ISIM ESL

ESL Simulation Software

ESL Reference Manual

Document Information

Version: 1.9.3
Date Published: March 2023

This document relates to ESL version 8.3.0

ISIM welcomes any suggestions to improve
the ESL Simulation Software and documentation

If you have any suggestions, or would like to point out
any errors or omissions, please contact us:

ISIM International Simulation Limited

161 Claremont Road
Salford
M6 8PA
UK

Tel: +44 (0) 161-736-5283

Email: info@isimsimulation.com
Web: https://www.isimsimulation.com

Table of Contents

CHAPTER 1

Introduction

1.1 Introduction

This document is the *Reference Manual* for the ESL simulation language and submodel library. Other ESL documents are: *User Guide and Tutorial* which provides a step-by-step introduction to both the graphical interface (ESL-Studio) and how to write programs directly in the ESL language and *Development Guide* which explains in detail how to use ESL.

The ESL simulation language was originally written to meet the requirements of the European Space Agency. It is a general purpose Continuous Systems Simulation Language (CSSL), with a comprehensive supporting software environment, which may be applied in any field where dynamic systems are to be studied.

The ESL software environment provides users with all the facilities to describe a mathematical model; execute the simulation and analyse the results. The core of this environment is the ESL language. It is a comprehensive procedural language extended to address the requirements of dynamic simulation.

ESL programs are presented in standard text files, which may be prepared using any suitable text editor. An alternative to the text file definition is to automatically generate error-free ESL programs using the ESL-Studio graphical user interface. Many models can be expressed in the form of block diagrams and ESL-Studio allows such diagrams to be constructed. ESL-Studio then applies rigorous integrity checking to ensure that an error-free program is generated. Users can then execute the simulation and analyse the results from within the ESL-Studio environment.

Therefore there are two completely different user interfaces to the ESL system: the conventional programming language to specify a simulation; or graphical driven input using ESL-Studio. Users may use either interface, without the need to understand the other, to undertake complete simulation projects. For some projects a mixture of the two approaches is an ideal answer. ESL-Studio allows textual code to be included alongside block-diagram elements. Both routes provide an excellent environment for ensuring integrity of a simulation. During processing, extensive diagnostic checks are made to ensure correct user programs, even to the extent of checking consistent use of all variables, including state, memory and algebraic variables.

ESL is recognised as a *natural model definition language*: the way to unambiguously define a simulation. The characteristics which help justify this claim are: the submodel concept; unambiguous model definition code presented in a modern programming style; the clear definition of non-linearities or discontinuities; full matrix, vector and array slice support; optional Transfer Function notation; linearization features; steady-state finders and, of great importance, the strict rules which are vigorously imposed by the ESL compiler.

To support these concepts ESL provides a very practical solution in the form of an *Interpreter* for fast turn-round of simulation programs under development with excellent run-time diagnostics and facilities, and a *Translator* for efficient production simulation runs. Following a simulation, comprehensive post-mortem graphical analysis can be performed using ESL-Studio (irrespective of whether the simulation was run in ESL-Studio or from the command line).

This manual contains: a formal specification of the ESL language; a list of the submodels contained in the Submodel Library and a summary of the ESL language syntax.

CHAPTER 2

ESL Language Specification

2.1 Introduction

This chapter presents a formal syntax specification of the ESL simulation language. The syntax specification is expressed in Modified Backus Naur Form (MBNF), and this is explained by means of examples and a discussion of the main semantic operations. Readers are referred to the Development Guide for a detailed discussion of the use of various ESL statements and constructs.

2.2 MBNF Syntax Convention

The MBNF syntax definition uses special conventions and symbols to express the syntax of the ESL language. The complete ESL syntax definition consists of a series of statements of the form:

```
syntax element = expression.
```

where *expression* comprises *lexical elements, syntax elements* and *meta-symbols***.**

Lexical elements, or "terminal symbols", (keywords, characters, operators etc..) are presented between double-quotes, for example: "INTEGER", "+", "1". In particular, keywords appear in upper-case characters.

Note: *This convention is used in the formal definition, but the ESL language is insensitive to* the *case of the characters. Therefore the user may type either "INTEGER", or "integer".*

Syntax elements define components of the language, and are shown in lower-case e.g. study_program, variable. The first syntax element of the ESL definition is program.

Meta-symbols**:**

Optional items are shown between square brackets, e.g. ["+"] denotes an optional operator token.

Optional repeated items are shown between braces, e.g. {"+" variable} indicates that "+ variable" may appear any number of times or not at all.

Alternatives:

Where a choice exists, the options are shown separated by a vertical bar, e.g.

```
real | integer | character
```

indicates that one of the three elements may be used.

Factorization:

Parenthesis are used as meta-symbols to group elements, or to factor out the common head of a set of alternatives, e.g.

```
x y | x w | x z
```

could be expressed as:

```
x ( y | w | z )
```

2.3 Lexical Elements

This section describes the basic lexical elements from which all ESL statements are constructed.

ESL statements may extend over several lines. The "end of line" is treated as a space character and lexical elements which must not have embedded spaces (e.g. identifiers, keywords, numbers etc.), must not be broken by line boundaries. Furthermore no lexical element may be split across a line boundary, this includes character strings.

2.3.1 Character sets

The following character sets are used in the ESL language.

```
upper_case_letter =
        "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
        "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
        "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".

lower_case_letter =
        "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
        "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
        "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".

digit =
        "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

special_character =
        """" | "(" | ")" | "*" | "+" | "," | "-" | "." | "/" |
        ":"  | ";" | "<" | "=" | ">" | "_" | "[" | "]".

space_character =
        " ".
```

ESL *special_characters* are used as delimiter tokens, or in expressions, and must not be used in variable naming (with the exception of underscore, "_").

Identifiers, such as variable names or reserved words containing *lower_case_letters* are interpreted as upper case, i.e. *abCD* and *ABCD* are treated as equivalent.

```
other_special_character =
        "!" | "'" | "£" | "$" | "%" | "&" | "?" | "@" | "\" |
        "^" | "{" | "|" | "}" | "~".
```

other_special_characters may be used only in character strings or comments.

2.3.2 Identifiers and variables

The ESL definition of an identifier is:

```
identifier =
        letter {letter | digit | "_"}.

letter =
        upper_case_letter | lower_case_letter.
```

Variables are defined formally in ESL:

```
variable =
        identifier ["(" subscript {"," subscript} ")" ].

subscript =
        expression [".." expression ].

expression                                    [Section: 2.12]
```

Variable names may be any length up to the maximum line length (132 characters), however only the first 28 characters are significant. The first character must be a letter; the remainder may be letters, digits or the underscore character. Note *other_special_characters* and the *space_character* are not permitted in variable names.

Subscripts are used when the variable is an array, to identify a specific element, or an array slice. *Subscript expressions* may be of type integer or real (truncated to integer).

The derivative variable is used to specify a differential equation, and its associated state variable:

```
derivative_identifier =
        identifier """ { """ }.


derivative_variable =
        derivative_identifier
        ["(" subscript {"," subscript} ")" ].
```

2.3.3 Character strings

A literal character string is a sequence of one or more characters enclosed within string bracket characters. The string bracket character may be included in the string if it appears twice as adjacent characters. The format is:

```
character_string =
        """" character {character} """" |
        "%"  character {character} "%".


character =
        letter | digit | space_character |
        special_character |
        other_special_character.
```

letter	[Section: 2.3.2]
digit	[Section: 2.3.1]
space_character	[Section: 2.3.1]
special_character	[Section: 2.3.1]
other_special_character	[Section: 2.3.1]

2.3.4 Keywords

ESL features four classes of keywords:

Keywords declaring a program or subprogram;

Keywords declaring a region;

Keywords used in statements.

These keywords are not permitted in any other context in an ESL program (e.g. as variables, model names etc.), except within comments or text strings.

2.3.4.1 Program and subprogram

The following keywords are used for program module declarations:

MODEL	[Section: 2.6.1]
SUBMODEL	[Section: 2.6.2]
SEGMENT	[Section: 2.6.3]
PROCEDURE	[Section: 2.6.4]
PACKAGE	[Section: 2.6.6]

The keywords:

STUDY
END_STUDY

are required at the start and end of a complete ESL program. ESL code which represents an embedded or remote segment starts with one of the program keywords:

```
EMBEDDED
REMOTE
```

2.3.4.2 Region keywords

The following keywords should appear with no terminator (";"). Each marks the beginning of a region, and termination of the previous region.

```
INITIAL
DYNAMIC
STEP
COMMUNICATION
TERMINAL
ANALYSIS
```

These keywords may only be used in modelling subprograms, Table 2-1 indicates their required and permitted use.

Region keywords, when used, must be entered in the above order, i.e. the STEP region, if present, must be appear before any COMMUNICATION, TERMINAL or ANALYSIS region.

	MODEL	SUB-MODEL	SEGMENT
INITIAL	Optional	Optional	Optional
DYNAMIC	Mandatory	Mandatory	Mandatory
STEP	Optional	Optional	Optional
COMMUNICATION	Optional	Optional	Optional
TERMINAL	Optional	Prohibited	Prohibited
ANALYSIS	Optional	Prohibited	Prohibited

Table 2-1 Sub-program permitted regions

2.3.4.3 Statement keywords

The following keywords are used by ESL in various statement constructs

AND	INCLUDE	REAL
CHARACTER	INTEGER	RESTART
CLEAR_SCREEN	INTERACT	RESUME
CLOSE	IOSTAT	RETURN
CONSTANT	LINEARIZE	REWRITE
CREATE	LOGICAL	SNAPSHOT
DELETE	LOOP	STOP
ELSE	NOSORT	TABULATE
ELSE_IF	NOT	TERMINATE
END	OPEN	THEN
END_IF	OR	TRANSFER
END_LOOP	PARAMETER	TRANSFER_MATRIX
END_PROCEDURAL	PLOT	TRIM
END_WHEN	PREPARE	TRUE
EXTERNAL	PRINT	USE
FALSE	PROCEDURAL	WHEN
FILE	READ	WHILE
FOR	READEL	

2.3.5 ESL numbers

The formal definition of ESL numbers is:

```
number=
        real_number | integer.

real_number =
        integer ("." (integer [exponent] | exponent) | exponent ).

exponent =
        ("E"|"e"|"D"|"d") ["+"|"-"] integer.

integer =
        digit {digit}.

digit                                              [Section: 2.3.1]
```

No spaces may be embedded in numbers. For real numbers the decimal point must not be the first or last character, and it is always required unless an exponent operator is included. The E and D exponent characters are treated identically, and indicate the power of ten by which the number is to be multiplied. The range of an integer is from -2147483648 to 2147483647, and the magnitude of real numbers is approximately 1.17510-38 to 3.4010+38.

2.3.6 ESL delimiters

A number of symbols have special significance in the ESL language. These are: the space " "; the semi-colon ";"; the assignment symbol ":="; the colon ":"; mathematical operators, e.g. "+"; relational operators, e.g. ">"; the range symbol ".."; brackets "( )", "[ ]", "/"; string bracket symbols (" or %); and the comment symbol "--". The use of each of these special symbols is described below.

Space-symbol: A space may not be embedded in any lexical token, e.g. identifier, keyword, number, character string etc.. An end-of-line is treated as a space, and multiple spaces may be freely used between lexical tokens to improve the appearance of a program.

Semi-colon: Each ESL statement must be terminated with a semi-colon, the statement may extend over several lines, e.g.:

```
IF X > Y THEN
  A:=X;
ELSE
  A:=Y;
END_IF;
```

The following keywords (which are not ESL statements) require no terminator:

```
STUDY
END_STUDY
INITIAL
DYNAMIC
STEP
COMMUNICATION
ANALYSIS
TERMINAL
```

The semi-colon is also used to separate different groups of argument type declarations in a subprogram declaration, e.g.:

```
SUBMODEL sub1(real:a,b ; integer:c := real:x,y);
```

Assignment symbol: The assignment symbol, ":=", has three purposes in ESL. It is used in assignments, declarations and sub-program calls. In the following it is used to assign the results of the right-hand side to the left-hand side:

```
Y := TRANSFER(1/(S+1)) * X;
X := 0.0;
y := A*x + D*u;
x,y := sub1(a,b);
```

In declarations it is used as a delimiter between the output and input arguments of MODELs, SUBMODELs, and SEGMENTs:

```
MODEL mod1(real:x,y := real:a,b);
SUMODEL submod1(real:x,y := real:a,b);
SEGMENT seg1(real:x,y := real:a,b);
```

In subprogram calls it is used to delimit the output and input arguments:

MODEL calls:

```
mod1(x,y := a,b);
```

SEGMENT calls:

```
seg1(x,y := a,b);
```

PROCEDURE (sub-routine) calls:

```
sub1(a := b,c);
```

For declarations and calls, the ESL convention is that output variables are placed before the ":=", and input variables after. If no ":=" token is present, then all arguments are interpreted as outputs.

