

NORD-100 SINTRAN III
SIMULA
Reference Manual

NORSK DATA A.S



NORD-100 SINTRAN III
SIMULA
Reference Manual

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.

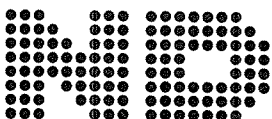
The information described in this document is protected by copyright. It may not be photocopied, reproduced or translated without the prior consent of Norsk Data A.S.

Copyright © 1980 by Norsk Data A.S.

PRINTING RECORD

[illegible]

NORD-100 SINTRAN III SIMULA - Reference Manual
Publ. No. ND-60.092.03



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

NOTICE

The information in this document is subject to change without notice. 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.

Copyright 1980 G. P. Philippot.

All rights reserved. Permission to create and distribute additional copies of this issue is granted on the condition that each copy is a complete reproduction of the manual, including this page.

Norsk Data A.S.
Postboks 4 Lindeberg gård
OSLO 10
Norway

The word SIMULA is a registered trade mark of the Norwegian Computing Center.

PREFACE

THE PRODUCT

This manual describes version 4.017 of the TPH SIMULA system for the NORD-100 computers, the October 1980 release.

The implementation of TPH SIMULA has been performed by Mr. G. P. Philippot of TPH Data A.S. All marketing rights for the NORD-100 have been transferred to Norsk Data A.S. in agreement with TPH Data A.S.

All questions and responses concerning the NORD-100 SIMULA language should be directed to Norsk Data's Marketing or Customer Support departments.

THE READER

This manual is intended for readers who need a description of the implementation of the SIMULA language for the NORD-100 computers.

PREREQUISITE KNOWLEDGE

A thorough knowledge of the SIMULA language is assumed, and a certain experience with the Sintran-III operating system is required to run a SIMULA program.

THE MANUAL

This manual is a reference manual describing differences between the TPH SIMULA implementation and the SIMULA Common Base Language, the extensions and restrictions. It also describes the use of the compiler and the external program representation. The manual is intended to be used as a reference text for the NORD-100 implementation, and each chapter is independent of the others.

RELATED MANUALS

SINTRAN III TIME-SHARING/BATCH GUIDE	ND-60.132
SINTRAN III REFERENCE MANUAL	ND-60.128

Contents

1. Introduction	7
1.1. Nord-100 version	7
1.2. Compiler description	8
1.2.1. Extensions	8
1.2.2. Restrictions	9
1.3. Operating environment	9
2. Precompiler	11
2.1. Flags	11
2.2. Options	11
2.3. Conditional compilation	12
2.4. End of file	13
2.5. Alternate source files	13
3. Source program	14
3.1. Delimiters	14
3.2. Identifiers	15
3.3. Constants	16
3.3.1. Numeric constants	16
3.3.2. Text constants	16
4. Procedures and classes	18
4.1. External quantities	18
4.2. Entrypoint quantities	19
4.3. Virtual procedures	20

5. Standard classes	21
5.1. Input/output	21
5.2. SIMSET	24
5.3. SIMULATION	24
6. Standard procedures	25
6.1. Quasi parallel sequencing	25
6.2. Arithmetic and conversion	26
6.3. Random draws	26
6.4. Editing and de-editing	27
6.5. Other text operations	27
7. Compilation and execution of programs	28
7.1. Elementary compile and execute procedure	28
7.2. Saving the binary code	29
7.3. Advanced compiler use	29
7.3.1. Terminal command language	29
7.3.2. Compiler option set	31
7.3.3. Compilation errors	32
7.4. Run time system	32
7.4.1. Debugging command language	32
7.4.2. Run time errors	34
7.4.3. Run time system options set	34
8. References	35
Appendix A: Example on use	36
Appendix B: Summary on standard identifiers	39

Appendix C: Compiler error messages	42
Appendix D: Run time error messages	45
Appendix E: Extended language	47
Appendix F: Non-Simula procedures	49
Appendix G: External procedure library	54

1. Introduction

The SIMULA language, developed by the Norwegian Computing Center (NCC), has achieved great acceptance in universities and computer teaching institutes for the exceptionally good program and data structuring capabilities, making it especially suitable for teaching programming techniques. The list processing and simulation capabilities offered by the SIMSET and SIMULATION system classes make SIMULA a programming language with capabilities far beyond those normally found with FORTRAN, COBOL, BASIC, etc.

This implementation of the SIMULA language makes, for the first time, these programming capabilities available within low cost minicomputer environments.

TPH SIMULA is an implementation of SIMULA, a general purpose language defined in 1967. SIMULA is defined in "SIMULA Common Base Language" (Ref. (1)), later referred to as the Common Base. As no attempt is to be made in the present manual to teach SIMULA, the reader is assumed to possess a thorough knowledge of SIMULA from the Common Base or other sources, e.g. (2).

This implementation of 1977 is basically machine independent. For practical purposes however, the manual is written for NORD-100 users.

1.1. Nord-100 version

The version described here compiles and executes programs on the NORD-100 computer. The compiler occupies 31K words of program plus dynamically allocated data, minimum 2K. The program consists of several independent phases, varying from 5 to 15K words, each performing different compilation tasks.

The compiler executes as a reentrant subsystem under the SINTRAN III/VS operating system, whereby many active users share common code. Utilizing the virtual storage concept of the NORD-100 SINTRAN III/VS, only those 1K pages that are actually necessary get allocated.

Programs can be written according to any of the DEC, IBM, or UNIVAC hardware notations, but it is recommended that they correspond to the standard hardware notation of SDG recommendation no. 4. Unless explicitly requested, lower case letters in identifiers are considered equivalent to upper case. The character set for use in program execution is the ASCII set, having rank values from 0 to 127 (decimal).

All system dependent details are related to the SINTRAN III operating system.

Section 7.1 gives a minimum of instruction for new users who want to run programs without reading the entire manual first.

1.2. Compiler description

The compiler reads programs written in SIMULA and translates them to binary relocatable code, binary absolute code, or instructions directly placed in memory, according to user commands. This section gives a summary of all known extensions and restrictions. Reasons and details are, in general, described elsewhere.

1.2.1. Extensions

An extended language called Zimula is available upon request, see appendix E.

External classes and procedures (in Simula, Fortran, or assembly code) have been implemented. An external class may be referenced on any block level of a subsequent compilation. A special compiler command specifies which user libraries are to be searched when looking for external quantities.

The while statement is (of course) implemented.

The hidden protected feature (see SDG recommendation no. 1, Attribute Protection) is included syntactically, but as yet not processed semantically. This was done to allow transfer of programs without having to delete the protection specifications.

Compiler directives, identified by % in column one, include the conditional compilation feature.

If all matches to a virtual procedure have the same number, types, and modes of parameters, then the calls for this procedure will be just as efficient as an ordinary procedure call.

Text variables are allowed to point to text constants. Any attempt to modify a text constant will be detected by the run time system.

Run-time checks for array bounds, qualifications, arithmetic overflow, and none can be switched off individually for any part of the program as desired.

An interactive debugging system is available, allowing breakpoints and a statement by statement execution.

1.2.2. Restrictions

The maximum length of an identifier is 72 characters, all of which are significant. This conforms to SDG recommendation no. 4 on hardware notation.

Source lines have a maximum length of 120 characters. Excess characters are ignored, as well as non-printable characters. (The TAB character is taken as one space.) Carriage return is the end-of-line signal.

Texts (both variable and constant) have a maximum length of formally 32761 characters, though overall program and data size may effectively restrict the length further.

Integer variables and constants have a range from -32768 to 32767, inclusive. Real variables and constants can have these values: From -1&4920 to -1&-4920, exact zero, and from 1&-4920 to 1&4920. They have 10 significant digits.

The number of nested expressions, procedure calls, etc. in each statement is restricted to 64. This number should be generous but is easily increased upon request.

Prefixing by system classes is only allowed with SIMSET and SIMULATION, but these prefixes can be used at any block level.

The indices to a multi-dimensional array are not checked individually. To save time and data space, only the resulting address is checked against the bounds.

Depending on programming style, the compiler in its present state can only take approx. 5000 lines of source text. In some cases, however, the user can use external compilation to split his program. The combined program size capacity is estimated to be over 8000 lines.

An array object may occupy a maximum of 49152 words total. Allowing for some overhead, this gives approx. 49000 integer elements, 16300 real elements, or 12000 text elements.

A block object (procedure, class etc.) may only occupy 512 words maximum. A compiler error message is given if the maximum space is exceeded. Note that the program code allocates temporary cells and that these can also violate said restriction.

1.3. Operating environment

Information in this section is subject to change without notice. TPH SIMULA under SINTRAN III consists of the following public files:

N10-SIMULA:BPUN	Binary absolute file containing the compiler (usually made reentrant under the name SIMULA)
SIMRT3:SIM	Definition of non-standard library routines
SIMERR:DATA	Error messages for use by the run time system
SIMERC:DATA	Error messages for use by the compiler
SIMBASE3:SIM	Definition of DIRECTFILE
SIMSET3:SIM	Definition of SIMSET
SIMULA3:SIM	Definition of SIMULATION
SIMRAND3:SIM	Random drawing routines
SIMMAT:SIM	Mathematical library
S-LOAD-DUAL:PROG	Loader for :PROG files with large programs

All enquiries concerning Simula system results should be accompanied by the version number printed by the compiler that was used. The version number must include both level and the edition, e.g. 4.011.

