TPH SIMULA

REFERENCE MANUAL

REVISION RECORD

Revision	Notes
9/77	Original Printing
8/78	Version two

ND-60.092.02

August 1978

NORSK DATA A.S.

Lørenveien 57, Postboks 163 Økern, Oslo 5, Norway

Norsk Data A.S. presents

PROGRAMMING SYSTEM FOR THE SIMULA LANGUAGE
NORD-10 SINTRAN III VERSION REFERENCE MANUAL

This manual reflects the software as of version 3.54 of TPH SIMULA, the April 1978 release.

Norsk Data A.S.
Lørenveien 57
OSLO 5
Norway

ND 60.092.02

Contents

Preface

This implementation of the SIMULA language for the NORD-10 minicomputer has been performed by mr. G. P. Philippot of TPH Data A.S. All marketing rights for the NORD-10 have been transferred to Norsk Data A.S. in agreement with TPH Data A.S.

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 in 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.

All questions and responses concerning the NORD-10 SIMULA language should be directed to Norsk Data's Marketing or Customer Support departments.

1. Introduction

TPH SIMULA is an implementation of SIMULA, a general-purpose high level 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 posess 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-10 users.

1.1. Nord-10 version

The version described here compiles and executes programs on the NORD-10 computer. The compiler occupies 21K words of program plus dynamically allocated data, minimum 2K. The program consists of several independent phases, varying from 5 to 10 K 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-10 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. 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 core, 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

For easy and efficient definition of symbolic constants, the literal declaration has been introduced to the hardware notation. See section 3.4.

External classes and procedures (in Simula 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.

Index to a switch is always checked against the bounds, causing a run-time error if violated.

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

Run-time checks for array bounds, qualifications, 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

Only the first 24 characters of an identifier are registered. Apart from this, it may have any length up to the end of line.

Source lines have a maximum length of 120 characters. Excess characters are ignored, as well as non-printable characters. 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 -10^{4920} to -10^{-4920}, exact zero, and from 10^{-4920} to 10^{4920}. They have 10 significant digits.

All matches to a virtual procedure should have the same number, types, and modes of parameters. This restriction is for efficiency; violation will slow down the procedure calls.

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. 3000 lines of source text. The problem is under investigation. In some cases, however, the user can use external compilation to split his program. The combined program size capacity is estimated to be over 4000 lines, a figure which will increase to 8000 when the full 128K addressing technique has been employed.

An array object may occupy a maximum of 4096 words total. Allowing for some overhead, this gives approx. 4000 integer elements, 1300 real elements, or 1000 text elements.

A block object (procedure, class etc.) may only occupy 128 words at present. Since this is a fairly narrow margin, the size will be increased to 512 words in a later release. A compiler error message is given if the maximum space is exceeded. Note that the program code allocates temporary calls 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:PROG	Executable program file for the compiler
SIMUPRE3:SYMB	Simula part of the run-time system
SIMRT3:SIM	Assembly part of the run-time system
SIMERR:DATA	Error messages for use by run-time system
SIMBASE3:SIM	Binary relocatable library
SIMSET3:SIM	Binary relocatable version of SIMSET
SIMULA3:SIM	Binary relocatable version of SIMULATION

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 the level and the edition, e.g. 3.54.

For efficient use on a Nord-10, 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 64K 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.

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. The simple definition by column 1 of each line has been made in order to save time spent in precompiler when most of the program does not use it.

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 true. 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 false. 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 -32767 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 of 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 scan, 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 scan, 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 console (not recommended), %EOF is the only way of terminating the source text.

2.5. Alternate source files

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

%COMPILE filename

subject to suppression by flags if specified. The filename may be replaced by a logical unit 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, excepting the operating system's possible limit to number of simultaneously opened files.

3. Source program

Your program is to be supplied as lines of maximum 120 printable ASCII characters each. 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 embedding 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.

To include the DEC notation, a special alternative for comments has been allowed: ! outside of a comment, character, or text constant is accepted as 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 represented in 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 tnese have several alternative representations in TPH SIMULA; some of them may even be represented by reserved words. Here is the set:

Common Base	TPH SIMULA (all alternatives given on the same line)
=	= 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. NOTE! The otherwise compatible hardware notation for UNIVAC disagrees on this point, i.e., the underscores are ignored. The maximum recognizable identifier length is 24, but any length may be used in the source program.