Qualifier symbol: The colon symbol ":", qualifies the variables to its right by the preceding keyword. It is used in the following cases:

```
REAL:        x,y,z;
CHARACTER:   a,b;
INTEGER:     c,d,e;
LOGICAL:     q;
FILE:        infile,outfile;
```

Separator Symbol: ESL uses the comma to separate similar variable or expressions within a statement, e.g.:

```
REAL: a, b, c;
PRINT "Output x = ",X," and y = ",Y;
```

Operator Symbols: the following symbols are used for arithmetic operations:

*	multiplication
/	division
-	subtraction or unary negation
+	addition
**	exponent
^	vector cross product
.	vector dot product

In arithmetic expressions, all operators must be explicitly included. The ESL compiler will check for the correct use of operators depending on the variable types used.

Note: *In certain cases the explicit multiplication operator may be excluded from the TRANSFER function statement where parenthesis is used.*

The multiplication symbol, "*", is also used in PROCEDURE or SUBMODEL declarations for implicit dimensioned arrays.

The multiplication symbol, "*", is also used to indicate repeated values for array elements, e.g.:

```
REAL:A(4)[4 * 0.0];
```

sets all four elements to 0.0.

Relational Operators: the following symbols are used in ESL for relational operations:

=	equality
/=	inequality
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal

Range symbol "..", is used with arrays to explicitly present both lower and upper bounds of a dimension, e.g.:

```
REAL: ARR(1 .. 2, 0 .. 10);
```

declares a two dimension array with subscript ranges 1 to 2 and 0 to 10. It is also used to define array slices, e.g.:

```
arr1(2..4, 2..4) := arr2(1..3, 2..4);
```

The range symbol is also used to indicate the range of a FOR statement, e.g.:

```
FOR i := 8 .. 15 LOOP ....
```

Bracket Symbols: round, "( )", square, "[ ]" and slash "/" brackets are used.

Round brackets are used in the following cases:

`MODEL mod (real:a:=real:b);`	subprogram declarations
`a:= b * (d + e);`	arithmetical expressions
`a:= sin(x) + c;`	function calls
`a:= sub1(b,c);`	sub-model calls
`mod1(a:=b);`	model, segment and procedure calls
`real:AA(10,10), BB(20,20);`	array declarations
`a:= AA(1,2);`	array subscripts
`AA(1..3,2..4):=BB(11..13,12..14);`	array slices
`read (char_var_prompt),a,b;READ`	statement for character variable prompt

Square brackets are used in the following cases:

`PLOT x,y,[y2],0,tfin,0,10;`	optional PLOT statement parameters.
`TRIM [TAU] := [TH''];`	indicating matrix or vector quantities.
`LINEARIZE AA,BB := [th,th'], [tau];`	matrix or vector quantities.
`INTEGER: ARR(2,2) [10,11,20,21];`	explicit row-major setting of arrays.

Slash brackets are used in the following cases:

`REAL:a /1.0/, b /1.0/;`	initialisation of variables
`INTEGER: ARR(2,2) /10,20,11,21/;`	initialisation of arrays, column-major

String Bracket Symbols: literal character strings are bracketed by either of the following:

```
" character string "
% character string %
```

The character used to indicate the start of the string must match that at the end. Use of either of the string bracket symbols within the string is permitted by using the alternative delimiter, or by presenting the character twice, e.g.:

`""""`	is a one character string:	`"`
`%"%`	is a one character string:	`"`

2.4 COMMENTS

Comments in ESL are placed after the double hyphen symbol, "--", and may appear anywhere on a line. All text to the right of the symbol is treated as a comment, e.g.:

```
X := 1.0 -- Set value of X
-- A comment may occupy a full line,
-- and continue with any characters including,
-- a further -- symbol.
```

LIBRARY comment

The LIBRARY directive is a special comment used only by ESL-Studio when processing ESL text submodel, or function procedure, files for inclusion as icons in a block diagram. It is used to specify any standard library submodel, other submodel, procedure, or package, which is used by that module. It causes ESL-Studio to generate INCLUDE statements to incorporate the required modules.

The syntax for the Library declaration is:

```
library_declaration =
        "--" "LIBRARY" file_identifier {"," file_identifier}.

file_identifier =
        character {character}.
```

Note that no terminating ";" is required, and file naming is computer dependent, a space character not regarded as valid in a filename.

2.5 INCLUDE Directive

The INCLUDE statement is not strictly an ESL statement, but a directive to the ESL compiler to indicate that the contents of the specified file should be included in a program at this position.

INCLUDE statements may appear at any point in the program as the only statement on the line, and are dealt with at the lexical level. Code included by this means is treated exactly as though it were written in original program. Further INCLUDE statements may be placed in the included file, that is, include files may be nested.

```
include_statement
        "INCLUDE" character_string ["-" ("L"|"l")] ";".

character_string                                    [Section: 2.3.3]
```

The character string specifies the file name (which is given the ".esl" extension by default). The file name must be bracketed by string bracket symbols " or %. The ESL compiler will try to open the file using the following sequence:

Current directory, with .esl extension if non specified;
Library directory, with .esl extension if non specified;
Current directory, with no extension if non specified.

If the file name incorporates a directory path only that directory will be searched.

Note: *File names are case sensitive on some systems, e.g. LINUX, the recommended practice is to use lower-case file names.*

The "-L" or "-l" option causes the included file to be added to any ESL listing file generated by the compiler, for example, if a "-lst" option is presented on the esl command line.

2.6 Program Structure

Four basic types of ESL program are permitted:

```
program =
        study_program | remote_program |
        embedded_program | non_program.
```

A **study_program** is the basic stand-alone executable simulation program. It must start with a STUDY keyword and be terminated with an END_STUDY. If no model or procedural subprograms are included then the study will contain procedural code only, [Section: 2.11], and by default the complete program will be the experiment, or experiment region. All subprograms must be declared prior to this region. The format is:

```
study_program =
        "STUDY"
        {program_unit}
        experiment
        "END_STUDY".

experiment                                          [Section: 2.9.7]
```

A **remote_program** is intended for concurrent processing, either on the same or a remote processor. It must begin with the REMOTE keyword and include one, and only one, SEGMENT as the last program module:

```
remote_program =
          "REMOTE"
          {package_specification |
          procedure_subprogram |
          function_subprogram |
          submodel_subprogram }
          segment_subprogram.
```

An **embedded_program** is intended to be called by programs written in FORTRAN. It must begin with the EMBEDDED keyword and its structure is identical to the remote_program.

```
     embedded_program =
          "EMBEDDED"
          {package_specification |
          procedure_subprogram |
          function_subprogram |
          submodel_subprogram }
          segment_subprogram.
```

package_specification	[Section: 2.6.6]
procedure_subprogram	[Section: 2.6.4.1]
function_subprogram	[Section: 2.6.4.2]
submodel_subprogram	[Section: 2.6.2]
segment_subprogram	[Section: 2.6.3]

A **non_program** may contain any program_unit, except an experiment, but the compiler will not produce an h-code file ".hcd". It is simply used to check the validity of the program. Note that the program does not start with a program keyword such as STUDY, i.e.:

```
     non_program =
          {program_unit}.
```

Of the seven program_unit types in ESL, five are referred to as subprograms. Note that all subprograms and packages must be declared before use.

```
     program_unit =
          package_specification |
          procedure_subprogram |
          function_subprogram |
          model_subprogram |
          submodel_subprogram |
          segment_subprogram |
          external_segment_declaration.
```

package_specification	[Section: 2.6.6]
procedure_subprogram	[Section: 2.6.4.1]
function_subprogram	[Section: 2.6.4.2]
model_subprogram	[Section: 2.6.1]
submodel_subprogram	[Section: 2.6.2]
segment_subprogram	[Section: 2.6.3]
external_segment_declaration	[Section: 2.6.3]

2.6.1 MODEL

The MODEL subprogram provides the user with the capability to describe the physical system, and it may only be called from the experiment region. Although any number of MODEL subprograms may be presented, only one may be active at a time. MODEL subprograms may be called with any number of arguments between parenthesis, empty parenthesis or no parenthesis at all. A MODEL must contain at least a DYNAMIC region and optionally up to one each of the other region types. The declaration of reserved variables is implicit in a MODEL.

```
model_subprogram =
        "MODEL" identifier argument_specification ";"
        declarations
        model_body
        "END" [identifier] ";".

model_body =
        ["INITIAL" statements]
        "DYNAMIC" dynamic_region_code
        ["TERMINAL" statements]
        ["ANALYSIS" statements].

dynamic_region_code =
        {model_statement ";"}
        ["STEP" statements]
        ["COMMUNICATION" statements].
```

identifier	[Section: 2.3.1]
argument_specification	[Section: 2.7]
declarations	[Section: 2.8]
model_statement	[Section: 2.6.1]
statements	[Section: 2.11]

The MODEL must be terminated by an END statement, with an optional identifier which should be the MODEL name.

The calling convention (from the experiment region) is to place the arguments in the same order as the declaration. This is formally described in the call statement:

subprogram_call	[Section: 2.11.2.6]

The actual argument types must match those used in the. Expressions may be used in the input argument list, and must be the same type as the corresponding declared, or formal, argument. The use of ":=" to separate output arguments from input arguments is mandatory in the model call.

2.6.2 SUBMODEL

The SUBMODEL provides the means to represent part of a system separately from the complexities of the remainder. It is the modelling equivalent of a subroutine or procedure in a procedural language like FORTRAN or Pascal, or in fact the ESL procedure. A SUBMODEL may be invoked several times from the DYNAMIC regions of a MODEL, SEGMENT, or other SUBMODELs. SUBMODELs differ from procedural subprograms in that separate instances of a given SUBMODEL are active simultaneously, and techniques are incorporated into the ESL implementation to avoid conflict between the data associated with different calls of the same SUBMODEL. That is, each invocation of a SUBMODEL has its own "private" data (for variables declared in the SUBMODEL) associated with the particular invocation.

SUBMODEL subprograms may be called with any number of arguments between parenthesis, empty parenthesis or no parenthesis at all. A SUBMODEL must contain at least a DYNAMIC region and optionally an INITIAL, a STEP, and a COMMUNICATION region. It may not include an ANALYSIS or TERMINAL region. A SUBMODEL structure is similar to that of a MODEL:

```
submodel_subprogram =
        "SUBMODEL" identifier argument_specification ";"
        declarations
        submodel_body
        "END" [identifier] ";".
```

```
submodel_body =
        ["INITIAL" statements]
        "DYNAMIC" dynamic_region_code.
```

identifier	[Section: 2.3.1]
argument_specification	[Section: 2.7]
declarations	[Section: 2.8]
statements	[Section: 2.11]
dynamic_region_code	[Section: 2.6.1]

The SUBMODEL must be terminated by an END statement, optionally followed by the SUBMODEL name. The declaration of reserved variables is implicit.

The formal calling convention requires the use of a "submodel_call_statement", from the DYNAMIC region of a MODEL, SEGMENT or other SUBMODEL:

```
submodel_call_statement =
        output_arguments ":="
        identifier "(" input_arguments ")" ";".
```

output_arguments	[Section: 2.11.2.6]
input_arguments	[Section: 2.11.2.6]
identifier	[Section: 2.3.1]

Calls to Submodels are formally "model statements".

ESL provides an extensive library of submodels that may be used in applications, before use the required submodel must be accessed via an INCLUDE declaration [Section: 2.5]. See chapter 3 for a complete list of submodels and their description.