For efficient use on a NORD-100, please observe that the physical memory available for swapping must be sufficiently large to accomodate the Simula system's working set, which can be up to 123K words depending on program size. A computer running Sintran III/VS will, in most configurations, need at least 96K words of memory.

The Sintran III/VS host system must allow 128K words of virtual memory per user. This requires a modification to the standard system, namely, the @CHANGE-BACKGROUND-SEGMENT-SIZE command has to be executed for each terminal that is to be allowed use of SIMULA compiler or programs.

2. Precompiler

Before reaching the compiler, your source program is inspected by a macro processor called the precompiler. All lines beginning with % are taken as command lines to this processor - therefore, take care so as to avoid e.g. comment lines to look like macro commands. However, a command line beginning with % followed by a space is ignored by the precompiler.

Macro commands serve two purposes: Changing compiler options from within the source text, and controlling the omission of specified sections of the program. The latter is called conditional compilation and is particularly useful when compiling the same source text for use on several different installations. The Simula run time system uses the precompiler in this way.

2.1. Flags

A flag has the value true or false and is identified by an identifier that may have any length as long as it does not exceed the source line. (The source line is limited to 120 characters.) The name tables being totally separated, there is no name conflict with respect to Simula program identifiers. A flag identifier may actually consist of any characters, not only letters or digits. A flag not yet defined, if referenced, gets the initial value false. There are two commands for definition of a flag:

%SET	identifier	The specified flag is set to <u>true</u> . It may have been defined and/or referenced before, but the now assigned value is valid from now on only.
------	------------	---

%RESET	identifier	The specified flag is set to <u>false</u> . Scope as explained above.
--------	------------	---

The use of flags is shown in section 2.3.

2.2. Options

Compiler options in this Simula system have values from -32768 to 32767, rather than the more common false or true. At any time, a specific option is considered to be set if the value is greater than zero, the initial value being zero. This allows selected program sections to be enclosed by increment/decrement of options, the effect of which can be controlled from the outside. Option assignments are defined in ch. 7.3.2. There are two macro commands for changing options:

`%SETOPT cccc` Increment all options mentioned in cccc.

`%RESOPT cccc` Decrement all options mentioned in cccc.

These commands correspond exactly to the console commands `>SETOPT` and `>RESOPT`, described in ch. 7.3.1.

2.3. Conditional compilation

We have now arrived at the precompiler's main task: Suppression of selected paragraphs of code, controlled by the current flag values. A general construction for conditional compilation looks like:

```
%IF flag
:
:
:   Simula source text,
:   first paragraph
:
%ELSE flag
:
:
:   Simula source text,
:   second paragraph
:
%FI flag
```

If the current value of flag is true, the first paragraph is compiled and the second is skipped. If false, the second is compiled instead. If either paragraph is to be empty, the corresponding `%IF` or `%ELSE` may be omitted. Within the paragraphs, other conditional compilations may occur as long as they do not use the same flag as the enclosing one. Example on a branch to be executed if flag A and/or B is true:

```
%ELSE A
%IF B
%FI A
    outtext("This is version A or B");
    outimage;
%FI B
```

The construction may seem rather odd and intuitively illegal, but it is based on knowledge of the one-pass operation of the precompiler, which is governed by these simple rules in its neutral state:

- Any `%FI` command is ignored.
- Any `%IF` with a true flag is ignored.
- Any `%ELSE` with a false flag is ignored.

- %IF with a false flag causes input skip, ignoring all lines until %ELSE or %FI with the same flag is encountered. Note that all references to other flags are ignored.
- %ELSE with a true flag causes input skip, ignoring all lines until %FI with the same flag is encountered.

According to this, the example works as follows: If A is true, the %IF B is ignored and we compile the two statements. The late %FI B is ignored. If A is false, the %ELSE A is ignored and thus the %IF B is checked. If even B is false, all text up to %FI B is ignored. This then is the only case where the statements are suppressed, for a true B would cause %IF B to be ignored, and as stated above, the stand-alone %FI A and %FI B do no harm.

2.4. End of file

The macro command %EOF may be used to terminate the source file. It has the same effect as if end-of-file had been encountered. If compiling directly from the terminal (not recommended), %EOF is the only way of terminating the source text.

2.5. Alternate source files

At any point in the source text, code from another file may be inserted by the command

%COMPILE filename

subject to suppression by flags. The filename may be replaced by a logical file number. If the %COMPILE is honored, a separate precompiler is created for the inclusion, having no knowledge of or effects on the flags of the surrounding precompiler. %COMPILE commands may be nested to any depth, up to the operating system's limit of simultaneously opened files.

3. Source program

Your program is to be supplied as lines of maximum 120 printable ASCII characters each. Though we recommend use of the standard hardware notation as of SDG recommendation no. 4, it may follow any of the DEC, IBM, or UNIVAC notations. This chapter describes the complete hardware notation and gives other information closely related to this. The general idea is to reserve a set of 64 words, containing begin, end, integer, and many others, thus eliminating the need for surrounding quotes or other means of recognition. In consequence, blanks cannot be allowed in identifiers and are needed between reserved words and user identifiers. These restrictions, according to most users, are insignificant with respect to the valuable time saved in the typing of programs.

According to SDG rec. no. 4, a special alternative for comments has been allowed: ! outside of a comment, character, or text constant, is accepted as a start of a comment, thus replacing the key word comment. This causes no restrictions at all on source programs, since the symbol ! would otherwise be illegal.

3.1. Delimiters

In describing delimiters, we deliberately omit those described in the Common Base by underlined words, since they are all simply coded as reserved words. A complete table of reserved words is given in the next section. We also omit the delimiters used in constants only.

What remains then is a set of delimiters represented by various non-alphabetic symbols in the Common Base. Many of these have several alternative representations in TPH SIMULA; some of them may even be represented by reserved words. Here is the set:

<u>Common Base</u>	<u>TPH SIMULA with alternatives</u>
=	= eq
≠	<> ne =/ =
<	< lt
>	> gt
≤	<= le
≥	>= ge
==	==
≠/=	≠/=
¬	not
^	and
v	or
≡	imp
≡	eqv
((
[((/
))
]) /)
:	: ..
;	; ., \$
:=	:= . = .. =
:-	:- . - .. -
+	+
-	-
x	*
/	/
÷	//
↑	** ^
.	.
,	,

3.2. Identifiers

An identifier begins with a letter (A-Z, a-z) and contains letters and digits. To improve readability, it may also contain the underscore character in any quantity. Underscores are significant. The maximum recognizable identifier length is 72; any length exceeding this is an error.

Letters of an identifier are internally converted to upper case unless compiler option U has been set. Option U does not apply to reserved words. The following identifiers are reserved (regardless of lower or upper case) and may not be used for any purpose other than those specified in the Common Base or in this manual:

ACTIVATE	AFTER	AND	ARRAY	AT
BEFORE	BEGIN	BOOLEAN	CHARACTER	CLASS
COMMENT	DELAY	DO	ELSE	END
EQ	EQV	EXTERNAL	FALSE	FOR
GE	GO	GOTO	GT	HIDDEN
IF	IMP	IN	INNER	INSPECT
INTEGER	IS	LABEL	LE	LONG
LT	NAME	NE	NEW	NONE
NOT	NOTEXT	OR	OTHERWISE	PRIOR
PROCEDURE	PROTECTED	QUA	REACTIVATE	REAL
REF	SHORT	STEP	SWITCH	TEXT
THEN	THIS	TO	TRUE	UNTIL
VALUE	VIRTUAL	WHEN	WHILE	

Note the reservation of hidden and protected. They are included for compatibility reasons, allowing transfer of programs using a recommended extension to Simula systems. However, TPH SIMULA does not as yet process such specifications, it merely tolerates them. There are plans for implementation, and the users will be notified when the feature is available.

3.3. Constants

Some constants are represented by reserved words - true, false, none, and notext. Since they are known from Common Base, they are not covered here. Instead, we will explain the notations for numeric and text constants.

3.3.1. Numeric constants

The standard forms for integer and real constants apply. Exponent sign for the real constant is &. Some examples of legal constants:

1 2 -1000 5.3 0.003 125&-3 -1&10

As already mentioned, integer variables and constants have a range from -32768 to 32767, inclusive. Real variables and constants can have these values: From -1&4920 to -1&-4920, exact zero, and from 1&-4920 to 1&4920. They have 10 significant digits.

3.3.2. Text constants

Character constants are enclosed in single quotes ('). Since non-printable characters of the source program are ignored, only the 95 printable ASCII characters are allowed. Examples:

' ' 'A' '(' 'n' ':' ''' ''''

The parameter range for char is 0 to 127, corresponding to the 7-bit ASCII code set. Codes 0 to 31 are the non-printable codes (control codes), codes 40 to 63 are ' ' to '?', codes 64 to 94 are upper case letters (and some symbols), codes 96 to 126 are their lower case equivalents, code 95 is '_', and code 127 is rubout. Character variables are initialized to code 0 (zero). The information given above should not be used by the programmer who wants a portable program, with the exception that all ASCII systems can be expected to have the same code definitions.

