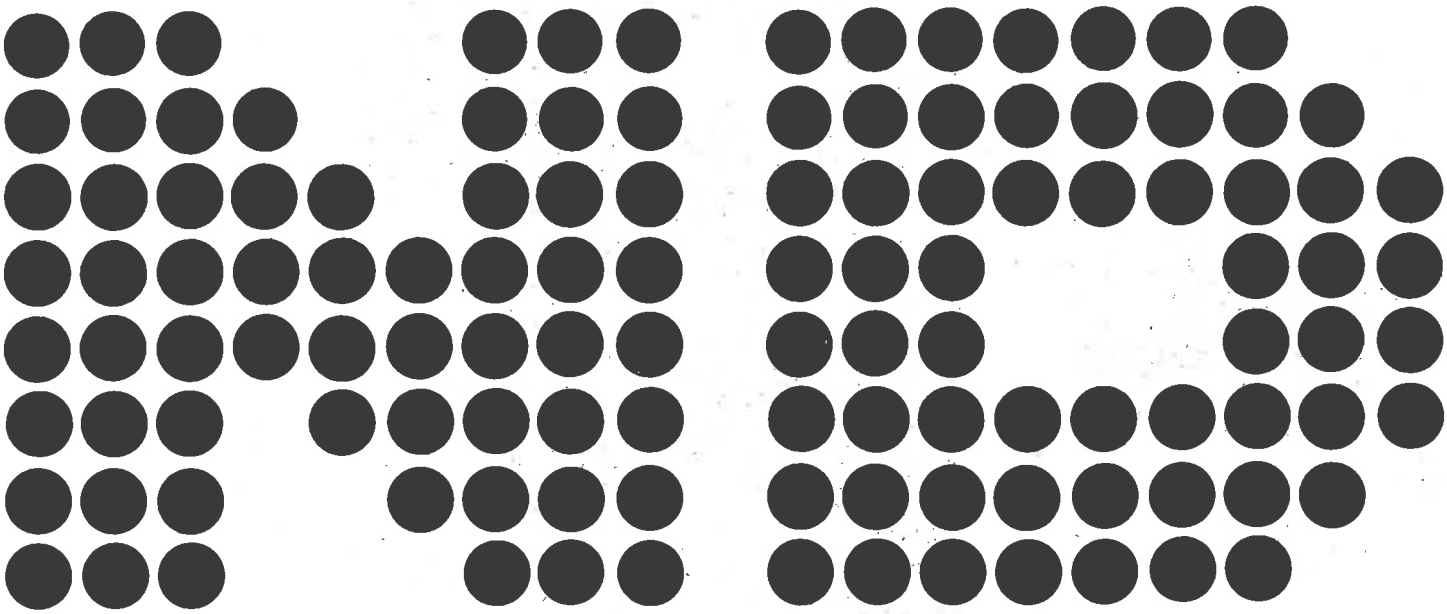
NORD-10/100
PASCAL Compiler
User's Guide

NORSK DATA A.S



NORD-10/100
PASCAL Compiler
User's Guide

NOTICE

PRINTING RECORD

Printing	Notes
12/80	Version 03

NORD-10/100 PASCAL Compiler – User's Guide
Publication No. ND-60.124.03

NORSK DATA A.S
P.O. Box 4, Lindeberg gård
Oslo 10, Norway

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

Contents

PREFACE

The product

This manual describes the NORD-10/100 Pascal compiler. The compiler is delivered in two versions, depending on the floating point hardware of the computer the compiler is run on, either 32-bit or 48-bit. The manual applies to both versions.

The reader

The reader is assumed to know the Pascal language, as this manual describes only the extensions and differences between NORD-10/100 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 Sintran-III 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. Only differences between Standard Pascal and NORD-10/100 Pascal are described. For complete examples of Pascal programs, refer to chapters 8 and 9. Compiler error messages and run time error messages are listed in Appendix A and B, respectively.

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. It is the first language which shows ability to bite into domains hitherto reserved for FORTRAN and BASIC. This ability is not only local, but is apparent on a world-wide scale.

This manual contains the information necessary to compile and execute Pascal programs on the NORD-10/100. 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.

Changes or additions relative to the previous version of this manual are indicated by a vertical bar in the margin.

The present chapter gives a general description of the NORD-10/100 Pascal system. The specific information necessary for the compilation and execution of Pascal programs is found mainly in chapters 2 to 4. Most of the chapters 5, 6 and 7 describe features for the more advanced use of NORD-10/100 Pascal.