2.6.3 SEGMENT

SEGMENTs provide a means of both emulating and actually running concurrent processes, and also running embedded ESL simulations. They are similar to a MODEL but they may only be called from the COMMUNICATION region of a MODEL, which ensures a fixed time period between each SEGMENT invocation. The declaration of reserved variables is implicit. A SEGMENT is defined as follows:

```
segment_subprogram =
        "SEGMENT" identifier argument_specification ";"
        declarations
        segment_body
        "END" [identifier] ";".

segment_body =
        ["INITIAL" statements]
        "DYNAMIC" dynamic_region_code.
```

statements	[Section: 2.11]
dynamic_region_code	[Section: 2.6.1]

A SEGMENT has its own reserved variables, and may use a different integration algorithm from the model. COMMUNICATION periods (CINT) may be set in the SEGMENT, but they must be selected to be consistent with the calling rate determined by the model.

Where an external, or remote, SEGMENT is used, it must have a prototype declaration in the main ESL study. This takes the form:

```
external_segment_declaration =
        "SEGMENT" identifier argument_specification
        "EXTERNAL" ";"
        [declarations segment_body]
        "END" [identifier] ";".
```

argument_specification	[Section: 2.7]
identifier	[Section: 2.3.1]

The local declarations and the segment body may be included, but they are ignored by the ESL compiler.

The calling convention for SEGMENTs is the same as that for a MODEL, but they may only be called from a model's communication region:

subprogram_call	[Section: 2.11.2.6]

2.6.4 PROCEDURE

Two types of procedural subprograms are defined in ESL, the function_subprogram and the procedure_subprogram, they are similar to FORTRAN functions and subroutines. In addition, external FORTRAN or C routines may be called.

Procedural subprograms contain procedural code, not modelling code. The scope of ESL reserved variables is not extended to procedures, and if they are required they must be declared with a USE RESERVED statement.

2.6.4.1 Procedure subprograms

The format for a procedure_subprogram is:

```
procedure_subprogram =
        "PROCEDURE" identifier
        ["("[argument_list]")"]";"
        procedure_specification
        "END"[identifier]";".

procedure_specification =
        declarations
        statements.
```

identifier	[Section: 2.3.1]
parameter_list	[Section: 2.7]
declarations	[Section: 2.8]
statements	[Section: 2.11]

No distinction is made in the declaration between input and output arguments, the ":=" symbol is not permitted here. Note that the argument list, and "( )" symbols may be omitted.

Procedures may be called from any procedural code region of the ESL program, this excludes direct calls from the DYNAMIC region of modelling subprograms:

subprogram_call	[Section: 2.6.1]

The call is identical in structure to the MODEL and SEGMENT, the ":=" symbol being used to distinguish between output and input arguments, but the ESL compiler does not check whether the procedure uses the formal arguments as implied by the ":=" symbol.

2.6.4.2 Function subprograms

The format of a function-subprogram is:

```
function_subprogram =
        "PROCEDURE" identifier
        "("[argument_list]")" "RETURN" type ";"
        procedure_specification
        "RETURN" expression ";"
        "END"[identifier]";".
```

identifier	[Section: 2.3.1]
parameter_list	[Section: 2.7]

type	[Section: 2.7]
procedure_specification	[Section: 2.6.4.1]

The RETURN statement must be placed at the end of the procedure; however, additional RETURNs may be placed elsewhere in the procedure-specification in which case the first encountered will provide the return value to the calling routine.

Function_subprograms are called from expressions [Section: 2.12] in procedural or modelling Code, and may return a single value, or an array result, of the RETURN type.

```
function_call =
        identifier "(" expression {"," expression} ")".
```

identifier	[Section: 2.3.1]
expression	[Section: 2.12]

2.6.4.3 External procedures

FORTRAN and C functions may be used by ESL programs when using the Translator option. Such functions must be declared with an EXTERNAL declaration statement [Section: 2.8.6] inside the calling subprogram. Once declared, such functions are called in the same way as ESL declared procedures and function procedures.

2.6.5 Standard functions

The following standard functions are part of the ESL language and are implicitly declared:

SIN	sine of argument (radians)
ASIN	arc-sine of argument
COS	cosine of argument (radians)
ACOS	arc-cosine of argument
ATAN	arc-tangent of argument
ATAN2	arc-tangent of two arguments
LOG & ALOG	natural logarithm of argument
EXP	exponential of argument
ABS	absolute value of argument
SQRT	square root of argument
RAND	pseudo-random number
INT	integer value of argument
LEN	returns total number of array elements
LEN_1	number of elements in first dimension of array
LEN_2	number of elements in second dimension of array
LEN_3	number of elements in third dimension of array
ACHAR	character value corresponding to ASCII code
IACHAR	ASCII code corresponding to character value argument
INV	inverse of square matrix
DET	determinant of square matrix
TRNSP	transpose of a matrix
SUB_STRING	returns position in first character string argument where second character string argument is encountered as a sub-string.

The ATAN2 function is an alternative form of ATAN which takes two arguments, and is equivalent to ATAN (arg1/arg2). The result is expressed in radians in the range $-\pi <$ result $<= \pi$, whilst ATAN gives a result in the range $-\pi/2 <=$ result $<= \pi/2$.

The LEN functions work on all array types, including character arrays and strings.

SUB_STRING returns a zero if the second character string argument is not located in the first sub-string argument (identical to FORTRAN INDEX).

RAND(X) produces a uniformly distributed real pseudo-random number in the range 0.0 to ABS(X), where X is a real. A negative or zero value for X re-seeds the number generator to start at the first number in the 4294967296 sequence. The formula used is:

$$RAND(X) = \frac{SX}{M}$$

where $S = (S_O B + C)\mathrm{mod}(M)$

S_O is the last seed,

B=69069,

C=1,

M=2^{32}

The INV function inverts a real or integer square matrices and returns a real square matrix

The DET function calculates the determinant of a real or integer square matrices and returns a real scalar result.

The TRNSP function transposes a two-dimension real, integer, logical or character matrix, and returns a matrix of the same type.

A singular matrix will cause the DET and INV functions to give a run time error.

Note: *Care should be taken in using ASIN and ACOS as they are not defined for arguments outside the range ±1.0. The ATAN function does not have such restrictions.*

2.6.6 PACKAGE

Packages provide a means of grouping data under one heading and making that data available to subprograms and the experiment. They are similar to FORTRAN named common blocks. The format of a PACKAGE definition is:

```
package_specification =
        "PACKAGE" identifier";"
        declarations
        "END" [identifier]";".
```

Declarations may include any of:

EXTERNAL	[Section: 2.8.6]
REAL,INTEGER,CHARACTER,LOGICAL,FILE	[Section: 2.8.5]
CONSTANTS	[Section: 2.8.2]
PARAMETER	[Section: 2.8.3]

and they may also include initialisation of the variables. A subprogram may access the variables in a PACKAGE by the USE statement [Section: 2.8.7]. The PACKAGE itself, however, may not reference other packages.

Note: *NOTE: Access to PACKAGE variables is restricted to the subprograms with a USE declaration. It is not automatically passed to any other subprograms.*

All variables declared in a PACKAGE are given the procedural variable classification [Section: 2.9.8.1].

```
PACKAGE pack1;
    EXTERNAL optim;
    EXTERNAL REAL:func23;
    CONSTANT REAL:pi/3.142/;
    CONSTANT CHARACTER:logo(3)/"ESL"/;
    INTEGER:i,j,k,l;
END pack1;
```

Example PACKAGE declaration

The package structure also provides access to ESL program data from non-ESL routines.

2.6.7 Reserved PACKAGE

ESL features a number of reserved variables as part of the language. An implicit USE RESERVED is assumed in the experiment and all modelling subprograms (MODEL, SUBMODEL and SEGMENT). If access is required from a procedural subprogram (procedure or function) then an explicit USE RESERVED is required.

Many of the Reserved PACKAGE variables are important control variables for the study, and their initial values may be set. There are also reserved variables for system purposes, and these may not be set by the user, i.e. they are treated as constants. The reserved PACKAGE variables which are for user purposes are shown in Table 2-2

Name	Initial value	Type	User setting	Description
T	0.0	real	yes	independent variable (normally time)
TSTART	0.0	real	yes	initial value of T at start of run
TFIN	10.0	real	yes	final value of T at end of run
CINT	1.0	real	yes	communication interval
DISERR	0.0001	real	yes	discontinuity detection error tolerance
INTERR	0.001	real	yes	integration error tolerance
ALGO	1	int.	yes	integration algorithm
NSTEP	1	int.	yes	minimum number of integration steps in CINT
DIS_ST	N/A	int.	no	indication of STEP region call
IEX_CM etc.			no	system use

Table 2-2 Reserved Package variables

Table notes:

1. ALGO may be set to the following values:

1 or	**RK5**	5th order variable step	
2 or	**RK4**	4th order fixed step	
3 or	**RK2**	2nd order fixed step	
4 or	**STIFF2**	2nd order stiff integration	
5 or	**GEAR1**	Gear's variable step stiff integration	
6 or	**GEAR2**	Gear's method, diagonal Jacobian	
7 or	**ADAMS**	Adams predictor-corrector	
8 or	**RK1**	Euler 1st order method	
21 or	**LIN1**	Linearization routine	[Section: 2.9.6]
22 or	**LIN2**	Linearization routine	[Section: 2.9.6]

2. DIS_ST is valid in the STEP region and indicates the reason why the step region has been invoked:

0	ordinary end of step
1	communication point
2	immediately before discontinuity
3	immediately after discontinuity

2.7 Subprogram Argument Declarations

The normal method of passing data between ESL subprograms is by an argument list. The optional argument_specification in a subprogram declaration is specified as:

```
argument_specification =
        [ "(" [output_argument_list]
        [ ":=" input_argument_list] ")" ].

output_argument_list =
        argument_list.

input_argument_list =
        input_argument_declaration
        { "," input_argument_declaration}.

input_argument_declaration =
        ["CONSTANT"] variable_type_declaration | file_declaration.

argument_list =
        argument_declaration
        { ";" argument_declaration}.
```

input_argument_declaration	[Section: 2.7]
argument_declaration	[Section: 2.7]
variable_type_declaration	[Section: 2.3.2]
file_declaration	[Section: 2.8.5]

Input arguments may include a CONSTANT specification which allows the ESL compiler the possibility of producing more efficient code. For example, the CONSTANT argument only needs to be passed to a submodel for the initial call as it is assumed to remain constant throughout a simulation run.

```
argument_declaration =
        variable_type_declaration | file_declaration.

variable_type_declaration =
        type ":"
        variable_declaration
        {"," variable_declaration}.

variable_declaration =
        identifier ["("(dimension_bounds|"*")
        {","(dimension_bounds|"*")}")"].

type =
        "REAL"|"INTEGER"|"LOGICAL"|"CHARACTER".
```

file_declaration	[Section: 2.8.5]
file_specifier	[Section: 2.8.5]
dimension_bounds	[Section: 2.8.4]
identifier	[Section: 2.3.1]
integer	[Section: 2.8.1]

Where an array is specified in the formal argument declaration of a PROCEDURE or SUBMODEL, the symbol "*" may be used in place of explicit dimension bounds. For a MODEL or SEGMENT the actual dimension information is required.

In SUBMODEL and PROCEDURE declarations the array upper dimension bounds are ignored, they are calculated from the dimension lengths of the actual array argument, that is the dimension lengths are inherited from the declaration in the calling subprogram.

Note: NOTE: *If the lower dimension bound is not unity, users are advised to always use implicit "*" dimension specifications for submodels and procedures (not models or segments).*

ESL provides flexibility in allowing differences between actual and formal array arguments in the case of SUBMODEL and PROCEDURE calls. The number of dimensions need not match, provided no attempt is made in the called subprogram to access array elements that do not exist. This is illustrated by:

Actual Argument	Formal Argument	Comment
`A(2,3)`	`F(*)`	assumes F(1..2)
	`F(*,*)`	assumes F(1..2,1..3)
	`F(*,*,*)`	assumes F(1..2,1..3,1..1)

In the first case, only elements of the first dimension of the actual argument are accessible, and in the last case the third dimension is assumed to have a length of unity. Subscripting and Slicing in the called routine must conform to the number of dimensions given in that subprogram's array declaration.