Text constants are enclosed in double quotes (") and may contain the same character set as character constants. A double quote within a text constant is coded as two double quotes. Two adjacent text constants (possibly on successive lines) are automatically concatenated into one. If more than one line is required for the constant, use of the concatenation feature is highly recommended. Some examples of text constants:

<u>Source code</u>	<u>Result</u>
"Abc"	Abc
"Quote: """	Quote: "
"Part one, " "part two"	Part one, part two

To allow control codes in character constants, there exists a coded version of character constant:

!n!

where n is 1, 2, or 3 digits, to be interpreted as a decimal number and not to exceed the value 127. This is interpreted as the character constant with ASCII code n. If n is more than 3 digits, or greater than 127, then the !n! sequence is only taken as it is. The coded version can be used in single character constants:

'!7!'

would mean the ASCII code for BELL. It can also be used in text constants:

"Abc!7!!7!!7!"

would mean the text Abc followed by the code for BELL three times.

4. Procedures and classes

This chapter describes how to create and use separately compiled modules. Creation is based on the concept "entrypoint quantities", which means, quantities that are available outside of the defining module. The corresponding concept for referencing such quantities is called "external quantities", indicating that the quantities in question are to be found outside of the referencing module.

At the end of the chapter, some remarks are given concerning special features of virtual procedures.

4.1. External quantities

Primarily for the benefit of assembly coding, two different forms of external quantity have been defined: Body substitution and complete substitution. For the ordinary user, only the latter is relevant, meaning that the quantity is to be completely defined in an externally compiled module, available at the time of compiling the referencing module. It is the only form that adheres to the philosophy of total safety in high level language programming.

Advanced system programmers are allowed access to the body substitution. This is a means of having the compiler generate a prototype for a routine, the code of which is external and usually FORTRAN or assembly coded. No responsibility is taken for the results or disasters caused by such routines. See appendix F for information on FORTRAN or assembly coded routines.

COBOL-type routine calls are not supported.

One or more external quantities for complete substitution are declared by the keyword external, followed by class or procedure (without prefix or type), followed by a list of identifiers (separated by commas and terminated by semicolon). Neither parameters nor body is given. Examples:

```
external procedure operate;
```

```
external procedure abool,bbool,cbool;
```

```
external class system;
```

In addition, within a block or class, any identifier that has been used for prefixing without ever being declared at that block or class is implicitly declared external class. This is why SIMSET and SIMULATION may be used without declaration, as in any other SIMULA implementation.

When encountering an explicit or implicit external declaration, the compiler starts searching the libraries. These are the system libraries (standard procedures or classes), followed by any library that the user may have specified by the >LIB command (see ch. 7). A library is the >BRF specified output from a separate compilation. See sec. 4.2 for description. Upon finding the requested quantity name, the compiler reads all necessary information about parameter list, type or prefix, and local attributes. Thus all use of the quantity can be fully checked at compile time, just as if it had been declared internally.

To maintain the safety of Simula, simply observe the following rule: Whenever making a change in a library file through an E-option compilation, recompile all modules that reference this file, then all modules that reference files now changed, etc., until the main program has been recompiled. Never assume your changes are insignificant for the calling modules. In any case, complaints where external quantities are involved will not be investigated unless all source files are submitted for recompilation. Remember that each new Simula release invalidates all >BRF produced files, but will not, of course, affect >BIN files.

4.2. Entrypoint quantities

This section describes how to create an entrypoint quantity written in Simula. If the source file contains one or more declarations instead of a main program block, no absolute program is produced. Instead, the specified BRF file becomes a valid user library. The source text for this type of compilation consists of one or more procedure or class declarations and no statements. Each of these declarations defines an entrypoint quantity which means the quantity, its prefix/type, and parameter list is known to a later compilation with the appropriate >LIB command. If a class, the local attributes are also known - and so on to any depth of local classes. Example showing the most usual user library definition:

```

class prefix; begin
  integer a,b,c;
  procedure p(i); integer i; begin
    real d,e,f;
    ...
  end;
class cl; begin
  integer g,h;
  procedure q; ... ;
end;
end;
    
```

The main program prefixed by "prefix" now has access to a, b, c, p, cl, and by remote accessing or inspect all attributes of cl: g, h, and q. In other words, exactly and with the same compile time checking as

if prefix were declared in the main program.

Caution: Although legal, the subsequent use of an external class on a block level other than that of its original compilation should be avoided. If not, the run time level changes cause about 50% increase in execution times.

4.3. Virtual procedures

For efficient use of virtual procedures, please note that all matches to a virtual procedure should have compatible parameter lists. That is, corresponding parameter positions should have the same type and transmission mode, and of course the number of parameters has to be equal.

It is legal to employ the more general call to virtuals. In the case of non compatible lists however, the procedure call will be substantially less efficient.

Note! If all the calls to a particular virtual procedure defined in an external module are made with compatible parameter lists throughout that module, then a subsequent compilation is not allowed to break this consistency for the procedure in question.

5. Standard classes

This chapter describes the implementation of standard classes, a set consisting of I/O classes and the SIMSET/SIMULATION classes. In the interest of machine independence, they are written as external Simula classes. Unfortunately, this leads to some minor restrictions which are foreseen by the Common Base: The I/O classes cannot be used for prefixing. Rest assured, however, that the efficiency thus gained justifies the sacrifice.

5.1. Input/output

The class identifier "FILE" has been made accessible to the user program, a valuable feature for such occasions as passing file parameters to procedures. Generation of a pure FILE object is not practical though, since open and close will be undefined. See (1).

The procedure "open" is defined as follows:

```
procedure open(t); text t; begin  
  image:-t;  
  :  
  :  
  :  
  end;
```

At the point of calling open, the file name parameter of the file object is examined. If it is a number, then the file object is associated with that number - interpreted as an octal logical file number. Some file must already have been opened to this file number.

If the file name is not a number, then the Simula system will try to open (via the SINTRAN system) a file of the given name. In this case, the contents of the t parameter to open is inspected for the presence of the following information items:

```
:tttt @a
```

If present, :tttt must start in column one and the name tttt must consist of maximum four characters. It means that the default file type is to be tttt instead of DATA as would be assumed otherwise.

If present, @a must either start in column one or be separated from :tttt by exactly one space. It means that the file is to be opened for access according to the value of the @a item:


```

@W  Write sequential
@R  Read sequential
@WX Write/read random
@RX Read random
@RW Read/write sequential
@WA Write append
@WC Write/read random common
@RC Read random common

```

If no @a is present, then the file is opened for access according to the file object's class:

```

infile      Read sequential
outfile     Write sequential
printfile   Write sequential
directfile  Write/read random

```

To indicate successful interpretation, the :tttt and @a fields of the t parameter to open will be space filled after use.

The procedure "close" is defined as follows:

```

procedure close; begin
  :
  :
  :
  end;

```

If the Simula system had to open a file when "open" was called, then this file is now automatically closed.

The image length used for sysin and sysout is 136. If, for example, a shorter sysout.image is wanted, the user will simply include in his program start the statement "image:-blanks(100);" or whatever he might want.

The parameter NAME to a FILE may be read afterwards by the procedure (local to class FILE)

```

text procedure id; ..... ;

```

usually returning a copy of NAME. If, however, NAME was a file name rather than a number, that name is only returned if the file object is currently in the closed state. When open, a text of 8 characters containing the system generated file number (in octal) is given. This feature is useful if several file objects are to access the same file number, and the Simula system is supposed to open it as described previously under the procedure "open".

In communicating directly with terminals, it might be desirable to operate on a character basis. For this purpose, the following procedures exist:

```
procedure directout(c); character c; ..... ;  
character procedure directin; ..... ;
```

The procedure `directout` transmits `c` to the terminal at once. If the terminal is in character mode, each character typed will be available at once, and it can then be read by calling `directin`.

Actual files associated with `infile`, `outfile`, or `printfile` may be of any type; note, however, that a `directfile` must be random accessible. Concerning the alternate use of `outfile/infile` and `directfile` for the same physical file, observe:

- On `outimage` to an `outfile`, the stripped image is written (zero length if possible) followed by carriage return, line feed.
- On `outimage` to a `directfile`, the image is not stripped before writing. All images on the file will therefore have the same length - equal to `image.length` at first call to open, plus carriage return and line feed.
- On `inimage` to an `infile`, the mass storage image must not contain any other characters than spaces in excess of the `infile`'s image length. The terminating carriage return is not included in the count, and any leading line feed is ignored.
- On `inimage` to a `directfile`, the mass storage image's length must equal the `directfile`'s `image.length` as it was at open time. The image length uniformity is necessary because otherwise we could not calculate the parameters for a random access positioning based on a `locate` call. Following an `inimage` or a `locate`, `ENDFILE` will be true if `LOCATION` now points into a non-existent mass storage block. This does not mean that `inimage` is necessarily allowed whenever `ENDFILE` is false: The physical mass storage blocks are usually far greater than the Simula image, so that in general only part of an existing block may contain valid images. You may object to this seemingly useless (but perfectly legal, according to Common Base) definition of `ENDFILE`, but considering system overhead, we think that fast operation is more important for most users.

The class `outfile` is extended by

```
procedure breakoutimage; ..... ;
```

having the same effect as `outimage`, except that no carriage return or line feed is given after the output. Thus having no spacing effect, the procedure is not virtual.

The set of editing and de-editing procedures has been extended by `inoct`, `outoct`, and `outzint`. See section 6.4.

