on DISPOSE or .RELEASE with a nil-valued pointer parameter.
- 1 POINTER IS OUTSIDE HEAP
Attempted access to data via a pointer which did not point to data within the heap, or call on DISPOSE or RELEASE with a pointer parameter that did not point within the heap.
- 10 PUTRAND ON INPUT FILE
Program attempted PUTRAND on a read only file.
- 28 RESET ON OUTPUT FILE
RESET was attempted on a write only file.
- 27 REWRITE ON INPUT FILE
REWRITE was attempted on a read only file.
- 8 SET ELEMENT OUT OF RANGE
Program attempted to construct a set with an element value not within the set type.
- 5 STACK-HEAP OVERFLOW
The program generated too much data by calling procedures recursively or with the NEW constructor.
- 2 SUBSCRIPT OUT OF RANGE
The index(es) to an array are outside the array bounds.
- 43 UNAUTHORIZED USE OF PASCAL
The soft-key for Pascal has not been entered in the SINTRAN system.
- 18 UNKNOWN LOGICAL UNIT
There is no file open on this logical unit.
- 11 WRONG I/O PARAMETER
Illegal specification of the formatting of a number.
- 41 WRONG LIBRARY VERSION
Either
 - a) program was compiled with one version of the Pascal compiler and loaded with another version of the Pascal library, or
 - b) N100: floating format (32-bit or 48-bit) in program and library are not the same, or
 - c) N100: two-bank program was loaded with one-bank library or vice versa.

Index

assembly routines	56, 62.
banks	2, 9, 53.
BOOLEAN	11.
BRKM	19.
CHAR	9, 11.
character	
parity	9, 35.
set	4, 9, 10.
CLEAR	28.
CLTE	22.
COBOL	17.
code file	25.
COMMAND	21.
comments	4.
COMPILE	25.
compiler commands	6.
compiletime errors	69.
conditional compilation	6.
conformant arrays	13.
CONNECT	38.
CONTINUE	32.
COSH	18.
DATE	19.
DISCONNECT	38.
ECHOM	19.
ENDIF	6.
EOF	7.
ERMSG	19.
EXIT	28.
extensions	16.
external	
functions	56, 61, 62.
procedures	16, 18, 56, 61,
	62.
FAULT	9, 34.
file	20, 21, 35.
type	4.
FIRST	23.
flags	6.
formal procedures	15.
FORTRAN	17, 43, 58.
FREEMEM	20.
generic functions	23.
GETRAND	41.
HALT	18.
HEAP	18, 20, 53, 60.
HELP	25.
hexadecimal constants	23.
HOLD	20.
identifier	5, 9.
IFFALSE	6.
IFTRUE	6.

implementation	53, 60.
INCLUDE	7.
INPUT	36.
inputoutput	35, 40, 58, 63.
INTEGER	11.
ISIZE	20.
keyword	5.
LAST	23.
LINESPP	10, 28.
list file	25.
LMAXINT	12.
LONGINT	11.
LONGREAL	11.
LROUND	12.
LTRUNC	12.
LUNIT	20.
MARK	18.
MAXREAL	12.
MDLFI	21.
module	5, 45, 56.
multiple source	7.
NDPascal	1.
octal	
constants	23.
10	36.
options	8, 27, 28.
OSIZE	21.
OUTPUT	36.
overlay	45.
packed	
files	36.
structures	12.
PAGE	10.
PLANC	17, 58.
POWER	18.
procedure parameters	13, 15.
program	
compilation	25, 29, 46, 64, 67.
execution	19, 27, 31, 53, 60, 66, 68.
heading	39.
loading	10, 17, 31, 49, 56, 61, 66, 68.
overlay	45.
sample	64, 67.
PUTRAND	41.
RANDOM	22.
access	41.
REABT	21.
REAL	9, 11.
realtime programs	43.

RELEASE	18.
RESET	6, 28.
RMAX	22.
ROBJENT	21.
RUN	27.
RUNMODE	20.
runtime errors	9, 33, 43, 73.
scratch files	39.
segment	60.
set	
command	6, 28.
type	2.
SETBT	22.
SETE	22.
SINH	18.
SINTRAN command	21, 29.
SMAX	22.
source	
file	25.
program	4.
special symbols	4.
STACK	20, 53, 60.
Standard	17.
files	36.
functions	18, 23.
identifier	5.
Pascal	1.
procedures	18.
types	11.
strings	13.
structured types	12.
syntax errors	26, 69.
terminal	19, 40.
TEXT	11, 35, 40.
TIME	19.
traps	15, 22, 34.
TUSED	19.
value	5, 16, 28.
variable initialization	16.
VERSN	20.