Note: *MODEL and SEGMENT declarations cannot use implicit dimension bounds, and their actual and formal array arguments must have identical dimension lengths. This is because of the special nature of the interface between procedural code and modelling subprograms.*

2.8 Declarations

The declaration statements follow the declaration of each subprogram, or package. All variables used in a subprogram must be declared, and the declarations may include initial values.

Declarations are permitted in the experiment region, the PACKAGE block, the declaration region of MODELs, SUBMODELs, SEGMENTs and PROCEDUREs. Declarations must appear before any executable statements, and are also required in subprogram argument lists; see MODEL, [Section: 2.6.1]; SUBMODEL, [Section 2.6.2]; SEGMENT, [Section: 2.6.3]; PROCEDURE [Section: 2.6.4].

Note: *Procedural subprograms initialise their variables once only, prior to execution. In contrast, simulation subprograms initialise their variables at the start of every simulation run, (unless explicitly directed not to by a RESUME, [Section: 2.11.2.11], or RESTART, [Section: 2.11.2.12], statement).*

The following declaration classes are defined:

```
declarations =
          { (external_declaration
          | file_declaration
          | constant_declaration
          | parameter_declaration
          | type_declaration
          | use_declaration
          | nosort_declaration) }.

type_declaration =
          type ":" declaration_variable
          [("/" | "[") aggregate ("/" | "]")]
          {"," declaration_variable
          [("/" | "[") aggregate ("/" | "]")]}.
```

```
declaration_variable =
        identifier
        [ "(" dimension_bounds {"," dimension_bounds} ")"].

dimension_bounds =
        ["-"] integer [".." ["-"] integer].

aggregate =
        aggregate_element {"," aggregate_element }.

aggregate_element =
        {(identifier | integer) "*"}
        (identifier | ["+"|"-"] (integer | real_number ) |
        "FALSE" | "TRUE" | character_string).
```

external_declaration	[Section: 2.8.6]
file_declaration	[Section: 2.8.5]
constant_declaration	[Section: 2.8.2]
parameter_declaration	[Section: 2.8.3]
use_declaration	[Section: 2.8.7]
nosort_declaration	[Section: 2.8.8]
type	[Section: 2.7]
identifier	[Section: 2.3.1]
dimension_bounds	[Section: 2.8.4]
integer	[Section: 2.8.1]
real_number	[Section: 2.8.1]
character_string	[Section: 2.8.1]

2.8.1 Type declarations

Optional REAL initialisation demands that no spaces may be embedded in numbers, the decimal point must not be the first or last character, but must be included unless an exponent operator is present. The number may be signed.

```
REAL: x,y,z;                        -- declaration, no values
REAL: a1[1.0],a2/1.0E5/,a3/1E-5/;-- declarations, values set
REAL: arr1(4,3,2);                  -- 3 dimensional array, no values
REAL: arr2(2,2,2)/8*0.0/;           -- 3 dimensional array, all values 0.0
REAL: arr3(3,3)[1.0,1.1,1.2,
             2.0,2.1,2.2,
             3.0,3.1,3.2];          -- 2 dimensional array, values set
```

Examples of REAL declarations

Optional INTEGER initialisation again does not allow spaces or any other characters, and may be signed.

```
INTEGER: zz,yy;                        -- declaration only
INTEGER: ww/2/,q1/565/;                -- declaration and values  set
INTEGER: arr21(2,3);                   -- array declaration
INTEGER: arr31(2,3)/1,2,3,4,5,6/;      -- array declaration, values set
```

INTEGER declaration examples

Optional LOGICAL initialisation requires the words FALSE or TRUE, abbreviations are not permitted. Examples of LOGICAL declarations are as follows:

```
LOGICAL: log1, log2;
LOGICAL: log3/TRUE/;
LOGICAL: log4/FALSE/;
```

LOGICAL declarations

Optional CHARACTER initialisation requires literal character string(s) to be presented. Note variables are defined as either as a string variable or an arrays of characters.

String variables:

a single character variable;
a one dimensional character string array;

Array of characters:

a two or three dimensional character array.

```
CHARACTER: A,B(6);                        -- character string variables
CHARACTER: A/"a"/;                        -- character string
CHARACTER: C(2,3);                        -- arrays of characters
CHARACTER: ZR(2,3)["abc","def"];
CHARACTER: ZC(2,3)/"ad","be","cf"/;
```

CHARACTER declarations

If character strings or arrays are assigned initial values in the declaration, then the values must exactly match the array length. For two or three dimension arrays the square brackets, "[ ]", indicate that the character values are stored in the normal row-major order, whilst the "/" delimiters indicate column-major order. In the example both ZR and ZC are initialised to the same values.

See the Development Guide, chapter 6, for details on the character handling features provided by ESL.

2.8.2 CONSTANT declarations

Any of the variable type declarations can also be declared as a CONSTANT:

```
constant_declaration =
        "CONSTANT" type ":" declaration_variable
        ("/" | "[") aggregate ("/" | "]")
        {"," declaration_variable
        ("/" | "[") aggregate ("/" | "]")} ";".

declaration_variable                      [Section: 2.8]
aggregate                                 [Section: 2.8]
```

The values for CONSTANT variables MUST be included with the declaration.

```
CONSTANT REAL: R1/1.5/;
CONSTANT INTEGER: I1/3/,I2/4/;
CONSTANT LOGICAL: GOOD/TRUE/,BAD/FALSE/;
CONSTANT CHARACTER: logo(3)/"ESL"/;
CONSTANT CHARACTER: product(2,10)["Simulation","Language  "];
```

CONSTANT declarations

2.8.3 PARAMETER declarations

A PARAMETER is semantically the same as a CONSTANT, and must be given a value in the declaration. However, alternative values of PARAMETERs may be set by a "simulation driver file", when ESL is run, which will overwrite the value set in the ESL program (ESL Development Guide chapter 11).

```
parameter_declaration =
        "PARAMETER" type ":" declaration_variable
        ("/" | "[") aggregate ("/" | "]")
        {"," declaration_variable
        ("/" | "[") aggregate ("/" | "]")}.
```

declaration_variable	[Section: 2.8]
aggregate	[Section: 2.8]

2.8.4 Array declarations

ESL arrays are limited to a maximum of three dimensions. Two dimension array declarations follow normal conventions by declaring the number of rows, and then the number of columns.

```
REAL: array_name(row_range, column_range);
```

Three dimension arrays require the "planes" to be specified first, then the rows and columns.

Any of the variable types may be declared as arrays by including parenthesis enclosing dimension bounds for each dimension, there are no special declaration statements keywords for arrays. Array initialisation is optional, however, all array elements must be set if initialisation is used.

Repeated initial values may be set to successive array elements by using the "*" operator. This may be combined with individual settings, e.g.:

```
REAL:Arr1(4, 0..3) [1.0, 14*0.0, 1.0];
```

will set all array elements to 0.0 except the first and last.

The lower-bound of an array dimension is assumed to be 1 unless it is explicitly given another value (which could be zero or negative). The upper-bound must be greater than, or equal to, the lower-bound.

```
REAL: r1(5)/5*0.0/;        -- 5 element, 1 dimension array of
                           -- reals all set to 0.0, subscripts
                           -- (1 to 5).

REAL: r2(5,5);             -- 25 element, 2 dimension array of
                           -- reals, not initialised, subscripts
                           -- (1 to 5, 1 to 5.
```

ARRAY declarations

```
INTEGER: Array0(0..3,4,-2..4) [140*0];
                                -- 140 element (4*5*7) 3 dimension
                                -- array, each element value zero,
                                -- subscripts (0 to 3, 1 to 4,-2 to 4)
INTEGER: Array1(0..3,5,-2..4);

INTEGER: IARR1(3,4,5);          -- 60 element, 3 dimensional array of
                                -- integers, subscripts:
                                -- (1 to 3, 1 to 4, 1 to 5)
```

INTEGER array declarations

```
CHARACTER:specials(5,10);         -- array of 5 words of 10 characters
                                  -- each, not initialised, subscripts
                                  -- (1 to 5, 1 to 10)
```

CHARACTER array examples

```
LOGICAL:log1(2,2,2)/TRUE,TRUE,
                    TRUE,TRUE,
                    TRUE,TRUE,
                    TRUE,TRUE,
                    TRUE,TRUE/;
```

LOGICAL array declarations

A three-element column, or row, array may be treated as a special vector, and have certain vector operations performed on it, [Section: 2.12].

```
REAL: vect1(3);            -- 3 element column vector
REAL: vect2(3,1);          -- 3 element column vector
REAL: vect3(1,3);          -- 3 element row vector
```

Vector Array declarations

Two alternative formats for initialisation are provided for matrices; the "//" delimiters imply the initialisation data is presented column by column (column-major) order:

```
REAL: A(3,4)/a11, a21, a31,
        a12, a22, a32,
        a13, a23, a33,
        a14, a24, a34/;
```

The alternative format allows a more natural row by row (row-major) order:

```
REAL: A(3,4)[a11, a12, a13, a14,
        a21, a22, a23, a24,
        a31, a32, a33, a34];
```

Users are advised to use row-major conventions to be compatible with ESL PRINT and READ statements ordering of array output.

2.8.5 FILE declarations

ESL allows the transfer to and from data files, of any of the data types defined. File specifiers are used to connect input/output operations, such as PRINT, READ, or TABULATE, with physical files. Once declared, the file-specifier may be connected and disconnected by means of the various file handling commands (CREATE [Section: 2.11.3.2], OPEN [Section: 2.11.3.1], REWRITE [Section: 2.11.3.3], CLOSE [Section: 2.11.3.4] and DELETE [Section: 2.11.3.5]). File-specifiers may be passed between subprograms in the same way as other types, and they may also be declared in PACKAGEs.

Note: *TABULATE and PREPARE also allow file names to be explicitly presented within the statement.*

File specifier syntax is:

```
file_declaration =
      "FILE" ":" file_specifier {"," file_specifier} ";".
```

```
file_specifier =
        identifier.

identifier                                    [Section: 2.3.1]
```

A file-specifier which is not connected to a specific file will direct any input or output to the user terminal.

2.8.6 EXTERNAL declarations

ESL may call external subroutines or functions written in FORTRAN or C. These externals must appear in an EXTERNAL declaration:

```
external_declaration =
        "EXTERNAL" [type ":"] identifier
        {"," identifier} ";".

type                                          [Section: 2.7]
identifier                                    [Section: 2.3.1]
```

Where the identifier is a name of an external procedure or function. This feature may only be used with the Translator option, it is ignored by the Interpreter.

2.8.7 USE declarations

Before the data in a PACKAGE can be accessed by a subprogram, it must be made known to that subprogram by the USE declaration. This specifies that all variables, constants, file specifiers and externals declared in the named PACKAGE are available to the subprogram. The syntax for the USE declaration is:

```
use_declaration =
        "USE" identifier {"," identifier} ";".
identifier                                    [Section: 2.3.1]
```

where the identifier is the name given to a declared PACKAGE.

The predefined package, RESERVED, containing the reserved variables (T, TSTART, ALGO etc.), is implicitly declared in all model subprograms and the experiment. It is not however implicitly available in procedural subprograms, and its use requires the declaration:

```
USE RESERVED;
```

Note: *Separate sets of reserved variables exist for a MODEL, and each SEGMENT. When SEGMENTS are not "remote" they inherit initial values of reserved variables from the calling MODEL, these values then may be individually set by each SEGMENT. Remote and embedded SEGMENTS initially have the default values for reserved variables.*

2.8.8 NOSORT declarations

The NOSORT declaration is only effective in MODEL, SUBMODEL and SEGMENT program modules where it inhibits the DYNAMIC region statement sorting mechanism. This is sometimes necessary where the user defined order of statement execution is to be strictly adhered to, e.g., for reasons of numerical accuracy. Only a single NOSORT declaration is needed within the declaration region of the program module.

```
nosort_declaration =
        "NOSORT" ";".
```

2.9 Modelling Regions

The MODEL, SEGMENT and SUBMODEL are the modelling programs which are divided into a number of regions.