NORD-10/100 Pascal has been implemented according to the definition in "Niklaus Wirth: The Programming Language Pascal. Revised Report. (1973)". Hereafter this language definition will be referred to as Standard Pascal.

NORD-10/100 Pascal is a superset of Standard Pascal, and has several extensions in relation to it. Especially, extensions have been introduced to make it convenient to compile and run Pascal programs in a time-sharing environment. There are also facilities for the generation and execution of real-time Pascal programs. 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 NORD-10/100 Pascal compiler was developed from the Pascal TRUNK compiler designed at ETH, Zurich. The compiler produces BRF code, which can be loaded by the Nord Relocating Loader and then executed. A program may refer to separately compiled procedures and functions written in Pascal, FORTRAN, NPL or assembly language.

1.2. The main machine dependent characteristics

A NORD-10/100 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 64K of memory. As a two-bank program, the program may occupy up to 64K in the instruction bank, and the data occupy up to 64K 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 NOBKS command.

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

The NORD-10/100 Pascal system has been constructed to run on NORD-10/100 computers with either 32-bit or 48-bit floating point arithmetic. Cross compilation is possible by using the compiler R option.

A variable of type set will by default occupy 8 words, i.e. a set can have up to 128 elements. The S option can be used to reduce the number of words occupied by set variables.

1.3. The main extensions

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

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 routines GETRAND and PUTRAND.

2. THE SOURCE PROGRAM

A Pascal source file must contain either

1) A full Pascal program, or
2) 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 will compile into an executable object program, while procedures, functions and modules will 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 are acceptable as input to the compiler.

A source input line must not exceed 100 characters. The Pascal compiler will indicate a longer line as an error.

2.1. Identifiers

An identifier may be of any length, but only the first 8 characters are significant. Within an identifier, lower and upper case letters will be treated as distinct, unless the U option is on (see section 2.4.3).

2.2. Keywords

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

Standard Pascal keywords:

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

Extra keywords in NORD-10/100 Pascal:

value module

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.

2.3. Standard identifiers

Following is a list of the standard identifiers in NORD-10/100 Pascal. A standard identifier may be thought of as having been 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	NIL	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 NORD-10/100 Pascal:

CONNECT	COSH	DISCONNECT	GETRAND
HALT	MARK	MAXREAL	POWER
PUTRAND	RELEASE	SINH	WRITEEOF

All standard identifiers are written with upper case letters.

2.4. Compiler commands

The source program text may contain commands to the compiler. A command is signalled by the character "$" in position one in 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.

2.4.1. Conditional compilation

The NORD-10/100 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 8 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.

2.4.2. Multiple source files

The $INCLUDE-command facilitates insertion in a program of source text from an alternate file. 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 4.

2.4.3. 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 Specify n-bank execution of program (n=1 or 2). Default value is n=1.

Ic Allow c as a legal character in an identifier. c must be in the set ['!','"','#','&','?','_','|','\'].

L* Generate listing. Default value is 1 (on).

M* List generated object code (MAC). Default value is 0 (off).

P* Program code dump. Default value is 0 (off). This option produces 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 Specify n-word real (n=2 or 3). Default value is 2 on NORDs with 32-bit floating point arithmetic, and 3 on NORDs with 48-bit arithmetic. A program that is to be cross-compiled, must not contain real constants.

Sn Specify n-word sets (n=1,2,...,8). All variables of type set will then occupy n words, and can have up to 16n elements. The option can only be used once in a program, and must appear before any reference to or use of set is made. n=1 will cause in-line code to be generated for most of the set operations. Default value is n=8 (up to 128 elements).

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.

Note that the NORD hardware does not facilitate checking of overflow on floating point arithmetic operations. Therefore, Pascal can only detect overflow on integer operations. As a special case, attempted floating division by zero is detected.

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. section 6.2). Default value is 1 (on).

Z* Initialize all variables 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:

(*$M+,S3,T-*) means:

M+ List object code.
S3 Sets will occupy 3 words (up to 48 elements).
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.

2.4.4. Program listing

The command

$LINESPP n

tells the Pascal compiler to print the program listing with n lines per page.

The command

$PAGE

gives new page in the program listing.

2.4.5. Special symbols

Some of the special symbols in Standard Pascal have one or more alternate representations in NORD-10/100 Pascal:

Standard Pascal	NORD-10/100 Pascal
{	{ or (*
}	} or *)
[	[or (.
]	] or .)
↑	↑ or @
and	and or &
not	not or ~

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

2.5. Extensions in NORD-10/100 Pascal

This section describes most extensions in NORD-10/100 Pascal. Refer to chapter 5 for I/O extensions. Real-time programs and overlay programs are described in chapters 7 and 8, respectively.