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 other purposes than those specified in Common Base or in this manual:

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

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.

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.

3.3. Constants

Some constants are represented by reserved words - true, false, none, and notext. Known from Common Base, they are not covered here. Instead, we will explain the notations for numeric and textual 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 0.0003 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 -10^{4920} to -10^{-4920}, exact zero, and from 10^{-4920} to 10^{4920}. They have 10 significant digits.

3.3.2. Textual 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. 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 set as for 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 on text constants:

Source code	Result
"Abc"	Abc
"Quote: """	Quote: "
"Part one, " "part two"	Part one, part two

3.4. Literal declaration

This section describes an extension to the hardware notation, available under TPH SIMULA only. It should be ignored if future transportation is to remain a possibility. 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.

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 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. It is a means of having the compiler generate a prototype for a routine whose code is external and usually assembly coded. No responsibility is taken for the results or disasters of such routines.

FORTRAN- or COBOL- type routine calls are not supported.

One or more external quantities for complete substitution are declared by the keywords external Simula, 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 Simula procedure operate;

external Simula procedure abool,bbool,cbool;

external Simula class system;

In addition, within a block or class that has no prefix, any identifier that has been used for prefixing without being ever declared at that block or class is implicitly declared external Simula 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 Simula 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 an E-option compilation. See sec. 4.2 for description. Upon finding the requested quantity name, the compiler reads all necessary information concerning 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 may invalidate 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 option E is set during compilation, no absolute program is produced. Instead, the specified BRF file becomes a valid user library. The source text for an E compilation usually consists of one main block containing one or more entrypoint declarations and no statements. The entrypoint declaration is any ordinary procedure or class declaration preceded by the keyword entrypoint. This makes the quantity, its prefix/type, and parameter list 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. Only a quantity which is itself marked entrypoint is excepted from this recursive tree of definitions, such a quantity and its tree being emitted separately with its own identification. Example showing the most usual user library definition:

```
begin
    entrypoint 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;
```

```
        end;
```

The main program prefixed by "prefix" now has access to a, b, c, p, cl, and by remote access 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 use of an external class on a block level other than that of its E-compilation should be avoided. If not, the run-time level changes cause about 50% increase in execution times.

4.3. Assembly code in cooperation with Simula

This section is for experts only. It is included in the manual at this point because the mechanisms involved are similar to those of external Simula procedures and classes. The average user is strongly advised to skip to the next section.

An external quantity for body substitution is declared with all parameters and a body without statements. Its 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 its leading external library only.

An assembly coded entrypoint quantity must conform to the Simula BRF format 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.