5.2. SIMSET

The system class SIMSET is an external class implicitly available from a system library. When used as a prefix, it is automatically declared local to the block enclosing the prefixed one. In other words, it is used according to the Common Base.

If more than one block has SIMSET declared local to it, each of these SIMSET versions is at compile time considered different from all the others. This means you are not allowed to do assignment between ref(link) variables of different versions, for example. However, in situations accepted by the compiler (such as formal procedure calls), the corresponding run time checks will accept, because there is only one SIMSET at run time.

Conclusive remark: There is nothing special or restrictive about SIMSET in TPH SIMULA.

5.3. SIMULATION

All information of the previous section applies to SIMULATION as well. In a block that has SIMULATION declared (implicitly), SIMSET is automatically declared by the same technique. The fact is relevant in cases of many versions of SIMSET and SIMULATION, giving the effect already mentioned.

SIMULATION uses quasi parallel sequencing. Refrain from using call, detach, or resume to avoid corrupting the SIMULATION mechanisms. Another caution: If SIMULATION is used as prefix in other blocks than the outermost one, then expect a 50% increase in execution time. This is due to administrative overhead in changing block levels within the same class body.

6. Standard procedures

Since this is no textbook, a complete list of standard procedures is not given here. Instead, we concentrate upon useful information about implementation defined items. A complete list of standard procedures and classes appears in appendix B, and the added (non-standard) procedures are described in appendix G.

6.1. Quasi parallel sequencing

The procedures call, detach, and resume are all implemented. Since there is presently some confusion as to the operation of detach, our version will be defined here along with the terminology of quasi parallel sequencing.

operating object

- object that is either the currently executing one, or dynamically enclosing it.

attached object

- object that is operating and has a return point to some calling object. This is true for all procedures, and for class objects just called by new and not yet returned.

detached object

- object that has no return point, but is rather a component of the nearest enclosing quasi parallel system and has a reactivation point when non-operating. It may be operating or non-operating. A prefixed block is initially (and always) detached as seen from inside, but of course has an implicit knowledge of how to return to its environment.

terminated object

- object that has passed through its final end. It can never again become operating. Any attempt to make it operating is an error. Otherwise, the terminated object is similar to a detached one.

The procedure "detach" is local to any class and will thus operate on the nearest statically (textually) enclosing class or prefixed block (possibly inspected), or in the case of X.detach, the object X. If that object is a prefixed block, detach is a no-operation. If it is an attached class, the class becomes detached and control is passed to the return point of that class. If it is a detached class, control is passed to the reactivation point of the nearest enclosing quasi parallel system of the class, that is, usually after the last resume statement of that system's main program. In both cases, the point after the detach statement becomes the reactivation point of the

class.

The procedure "call" has a parameter that must be a detached and non-operating class object. It causes the object to become attached at the point of the call statement, then passes control to the reactivation point of the class object.

The procedure "resume" has a parameter X that must be a detached and non-operating class object. The nearest enclosing quasi parallel system S of that object is sought out. Then the operating component Y of that system is found, and by previous definitions the resume statement is dynamically enclosed by Y. The effect of the statement is to swap X and Y such that Y becomes non-operating with a reactivation point after the resume statement, and X becomes operating. Control is passed to the reactivation point of X.

The above definitions are given for the general case of a number of enclosing systems of any complexity. In a very simple model of only one prefixed block (the outermost block of a program, the main program, automatically is) and a few class objects that execute call, detach, and resume directly and moderately in their bodies, we may explain in other terms:

A detach has no effect in the main program, in a class object called by new or call returns to after the new or call, in a class object restarted by resume returns to the main program's last resume (which may or may not be the one that started the class object).

A call on a class object also starts execution following the object's last detach or resume, but the object itself (by detach or resume) selects which object is next to be executed. Resume will execute the parameter, while detach will execute the main program.

6.2. Arithmetic and conversion

Rank or isorank (a new standard procedure introduced by SDG recommendation no. 4) converts from character to integer and may give values from 0 to 127. Isochar (also introduced by SDG rec. no. 4) converts the other way; its integer parameter must be of value 0 to 127. Char also converts from integer to character, but accepts any parameter - this may be used for creating certain special character codes. Besides, char is much more efficient than isochar.

6.3. Random draws

According to Common Base, the U parameter to random drawing procedures is of type integer. On a 16 bit computer, this gives a random stream of 65536 different values.

Note: The U parameter must be a simple variable of the correct type, since it is called by name.

6.4. Editing and de-editing

The result of a `getint` or `getfrac` must be within the range for integer (see section 3.3.1), and the result of a `getreal` must be within the range for real (same section). Otherwise a run time error message is given. On editing (`put...`) the field must have enough space for the number, or else the field will be filled with asterisks, and a warning will be printed after program termination.

The exponent given by `putreal` has the form "&+nnnn", i.e. four digits are needed because of the wide exponent range of the NORD-100 hardware.

We have added a non-standard procedure "`getoct`" operating like `getint`, except that the base for de-editing is 8 rather than 10. Similarly, "`putoct`" will edit using base 8, and will not suppress leading zeroes, but otherwise resembles `putint`. The procedure "`putzint`" operates like `putint`, but will not suppress leading zeroes. Corresponding `inoct`, `outoct`, and `outzint` procedures have been added to the file classes.

6.5. Other text operations

Local to text variables, we have added a non-standard procedure "`trim`" operating like `strip`, but removing leading blanks rather than trailing ones.

7. Compilation and execution of programs

Having called the compiler by the @SIM command, the user communicates with Simula through a set of console commands, each solicited by the pointed bracket ">". Special conditions concerning options, files, etc. may then be set up prior to compilation.

If the compilation was successful, the produced code normally resides as instructions in core. (Assuming option V was specified.) The program will start automatically, and the run time system prints: Ready for Simula execution. If on line debugging was requested, the Simula system prints an asterisk and waits for the first debugging command. On passing through the last end, the CPU time used (and number of garbage collections, if any) will be printed and control is passed back to the operating system.

This chapter describes all available compiler commands, options, and debugging commands.

All editing facilities and file name abbreviations of SINTRAN III are available during typing of commands and input data to the compiler, as well as to an executing Simula program. However, compiler commands cannot yet be abbreviated.

7.1. Elementary compile and execute procedure

This section is a short introduction for use under the SINTRAN III operating system. To prepare your source program, use QED (Quick Editor) and write the program text into the editor. Use any combination of DEC, IBM, or UNIVAC hardware notation, but no line should start with % in column one as this is used for the compiler command facility. Save your program on a file, for example, PROGRAM:SYMB, and exit from QED. Assuming a listing is wanted, conversation should now proceed as follows (user's key-in is underlined):

```
@SIM
TPH SIMULA 4.017
>SETOPT VS
>LIST L-P
>COMPILE PROGRAM
```

Your listing should now go to the printer. If errors were found, re-enter QED and correct the PROGRAM:SYMB file.

If your program was compiled without errors, the >COMPILE command proceeds to assemble the object code into memory. This is indicated by the message LOADING PROGRAM. Then the program starts automatically with the message

Ready for Simula execution

followed by any output your program might generate, or if so programmed, you will now type your input to the Simula program. Input goes to the program only after giving the carriage return. After program end, the system prints used CPU time, e.g.

CPU seconds used: 5.017
Exit Simula

and returns to the operating system.

7.2. Saving the binary code

Preparations for saving binary code are best made by using the compiler command >PROG BINSIM (say) prior to >COMPILE PROGRAM. The first execution can proceed as usual, while subsequent runs should be started thus (note that the name BINSIM was chosen as an arbitrary example of file name):

@BINSIM

As seen above, typing the file name given in the >PROG command as a command to the operating system, the program starts execution directly. The file BINSIM:PROG must exist in advance, so you would have to create it the first time.

7.3. Advanced compiler use

This section is intended for Simula experts and other skilled programmers who only want the strictly formal definitions of commands and options. The run time section is similar in aim.

7.3.1. Terminal command language

Currently, this set of commands is implemented:

>SETOPT	cccc	Increment all compiler options mentioned in cccc. An option is set when its value is greater than zero. The initial value is zero. See next section for a list of options.
>RESOPT	cccc	Decrement all compiler options mentioned in cccc.
>RTS	dddd	Set to <u>true</u> all run time system options mentioned in dddd. Run time system options can only have the values <u>true</u> or <u>false</u> . See section 7.4.3 for a list of options.

- >LIB filename The specified filename, presumably containing BRF code from a previous Simula compilation of an external module, is registered as a potential library file to be checked when searching for external quantities. The user may give any number of >LIB commands; the most recently specified file will be searched first.
- >ASM filename Specifies that object code (in symbolic assembly) be written to filename. This is mainly for testing purposes.
- >BRF filename Specifies that binary relocatable code be written to filename. This is for use when compiling external modules, though even a main program can be saved as BRF code (and loaded by >LOAD).
- >BIN filename Specifies that binary absolute code (in the standard NORD-100 hardware format) be written to filename. Such code can be executed by giving the @PLACE filename command (remember the type :BIN) followed by a @GO 0 command.
- >PROG filename Specifies that binary absolute code (in the standard Sintran @DUMP format) be written to filename. This is the recommended way of saving the object code. If the program is very large, it can only be started by @S-LOAD-DUAL filename. Otherwise, @filename is enough. The compiler will inform you if your program is of the large kind.
- >LIST filename Specifies that listing be written to filename. The listing includes error messages and byte output, where appropriate.
- >RUNOFF filename Specifies that documentation output be written to filename. Such output is generated instead of compilation if option M is set at >COMPILE time.
- >COMPILE filename Do full compilation according to current options. The source code is taken from filename; the default source being the terminal. (In batch jobs, the batch input file.) The compilation stops at end-of-file or the %EOF macro command (see precompiler chapter). If option V is specified and no errors were detected, the program is automatically executed. Note: Only one >COMPILE or >LOAD command is allowed. After such command, >EXIT is performed implicitly unless there is an automatic execution instead.
- >LOAD filename Binary relocatable code from filename is loaded, and if V option, executed. The loaded absolute program can also go to a >BIN or >PROG specified file.