2.5.1. Variable initialization

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

Records, sets and pointers may not be initialized.

The syntax for initialization is:

```
<variableinit>::=       value {<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 array of CHAR, a string constant must be enclosed in parentheses as shown in the last example.

2.5.2. Standard procedures and functions

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 an integer, x↑y is (in principle) calculated by repeated multiplication. 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 a string parameter. HALT will write this string on the terminal and abort 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>) will assign the address of the current heap top to <ptr>. The call RELEASE(<ptr>) will deallocate everything on the heap which is beyond the value of <ptr>.

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

2.5.3. 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 3 with echomode in the A register. This will define the echo mode for the terminal as specified in the Sintran manual.

BRKM

External declaration:

```
procedure BRKM(breakmode: INTEGER); extern;
```

Executes MON 4 with breakmode in the A register. This will define the break mode for the terminal as specified in the Sintran manual.

ERMSG

External declaration:

```
procedure ERMSG(errorno: INTEGER); extern;
```

Executes MON 64 with errorno in the A register. 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.

RANDOM

External declaration:

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

This function gives a uniformly distributed pseudo random number in the interval <0,1>. Each new value is calculated from the value of the parameter. This 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.

NOBANKS

External declaration:

```
function NOBANKS: INTEGER; extern;
```

Gives the number of banks (1 or 2) used by the running program.

RUNMODE

External declaration:

```
function RUNMODE: INTEGER; extern;
```

Gives the execution mode of the running program:

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

LUNIT

External declaration:

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

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

2.5.4. External Pascal routines

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

The generated BRF file may be loaded with any Pascal main program which contains extern declarations of one or more of the Pascal routines. Only those routines which are actually referred, are loaded (each external Pascal routine contains a LIBR <entrypoint> loader directive).

External routines may use extern 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 argument list of the extern declaration and of the separately compiled procedure.

A file of Pascal routines may be headed by constant, type and var definitions. The var 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 Pascal does not check that the definitions are consistent with

corresponding definitions in the main program. It is therefore strongly recommended to use the $INCLUDE facility to incorporate global definitions in an external program module.

2.5.5. External FORTRAN routines

Separately compiled FORTRAN subroutines may be called from a Pascal program. A FORTRAN routine must be declared in the Pascal program with a procedure or function heading, and a body consisting of the word "FORTRAN". Example:

```
procedure ROUTINE(var x, y: REAL); FORTRAN;
```

Parameters of any type and kind, except Pascal procedure or function names, may be transmitted to the FORTRAN routine; however, no check is made that the parameters are consistent with the formal parameters of the FORTRAN routine. Parameters which are specified as var, or which occupy more than 8 words, are transmitted by reference. Value parameters occupying 8 words or less are transmitted by value.

FORTRAN routines may only be called from one-bank Pascal programs.

When loading modules for a mix of Pascal and FORTRAN programs, the following order must be observed:

1) Pascal main program
2) Pascal and FORTRAN external routines
3) FORTRAN library
4) Pascal library

2.5.6. 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.

2.5.7. Miscellaneous extensions

The compiler accepts octal constants. The syntax for an octal constant is

{d}*dB

where d is an octal digit.

MAXREAL is a standard real constant with a value equal to the largest possible floating point value (approximately 10↑4930 and 10↑76 for 48- and 32-bit floating point numbers, respectively).

2.6. Implementation dependent features

2.6.1. Structured types

Variables of structured types (records and arrays) may be assigned to and compared, provided the variable type is not packed or contain packed variables. Variables of type packed array [...] of CHAR are excepted from this restriction.

2.6.2. 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 word. No single variable will cross word boundaries. Also, a record or an array will always start at a new word boundary.

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

A variable within a packed structure cannot be used as a var parameter to a procedure.

See chapter 5 for information on packed files.

2.6.3. Strings and character arrays

In Standard Pascal, a string constant with n characters is automatically given 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 array [...] of CHAR where the lower bound is different from 1. In NORD-10/100 Pascal such assignment or substitution

will be legal provided the length of the string is equal to the length of the array.

2.6.4. Formal procedures

A formal procedure may only have value parameters. On entry to a formal procedure, the actual parameters are checked 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.

3. PROGRAM COMPILATION

The Pascal compiler is invoked by the command

```
@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 "$".