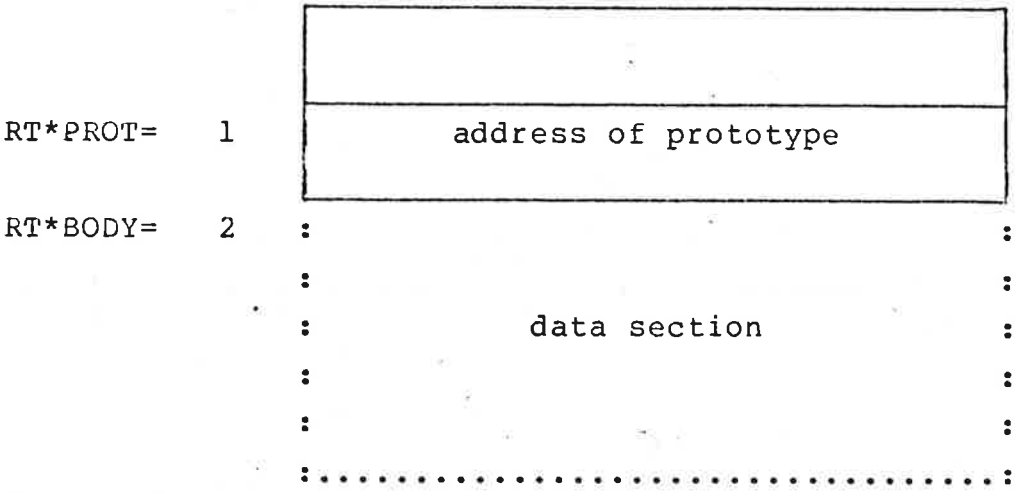
- Its return, parameter, and local variable cells are found starting at address RT*BODY (see later definition) relative to the B register. For example, the standard procedure rank has the following assembly code:

```
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.

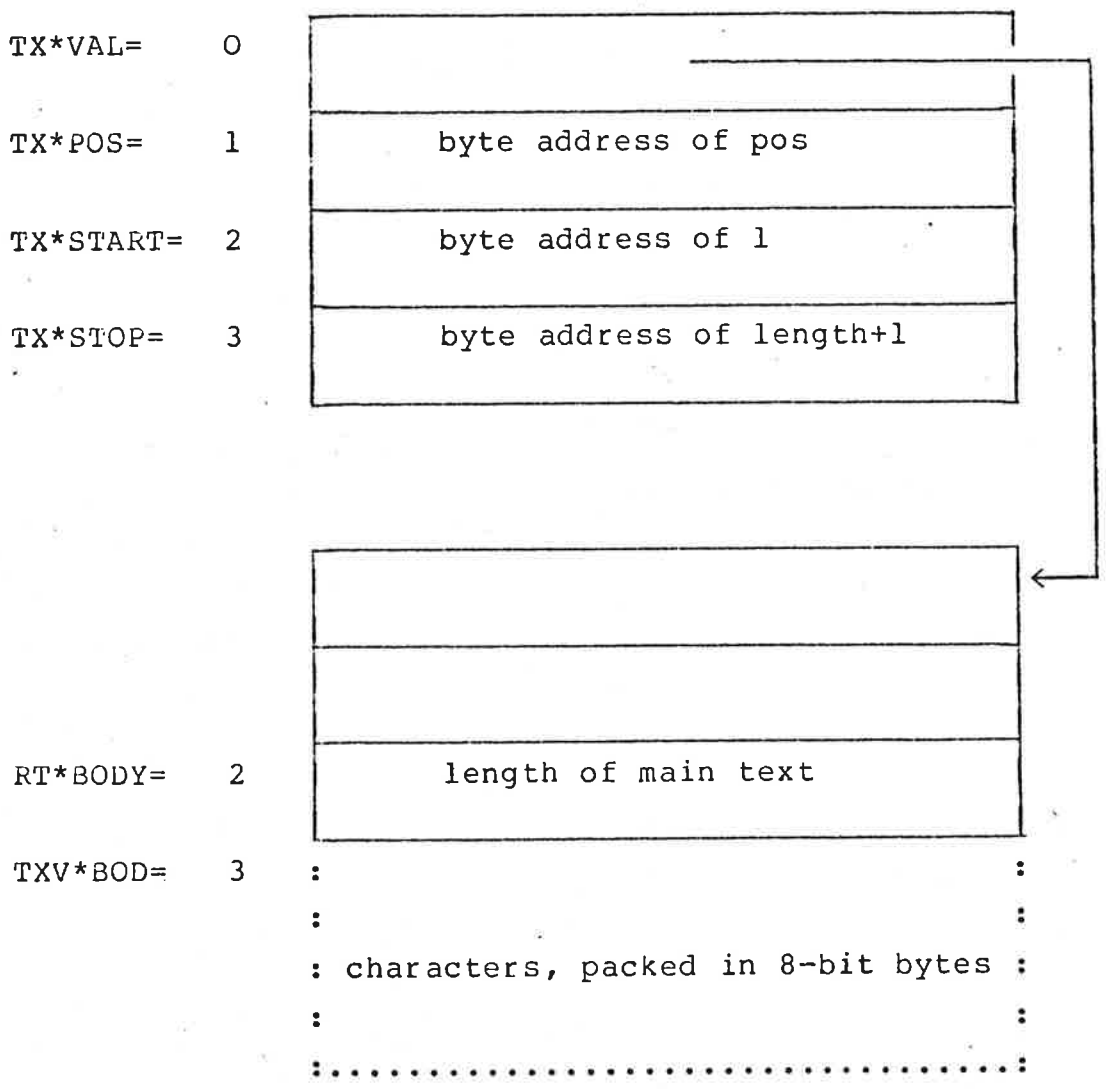
An object of a block, procedure or class has this format:



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 two words. For other parameters and local variables, we have the following requirements:

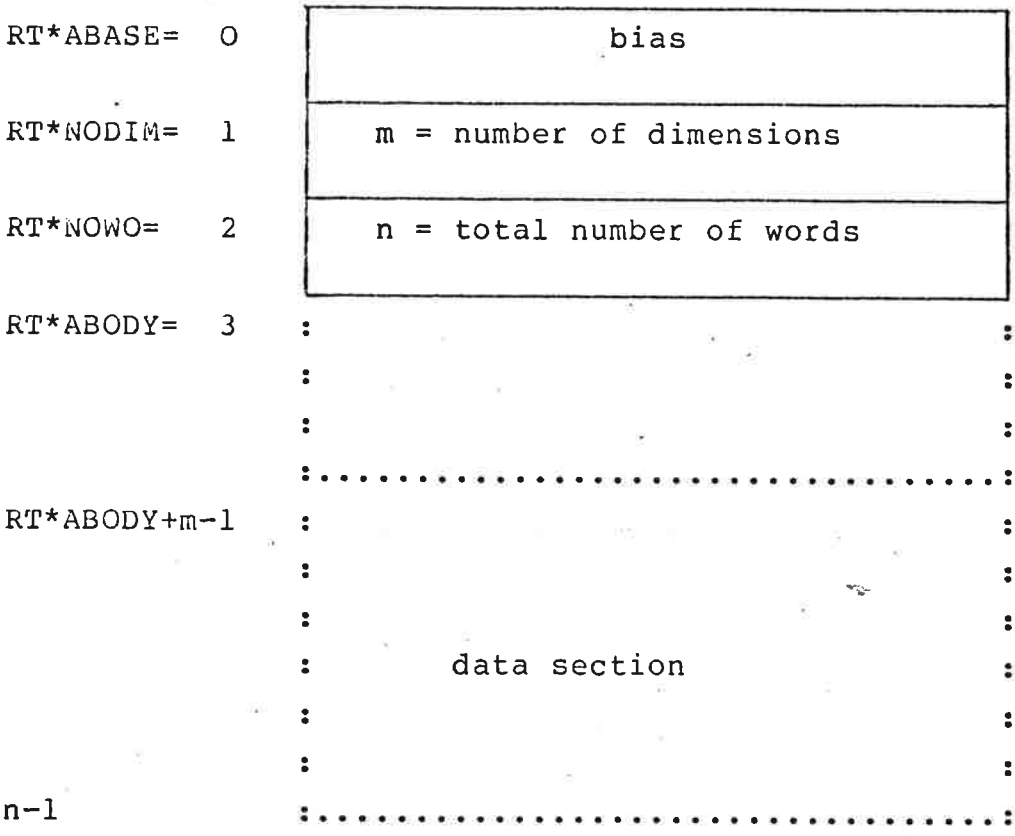
array of any type:	1 word
single variable:	
ref	1 word
integer	1 word
Boolean	1 word
character	1 word
real	3 words
text	4 words

The text reference and object are formatted as



The byte addresses TX*POS, TX*START, TX*STOP are all given in bytes relative to the word address kept in TX*VAL.

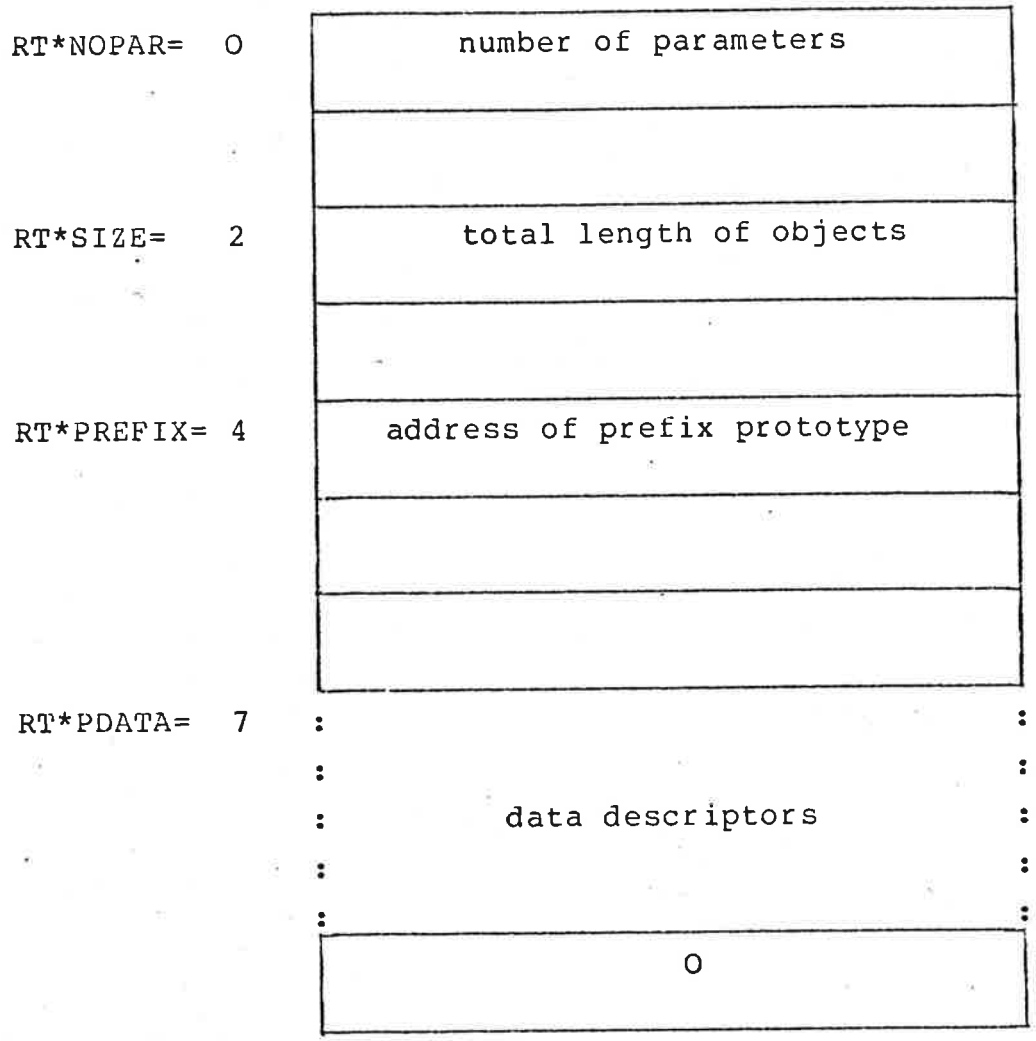
An array object has the following format:



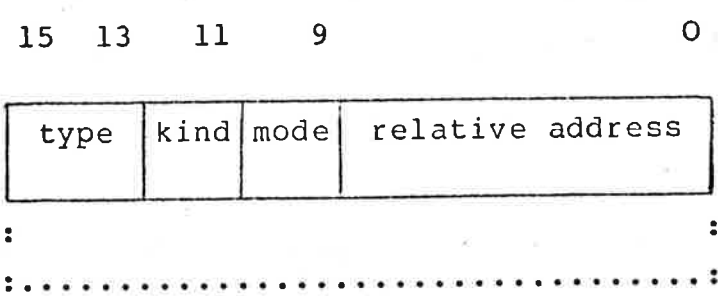
It will be seen from the above that the data section of a one-dimensional array starts at RT*ABODY. The data section contains variables as described for block objects. The elements are laid out by ascending indices, the last index being incremented first.

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)/elementsize.

Finally, we give the format of a prototype:



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:



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

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

4.4. 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 same type and transmission mode, and of course the number of parameters has to be equal.

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

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 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.