>EXIT Return to job monitor.

Wherever "filename" is specified above, the file number of an opened file may be used instead. The following default file types are used if the compiler must open the file:

>LIB	SIM
>ASM	SYMB
>BRF	SIM
>BIN	BIN
>PROG	PROG
>LIST	SYMB
>RUNOFF	SYMB
>COMPILE	SYMB
>LOAD	SIM

NOTE: The binary relocatable files have type SIM. The format is not compatible with other software because a SIM file contains Simula attribute information.

7.3.2. Compiler option set

The following options are implemented:

- A Array bounds not to be checked at run time
- D Debugging symbol table produced
- F Flags for begin/end are abbreviated and given in left margin, or completely removed (option FF)
- G Generation of symbolic code to file specified by >ASM
- I Inhibit the generation of line numbers
- K Kill the compilation on the first error detected
- L List all input lines, including macro commands
- M Produce a RUNOFF document file, rather than compiling a program
- N No checks for arithmetic multiply overflow (>RESOPT N: Check add/subtract as well)
- O On-line debugging by interaction with run time system via terminal
- Q Remove qualification checks
- R Remove none checks

- S Source listing, suppressing macro commands and the lines suppressed by such commands (see precompiler chapter)
- T Time and space requirements printed after compilation
- U Upper and lower case in non-reserved words recognized as being different
- V Automatic execution (compile-and-go)
- W Warning messages suppressed
- X Experimental compilation for system maintenance. Not to be used by others.
- Y Print loader map (primarily useful for system maintenance)
- Z Simula extended language, see appendix E.

In addition, there are some non-letter options which are only for maintenance of the Simula system. They are described in the internal technical documentation.

7.3.3. Compilation errors

Messages are given along with the source listing (syntax errors) or at the end, identified by line number (semantic errors). They are believed to be sufficiently instructive so as to make an explanation here unnecessary. A complete list appears in appendix C.

7.4. Run time system

7.4.1. Debugging command language

In a program, or part of a program, that has been compiled with option O set, the run time system is given control at the end of each source statement. The line number is printed on the terminal as follows:

Line n
*

One of the following commands may then be given:

- *HELP Print a list of commands
- * (carriage return only) Execute next statement
- *STEP Same as above

*STEP n	Execute n next statements unless breakpoint is reached
*BREAK n	Set breakpoint after first statement of line n
*BREAK	Remove breakpoint
*GO	Do not stop until breakpoint (if any) is reached
*GO n	Same as: *BREAK n *GO
*DYN n	Print the chain of calling or enclosing block instances, up to a maximum of n (default 20) instances. This command is very useful in showing where we are in an error situation. The object addresses given can be used as parameters to the *OBJECT command.
*RESTART	Close all files, reset all system data, and repeat execution of the entire user program, beginning with the message Ready for Simula execution.
*SYSIN file	Close the current sysin, and reopen it to given filename
*SYSOUT file	Close the current sysout, and reopen it to given filename
*TRACE n m	Set up tracing facilities registering, from now on, each execution of statements from line n to line m. Compiler option O must have been set for statements to be registered.
*HISTO i n m	Stop tracing. List the results on printer, showing line n to m as a bar graph and grouping i lines in each bar. Default i is 1, default n, m are those of last *TRACE command.
*OBJECT name	Dump the identified object on the terminal. The name must identify a ref variable that is visible from where we are (according to *DYN). Repeated dot accessing is allowed, as well as array element access with a decimal constant as index. *OBJECT without parameter will dump the current object. Note: Names of variables are only known if the module was compiled with D option.
*VAR name	Print the value of the identified variable, visible from where we are. Dot access and array access is allowed just as for *OBJECT.
*DRIVER a	Dump the driver at octal address a. Mostly intended for system maintenance, this command requires a correct driver address as given by some *OBJECT command. *DRIVER without parameter will dump the current driver.

*EXIT Terminate the user program

Other commands than the ones mentioned above give:

ILLEGAL DEBUG COMMAND

*

7.4.2. Run time errors

In the event of a run time error, a self explanatory message is printed on the terminal. Then the debugging command processor is entered. A carriage return, *STEP, or *GO will cause final Simula program termination, or, if desirable, the entire program can be repeated by *RESTART. A list of run time error messages appears in appendix D.

7.4.3. Run time system options set

The following options are implemented:

- N Suppress the "Ready for Simula execution" message and the termination messages from the run time system. Useful for writing system software in Simula.
- O On-line debugging mode before program start. The run time system will print "Pre-program conversation requested.", then Line 0 and the usual asterisk. The user may then use any number of debugging commands prior to starting his program with the *GO command. This feature is especially useful for changing the input/output files by *SYSIN and/or *SYSOUT.

8. References

- (1) O.-J. Dahl, B. Myhrhaug, K. Nygaard:
COMMON BASE LANGUAGE (S-22)
Norwegian Computing Center
- (2) G. Birtwistle et al.:
SIMULA BEGIN
Studentlitteratur
- (3) SINTRAN III Users Guide (ND-60.050)
Norsk Data A.S.

Appendix A: Example on use

The following batch job creates a short Simula program, tries to compile it, then goes back to the editor to correct it, and finally succeeds in compiling and executing the program. The program just adds three numbers from the job stream and prints the result. This use of batch job is a bit artificial of course, since we have to know in advance that the first compilation will fail. It is, however, believed to demonstrate quite well how a typical terminal session could proceed for a novice user. The batch job file looks like this (<L> means control-L, <ESC> means escape):

```
@ENTER TPH,,,60
```

```
@CC USE EDITOR TO TYPE IN THE FILE:
```

```
@QED
```

```
I
```

```
BEGIN
```

```
    OUTINT(ININT+ININT+ININ,10);
```

```
END;
```

```
<L>
```

```
W"ADD3"
```

```
F
```

```
@CC TRY TO COMPILE IT
```

```
@SIM
```

```
SETOPT SV
```

```
COMPILE ADD3
```

```
@CC CORRECT THE ERROR BY USING EDITOR
```

```
@QED
```

```
RADD3
```

```
S/ININT,/ININ,/
```

```
W*
```

```
F
```

```
@CC SECOND ATTEMPT TO COMPILE AND RUN THE PROGRAM
```

```
@SIM
```

```
SETOPT SV
```

```
COMPILE ADD3
```

```
5 50 500
```

```
@CC OK, LOG OUT
```

```
<ESC><ESC>
```

Running it on the computer, we get the following printout from the whole job:

@CC USE EDITOR TO TYPE IN THE FILE:

@QED

QED 4.2

*I

INS >BEGIN

INS > OUTINT(ININT+ININT+ININ,10);

INS > END;

INS >*

*W"ADD3"

26 WORDS WRITTEN

*F

@CC TRY TO COMPILE IT

@SIM

TPH SIMULA 4.011

>SETOPT SV

>COMPILE ADD3

TPH SIMULA (SINTRAN III) LEVEL 4 ED 011

00001 BEGIN

00002 OUTINT(ININT+ININT+ININ,10);

00003 END;

*** LINE 00002 ERROR: MISSING DECLARATION OF ININ ***

ERROR MESSAGES GIVEN: 00001

@CC CORRECT THE ERROR BY USING EDITOR

@QED

QED 4.2

*RADD3

26 WORDS READ

*S/ININT,/ININ,/
ND-60.092.03

1 SUBSTITUTION

W

27 WORDS WRITTEN

*F

@CC SECOND ATTEMPT TO COMPILE AND RUN THE PROGRAM

@SIM

TPH SIMULA 4.011

>SETOPT SV

>COMPILE ADD3

TPH SIMULA (SINTRAN III) LEVEL 4 ED 011

00001 BEGIN

00002 OUTINT(ININT+ININT+ININT,10);

00003 END;

LOADING PROGRAM

Ready for Simula execution

5 50 500

555

CPU seconds used: 0.040

Exit Simula

@CC OK, LOG OUT

Appendix B: Summary on standard identifiers

All standard procedures and classes are listed, sorted alphabetically on their names. Those that are not part of the Common Base (or recommended extensions) are marked by an asterisk in the "Extra" column. Note: SIMSET and SIMULATION and their attributes have not been included in the list, since they are fairly well concentrated in their Common Base definitions. Types have been abbreviated: integer to int, Boolean to Bool, character to char. Note that this Simula system also contains certain library procedures that have to be declared external prior to use. These are described in appendix G.