The available commands are:

```
HELP
COMPILE
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 2.4).

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 orders 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 parameter list may be omitted, in which case the command processor will ask the user to specify the files one by one.

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.

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

The listing contains:

in column 1: Source line number (decimal).

in column 2: Relative program and variable addresses (octal).

in column 3: 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 4: The source program.

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 will indicate 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 will be indicated by a line containing the text **SKIP* at the left, and hyphens under the skipped text. Lines containing syntax errors will in addition be written on 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 can be found in appendix A.

<code file> is the file on which the BRF output will be written. The <code file> parameter may be omitted, in which case no object code will be generated.

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

3.4. OPTIONS

The OPTIONS command is used to set compiler options. The command and the options are described in section 2.4.3.

3.5. 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 2.4.1.

3.6. EXIT

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

3.7. LINESPP

The LINESPP command is described in section 2.4.4.

3.8. VALUE

The command

```
$VALUE OPTIONS
```

lists the current value of all options.

The command

```
$VALUE FLAGS
```

lists the current value of all flags.

3.9. Program compilation example

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

```
Terminal input/output                    Comments

@PASCAL                                  Call Pascal compiler
PASCAL/NORD-10/100 VERSION F 80-11-04     Identifying text
$OPTION B2,T-                            Compile for 2-bank execution and
                                         suppress generation of test
                                         instructions.
$SET PARIS                               Generate "PARIS" version of
                                         program. (Assumes source file
                                         contains $IFTRUE and $IFFALSE
                                         tests on flag with name PARIS.)
$COMPILE                                 Compile
Source file=MYPROG                       Source is MYPROG
List file=LINE-PRINTER                   Listing to line printer
Code file=MYPROGCODE                     BRF code goes to MYPROGCODE
  NO ERRORS                              Message from compiler
  24.32 SECONDS COMPILATION TIME
$EXIT                                    Exit
@                                        Control to SINTRAN
```

4. PROGRAM LOADING AND EXECUTION

4.1. Program loading

A compiled NORD-10/100 Pascal program must be loaded by the NRL loader before it can be executed. The reader should consult the NRL manual for details concerning the loader and the loading process. Here we will just give an example of how a Pascal program is loaded and executed:

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

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. FINLIBR) must be loaded between the main program and the Pascal library.

The NRL command PROGRAM-FILE should be used with great care due to limitiations 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.

Further information on how a running Pascal program utilizes memory, and how to make an absolute program, can be found in chapter 6.

4.2. Run-time errors

If a program attempts to do an illegal operation, the program will abort 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 can be found in appendix B.

The error message will indicate 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 FORTRAN subroutine or function, 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 progam addresses in the listing by the help of a loader map. The loader map can be acquired by the NRL *ENTRIES-DEFINED command.

If the program aborts with the error message STACK-HEAP OVERFLOW, then your program needs more space for data. If the program was compiled with the B1 option, you may reload and run the program in two-bank mode (cfr. section 6.1).

A Pascal main program may contain the declaration of a procedure

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

The effect is that when a run-time error occurs, FAULT will be called. The error number, and source line number and object code address of the error are the actual parameters. 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 will be done.

The error numbers are found in appendix B.

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.

5. INPUT/OUTPUT

Input/output is that part of a programming language which is most operating system dependent. Several design and implementation decisions therefore have to 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 . . . file of . . . is not allowed.

2) File variables, or structures containing file variables may not be generated with the NEW constructor.

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 of maximum 136 characters each.

Note: In NORD-10/100 Pascal, 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) will be taken as a line separator. An LF character (value 12 octal) following CR will be ignored. According to Standard Pascal, EOLN(<file>) will become TRUE when a READ(<file>,c) reads the last character before the CR. When EOLN(<file>) is TRUE, the next READ(<file>,c) will deliver the space character (value 40 octal).

On output, WRITELN will write the two characters CR and LF.

The editing specifications in READ and WRITE are extended to enable I/O of the octal representation of integers. In READ, an integer parameter may be followed by a :n specification, while in WRITE, an integer parameter may have a :n specification after the :<field width>

specification. In both cases, if n has the value 8, the octal representation of the integer will be read or written. If n is not equal to 8, decimal conversion will be performed.

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>. A <field width> of value zero gives the default field.

	w = 0	0 < w < p	p <= w (1)
integer	6	p	w
real	16 (2)	16(2),p(3)	w
Boolean	5	w (4)	w
character	1	w	w
string	p	w (4)	w

(1) Blank fill to the left
(2) Floating point representation
(3) Fixed point representation
(4) The initial w characters of the string ('FALSE' and 'TRUE' when Boolean)

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 on a non-packed file, a whole number of 16-bit words will always be transferred.

In the declaration

packed file of T,

the key-word packed will have an effect only if values of type T occupy 8 bits or less. In these cases, PUT and GET will operate as follows:

If the values of type T occupy 6,7 or 8 bits:
Transfer one value.

If the values of type T occupy 1, 2, 3, 4 or 5 bits:
Pack (unpack) the maximum number of values in one 16-bit word.
Transfer a word when it is full (PUT) or empty (GET).

Be aware that on reading a file of this kind, it may be the case that EOF is found too late, if the last word was not completely filled when the file was written.

5.1.4. Non-TEXT files

When f is not of type TEXT,