At file object generation time, NAME is allowed to have two syntactically different values:

- If its first non-blank character is a digit or a minus sign, the value will be interpreted as an octal logical unit number (lun) of a file already opened via the operating system.

- In the other case, NAME should represent a legal file name of the directory, accessible to this user and not yet opened via the operating system. It will

automatically be thus opened at the time of calling the open procedure. At close time it is again closed, contrary to the lun specified case where user is responsible for both open and close versus the operating system.

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 lun, 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 lun (in octal, signed) is given. This feature is useful if several file objects are to access the same lun, and the Simula system is supposed to open it as described above.

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; a directfile must be random accessible unless used in a purely sequential manner. 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, 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 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.

5.2. SIMSET

The system class SIMSET is an external Simula class implicitly available from a system library. When used for prefixing, 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 have SIMSET declared local to them, 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 (or prefixing the entire program), 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 as to added procedures or other implementation defined items. A complete list of standard procedures and classes appears in appendix B.

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. Attempt to make it operating is an error. Otherwise, the terminated objects is similar to a detached one.

The procedure "detach" is designed to operate on the nearest statically (textually) enclosing class or prefixed block (possibly inspected). 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 rectivation 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.

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 starts executing following the last detach (or resume) that the object made, and the next detach of the object causes return.

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

6.2. Arithmetic and conversion

Rank converts from character to integer and may give values from 0 to 127. Char converts the other way; its integer parameter must be of value 0 to 127. Currently if parameter to char is out of range, no message occurs but the extra bits are masked off to make a legal character result. There exists an extra conversion procedure

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

returning c unless c is in the lower case range (greater than _), in which case the upper case equivalent of c is returned.

6.3. Random draws

According to Common Base, the parameter U to random drawing procedures is of type integer. On a 16 bit computer, this gives very poor random streams. To obtain better draws, use a real variable initialized within the range 0 to 1, and call the extra set of procedures which have an appended r to their identifiers. For example, randintr is used instead of randint. Each procedure has to be declared external Simula procedure.

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

6.4. 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. Being non-standard, they all have to be declared external Simula procedure. Each procedure is described by its Simula definition, followed by a short explanation. Knowledge of the operating system is assumed when necessary.