2.9.1 INITIAL region

The INITIAL region is optional in MODEL, SUBMODEL and SEGMENT subprograms and must appear before the DYNAMIC region. The INITIAL region is executed once at the start of a simulation run only, that is, once for each call of the MODEL from the experiment region. It may be used to set simulation reserved variables such as TSTART, TFIN, ALGO etc., instead of allowing the experiment to perform this function. For segments it is strongly advised that reserved variables which control the run are set here. The INITIAL region may contain procedural statements only [Section: 2.11].

2.9.2 DYNAMIC region

The DYNAMIC region is mandatory in a MODEL, SUBMODEL and SEGMENT subprogram, and must precede any STEP, COMMUNICATION or TERMINAL regions. It must contain non-procedural, modelling code, only [Section: 2.10] which describes mathematically the dynamics of the physical system being modelled.

2.9.3 STEP region

The STEP region is optional in each of the modelling subprograms, and if used must be used immediately after the DYNAMIC region, that is, before any COMMUNICATION region. It contains procedural code only [Section: 2.11]. Statements in the STEP region are executed at the end of every successful integration step, this includes points immediately before and after a discontinuity. Execution of the STEP region is determined by the number of discontinuities and the integration algorithm. For fixed-step algorithms the reserved variables CINT and NSTEP [Section: 2.6.7], determine the basic step-size (CINT/NSTEP), and hence the basic frequency of the STEP region execution. For variable-step algorithms the maximum step-size is the same, but the algorithm may well use a smaller steps to satisfy error criteria. The variable DIS_ST [Section: 2.6.7] indicates the reason for the STEP region execution.

2.9.4 COMMUNICATION region

The COMMUNICATION region is optional in each of the modelling subprograms, and if used must appear immediately after the DYNAMIC and any STEP region. It contains procedural code only [Section: 2.11], and is executed at precise communication intervals as set by CINT [Section: 2.6.7]. Calls to SEGMENTS [Section: 2.6.3] may only be made from the MODEL COMMUNICATION region.

2.9.5 TERMINAL region

The TERMINAL region may only appear in a MODEL, before any ANALYSIS region (if one exists), or as the last region of the model. It is executed once only at the end of the simulation run. It contains procedural code which is executed prior to returning to the experiment code which called the model.

2.9.6 ANALYSIS region

The ANALYSIS region may only appear in a MODEL subprogram, as the last region of the model. It contains procedural code [Section: 2.11], and is used to perform steady-state and linearization functions. It may contain only one TRIM [Section: 2.11.2.8] and only one LINEARIZE [Section: 2.11.2.7] statement, but other procedural statements may be freely used. The ANALYSIS region is invoked by setting the reserved variable ALGO [Section: 2.6.7] to LIN1 or LIN2 either in the INITIAL region, or the experiment prior to the MODEL call.

2.9.7 Experiment region

The experiment region is mandatory in a study_program, and is the final region of the program. No keyword is used to indicate the start of the experiment, and the region contains procedural code [Section: 2.11]. This region is the only one from which a MODEL may be called, and is designed to perform an experiment on the model. A SEGMENT cannot be called from the experiment (it must be called from a model COMMUNICATION region), but other procedure_subprograms may be freely called. Reserved variables are implicitly declared in an experiment.

```
experiment =
        declarations
        statements.
```

declarations	[Section: 2.8]
statements	[Section: 2.11]

2.9.8 Variable classification

ESL assigns to each variable a classification which determines the usage of that variable. When arguments are passed between modelling subprograms, ESL checks not only the variable type, but also the classification to ensure that variable usage is consistent.

Model variables are:

memory Variables
algebraic variables

and are those variable types that are declared in, and are local to MODELs, SUBMODELs and SEGMENTs.

Memory variables are further classified as:

simulation parameters
state variables

and are those variables whose value depends on previous rather than current conditions within the simulation.

2.9.8.1 Procedural variables

These variables are used for basic computational purposes rather than modelling. They are declared and set either in the experiment region, a procedure_subprogram or in a PACKAGE. Procedural variables declared in a PACKAGE may be accessed in a subprogram through a USE statement. They may be freely used with the single exception that they not be set in modelling statements in a DYNAMIC region.

2.9.8.2 Simulation parameters

Simulation parameters are declared in a modelling subprogram, and are given a value (at declaration or in the INITIAL region), and then only changed in special circumstances. As far as the integration is concerned they remain constant throughout each step. They may be modified in procedural regions (e.g. STEP, COMMUNICATION, or in the bodies of WHEN blocks). when changed, e.g. in a WHEN body, the integration will then "see" a new constant value for the simulation parameter. Simulation parameters may inherit the classification from a submodel, by appearing as an output argument in the submodel call where the formal argument is classified as a simulation parameter.

2.9.8.3 Algebraic variables

Algebraic variables (classed as model variables) are declared in model subprograms and may only be set at one point in the DYNAMIC region. They may not be set in any procedural region other than as an output of a PROCEDURAL block. Derivatives of state variables are algebraic variables. It is necessary to ensure that algebraic variables are set before they are used, ESL automatically sorts the DYNAMIC region statements to ensure this is condition is satisfied. Algebraic variables may inherit the classification from a submodel, by appearing as an output argument in the submodel call where the formal argument is classified as algebraic.

```
SUBMODEL boxarea(real:area,rateout:=real:len,bredth);
real:rate;
INITIAL
rate:=0.0;
DYNAMIC
area:=len*bredth;
rate':=area;
rateout:=rate';
END boxarea;
-- area, rate'and rateout are an Algebraic variables
```

Example of Algebraic Variables

2.9.8.4 State variables

State variables, which are classed as memory variables, occur where differential equations are specified. The state variable is declared in a modelling subprogram and must be given an initial value at its declaration or in the INITIAL region. A variable becomes a state when it appears in a "prime" notation modelling statement in a DYNAMIC region. In the example below;

> **X** is declared and initialised as a state variable, and X' is an algebraic variable because it is set in the DYNAMIC region;
>
> **Y** is declared and initialised as a state variable, but because a double derivative is required, Y'' is the algebraic variable (because it is set in the DYNAMIC region), and Y' is another state variable which must also be initialised.

```
MODEL example(:=real:error);
real:X,Y;
INITIAL
X:=0.0;
Y:=0.0;
Y':=0.0;
DYNAMIC
X':= -X+sin(error);
Y'':= -2*Y'-Y;
END example;
```

Use of State Variables

State variables may inherit the classification from a submodel, by appearing as an output argument in the submodel call where the formal argument is classified as a state.

2.9.8.5 CONSTANT

There are two classes of constant, those explicitly declared as such and those variables which assume the constant status locally inside a subprogram. Explicit constants are declared by prefixing the type with the keyword:

CONSTANT REAL:

CONSTANT INTEGER:

CONSTANT LOGICAL:

CONSTANT CHARACTER:

and must be given their value with the declaration, [Section: 2.8.2]. Once declared constants cannot be changed.

Variables (of any type or classification) passed as inputs to a modelling subprogram are also treated as constants within that subprogram. Attempts to modify the value within the subprogram will cause an error.

```
SUBMODEL circarea(real:area:=real:radius);
CONSTANT REAL:pi/3.14159/;
DYNAMIC
area:=pi*radius**2;
end circarea;

-- pi         declared as CONSTANT
-- radius     treated as local constant
```

Examples of constant types

It is also possible to explicitly declare a modelling subprogram input_argument as a CONSTANT which is interpreted as meaning that argument remains constant throughout a simulation run. ESL exploits this information to produce more efficient code.

2.10 Modelling Code

Model statements define the dynamics of the system and appear in the DYNAMIC regions of MODEL's, SUBMODEL's and SEGMENT's.

```
model_statement =
        model_variable_statement |
        submodel_call_statement |
        procedural_model_block |
        when_statement.

model_variable_statement =
        model_variable ":="
        expression | if_clause | transfer_expression | transfer_matrix_expression.

model_variable =
        identifier | derivative_identifier.
```

submodel_call_statement	[Section: 2.6.2]
procedural_model_block	[Section: 2.10.6]
when_statement	[Section: 2.10.5]
expression	[Section: 2.12]
if_clause	[Section: 2.10.3]
transfer_expression	[Section: 2.10.2]
identifier	[Section: 2.3.1]
derivative_identifier	[Section: 2.10.1]

2.10.1 Differential equations

Differential equations may be presented in ESL in derivative form e.g.:

```
x'':= -k*x'-x+1.0;
```

In this example x'' will be classified as algebraic and x and x' as state variables.

Alternatively, differential equations may appear in integral form (using the INTEG submodel) or by specifying the corresponding transfer functions.

2.10.2 Transfer functions

In an ESL transfer function statement, the transfer expression has the form:

```
transfer_expression =
        "TRANSFER" "(" ( [gain]
        transfer_factor {transfer_factor} | gain )
        "/"( [pole] transfer_factor {transfer_factor} | pole )
        {"," initial_expression} ")" "*" input_expression ";".

gain =
        [unary_operator] coefficient.

coefficient =
        ["-"|"+"] identifier | number.

transfer_factor =
        "(" [unary_operator] transfer_term
        {adding_operator transfer_term } ")".

transfer_term =
        coefficient ["*" pole] | pole.

pole =
        ("S" | "s") ["**" integer ].

initial_expression =
        expression.

input_expression =
        expression.

unary_operator                                  [Section: 2.12]
expression                                      [Section: 2.12]
identifier                                      [Section: 2.3.1]
number                                          [Section: 2.8.1]
```

The initial_expression refers to initial conditions of the state variables and is optional, and if omitted values of 0.0 will be assumed. The input_expression is mandatory.

Multiplication is implied between bracketed transfer factors; the transfer function gain and any following factor, and between the origin pole (if present) and following factors (see examples). The S operator must appear after the coefficient in the transfer term. There is no limit to the number of factors present in the numerator or denominator, nor in the order of the Laplacian operator S, except that the order of S in the numerator must be equal to, or less than, the order of S in the denominator.

Transfer Function	ESL Representation
$\frac{-K}{s}$	`-K/s`
$\frac{10.0}{s^2}$	`10.0/s**2`
$\frac{(s+a)}{(2s+b)}$	`(s + a)/(2*s + b)`

$\dfrac{gain(2s^2+0.5s+6)}{s(s+a)(bs^2+cs+d)}$	`gain(2*s**2 + 0.5*s + 6)/`
	`s(s + a)(b*s**2 + c*s + d)`

$\dfrac{(1+0.1s)(1+0.2s)}{s^2(s+a)(bs^2+cs+d)}$	`(1 + 0.1*s)(1 + 0.2*s)/`
	`s**2(s + a)(b*s**2 + c*s + d)`

Examples of ESL transfer function notation

2.10.3 Multivariable transfer functions

In an ESL multivariable transfer function statement, the transfer matrix expression has the form:

```
transfer_matrix_expression =
        "TRANSFER_MATRIX" "(" denominator
        " [" matrix_row { ";" matrix_row } "]" ")" "*" input_expression ";".

denominator =
        ([ pole ] transfer_factor { transfer_factor } | pole ).
matrix_row =
        numerator { "," numerator }.

numerator =
        (([ gain [ "*" zero ] | zero ]) transfer_factor { transfer_factor } | (gain [ "*" zero ] | zero )).

zero =
        ( "S" | "s" ) [ "**" integer ].
```

2.10.4 IF clause

An IF-clause allows alternative expression values to be assigned to a model variable dependent on the value(s) of logical "control" expression(s). A discontinuity occurs when the control expression changes state.

```
if_clause =
        "IF" logic_expression "THEN" expression
        {"ELSE_IF" logic_expression "THEN" expression}
        "ELSE" expression.

logic_expression                              [Section: 2.10.3]
expression                                    [Section: 2.12]
```

The if_clause is allowed only in DYNAMIC regions of a MODEL, SUBMODEL or SEGMENT. The logical_expression corresponding to the first IF, or ELSE_IF, that is true will cause the corresponding expression to be assigned to the model variable. If no logical_expression is true, the ELSE expression is assigned.