```
READ(f,x)
```

is equivalent to

```
begin x := f↑; get(f) end
```

and

```
WRITE(f,x)
```

is equivalent to

```
begin f↑ := x; put(f) end
```

5.2. Association to external files

The procedures CONNECT and DISCONNECT have been implemented in NORD-10/100 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 5 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 an 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).

<status> is an integer variable where status for the CONNECT operation will be 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.

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, otherwise abort the program.

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

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.

5.2.3. Program heading parameters

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

For the file INPUT:

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

For the file OUTPUT:

```
CONNECT(OUTPUT,1,'SYMB','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 will be 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.

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, and set EOLN to TRUE. 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.

In a READ operation from the terminal, a number syntax error will not result in a program abortion. Instead, the message

```
ILLEGAL NUMBER SYNTAX
```

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

An input TEXT file associated with the terminal will be given logical unit number zero. This enables editing of the terminal input with CTRL A and CTRL Q.

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. Each of these procedures has two parameters:

<file> and <block number>

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 block size is equal to the number of words occupied by the file component type. This block size is determined when the file is opened by a call on CONNECT.

RESET and REWRITE have no effect on random access files.

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

5.5. WRITEEOF

WRITEEOF takes a file variable as a parameter. The procedure will write an end-of-file mark on the file, provided this operation is meaningful for the kind of medium on which the file resides.

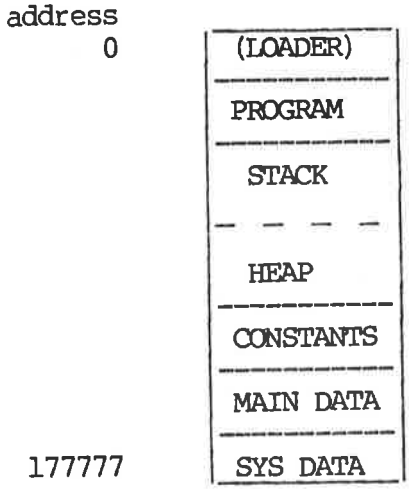
6. IMPLEMENTATION DESCRIPTION

This chapter will give some information on how the NORD-10/100 Pascal system works internally to enable more advanced use of the system. Be aware that most of the features described in this chapter are very NORD-10/100 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 NORD-10/100 Pascal.

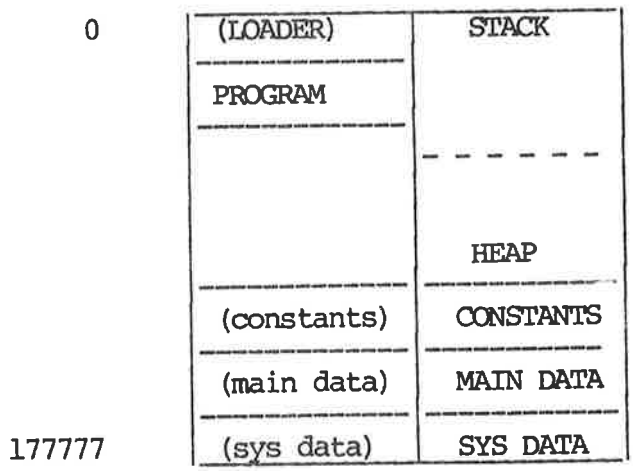
6.1. Memory layout

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

One-bank program



Two-bank program



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 5CRTL and 5CRTD.

The object program and Pascal library are identical in the one- and two-bank versions. When running, the system detects the actual execution mode by sensing bit zero in the STATUS register.

The decision whether to run a program in one-bank or two-bank mode may be postponed till the time when the program is to be loaded. Before loading, enter the command

```
*DEFINE NOBKS n
```

where n is 1 or 2. This will result in one- or two-bank execution respectively. A definition of NOBKS takes precedence over the compile-time B option.

One-bank programs

In a one-bank execution, Pascal will place 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 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).