```
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(i); integer i; ..... ;
```

Suspends execution of the program until i*0.1 seconds of wall clock time have passed. For NORD-10 execution, the maximum value of i is 32767.

```
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 that file is available for opening, the procedure opens it and returns the logical unit number (see operating system's manual). If not, 0 is returned.

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

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

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

```
integer i;
for i:=getexfile("SEMAPHORE",4)
     while i=0 do timewait(10);
:
:    (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.

6.5. 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 a run-time error message is given. As this action is not in accordance with Common Base, it will be changed to an asterisk fill and a warning after program termination. Information on this will follow the new release.

The exponent given by putreal has the form "&+nnnn", i.e. four digits are needed because of the wide exponent range of the Nord-10 hardware.

7. Compilation and execution of programs

Having called the compiler by the @N10-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 last end, the CPU time used 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.

7.1. Elementary compile and execute procedure

This section is a quick introduction to 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 certain macro processing facilities. 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-ins underlined):

```
@N10-SIM
TPH SIMULA 2.00140
>SETOPT VS
>LIST L-P
>COMPILE PROGRAM
```

Your listing should now go to the printer. If errors were found, leave the compiler by >EXIT, then 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 will take a minute or so, 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 >BIN BINSIM (say) prior to >COMPILE PROGRAM. The first execution can proceed as usual, while subsequent runs should be started thus:

```
@PLACE BINSIM:BIN
@GO 0
```

The starting address of the Simula program is 0 (zero). The program placed in memory by @PLACE may also be saved for later executions by the SINTRAN III @DUMP command. By typing the file name given in the @DUMP command as a command to the operating system, the program starts execution directly.

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. Console command language

Currently, this set of commands is implemented:

>SETOPT	cccc	Increment all compiler options mentioned in cccc. Option is set when value is greater than zero. Initial value is zero. See next section for list of options.
>RESOPT	cccc	Decrement all compiler options mentioned in cccc.
>RTS	dddd	Set to true all run-time system options mentioned in dddd. Run-time system options can only have the values true or false. See section 7.4.3 for a list of options.

>LIB	lun	Logical unit lun, 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. %SIM
>ASM	lun	Specifies that object code (in symbolic assembly) go to logical unit lun. This is mainly for testing purposes.
>BRF	lun	Specifies that binary relocatable code go to logical unit lun. This is for use when compiling external modules, though even a main program can be saved as BRF code (and loaded by >LOAD). %SIM
>BIN	lun	Specifies that binary absolute code (in the standard NORD-10 hardware format) go to logical unit lun. This is the recommended way of saving the object code. %BIN
>LIST	lun	Specifies that listing go to logical unit lun. The listing includes error messages and byte output, where appropriate.
>RUNOFF	lun	Specifies that documentation output go to logical unit lun. Such output is generated instead of compilation if option M is set at >COMPILE time.
>COMPILE	lun	Do full compilation according to current options. Source is from lun; default source is the terminal. (In batch jobs,

the batch input file.) 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 lun — Binary relocatable code from lun is loaded, and if V option, executed. :SIM

>EXIT — Return to job monitor.

Wherever "lun" is specified above, an existing file name may be used instead. It will be opened automatically, and closed at >EXIT time. The following default file types are used:

>LIB	SIM
>ASM	SYMB
>BRF	SIM
>BIN	BIN
>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
C	Cross-reference listing*
D	Debugging symbol table produced
E	External compilation
F	Flags for begin/end are abbreviated and given in left margin
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
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 line number table and loader map (primarily useful for system maintenance)

*Not implemented in current release.

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.

Note: Certain messages are only intended for Simula system maintenance, i.e. they show the occurrence of error situations in the compiler itself. Such messages are always preceded by the word INTERNAL. Please submit a complaint to the Customer Support department if you encounter such a message.

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 end of each user statement. The line number is printed on the terminal as follows:

```
LINE n
*
```

One of the following commands may then be given:

*	(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 dynamic chain. For the ordinary user, only the column giving the line numbers is significant. Wandering down this column, ignoring zero numbers, he will see the chain of returns from his currently executing block instance (topmost number) to the main program (bottom number). A maximum of n lines is printed. Default n is 20.
*RESTART	Close all files that were opened by Simula run-time system, reset all system data, and repeat execution of the entire user program, beginning with the message Ready for Simula execution.
*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 a	Dump the object at octal address a on the terminal. The command is a preliminary aid, and a must be correctly specified, or else the run-time system may blow up. Thus a must be found in the "Object" column of *DYN output.

*SYSIN lun	Change the sysin file to lun, which may be a file name or a logical unit number
*SYSOUT lun	Change the sysout file to lun
*HELP	List all of the above commands on the terminal

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 option set

The following option is implemented:

O	On-line debugging mode before program start. The run-time system will print "Pre-program conversation requested.", then LINE O 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-60.050.06)
Norsk Data A.S.

Appendix A. Example on use

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.

Type	Name	Local to	Parameters	Extra
real	abs		real	
real	arctan		real	
real	arctan2		real,real	*
text	blanks		int	
	breakoutimage	outfile		*
	call		ref(any class)	
char	char		int	
Bool	checkset		int,int	*
int	clock			*
	close	file		
text	copy		text	
real	cos		real	
real	cosh		real	
text	date			*
	detach			
Bool	digit		char	
file class	directfile			
char	directin	directfile		*
char	directin	infile		*
	directout	directfile	char	*
	directout	outfile	char	*
int	discrete		real array,int	
Bool	draw		real,int	
	eject	printfile	int	

Bool	endfile	directfile		
Bool	endfile	infile		
real	entier		real	
real	erlang		real,real,int	
real	exp		real	
class	file		text	
char	getchar	text		
int	getfile		text,int	*
int	getfrac	text		
int	getint	text		
int	getoct	text		*
real	getreal	text		
int	histd		real array,int	
	histo		real array,real array,real,	
int	iand		int,int	*
text	id	file		*
char	inchar	directfile		
char	inchar	infile		
int	incommand	infile	text array,int,ref(outfile)	
				*
file class	infile			
int	infrac	directfile		
int	infrac	infile		
	inimage	directfile		
	inimage	infile		
int	inint	directfile		
int	inint	infile		
int	inoct	directfile		*
int	inoct	infile		*
real	inreal	directfile		
real	inreal	infile		
text	inrest	infile		*
text	intext	directfile	int	
text	intext	infile	int	
Bool	lastitem	directfile		
Bool	lastitem	infile		

int	length	file		
int	length	text		
Bool	letter		char	
int	line	printfile		
real	linear		real array,real array,int	
	linesperpage	printfile	int	
real	ln		real	
	locate	directfile	int	
int	location	directfile		
	longidiv		int,int,int,int,int	*
text	main	text		
int	mod		int,int	
Bool	more	file		
Bool	more	text		
real	negexp		real,int	
real	normal		real,real,int	
	open	file	text	
	outchar	directfile	char	
	outchar	outfile	char	
file class	outfile			
	outfix	directfile	real,int,int	
	outfix	outfile	real,int,int	
	outfrac	directfile	int,int,int	
	outfrac	outfile	int,int,int	
	outimage	directfile		
	outimage	outfile		
	outint	directfile	int,int	
	outint	outfile	int,int	
	outoct	directfile	int,int	*
	outoct	outfile	int,int	*
	outreal	directfile	real,int,int	
	outreal	outfile	real,int,int	
	outtext	directfile	text	
	outtext	outfile	text	
int	poisson		real,int	