2.10.5 WHEN block

The WHEN statement may only be used in the DYNAMIC regions of MODELs, SUBMODELs or SEGMENTs. Its function is to detect events, and when they occur to execute the body of the WHEN, that is procedural statements bracketed by WHENs and END_WHEN. The logical expression, or "trigger condition" is monitored, and when, and only when, the condition becomes true will the WHEN body be executed.

```
when_statement =
        "WHEN" logic_expression "THEN"
                statements
        {"WHEN" logic_expression "THEN"
                statements}
        "END_WHEN".

logic_expression =
        expression.
```

expression	[Section: 2.12]
statements	[Section: 2.11]

```
WHEN X>XMAX THEN
  MAX:=TRUE;
  N:=N+1;
WHEN X<=XMAX THEN
  MAX:=FALSE;
  M:=M+1;
END_WHEN;
```

Example WHEN statement

2.10.6 PROCEDURAL block

PROCEDURAL blocks allow procedural_code to be placed in the DYNAMIC region of MODEL's, SUBMODEL's or SEGMENT's.

```
procedural_model_block =
        "PROCEDURAL" ["(" [ output_list ]
        [":=" input_list] ")"]";"
        statements
        "END_PROCEDURAL" ";".

output_list =
        identifier {"," identifier}.

input_list =
        model_variable {"," model_variable}.
```

identifier	[Section: 2.3.1]
statements	[Section: 2.11]
model_variable	[Section: 2.10]

The output_list and input_list enable ESL to sort the block within the DYNAMIC region, as a single statement, to ensure it is executed at a point where its input_list variables have values. Such blocks should only be used where absolutely necessary (e.g. to set individual elements of an array), because code within the blocks is not subject the rigorous modelling checks. It is essential that the output and input lists correctly reflect the use of the block.

2.11 Procedural Statements

Procedural statements are permitted in any region of any subprogram except the DYNAMIC region, unless included inside a PROCEDURAL block or a WHEN statement.

```
statements =
          {procedural_statement}.
procedural_statement =
            procedural_assignment_statement
          | if_statement                          [Section: 2.11.2.1]
          | loop_statement                        [Section: 2.11.2.2]
          | terminate_statement                   [Section: 2.11.2.3]
          | return_statement                      [Section: 2.11.2.4]
          | stop_statement                        [Section: 2.11.2.5]
          | subprogram_call                       [Section: 2.11.2.6]
          | open_statement                        [Section: 2.11.3.1]
          | create_statement                      [Section: 2.11.3.2]
          | rewrite_statement                     [Section: 2.11.3.3]
          | close_statement                       [Section: 2.11.3.4]
          | delete_statement                      [Section: 2.11.3.5]
          | print_statement                       [Section: 2.11.3.7]
          | read_statement                        [Section: 2.11.3.11]
          | readel_statement                      [Section: 2.11.3.12]
          | prepare_statement                     [Section: 2.11.3.10]
          | tabulate_statement                    [Section: 2.11.3.9]
          | plot_statement                        [Section: 2.11.3.8]
          | clear_screen_statement                [Section: 2.11.3.8]
          | interact_statement                    [Section: 2.11.2.14]
          | trim_statement                        [Section: 2.11.2.8]
          | linearize_statement                   [Section: 2.11.2.7]
          | eigenvalue_statement                  [Section: 2.11.2.9]
          | optimize_statement                    [Section: 2.11.2.10]
          | resume_statement                      [Section: 2.11.2.11]
          | restart_statement                     [Section: 2.11.2.12]
          | snapshot_statement.                   [Section: 2.11.2.13]

procedural_assignment_statement                   [Section: 2.11.1]
```

2.11.1 Assignment statement

An assignment statement replaces the current value of a variable with a new value specified by an expression. The syntax is:

```
procedural_assignment_statement =
          variable | derivative_identifier | derivative_variable
          ":=" expression.

variable                                          [Section: 2.3.2]
primed_identifier                                 [Section: 2.10.1]
derivative_variable                               [Section: 2.10.1]
expression                                        [Section: 2.12]
```

Table 2-3 indicates which type expressions (RHS) may be assigned to the variable (LHS) for all but the trivial same type case.

Variable	Expression	Valid	comments
real	integer	yes	
real	logical	yes	true=1.0 false=0.0
real	character	no	
integer	real	yes	truncated
integer	logical	no	
integer	character	no	
logical	real	no	
logical	integer	no	
logical	character	no	
character	real	no	
character	integer	no	
character	logical	no	

Table 2-3 Valid assignment types

Arrays of all types may be the subject of an assignment, character assignments may perform space filling or truncation, see [Section: 2.8.1].

2.11.2 Control statements

The following controlling statements may only appear in procedural code regions, and may change the sequence of operations of ESL programs.

2.11.2.1 IF statement

An IF statement determines the choice of execution based on the truth of logical expressions.

```
if_statement =
        "IF" logic_expression "THEN"
        statements
        {"ELSE_IF" logic_expression "THEN"
        statements}
        ["ELSE"
        statements]
        "END_IF" ";".

expression                                  [Section: 2.12]
statements                                  [Section: 2.11]
```

IF statements may be nested.

```
IF value1 >= value2 THEN
  IF logic1 THEN
    count:=count+1;
  END_IF;
ELSE_IF value1 < value2 OR value1 <= 0.0 THEN
  count:=0;
  logic1:=FALSE;
ELSE
  logic1:=FALSE;
END_IF;
```

Example IF statement

2.11.2.2 LOOP statement

A LOOP statement specifies that a sequence of statements in a basic loop is to be executed repeatedly, zero or a number of times. Execution of the loop is terminated when either the iteration specification of the loop is exhausted, or due to a TERMINATE statement within the loop. Syntax for a LOOP is:

```
loop_statement =
        [iteration_specification] basic_loop.

iteration_specification =
        ("FOR" identifier ":=" expression ".." expression
        ["STEP" expression]) |
        ("WHILE" logic_expression).

basic_loop =
        "LOOP"
        statements
        "END_LOOP" ";".
```

expression	[Section: 2.12]
identifier	[Section: 2.3.1]
statements	[Section: 2.11]

The identifier and expressions for the FOR and STEP parts must be matching real or integer types. The LOOP can be used in three distinct ways:

1. With no conditional "iteration_specification", using a TERMINATE or INTERACT to break out.
2. With a FOR "iteration_specification", to control the number of times the loop is executed.
3. With a WHILE "iteration_specification", which tests whether to undertake the next pass of the loop, or exit.

Use of STEP with the FOR statement is optional, and if omitted, the variable identified will be increment (or decrement) by one for each iteration.

```
LOOP
  X:=X+1;
  PRINT "current value of X is ",X;
  TERMINATE X>10;
END_LOOP;

FOR ANGLE:= 0.0 .. PI/2.0 STEP 0.1
LOOP
  ARCLENGTH:=RADIUS*ANGLE;
  AREA:=0.5*(RADIUS**2)*ANGLE;
  PRINT ANGLE,ARCLENGTH,AREA;
END_LOOP;

WHILE CURRENT<=10.0 LOOP
  PROCESSOR (CURRENT,VOLTAGE);
  PRINT CURRENT,VOLTAGE;
END_LOOP;
```

Examples of LOOP statement

2.11.2.3 TERMINATE statement

The TERMINATE statement causes the explicit termination of either a LOOP, or the simulation run, depending on its position. The syntax is:

terminate_statement =
    "TERMINATE" *logic*_expression ";".

logic_expression	[Section: 2.10.5]

Termination occurs when executed with the logical condition of true.

2.11.2.4 RETURN statement

A RETURN statement is mandatory as the last statement in a function_subprogram, [Section: 2.6.4.2], returning the value defined in the accompanying expression. Additional RETURN statements may be included. RETURNs are optional in a procedure_subprogram, the END causes a return [Section: 2.6.4.1]. The syntax is:

return_statement =
    "RETURN" [expression] ";".

expression	[Section: 2.12]

The expression may be any type, including an array, e.g.:

```
RETURN X>=Y;
RETURN INV(ARRAY);
```

2.11.2.5 STOP statement

A STOP statement may be located in any procedural code and causes an immediate termination of an ESL study.

stop_statement =
    "STOP" ";".

2.11.2.6 Subprogram call statement

A call statement enables procedures, or procedure functions, (internal or external) to be invoked from procedural code, a MODEL to be invoked from the experiment, or a SEGMENT to be invoked from the COMMUNICATION region of the model.

```
subprogram_call =
          identifier["("output_arguments
          [":="input_arguments]")"]";".

output_arguments =
          variable | derivative_variable
          {"," variable | derivative_variable}.

input_arguments =
          expression {"," expression}.

expression                                    [Section: 2.12]
identifier                                    [Section: 2.3.1]
```

Actual arguments in a call, and corresponding formal (dummy) arguments in the declaration of a subprogram must agree with respect to type, and in the case of arrays, the dimensions must be consistent, see [Section: 2.8.4] for further details.

Declarations of procedural subprograms do not distinguish between input and output arguments, but in the call the symbol ":=" must be used to separate the output from input arguments. ESL does not check that the procedure uses the arguments as specified, e.g. as outputs or inputs. It is the users responsibility to ensure that the procedure definition is consistent with the call, and that actual output arguments are variables (not expressions) which may be set.

2.11.2.7 LINEARIZE statement

One LINEARIZE statement only may be included in the ANALYSIS region of a MODEL. It is used to compute the matrices for the state-space form of a linearized model at steady-state conditions. The ANALYSIS region is invoked by calling a model with the reserved variable ALGO set to LIN1 or LIN2 (see next section).

```
linearize_statement =
          "LINEARIZE" a_matrix "," b_matrix ":="
          "[" state_vector "]" "," "[" input_vector "]"
          [ c_matrix "," d_matrix ":=" "[" output_vector "]" ] ";".

state_vector =
          ( identifier | derivative_identifier )
          {","( identifier | derivative_identifier ) }.

input_vector =
          identifier {","identifier}.

output_vector =
          identifier {","identifier}.

a_matrix =
          identifier.

b_matrix =
          identifier.

c_matrix =
          identifier.

d_matrix =
          identifier.

identifier                                    [Section: 2.3.1]
```

The c_matrix and d_matrix and output_vector are only required if both state and output equations are needed. Each of the vector identifiers may represent any number of scalar variables, one dimensional arrays or a mixture of these. The arrays for the state-space matrices must have been previously declared and have correct dimensions.

State space form:

```
x' = A*x + B*u
y  = C*x + D*u
```

where

x is the state vector
u is the input vector
y is the output vector
A,B,C,D are the state matrices

ESL format:

```
LINEARIZE A,B:=[x,z,Brr],[Urr,u1,u2]
          C,D:=[y1,y2];
```

where x,z,u1,u2,y1 and y2 are scalars and Brr and Urr are arrays.

Linearization example

2.11.2.8 TRIM statement

One TRIM statement only may be included in the ANALYSIS region of a MODEL. This employs one of two minimisation algorithms to determine the steady-state of a model. The statement form is:

```
trim_statement =
        "TRIM" "[" control_vector "]" ":="
        "[" derivative_vector "]" ";".

control_vector =
        ( identifier | derivative_identifier )
        {"," ( identifier | derivative_identifier ) }.

derivative_vector =
        ( identifier | derivative_identifier )
        {"," ( identifier | derivative_identifier ) }.

identifier                                        [Section: 2.3.1]
```

For any steady-state condition, known state and/or input variables are supplied to the algorithm, which then determines the values of the other supplied variables to achieve the steady-state condition. The ANALYSIS region is invoked, and options selected, by calling a model with the reserved variable ALGO set as follows:

LIN1	Newton-Raphson algorithm
LIN2	Simplex algorithm

The control_vector variables may be "state" or "simulation parameter" and for the derivative_vector, "algebraic", [Section: 2.9.8].