Two-bank programs

A two-bank program is loaded exactly as a one-bank program. Before execution starts, the CONSTANTS, MAIN DATA, and SYS DATA areas are moved 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 two-bank program, use the SET-LOAD-ADDRESS command to minimize the space between the PROGRAM and CONSTANTS areas.

A two-bank program will usually be slower than a one-bank program due to the necessary ALTON and ALTOFF monitor calls within the Pascal library.

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 area. The result of doing one of the definitions and omitting the other is undefined.

6.2. 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 main program or a separately compiled file: The name given by the programmer. Note that the loader uses 7-letter names, so that these identifiers ought to be distinct within the 7 first letters. The compiler can be ordered to make the procedure and function names anonymous by turning the X option off.

Procedures/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 invented characters, to make entry point names distinct.

Non-local labels: These have the form LABLdd+ where dd are invented 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 procedure or function with which this common area is associated. dd are invented characters.

6.3. Procedure and function calls

The following information on how procedure and function calls are handled by 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 loc
             system loc
             system loc
             function value
             parameter(1)
             parameter(2)
             . . .
             parameter(n)
```

In a proper Pascal procedure, the three system locations are used to contain Static Link, Dynamic Link, and Return Address.

The function value occupies 0 words if the object is a procedure; 1, 2, or 3 words if the object is a function.

parameter(i) can have the following form:

when var parameter	reference to actual
when value parameter	k-word value if k<=8
	reference to actual if k>8

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).

6.4. Input/Output

To save I/O execution time, the Pascal system 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. The first n disk files which the program CONNECTs for sequential I/O will then be accessed via buffers.

By default the number of buffers, n, is equal to 3. To redefine this number, either to save space, or to access more than 3 files via buffers, enter the command

```
*DEFINE NOBUF n
```

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

7. REAL-TIME PROGRAMS

Any 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 6.1). The placement and size of this area may be determined by the user if some other allocation is desired (cfr. section 6.1).

For a real-time program, RUNMODE is equal to 3 (cfr. section 2.5.3).

In case the real-time program does not access files, space may be saved by entering the command

```
*DEFINE-SYMBOL NOBUF 0
```

before loading the program (cfr. section 6.4).

Real-time FORTRAN routines may not be called from a Pascal program.

8. OVERLAY PROGRAMS

Large program systems written in 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 G) for details concerning the overlaying of programs.

8.1. 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 GARP(var w: world); . . .

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 extern 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.

8.2. 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);
VAR A,B,C: ARRAY [1..10,1..10] OF REAL;
    I: INTEGER;

  PROCEDURE RESULT;
  VAR I,J: INTEGER;
  BEGIN
    FOR I := 1 TO 10 DO
    BEGIN
      FOR J := 1 TO 10 DO WRITE(C[I,J]:7);
      WRITELN
    END
  END (*RESULT*);

  MODULE PROCEDURE INIT;
  VAR I,J: INTEGER;
  BEGIN
    FOR I := 1 TO 10 DO
    BEGIN A[I,J] := SQR(I)*J;
      B[I,J] := LN(I)*SQR(J)
    END;
    RESULT
  END (*INIT*);

  MODULE FUNCTION FACTORIAL(I: INTEGER): INTEGER;
  BEGIN
    IF I <= 1 THEN FACTORIAL := 1
    ELSE FACTORIAL := I*FACTORIAL(I-1)
  END (*FACTORIAL*);

  MODULE PROCEDURE ACCUM; EXTERN;

BEGIN (*MAIN PROGRAM*)
  INIT;
  FOR I := 1 TO 10 DO WRITELN(FACTORIAL(I):10)
  ACCUM
END.
```

External module:

```
VAR A,B,C: ARRAY [1..10,1..10] OF REAL;
    I: INTEGER;

MODULE PROCEDURE ACCUM;
VAR I, STATUS: INTEGER;

  PROCEDURE RESULT; EXTERN;

  PROCEDURE ROW(J: INTEGER);
  VAR K: INTEGER;
      SUM: REAL;
  BEGIN SUM := 0.0;
    FOR K := 1 TO 10 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: INTEGER;
  BEGIN STATUS := 0;
    FOR J := 1 TO 10 DO ROW(J)
  END (*COLUMN*);

  MODULE PROCEDURE WRITCOL(I: INTEGER);
  VAR J: INTEGER;
  BEGIN
    FOR J := 1 TO 10 DO WRITE(C[I,J]:12);
    WRITELN
  END (*WRITCOL*);

BEGIN (*ACCUM*)
  FOR I := 1 TO 10 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 such a case the call is executed as a normal procedure function call (FACTORIAL)

Compilation of the example programs:

```
@PASCAL
PASCAL/NORD-10/100 VERSION F 80-11-04
$COMPILE EXAMPLE LINE-PRINTER "EXAMPLE"
Codefile for module INIT     : "INIT"
Codefile for module FACTORIAL : "FACTORIAL"
NO ERRORS
     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
     1.20 SECONDS COMPILATION TIME
$EXIT
```

8.3. 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 6.1). 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).

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

A file containing an overlay program (:PROG file) should not be renamed with the Sintran-III 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 8.2:

```
@NRL
RELOCATING LOADER LDR-1935G
*IMAGE-FILE 100
*OVERLAY-GENERATION 10
*DEFINE STACK 0
*DEFINE HEAP 150000
*DEFINE NOBKS 2
*LOAD EXAMPLE PASCAL-LIB
FREE: 007534-174625
*OVERLAY-ENTRY (1) INIT
*LOAD INIT
OVERLAY 1 LEVEL  1 COMPLETED. AREA: 007534-007655
  5LDAT=007534      INIT=007534      HEAP=150000
*OVERLAY-ENTRY (1) FACTORI
*LOAD FACTORI
OVERLAY 2 LEVEL  1 COMPLETED. AREA: 007534-007573
FACTORI=007534
*OVERLAY-ENTRY (1) ACCUM
*LOAD ACCUM
OVERLAY 3 LEVEL  1 COMPLETED. AREA: 007534-010022
ROW FS*=007534      ACCUM=007700   ACCUFQ&/174615
*OVERLAY-ENTRY (2) COLUMN
*LOAD COLUMN
OVERLAY 4 LEVEL  2 COMPLETED. AREA: 010023-010057
 COLUMN=010023
*OVERLAY-ENTRY (2) WRITCOL
*LOAD WRITCOL
OVERLAY 5 LEVEL  2 COMPLETED. AREA: 010023-010100
WRITCOL=010023
*DUMP "EXAMPLE"
*EXIT
```

9. SAMPLE Pascal PROGRAM

```
@PASCAL
PASCAL/NORD-10/100 VERSION F 80-11-21
$COMPILE PASSCAN, TERMINAL, "PASSCAN"

 PASCAL/NORD-10/100 VERSION F 80-11-21                          80-12-03

 1  000000          PROGRAM PASSCAN (OUTPUT);
 2  000236          (* TIMES THE AVERAGE OF N X N ACCESSES *)
 3  000236          CONST  MAXARRAY = 1000;
 4  000236                 CHUNK    = 200;
 5  000236          VAR    X,Y,K : INTEGER;
 6  000241                 Z      : REAL;
 7  000244                 STIME  : REAL;
 8  000247                 ETIME  : REAL;
 9  000252                 TABLE : ARRAY [ 1 .. MAXARRAY ] OF REAL;
10  006142
11  006142  L1     FUNCTION TUSED : REAL; EXTERN;
12  177606
13  000000  B1     BEGIN
14  000000             K := CHUNK;
15  000010  R2         REPEAT
16  000010  B3            FOR X := 1 TO K DO BEGIN
17  000020                       STIME := TUSED;
18  000025                       FOR Y := 1 TO K DO Z := TABLE[ Y ];
19  000051                       ETIME := TUSED;
20  000056                       TABLE [ X ] := ETIME - STIME
21  000066  E3               END ;
22  000076                   Z := 0;
23  000103                   FOR X :=1 TO K DO  Z := Z +  TABLE[ X ];
24  000152                   Z :=  Z / K;
25  000163                   WRITELN (' AVERAGE TUSED TO ACCESS ', K ,
26  000177                            ' X ', K ,' ELEMENTS =',Z:8:4);
27  000223            K := K + CHUNK;
28  000231  U2        UNTIL K > MAXARRAY;
29  000237  E1     END.

 NO ERRORS
     1.46 SECONDS COMPILATION TIME
$EXIT
@NRL
RELOCATING LOADER LDR-1935G
*LOAD PASSCAN PAS-LIB
FREE: 030146-170501
*RUN