<u>Type</u>	<u>Name</u>	<u>Local to</u>	<u>Parameters</u>	<u>Extra</u>
<u>real</u>	abs		<u>real</u>	
<u>real</u>	arctan		<u>real</u>	
<u>text</u>	blanks		<u>int</u>	
	breakoutimage	outfile		*
	call		<u>ref</u> (any class)	
<u>char</u>	char		<u>int</u>	
	close	file		
<u>text</u>	copy		<u>text</u>	
<u>real</u>	cos		<u>real</u>	
<u>real</u>	cosh		<u>real</u>	
	detach			
<u>Bool</u>	digit		<u>char</u>	
<u>file class</u>	directfile			
<u>char</u>	directin	directfile		*
<u>char</u>	directin	infile		*
	directout	directfile	<u>char</u>	*
	directout	outfile	<u>char</u>	*
<u>int</u>	discrete		<u>real array</u> , <u>int</u>	
<u>Bool</u>	draw		<u>real</u> , <u>int</u>	
	eject	printfile	<u>int</u>	
<u>Bool</u>	endfile	directfile		
<u>Bool</u>	endfile	infile		
<u>real</u>	entier		<u>real</u>	
<u>real</u>	erlang		<u>real</u> , <u>real</u> , <u>int</u>	
<u>real</u>	exp		<u>real</u>	
<u>class</u>	file		<u>text</u>	
<u>char</u>	getchar	<u>text</u>		
<u>int</u>	getfrac	<u>text</u>		
<u>int</u>	getint	<u>text</u>		
<u>int</u>	getoct	<u>text</u>		*
<u>real</u>	getreal	<u>text</u>		
<u>int</u>	histd		<u>real array</u> , <u>int</u>	
	histo		<u>real array</u> , <u>real array</u> , <u>real</u> , <u>real</u>	

<u>text</u>	id	file	*
<u>char</u>	inchar	directfile	
<u>char</u>	inchar	infile	
<u>int</u>	incommand	infile	<u>text array, int,</u> <u>ref(outfile), text</u> *
file	<u>class</u>	infile	
<u>int</u>	infrac	directfile	
<u>int</u>	infrac	infile	
	inimage	directfile	
	inimage	infile	
<u>int</u>	inint	directfile	
<u>int</u>	inint	infile	
<u>int</u>	inoct	directfile	*
<u>int</u>	inoct	infile	*
<u>real</u>	inreal	directfile	
<u>real</u>	inreal	infile	
<u>text</u>	inrest	infile	*
<u>text</u>	intext	directfile	<u>int</u>
<u>text</u>	intext	infile	<u>int</u>
<u>Bool</u>	lastitem	directfile	
<u>Bool</u>	lastitem	infile	
<u>int</u>	length	file	
<u>int</u>	length	text	
<u>Bool</u>	letter		<u>char</u>
<u>int</u>	line	printfile	
<u>real</u>	linear		<u>real array, real array,</u> <u>int</u> <u>int</u> <u>real</u> <u>int</u>
	linesperpage	printfile	
<u>real</u>	ln		<u>real</u> <u>int</u>
	locate	directfile	
<u>int</u>	location	directfile	
<u>text</u>	main	text	
<u>int</u>	mod		<u>int, int</u>
<u>Bool</u>	more	file	
<u>Bool</u>	more	text	
<u>real</u>	negexp		<u>real, int</u> <u>real, real, int</u>
<u>real</u>	normal		<u>real, int</u> <u>real, real, int</u>
	open	file	<u>text</u> <u>char</u> <u>char</u>
	outchar	directfile	
	outchar	outfile	
file	<u>class</u>	outfile	
	outfix	directfile	<u>real, int, int</u> <u>real, int, int</u>
	outfix	outfile	<u>real, int, int</u> <u>int, int, int</u>
	outfrac	directfile	<u>int, int, int</u> <u>int, int, int</u>
	outfrac	outfile	<u>int, int, int</u> <u>int, int, int</u>
	outimage	directfile	
	outimage	outfile	
	outint	directfile	<u>int, int</u> <u>int, int</u>
	outint	outfile	<u>int, int</u> <u>int, int</u>
	outoct	directfile	<u>int, int</u> <u>int, int</u>
	outoct	outfile	<u>int, int</u> <u>int, int</u>
	outreal	directfile	<u>real, int, int</u> <u>real, int, int</u>
	outreal	outfile	<u>real, int, int</u> <u>real, int, int</u>
	outtext	directfile	<u>text</u> <u>text</u>
	outtext	outfile	<u>text</u> <u>text</u>

<u>int</u>	poisson		<u>real, int</u>	
<u>int</u>	pos	<u>file</u>		
<u>int</u>	pos	<u>text</u>		
<u>outfile</u>	<u>class</u> printfile			
	putchar	<u>text</u>	<u>char</u>	
	putfix	<u>text</u>	<u>real, int</u>	
	putfrac	<u>text</u>	<u>int, int</u>	
	putint	<u>text</u>	<u>int</u>	
	putoct	<u>text</u>	<u>int</u>	*
	putreal	<u>text</u>	<u>real, int</u>	
	putzint	<u>text</u>	<u>int</u>	*
<u>int</u>	randint		<u>int, int, int</u>	
<u>int</u>	rank		<u>char</u>	
	resume		<u>ref</u> (any class)	
	setpos	<u>file</u>	<u>int</u>	
	setpos	<u>text</u>	<u>int</u>	
<u>int</u>	sign		<u>real</u>	
<u>real</u>	sin		<u>real</u>	
<u>real</u>	sinh		<u>real</u>	
	spacing	<u>printfile</u>	<u>int</u>	
<u>real</u>	sqrt		<u>real</u>	
<u>text</u>	strip	<u>text</u>		
<u>text</u>	sub	<u>text</u>	<u>int, int</u>	
<u>ref</u> (infile)	sysin			
<u>ref</u> (printfile)	sysout			
<u>text</u>	trim	<u>text</u>		*
<u>real</u>	uniform		<u>real, real, int</u>	

Appendix C: Compiler error messages

In the texts shown below, "... " indicates that some identifier is inserted in the message when printed. The following messages can be given at compile time:

INCORRECT PROGRAM START
GARBAGE AFTER LAST END IGNORED
MISSING STATEMENT AFTER OTHERWISE
MISSING FOR ELEMENT
MISSING WHILE EXPRESSION
:- WITH STEP NOT ALLOWED
MISSING STEP EXPRESSION
MISSING UNTIL
MISSING UNTIL EXPRESSION
IF EXPRESSION MISSING
USELESS ELSE
MISSING THEN
INSPECT EXPRESSION MISSING
MISSING DO/WHEN
MISSING LABEL
MISSING FOR VARIABLE
DOUBLE DECLARATION OF ...
UNKNOWN QUALIFICATION: ...
ILLEGAL QUALIFICATION: ...
UNKNOWN OR CIRCULAR PREFIX: ...
WRONG TYPE, I EXPECTED TO SEE ...
MISSING DECLARATION OF ...
MISSING SPECIFICATION OF ...
MISSING PROCEDURE/ARRAY IDENTIFIER
THIS IS NOT PROCEDURE OR ARRAY IDENTIFIER: ...
THIS IS NOT CLASS IDENTIFIER: ...
UNIMPLEMENTED FEATURE USED
NOT REFERENCE BEFORE DOT
NO ATTRIBUTE CALLED ...
RUN-TIME REGISTER SHORTAGE
CANNOT EVALUATE ...
MISSING FOR ASSIGNMENT
MISSING DO
ILLEGAL GOTO
WRONG NUMBER OF PARAMETERS TO ...
UNDEFINED OPERATION
ILLEGAL USE OF QUA
MISSING LITERAL EXPRESSION
ILLEGAL EXPRESSION FOR LITERAL
NO EXPRESSION AFTER ASSIGNMENT
MISSING THEN - EXPRESSION
MISSING ELSE - EXPRESSION