int	pos	file		
int	pos	text		
outfile class	printfile			
	putchar	text	char	
	putfix	text	real,int	
	putfrac	text	int,int	
	putint	text	int	
	putoct	text	int	*
	putreal	text	real,int	
	putzint	text	int	*
int	ralb		real array	*
int	randint		int,int,int	
int	rank		char	
int	raub		real array	*
	relfile		int	*
	resume		ref(any class)	
	rtoff		char	*
	rton		char	*
Bool	rtopt		char	*
	setpos	file	int	
	setpos	text	int	
int	sign		real	
real	sin		real	
real	sinh		real	
	sintran		text	*
	spacing	printfile	int	
real	sqrt		real	
text	strip	text		
text	sub	text	int,int	
ref(infile)	sysin			
ref(printfile)	sysout			
int	tablook		text array,int,text	
				*
real	timeused			*
	timewait		int	*
real	trickreal		int,int,int	*

text	tricktext		int,int,int,int	*
text	trim	text		*
real	uniform		real,real,int	
char	upper		char	*

Appendix C. Compiler error messages

In the texts shown below, "..." indicates that some identifier is inserted in the message when printed. A leading "INTERNAL" indicates that a Simula system error has occurred, possibly due to other errors. 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 ...
INTERNAL: WANTED IDENTF, GOT ...
UNKNOWN QUALIFICATION: ...
ILLEGAL QUALIFICATION: ...
UNKNOWN OR CIRCULAR PREFIX: ...
INTERNAL: CANNOT FORMTYPE ...
WRONG TYPE, I EXPECTED TO SEE ...
MISSING DECLARATION OF ...
MISSING SPECIFICATION OF ...
MISSING PROCEDURE/ARRAY IDENTIFIER
THIS IS NOT PROCEDURE OR ARRAY IDENTIFIER: ...
UNIMPLEMENTED FEATURE USED

INTERNAL: GENERATE/GENEREF PARA=...
INTERNAL: NO FREE RT REGISTER
INTERNAL: GETSINT FAILURE
NOT REFERENCE BEFORE DOT
INTERNAL: RDETACH ON LOCKED REG
NO ATTRIBUTE CALLED ...
RUN-TIME REGISTER SHORTAGE
CANNOT EVALUATE ...
MISSING FOR ASSIGNMENT
MISSING DO
ILLEGAL GOTO
WRONG NUMBER OF PARAMETERS TO ...
UNDEFINED OPERATION
INTERNAL: GENERATE/GENEREF WILD DISPLAY NO.
ILLEGAL USE OF QUA
MISSING LITERAL EXPRESSION
INTERNAL: LOST LITERAL
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
DOUBLE TYPE SPECIFICATION
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
INTERNAL: WILD INTERPASS 2-3 BYTE
MISSING KEY WORD: PROCEDURE/CLASS
ILLEGAL THIS
LARGE INTEGER CONSTANT CONVERTED TO REAL
*** INTERNAL *** UNDEFINED INTERNAL SYMBOL IN PASS 3 A
INTERNAL: FATAL SINTRAN III INTERFACE ERROR
NO SPACE IN BLOCK OBJECT FOR THE VARIABLE ...
NO SPACE IN BLOCK OBJECT FOR ALL THE TEMPROARIES THAT ARE NEEDED
THIS IDENTIFIER IS UNACCESSIBLE IN QUICK PROCEDURE: ...
ILLEGAL ASSIGNMENT TO PROCEDURE IDENTIFIER ...
LIBRARY FILE DID NOT CONFORM TO SIMULA BINARY FORMAT
EXTERNAL CLASS/PROCEDURE DOES NOT APPEAR AS SUCH ON THE LIBRARY FILE
EXTERNAL CLASS/PROCEDURE NOT FOUND ON ANY LIBRARY
LIBRARY FILE CONTAINS EXTRA DEFINITION OF ...
NO LIBRARY FILE DEFINES ...

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:". Most of the internal error messages can never be evoked by user programming, as they require erroneous

conditions that are possible only in the test version.

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 or negative parameter to LOCATE
Attempt to passivate last process
Evtime called for idle process
Reactivate caused passivation of last process

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 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 4096 machine words)
You have exchanged lower and upper bounds in array declaration
Illegal goto from detached object or to inspected object
Array or switch 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

NORSK DATA A.S.
Lørenveien 57 - Postboks 163, Økern
OSLO 1

COMMENT AND EVALUATION SHEET

ND-60.092.0.2 TPH 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 pre-addressed form and post it. Please be specific wherever possible.

FROM _____

_____

_____