 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
```

APPENDIX A 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 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
101: Identifier declared twice
102: Low bound exceeds high bound
103: Identifier is not of appropriate class
104: Identifier not declared
105: Sign not allowed
106: Number expected
107: Incompatible subrange types
108: File not allowed here
109: Type must not be real
110: Tagfield type must be scalar or subrange
111: Incompatible with tagfield type
112: Index type must not be real
113: Index type must be scalar or subrange
114: Base type must not be real
115: Base type must be scalar or subrange
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 scalar, subrange or pointer
121: File value parameter not allowed
122: Forward declared function; repetition of result type not allowed
123: Misssing result type in function declaration
124: F-format for real only
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 parameter function does not agree with declaration
129: Type conflict of operands
130: Exepression is not of set type
131: Tests on equality allowed only
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 scalar or subrange
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 scalar
149: Index type must not be integer
150: Assignment to standard function is not allowed
151: Assignment to formal function is not allowed
152: No such field in this record
153: Type error in read
154: Actual parameter must be a variable
155: Control variable must not be formal or global
156: Multidefined case label
157: Too many cases in case statement
158: Missing corresponding variant declaration
159: Real or string tagfields not allowed
160: Previous declaration was not forward
161: Again forward declared
162: Parameter size must be constant
163: Missing variant in declaration
164: Substitution of standard proc/func 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 was redeclared
172: Undeclared external file

173: Fortran procedure or function expected
174: Pascal procedure or function expected
175: Missing file 'INPUT' in program heading
176: Missing file 'OUTPUT' in 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 not defined
184: Illegal jump to label
185: Variant already defined
190: Type must be scalar, subrange or array
191: Value list too long
193: Modules can not be forward declared
201: Error in real constant: digit expected
202: String constant must not exceed source line
203: Integer constant exceeds range
204: 8 or 9 in octal number
205: Real number overflow
206: Real number underflow
207: Too many decimals
208: String constant of zero length not allowed
250: Too many nested scopes of identifiers
251: Too many nested procedures and/or functions
252: Too many forward references of procedure entries
253: Procedure/function too long
254: Procedure/function has too many long constants
255: Too many errors on this source line
256: Too many external references
257: Too many externals
258: Too many local files
259: Expression too complicated
260: Procedure/function has too many local variables
261: Too many nested scopes of overlays
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
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 command
381: Unknown command
382: Ambiguous command
383: Too many flags
384: Too deep nesting of INCLUDE files
385: INCLUDE open error
386: Missing file name in INCLUDE
387: Codefile open error
390: EOF encountered on source file

398: Implementation restriction
399: Variable dimension arrays not implemented
400: Internal error (MOAVATTR, 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 is not defined.

23 ARGUMENT TO SIN/COS TOO BIG
Lost accuracy makes the function result meaningless.

25 ARGUMENT TO SINH/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) integer arithmetic operations,
b) floating division by zero, or
c) conversion of real 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 ON MARK/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.

13 ILLEGAL NUMBER SYNTAX
The number being read did not have the correct syntax.

6 ILLEGAL PARAMETERS TO FORMAL PROC/FUNC
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.

9 INPUT RECORD TOO LONG
A TEXT file record must not exceed 135 characters.

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.

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.

0 POINTER IS NIL
Attempted access to data via a pointer with the value NIL, or call on 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.

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 OUTSIDE 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. Running the program in two banks (see section 1.2) may solve the problem.

2 SUBSCRIPT OUT OF RANGE
The index(es) to an array are outside the array bounds.

18 UNKNOWN LUN
There is no file open on this logical unit.

11 WRONG I/O PARAMETER
Illegal specification of the formatting of a number.

Index

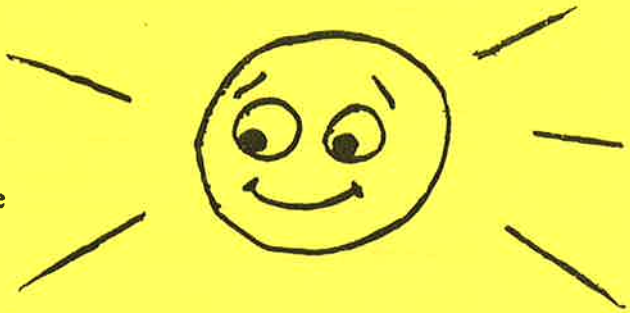
* * * * * * * * * * SEND US YOUR COMMENTS!!! * * * * * * * * * *

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

Please let us know if you

* find errors
* cannot understand information
* cannot find information
* find needless information

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



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

Manual name: Nord-10/100 PASCAL COMPILER Manual number: ND-60.124.03

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

Do you have suggestions for improving this manual? _____

Your name: _____ Date: _____
Company: _____ Position: _____
Address: _____

What are you using this manual for? _____

Send to: Norsk Data A.S.
Documentation Department
P.O. Box 4, Lindeberg Gård
Oslo 10, Norway

Norsk Data's answer will be found on reverse side

Answer from Norsk Data _____

Answered by _____ _____ _____ _____ Date _____

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