2.11.2.9 EIGENVALUE statement

The EIGENVALUE statement is a general procedural code statement to determine the eigenvalues of any real square matrix. An n x n matrix A has an eigenvector x and corresponding eigenvalue λ if:

$$\boldsymbol{A}x = \lambda\boldsymbol{x}$$

It follows that the eigenvalues are the n roots of the characteristic equation:

$$\det|\boldsymbol{A} - \lambda\boldsymbol{I}| = 0$$

The EIGENVALUE statement has the form:

```
eigenvalue_statement =
        "EIGENVALUE" eigenvalue_array ":="
        system_matrix ";".

eigenvalue_array =
        identifier.

system_matrix =
        identifier.
```

The following example illustrates the calculation of eigenvalues.

```
EIGENVALUE L := A;
```

where:

A	is the real n x n matrix whose eigenvalues are to be determined;
L	is a real n x 2 array to hold the eigenvalues.

On return from the statement, L(1..n, 1) will contain the real parts of the eigenvalues, and L(1..n, 2) will contain the imaginary parts (if the eigenvalues are complex).

2.11.2.10 OPTIMIZE statement

The OPTIMIZE statement uses the Simplex algorithm to optimize a system expressed as an ESL model or procedure. The output argument of the module is the "performance function" to be minimised, and the input arguments are the parameters to be determined. The form of the statement is:

```
optimize_statement =
        "OPTIMIZE" identifier "(" variable ":="
        variable {variable} ")" ";".
```

An example OPTIMIZE statement is:

```
OPTIMIZE subprog_name(cost := par1, par2, ... );
```

where:

subprog_name	is the name of the ESL model or procedure to be optimized;
cost	is the performance function output, to be minimised;
par1, par2,...	are the input parameters.

The OPTIMIZE argument list must match exactly that of the subprogram definition and all the arguments must be of type real. Further, the input parameters must be variables, not expressions, and be given initial values before the optimization call. For example:

```
STUDY
  MODEL CONTROLLER(REAL: cost := REAL: g1, g2, g3);
            ......
            ......
  END CONTROLLER;
-- EXPERIMENT
  REAL: perform, gain1, gain2, gain3;
  gain1 := 0.1; gain2 := 0.1; gain3 := 0.2;
  OPTIMIZE CONTROLLER(perform := gain1, gain2, gain3);
  PRINT "performance function", perform;
  PRINT "optimum gains", gain1, gain2, gain3;
END_STUDY
```

2.11.2.11 RESUME statement

A RESUME statement may only be used in the experiment region to call a MODEL. The format is:

```
resume_statement =
        "RESUME" subprogram_call ";".

subprogram_call                                 [Section: 2.11.2.6]
```

A RESUME statement will invoke the MODEL to continue the previous simulation run from the conditions which prevailed when the model last completed a simulation. It will bypass the INITIAL region, using the results at the end of the previous simulation for its initial conditions. It can be used only if there has been a previous call to the model, and TFIN normally needs increasing to allow the simulation to proceed

2.11.2.12 RESTART statement

A RESTART statement is used from the experiment region only to call a MODEL. The format is:

```
restart_statement =
        "RESTART" subprogram_call ";".

subprogram_call                                 [Section: 2.11.2.6]
```

A RESTART statement will invoke the MODEL to restart a simulation from the conditions which prevailed when the model last completed a simulation. It will bypass the INITIAL region, using the results at the end of the previous simulation for its initial conditions. It can be used only if there has been a previous call to the MODEL, and is similar to a RESUME, but resets the T variable to TSTART and forces a new run for PREPARE statement files.

2.11.2.13 SNAPSHOT statement

A SNAPSHOT statement may appear within procedural code to produce a complete copy of the simulation in a "snapshot" file (default extension ".snp") so that the simulation may be restarted from that state in the future. The statement would normally appear in a model communication region, or the experiment following the model call. It is only effective in cases where a single model call (possibly within a LOOP) is made from the experiment. The format is:

```
snapshot_statement =
        "SNAPSHOT" [file_name] ";".

file_name                                       [Section: 2.8.5]
```

If the file_name is omitted, the user will be prompted at run-time. A snapshot file may also be taken during the INTERACT service.

2.11.2.14 INTERACT statement

An INTERACT statement may be used within procedural code to pass control to the user at a pre-determined point when running a simulation. It provides the user with the ability to monitor, or set variables, and to control the program execution. The format is:

```
interact_statement =
          "INTERACT" ";".
```

The INTERACT service is also invoked by the user pressing the "break" keys, and in the event of most run-time errors.

2.11.3 Input - Output statements

All statements dealing with input and output in ESL are procedural code. The file_name to be connected to a file-specifier ([Section: 2.8.5]) by an OPEN, CREATE or REWRITE statement may be represented as a literal string (between quotes), a blank string (between quotes) or a one dimensional character array or expression. If a blank string is used, the program will prompt the user for a file name at run-time. Note that if a "file_status" option is used the program must be prepared to deal with error condition, otherwise ESL will attempt to resolve problems by interaction with the user, or by giving an error message and halting.

2.11.3.1 OPEN statement

The OPEN statement connects a file-specifier to an existing file for reading:

```
open_statement =
          "OPEN" file_specifier "," file_name
          ["," file_status] ";".

file_name =
          character_expression.
```

expression	[Section: 2.12]
file_specifier	[Section: 2.8.5]
file_status	[Section: 2.11.3.6]
identifier	[Section: 2.3.1]

2.11.3.2 CREATE statement

The CREATE statement will create a new file with the name specified and connect it to a file-specifier. If a file of the name specified already exists, an error condition occurs. The format is:

```
create_statement =
          "CREATE" file_specifier "," file_name
          ["," file_status] ";".
```

file_specifier	[Section: 2.8.5]
file_name	[Section: 2.11.3.1]
file_status	[Section: 2.11.3.6]

2.11.3.3 REWRITE statement

The REWRITE statement will create a new or overwrite an existing file of the name specified and connect it to a file-specifier. The format is:

```
rewrite_statement =
          "REWRITE" file_specifier "," file_name
          ["," file_status] ";".
```

file_specifier	[Section: 2.8.5]

file_name	[Section: 2.11.3.1]
file_status	[Section: 2.11.3.6]

2.11.3.4 CLOSE statement

The CLOSE statement will close the file connected to the designated file-specifier, and make that specifier available for other use. The format is:

```
close_statement =
        "CLOSE" file_specifier ";".
```

file_specifier	[Section: 2.8.5]

On normal termination from an ESL program, all files are automatically closed.

2.11.3.5 DELETE statement

The DELETE statement will close and delete a file which is not connected to a file specifier. The format is:

```
delete_statement =
        "DELETE" file_name ["," file_status] ";".
```

2.11.3.6 File status

This is an optional parameter that may be appended to an OPEN, CREATE, REWRITE, DELETE or READ statement. Any errors encountered in executing the command are returned in the integer variable indicated in the statement. The format is:

```
file_status =
        "IOSTAT" "=" integer_identifier.
```

identifier	[Section: 2.3.1]

The possible return values are given in Table 2-4.

If the IOSTAT statement is not included, any errors encountered will cause the program to interact with the user to resolve the problem, or to halt with an appropriate error message.

2.11.3.7 PRINT statement

The PRINT statement allows text, values of variables or expressions to be printed on the terminal or output to a file. For output to a file a file-specifier must be connected to the file for output (by CREATE or REWRITE). The format is:

```
print_statement =
        "PRINT" [ (file_specifier | print_element)
        {"," print_element }] ";".

print_element =
        expression [":" print_format_control]
        | "/" {"/"}
        | "-/".

print_format_control =
        ["-"] integer ["." integer].
```

file_specifier	[Section: 2.8.5]
expression	[Section: 2.12]
integer	[Section: 2.8.1]

The output list expressions may be of type real, integer, logical or character and may be any array type or slice. Arrays are printed in row-major order and any format control apply to all elements in the array.

The output stream may extend over the maximum line length and further lines are output until the stream is exhausted. If the line is split then logical and numerical values are adjusted to fit on the same line, whereas character strings may be split.

Return value	Status
0	Operation was performed without error.
1	Eol was encountered when a data value was expected. A data value terminated by Eol does not cause the IOSTAT variable to be set to Eol.
2	End-of-file was encountered.
3	Error occurred converting input data to internal form - e.g. illegal number format.
4	Failure to open file.
5	Create file failure - file already exists?
6	Failure to delete file.
7	Failure to create file.
8	Maximum file channels in use (>20).
9	Too many direct access files (installation error).
10	File inaccessible - illegal name, already open?

Table 2-4 IOSTAT Errors

By default the PRINT statement will output data in the following format:

REAL values:	FORTRAN G13.5 (field width 13, 5 significant digits);
INTEGER values:	FORTRAN I9,4X (field width 9, followed by 4 spaces);
LOGICAL values:	13 spaces (centred).

To override the default format, the value to be output is followed by a colon and then a whole number and possibly a decimal part. That is. :m.n (e.g. :12.5). This is interpreted as:

REAL values: if **:m.n** n =/ 0 then (FORTRAN Fm.n) - field width m, with n decimal places;
else if **:m.0** or **:m** then
    if m > 8 then FORTRAN Gm.(m-8)
    else_if m = 8 then FORTRAN G8.1
    else_if m < 8 then FORTRAN G13.5 (the default format).

INTEGER values: m (FORTRAN Im) - field width m, right justified.

LOGICAL values: m - field width, right justified.

CHARACTER: are not influenced by format control.

Note that a negative format specifier (e.g. :-m.n1 or :-m) suppresses all spaces in output.

The maximum field width, (m), is restricted to 24 characters.

Note: *NOTE: Formatting real values requires a field width large enough for: possible negative sign; leading zero; decimal point; n significant figures; and four positions for a possible exponent, which may left blank.*

The PRINT statement may also include line control characters:

/	forces a new line
-/	suspends a new line, next print will append

Type	Format	Example
real	G13.5 field width 13 and 5 significant digits	▾▾202.45▾▾▾▾ ▾▾0.12345e-10 ▾-0.31415e 01
integer	I9,4X right justified field width 9 followed by 4 spaces	▾▾▾▾▾▾▾▾3▾▾▾▾ ▾▾▾▾▾▾▾32▾▾▾▾ ▾▾▾▾▾▾321▾▾▾▾
logical	centred in field width of 13 characters	▾▾▾▾FALSE▾▾▾▾ ▾▾▾▾▾TRUE▾▾▾▾
character	lower case	AbCde becomes abcde

Table 2-5 PRINT default output

2.11.3.8 PLOT statement

The PLOT statement outputs specified data to the screen in the form of a graph. The first execution draws the graph axis, and remaining executions plot each of the specified variables or expressions. Repeated simulation runs from the experiment suppress the graph axis plotting allowing multiple runs to be shown. If the PLOT statement is placed in the STEP or COMMUNICATION regions, or in a subprogram called from these regions, each plotted point is joined to the previous point by a line. If placed in other procedural code regions, symbols are plotted at the data points.

The format of the PLOT statement is as follows:

```
plot_statement =
        "PLOT" [ plot_title ","]
        independent_variable ","
        dependent_variable
        ( "," [more_depend_var] | more_depend_var)
        x_min "," x_max ","
        y_min "," y_max ";".

independent_variable =
        model_variable | expression.

more_depend_var =
        "[" dependent_variable
        { "," dependent_variable } "]" [","].

dependent_variable =
        model_variable | expression.
x_min =
        expression.

x_max =
        expression.

y_min =
        expression.

y_max =
        expression.
```

plot_title	[Section: 2.11.3.8]
model_variable	[Section: 2.10]
expression	[Section: 2.12]
character_string	[Section: 2.8.1]

Multiple PLOT statements are permitted but only one may be active. In order to allow previous plots to be removed, a CLEAR_SCREEN statement is provided:

```
clear_screen_statement =
        "CLEAR_SCREEN" ";".
```

The next PLOT executed will become active.

2.11.3.9 TABULATE statement

The TABULATE statement will output specified data in a tabular format, either to the terminal or a file. If a file name is specified it is presented as a character string, variable or expression. If no extension is given, ".tab" will be appended, and if it is a blank character string, then the name of the program file (the .esl file) is used but with a .tab extension. Any existing file of the same name will be overwritten. The format is:

```
tabulate_statement =
        "TABULATE" [ file_specifier |
        file_name ","]
        output_value {"," output_value} ";".

output_value =
        model_variable | expression.
```

file_specifier	[Section: 2.8.5]
file_name	[Section: 2.11.3.1]
model_variable	[Section: 2.10]
expression	[Section: 2.12]

TABULATE output may include any numerical or logical expressions. If a file_name is specified, any existing file of that name will be overwritten. A TABULATE statement placed in the STEP or COMMUNICATION regions will produce a heading at the start of each simulation run. If the TABULATE statement is not invoked during a simulation run, then each time it is executed the heading line will be output prior to each data output. If the INTERACT option is used to repeat the run, the previously stored data will be overwritten.