MISSING OPERAND
EXTRA (
MISSING)
MISSING CLASS ID
MISSING REMOTE ID
MISSING PARAMETER
SERIOUS SYNTAX ERROR - REST OF STATEMENT IGNORED
GARBAGE IN VIRTUAL LIST
MISSING IDENTIFIER
TYPE ON SWITCH/LABEL SPECIFICATION
ILLEGAL VIRTUAL SPECIFICATION
DOUBLE MODE SPECIFICATION FOR
DOUBLE TYPE SPECIFICATION FOR
SPECIFICATION ILLEGAL FOR CLASS
SPECIFICATION FOR NON-EXISTENT PARAMETER ...
GARBAGE IN PARAMETER SPECIFICATION
MISSING COLON AFTER VIRTUAL
NO VIRTUAL SPECIFICATIONS GIVEN
GARBAGE AFTER BODY
MISSING ARRAY BOUND
MISSING BOUND DELIMITER
MISSING RIGHT BRACKET
GARBAGE AFTER DECLARATION
INCOMPATIBLE TYPES IN EXPRESSION, ASSIGNMENT, OR PARAMETER
INCONSISTENT PARAMETER LISTS TO VIRTUAL PROCEDURE
MISSING EXPRESSION IN ACTIVATE STATEMENT
WRONG NUMBER OF SUBSCRIPTS TO ARRAY ...
(INTERNAL, WARNING ONLY) LEFT OVER TEMPORARY ...
MISSING := AFTER SWITCH
MISSING TO
MISSING KEY WORD: PROCEDURE/CLASS
ILLEGAL THIS
LARGE INTEGER CONSTANT CONVERTED TO REAL
NO SPACE IN BLOCK OBJECT FOR THE VARIABLE ...
NO SPACE IN BLOCK OBJECT FOR ALL THE TEMPORARIES THAT ARE NEEDED
THIS IDENTIFIER IS UNACCESSIBLE IN QUICK PROCEDURE: ...
ILLEGAL ASSIGNMENT TO PROCEDURE IDENTIFIER ...
PROGRAM WAS TOO LARGE FOR COMPILER STORAGE - COMPILATION ABORTED
ILLEGAL SYNTAX FOR VARIABLE
I DO NOT KNOW OF ANY IN-LINE PROCEDURE ...
CONSTANT DATA AREA WAS TOO LARGE TO BE LOADED
PROGRAM AREA WAS TOO LARGE TO BE LOADED
TEXT VALUE CONSTANT NOT ALLOWED IN == RELATION
ILLEGAL USE OF ARRAY IDENTIFIER ...
ILLEGAL ASSIGNMENT TO EXPRESSION OR PROCEDURE
ILLEGAL REDECLARATION OF PREFIX TO EXTERNAL CLASS
STANDARD PROCEDURE IDENTIFIER IS NOT ALLOWED AS ACTUAL PARAMETER
MISSING SEMICOLON BEFORE INNER
LIBRARY FILE DID NOT CONFORM TO SIMULA BINARY FORMAT
EXTERNAL CLASS/PROCEDURE DOES NOT APPEAR AS SUCH ON THE LIBRARY FILE
LIBRARY FILE CONTAINS EXTRA DEFINITION OF ...
NO LIBRARY FILE DEFINES ...
UNKNOWN LANGUAGE IN EXTERNAL DECLARATION
MISSING TYPE AFTER SHORT OR LONG
ATTEMPTED DOT ACCESS ON CLASS WITH LOCAL CLASSES

TOO MANY PARAMETERS TO ...
THIS PROGRAM HAS EXTRA BEGIN OR MISSING END
TOO LONG IDENTIFIER
EXTERNAL MODULE HAD COMPILATION ERRORS
EXTERNAL MODULE COMPILED WITH OBSOLETE SIMULA RELEASE
SHORT/LONG HAS NO EFFECT IN THIS IMPLEMENTATION
PROGRAM AND DATA AREAS TOO LARGE FOR BPUN FORMAT
SYMBOL TABLE TOO LARGE - TRY SMALLER >DATA PARAMETER

Some of these messages may be obsolete, thus impossible to evoke.
There are a few warning messages included in the list; the message is
then accompanied by "WARNING:" rather than "ERROR:".

Appendix D: Run time error messages

The following messages can be given at run time:

Illegal parameter to LN
Illegal parameter to EXP
Illegal parameter to SQRT
Illegal parameter to ARCTAN
Illegal parameter to SIN or COS
Illegal parameter to SINH or COSH
Illegal operands to **
No numeric item found in text
Illegal number syntax
Illegal parameter to editing or de-editing routine
Ref before dot or qua was equal to none
Left side of text assignment has too short text object
No match to virtual attribute in this class
Parameter to CALL or RESUME was terminated
Parameter to CALL or RESUME was already operating
Too large array declared (max. is 49152 machine words)
You have exchanged lower and upper bounds in array declaration
Illegal goto from detached object or to inspected object
Array index out of range
Object in reference assignment or qua was of wrong class
Too small field for editing
Wrong number of parameters to formal or virtual procedure
Actual parameter to formal or virtual procedure is of wrong kind
Actual parameter to formal or virtual procedure is of wrong type
Sorry, we just ran out of memory space for your data
Wrong number of actual dimensions in array parameter
MORE is false in GETCHAR
MORE is false in PUTCHAR
SUB parameter(s) out of range
No numeric item found in text
File not closed at calling OPEN
Requested file is busy or non-existent
File already closed at calling CLOSE
File not open at calling SAVE
File not open at INIMAGE
ENDFILE is true at INIMAGE
Image of infile is too short
Lastitem is true at ININT
Lastitem is true at INFRAC
Lastitem is true at INREAL
File not open at OUTIMAGE
File not open at LOCATE
Wrong physical directfile image length
Too large directfile image

File not open at EJECT
Zero parameter to LOCATE
Attempt to passivate last process
Evttime called for idle process
Reactivate caused passivation of last process
DRUM ALAS
Parameter to BLANKS was negative or greater than 8186
Result of real to integer conversion was too large
Unimplemented feature used - please report
Run time system error detected - please report
Detach called for non-operating class object
ERLANG parameter(s) zero or negative
HISTO: First array must be longer than second array
Wrong parameters to LINEAR
Error in FORTRAN subroutine or function
Illegal parameter to isochar
Arithmetic operation caused overflow
Parameter to CALL or RESUME was none
Attempted assignment to constant or expression
Unsuccessful input/output request
Attempt to activate terminated process

Appendix E: Extended language

When option Z is set, certain extra features are available. For this purpose, the set of reserved words is extended by:

ENTRYPPOINT INLINE LITERAL QUICK XOR

It is strongly emphasized that the "Zimula" language is only available under the TPH SIMULA compiler. Thus, any program using this language is not portable. The extra features of Zimula are described in the following paragraphs.

Literal declaration

The set of legal declarations is extended by <literal declaration>, defined thus:

```
<literal declaration> ::= literal <literal list>;  
<literal list> ::= <literal item> /  
                  <literal item>, <literal list>  
<literal item> ::= <identifier> = <expression>
```

Each <literal item> has the following semantics: The <expression> may be of any type, but it must contain constants only. The <identifier> becomes a symbolic constant. It has the usual scope of normal identifiers, with one important restriction: It is only referenceable after its definition, i.e. in textually succeeding source code. According to this, one may e.g. define a set of symbolic codes thus:

```
literal atype=1, btype=atype+1, ctype=btype+1, dtype=ctype+1;
```

The main justification for literals is the advantage in code efficiency over the only other method of generating symbolic constants: Declaration and initialization of variables. The improvement in readability is of course also of some value.

The xor operator

The xor is a logical operator between Boolean expressions. Its result is the exclusive or of the operands, i.e. true if and only if they are different.

Quick procedures

The quick procedure is a procedure which is callable 12 times faster than the ordinary one. To achieve this, certain restrictions apply:

The procedure object is limited in size to 32 words, the parameters are limited to 3 words (one real, or 3 integer/ Boolean/ character/ ref/ array), no name parameters are allowed, and the quick procedure body may only access variables local to the surrounding block. Inspection is not allowed, and one may not call non-quick procedures.

The quick procedure is declared by preceding an ordinary procedure declaration by the keyword quick.

Inline procedures

The inline procedure is only for run time system use. See internal documentation.

Entrypoint procedures

When Z option is set, E option is necessary to signal a separate compilation. In this case, the keyword entrypoint must precede any declaration that is to be known outside of the module. If specifying both entrypoint and quick for the same procedure, entrypoint must precede quick.

Appendix F: Non-Simula procedures

This section describes non-Simula procedures. Such procedures are called at the user's own risk. Especially those coded in assembly code should only be created by experienced system programmers.

Fortran coded procedures

To use a Fortran subroutine or function as a procedure in Simula, let the Fortran compiler compile the Fortran module in reentrant mode (to make it compatible with Simula data allocation), then specify the resulting BRF file in a >LIB command to the Simula compiler. In the calling Simula program, the Fortran procedures are declared thus (those with type are functions, others are subroutines):

```
external Fortran integer procedure ifunc,jfunc;  
external Fortran real procedure r;  
external Fortran procedure psubr,qsubr;
```

Due to incompatibilities in storage allocation, the Fortran procedures must not use READ, WRITE, or FORMAT statements. If the user should discover other restrictions, we would be happy to receive an error report.

Assembly coded procedures

An external assembly coded routine with fixed parameter list is declared with all parameters and a body without statements. The source language is specified as "library". Examples:

```
external library procedure extrick;;  
  
external library Boolean procedure comp(a,b);  
  character a,b;;  
  
external library class help; begin  
  integer alfa,beta;  
  end;
```

Thus the syntax differs from a normal class or procedure by the leading "external library" only.

An assembly coded quantity must conform to the Simula BRF format (as produced by the Simula compiler itself) and appear in a >LIB command. Users are strongly discouraged from attempting to produce such files.

The assembly code for the body is subject to the following rules:

- It is headed by a label equal to the procedure or class name.
- It is terminated by a JPL I (EXIT& instruction. Note that this makes it hard to produce by means of the MAC assembler.
- It must not disturb the B register, which points to the procedure or class object.
- Its return, parameter, and local variable cells are found starting at address RT*BODY (see later definition) relative to the B register. For example, the standard procedure "rank" could have the following assembly code:

```

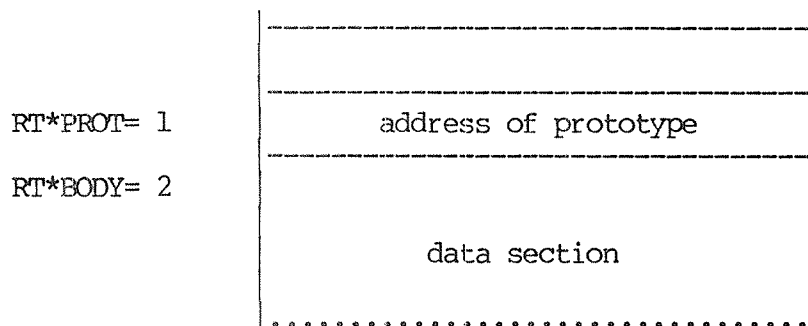
RANK,      LDA      RT*BODY+1,B
           STA      RT*BODY,B
           JPL I    (EXIT&
)FILL

```

For proper automatic inclusion at compilation time, the assembly body must reside on a BRF formatted file with the necessary external reference and entrypoint definition specifications. The file must be mentioned in a >LIB command prior to >COMPILE (see ch. 7).

We will now give the necessary technical information on the run-time data structures. The user is warned that all definitions are subject to change without notice. According to good programming practice, you should always reference given displacements by symbols, never by numbers. Cells that are irrelevant to users have been left blank, thus avoiding unnecessary complicated diagrams.

An object of a block, procedure or class has this format:



The data section contains parameters in order of parameter list appearance, followed by local variables in order of declaration. A parameter by name always occupies one word. For other parameters and local variables, we have the following space requirements:

array of any type: 1 word

single variable:

<u>ref</u>	1 word
<u>integer</u>	1 word
<u>Boolean</u>	1 word
<u>character</u>	1 word
<u>real</u>	3 words
<u>text</u>	4 words

The text reference is formatted as

TX*VAL=	0	pointer to text object
TX*POS=	1	byte address of pos
TX*START=	2	byte address of 1
TX*STOP=	3	byte address of length+1

Since TPH SIMULA always packs character and Boolean arrays, the text object does not need to have a special format. It is simply stored as a character array. The byte addresses TX*POS, TX*START, TX*STOP are all given in bytes relative to the word address kept in TX*VAL.

An array object has the following format, where n = total number of bytes if character or Boolean, total number of words otherwise:

RT*ABASE=	0	bias
RT*NODIM=	1	m = number of dimensions
RT*NOWO=	2	n
RT*ABODY=	3	
RT*ABODY+m-1	
		data section
n-1 (if words)	

It will be seen from the above that the data section of a one-dimensional array starts at RT*ABODY. The data section contains variables as described for block objects. The elements are laid out by ascending indices, the last index being incremented first.

To get the effective displacement within the array object, the bias is added to the product of index and element size (if multidimensional, things get more complicated). In this way you may calculate the lower bound (for $m=1$) as: $(RT*ABODY-bias)/elementsiz$.

Finally, we give the format of a prototype:

RT*NOPAR= 0	number of parameters
RT*SIZE= 2	total length of objects
RT*PREFIX= 4	address of prefix prototype
RT*PDATA= 7	data descriptors
	0

The prefix address is zero if the prototype is not prefixed. Each data descriptor is confined to one or two non-zero words, formatted as follows:

Bit no.:	15	13	11	9	0
	type :kind:mode:relative address				
				

The second cell is present if and only if type is ref. Although some of this information is irrelevant to users, the full set of codes is listed below.

Type:	0	<u>Boolean</u>	Kind:	0	simple variable
	1	<u>character</u>		1	<u>array</u>
	2	<u>integer</u>		2	<u>procedure</u>
	3	<u>real</u>		3	<u>class</u>
	4	<u>ref</u>	Mode:	0	<u>local variable</u>
	5	<u>text</u>		1	parameter by <u>value</u>
	6	(untyped)		2	parameter by <u>reference</u>
	7	<u>label</u>		3	parameter by <u>name</u>

Again, if you insist on using any of these codes, you are in for trouble unless you give them symbolic names. Good luck!

Appendix G: External procedure library

The TPH SIMULA run time system contains a number of non standard library routines. To use any of these routines, simply declare it as an external procedure. The following sections describe their functions. Each procedure is described by its Simula definition, followed by a short explanation.

Operating system interface

Procedures mentioned in this section are all extensions made for communication with the operating system. They should not be used if portability is to be retained, though conditional compilation (ch. 2) may solve that problem. Knowledge of the operating system is assumed when necessary.

procedure Sintran(t); text t; ;

The text t must contain a legal command to the operating system. The leading @ must not be present. All parameters must be given, to avoid Sintran's asking for more. NOTE: If the command is wrong, there is no way of receiving an error status. Presently, Sintran in that case will require an extra blank line to be typed in from the terminal. Thus, the programmer should make sure commands are always correct.

real procedure timeused; ;

Gives, in seconds, CPU time used by this program up to now.

text procedure date; ;

Returns a new text object of 8 characters, formatted yy-mm-dd where yy is the current year modulo 100, mm is current month, and dd is current date.

integer procedure clock; ;

Returns an integer that, if edited as 4 decimal digits, gives hhmm where hh is current hour and mm is current minute.

procedure timewait(r); real r; ;

Suspends execution of the program until r seconds of wall clock time have passed.

```
integer procedure getfile(t,a);  
  text t; integer a; ..... ;
```

The parameter t is the name of a supposedly existing file. The parameter a is the access mode as defined in (3). The default type of the file is DATA. If the file is available for opening, the procedure opens it and returns the logical unit number (see operating system's manual), which is always positive. If not available, a negative error code is returned.

```
integer procedure getexfile(t,type,a);  
  text t,type; integer a; ..... ;
```

Same as getfile, but the default type is given in "type", which must contain ' as its last character.

```
procedure relfile(n); integer n; ..... ;
```

A close request on logical unit n is made to the operating system.

For example, a file called SEMAPHORE:FLAG can be waited for, opened, and later closed:

```
integer i;  
for i:=getexfile("SEMAPHORE","FLAG'",4) while i<0 do  
  timewait(1);  
:  
:   (critical program section)  
:  
relfile(i);
```

This ensures that several jobs in the system may run the same program, even though the critical program section should not be executed by more than one job at any point of time.

Arithmetic routines

This section describes some special manipulations upon simple variables. Most of them are machine dependent and should not be used for portable programs.

```
character procedure upper(c); character c; ..... ;
```

If c is one of the lower case letters or special characters, upper will return the upper case equivalent. Otherwise c itself is returned.

```
Boolean procedure checkset(i,mask);  
  integer i,mask; ..... ;
```

Returns true if the logical AND between the two parameters is non-zero.

```
integer procedure bitset(i,n);  
  integer i,n; ..... ;
```

Bitset returns the value of i after setting bit no. n. Bits are numbered from right to left, starting on zero. There are 16 bits in the NORD-100 integer.

```
integer procedure shift(i,n);  
  integer i,n; ..... ;
```

Shift returns the value of i as shifted to the right, zero filling, n places.

```
integer procedure iand(a,b);  
  integer a,b; ..... ;
```

Iand is the result of a logical AND between a and b.

```
procedure longidiv(quot,rem,i,j,div);  
  name quot,rem;  
  integer quot,rem,i,j,div;  
  ..... ;
```

An integer division between i*j and div is made. The result goes to quot, the remainder to rem.

```
integer procedure ralb(a); array a; ..... ;
```

Ralb returns the lower bound of the one-dimensional real array a.

```
integer procedure raub(a); array a; ..... ;
```

Raub returns the upper bound of the one-dimensional real array a.

```
integer procedure caub(a);  
  character array a; ..... ;
```

Caub returns the upper bound of the one-dimensional character array a.

real procedure arctan2(a,b); real a,b; ;

Same as arctan(a/b), but returning the angle in correct quadrant.

text procedure tricktext(a,b,c,d);
integer a,b,c,d; ;

The user is not advised to call this routine. It will build up a text reference from the four given parameters, formatted as shown in app. F, but if such a reference would exist for some time with a wrong address, the garbage collector would feel very bad about it.

real procedure trickreal(a,b,c);
integer a,b,c; ;

Trickreal builds up a real variable from the three given parameters. It is not very interesting for a user, and one should indeed avoid creating a non-normalized real variable -- this would give very strange results.

Run time system interface

The run time system maintains a list of 15 options, each identified by a letter. The range is A to O. They can only assume Boolean values. Three procedures are available for checking or changing these options during execution:

Boolean procedure rtopt(c);
character c; ;

The value of the option identified by c is returned.

procedure rton(c); character c; ;

The option c is set to true.

procedure rtoff(c); character c; ;

The option c is set to false.

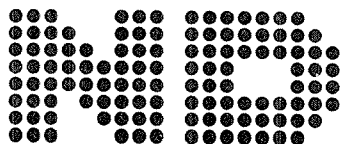
In certain very special cases, it might be desirable to force a garbage collection to occur, triggered by the user program. Note: In most cases, use of this feature would only make things worse! The mechanism for this is:

procedure collect; ;

Other utilities

```
integer procedure tablook(A,n,t);  
  text array A;  
  integer n;  
  text t;  
  ..... ;
```

The array A, which should have elements from 1 to n, contains a table of command names (for example). Tablook will try to find a match for t in this table, returning the index of the match. The parameter t may be abbreviated as long as it is not ambiguous. If ambiguity exists, -1 will be returned. If no match could be found at all, 0 is returned.



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

COMMENT AND EVALUATION SHEET

ND-60.092.03

Nord-100 Sintran III Simula
Reference Manual

In order for this manual to develop to the point where it best suits your needs, we must have your comments, corrections, suggestions for additions, etc. Please write down your comments on this preaddressed form and mail it. Please be specific wherever possible.

FROM

.....

.....

– we make bits for the future