A facility is provided in the convertDisplayFile program to convert between TABULATE and PREPARE file formats.

2.11.3.10 PREPARE statement

The PREPARE statement saves the values of specified variables in a non-text format for use by ESL-Studio and ESL-Displays or the convertDisplayFile program. The file name may be specified as a character string, variable or expression. If no extension is given, ".dsp" will be appended, and if it is a blank character string, then the name of the program filename (the .esl file) is used but with a .dsp extension. Any existing file of the same name will be overwritten. The format is:

```
prepare_statement =
        "PREPARE" file_name ","
        [plot_title ","]
        [plot_subtitle ","]
        output_value "," [value_title ","]
        {output_value "," [value_title "," ]} ";".
```

```
plot_title =
        character_expression.

plot_subtitle =
        character_expression.

value_title =
        character_expression.
```

output_value	[Section: 2.11.3.8]
file_name	[Section: 2.11.3.1]
character_string	[Section: 2.8.1]

The output values may be any numerical or logical expressions or arrays. The plot_title, plot_subtitle and value_title may be character strings, character variables (one dimensional character arrays) or character expressions of length up to eighty characters for the plot titles, and twenty characters for the value titles. A PREPARE statement in the STEP or COMMUNICATION regions causes data for each run to be stored in the named file. These data sets may be accessed by ESL-Studio and ESL-Displays or the convertDisplayFile program. If the INTERACT option is used to repeat the run, the previously stored data will be overwritten. If the PREPARE statement is not invoked during a simulation run, then each time it is executed further data is added to the prepare file to form a single data set.

2.11.3.11 READ statement

The READ statement is used to input data, either from a file or the terminal. If a file_specifier is present, it must have been previously connected to an existing file by an OPEN statement. The format is:

```
read_statement =
        "READ" [ ( file_specifier | prompt | read_element )
        {"," read_element}
        ["," file_status] | file_status ] ";".

prompt =
        character_string | "(" expression ")".

read_element =
        variable [":" read_format_control ].

read_format_control =
        ["-"] integer ["." integer].
```

file_status	[Section: 2.11.3.6]
file_specifier	[Section: 2.8.5]
character_string	[Section: 2.8.1]
expression	[Section: 2.12]
variable	[Section: 2.3.2]
integer	[Section: 2.8.1]

If no file_specifier, and no prompt, are given the READ statement will take input from the terminal, and prompt the user with the names of the specified read_elements. If the input requirements are not met, ESL will prompt the user for the outstanding data.

Unless otherwise specified ESL will use free-format for input. With free format each data item may be preceded by spaces, and must be terminated with a space, comma, equal sign or eol (end-of-line). In particular:

> Character strings may be presented without sting quotes (" or %) provided they do not contain embedded spaces. On input the ESL variable will be space filled or truncated as necessary, with no case conversion.

Logical input is treated as a character string, which must match one of the following:

True	False
true	false
tru	fals
tr	fal
t	fa
yes	f
ye	no
y	n
1	0

Valid integers may be preceded by an optional plus or minus, immediately followed by the integer value. No embedded spaces are allowed. Any decimal point will cause the rejection of the value.

Valid reals may be preceded by an optional plus or minus. They may be entered as integers, or may start, or end, with a decimal point, and may also include an exponent, e.g. -4.12e-5.

Formatting of inputs is provided by following the variable name with a colon, a whole number with an optional decimal part, i.e. :m.n (e.g. :12.1) or :m (e.g. :8).

For the whole part m:

An m of zero (i.e. 0.0 or 0.1) is the default situation and free format input applies.

A positive m determines a fixed field of that width, e.g. var:4.0 means var is to be in a field of 4 characters.

A negative m (i.e. :-1, :-1.0 or :-1.1) means free format but only that line is to be analyzed, the reading of a subsequent line is not permitted to satisfy the current list item.

The decimal part n applies to character string input only and indicates conversion to upper case, as:

m.0 means no conversion of case (the default);

m.1 conversion to upper-case.

For fixed field format, leading and trailing spaces are ignored for numerical and logical values, and a field which comprises entirely of spaces is considered to give a data value of zero or false for logical items. Furthermore if a valid number, which is properly delimited, is encountered before the end of the field, the remainder of the field is ignored.

For character string variables read in fixed format the m input characters are assigned to the string variable with truncation or space extension if the length of the variable string is different from n. Note that spaces, commas, and equals characters are permitted in these fixed format strings.

If an eol (end-of-line) is encountered during fixed format reading when a new data field is expected an error condition exists, and if present the IOSTAT variable will be set to indicate the eol. If the IOSTAT variable is not present the program continues without reporting an error. Any remaining input list items will be given their default values, i.e. zero for numbers, space for characters, and false for logical.

If the start of a fixed format field exists, but there are less than n characters before an eol is encountered, the input field is considered to be extended by spaces until it is "n" characters long. This case is not considered an error and the IOSTAT variable is not set to indicate eol.

2.11.3.12 READEL statement

The READEL statement will read one variable element at a time from the input buffer which must have been previously filled with a READ statement.

readel_statement =

1. "READEL" [read_element {"," read_element}

1.1.1.1.1. ["," file_status] | file_status] ";".

read_element	[Section: 2.11.3.11]
file_status	[Section: 2.11.3.6]

2.12 Expressions

Expressions are specified in many syntactic definitions, the ESL expression is similar to that found in other languages such as FORTRAN.

```
expression =
        logical_term {"OR" logical_term}.

logical_term =
        logical_factor {"AND" logical_factor}.

logical_factor =
        ["NOT"] logical_primary.

logical_primary =
        simple_expression
        [relational_operator  simple_expression].

simple_expression =
        [unary_operator] term {adding_operator term}.

term =
        factor {multiplying_operator factor}.

factor =
        primary {exponentiating_operator primary}.

primary =
        "(" expression ")" |
        function_call |
        variable |
        derivative_identifier | derivative_variable
        number |
        character_string |
        "FALSE" |
        "TRUE".

relational_operator =
        "=" | "/=" | "<" | "<=" | ">" | ">=".

adding_operator =
        "+" | "-".

unary_operator =
        "+" | "-".

multiplying_operator =
        "*" | "/" | dot_product | cross_product.
```

```
exponentiating_operator =
        "**".

dot_product =
        ".".

cross_product =
        "^".
```

variable	[Section: 2.3.2]
derivative_identifier	[Section: 2.3.2]
derivative_variable	[Section: 2.3.2]
number	[Section: 2.8.1]
character_string	[Section: 2.8.1]

The following groups are defined in decreasing order of precedence in Table 2-6.

( )	parenthesis
. ^	vector operators
**	exponentiating operator
* /	multiplication and division operators
+ -	unary plus and minus
+ -	addition and subtraction operators
= /= < <= > >=	relational operators
NOT	unary logical operator
AND	logical operator
OR	logical operator

Table 2-6 Operator precedence

For operators at the same precedence level, evaluation is left to right with the sole exception of:

```
A**B**C
```

which is treated as:

```
A**(B**C)
```

CHAPTER 3

Submodel Library

This chapter contains an alphabetic list of submodels and procedures available in the ESL library. The declaration and comment specifying each module are presented. The text of this chapter was generated automatically by extracting text directly from the library files.

3.1 ABSX

```
SUBMODEL ABSX(REAL:y := REAL:x);

-- Outputs an absolute value of a real input. This is a
-- submodel version of the standard function ABS which treats a
-- change in sign of the input as a discontinuity.
-- The calling sequence is:
--
--    y:= ABSX(x)
--
-- where:
--    x is the input variable;
--    y is given a value such that:
--
--          y = x, if x >= 0.0,
--          y = -x, if x < 0.0.
--
-- The output is an algebraic variable.
```

3.2 AFGEN0

```
SUBMODEL AFGEN0(REAL:y := INTEGER:N; REAL:TABLE(2,*),x);
-- Searches a table of x-y coordinate values (ascending x order)
-- and returns a zero order interpolation corresponding to input
-- x. That is the y-table-value corresponding to the largest
-- x-table-value which is less than or equal to x. If x is less
-- than the first x-table-value then the first y-table-value is
-- returned. The calling sequence is:
--
--    y:= AFGEN0(N,TABLE,x)
--
-- where:
--
-- N is the number of elements in the array;
-- TABLE is the table of values which represents a two row
--       n column matrix:
--       row 1 represents the input values, which
--       must be in ascending order,
--       row 2 represents the corresponding function values;
--
-- x  is the input value;
-- y is returned value.
-- The output is a memory variable.
```

3.3 AFGEN1

```
SUBMODEL AFGEN1(REAL:y := INTEGER:N; REAL:TABLE(2,*),x);
-- Searches a table of x-y coordinate values and finds
-- which values span the input value of x and performs a
-- first order interpolation to obtain a value for y. The
-- calling sequence is:
```

```
--
--    y:= AFGEN1(N,TABLE,x)
--
-- where,
--
-- N is the number of elements in the array,
-- TABLE is the table of values which represents a two row
--       n column matrix;
--       row 1 represents the input values, and which
--       must be in ascending order,
--       row 2 represents the corresponding function values
-- x  is the input value,
--
-- y is calculated by use of the equation,
--
--    y = ycl+(x-xcl)*grad
--
-- where grad = (ycu-ycl)/(xcu-xcl), and
--
-- ycu - TABLE(2,i) array upper segment value,
-- ycl - TABLE(2,i-1) array lower segment value,
-- xcu - TABLE(1,i) array upper segment value,
-- xcl - TABLE(1,i-1) array lower segment value.
--
-- The output is an algebraic variable.
```

3.4 AFGEN2

```
SUBMODEL AFGEN2(REAL:y := INTEGER:N ;REAL:TABLE(2,*),x);
-----------------------------------------------------------------
-- Searches a table of x-y coordinate values and finds
-- which values span the input of x and performs a second
-- order interpolation to obtain a value for y.
-- Over any segment, a number of simple polynomial fits
-- is found and the interpolated function, y, is a
-- weighted average over two of these; the weighting
-- being a function of the independent variable.
-- The weighting function has a zero slope at the points
-- where the interpolating polynomials switch over. In
-- the range TABLE(1,i) to TABLE(1,i+1), two quadratics
-- are used; Qn fits TABLE(2,i-1), TABLE(2,i) and
-- TABLE(2,i+1) and Q1n fits TABLE(2,i),
-- TABLE(2,i+1) and TABLE(2,i+2). Qn and Q1n overlap in the
-- range TABLE(1,i) to TABLE(1,i+1). In this range,
--
--    y = w*Qn+(1-w)*Q1n
--
-- where,
--
-- w = 1-3*s**2+2*s**3, is the weighting function,
-- s = (x-TABLE(1,i))/(TABLE(1,i+1)-TABLE(1,i)),
--     the independent variable,
--
-- s has a normalised range [0,1] and,
-- w(0) = w(1) = w'(0) = w'(1) = 0.0.
--
-- The calling sequence is:
--
--    y:= AFGEN2(N,TABLE,x)
--
-- where,
--
-- N is the number of elements in the array,
-- TABLE is the table of values which represents a two row
--       n column matrix;
--       row 1 represents the input values and which
--       must be in ascending order,
--       row 2 represents the corresponding function values
-- x is the input value.
--
-- The output is an algebraic variable.
```

3.5 BISTBL

```
SUBMODEL BISTBL(LOGICAL:y := CONSTANT LOGICAL:IC;
                                      LOGICAL:reset,set,clock,x);

-- Logical bistable storage device which stores the
-- logical data input (x) as the clock input becomes TRUE.
-- A 'set' input of TRUE causes a TRUE to be stored and
-- inhibits the normal operation. Similarly, a reset value
-- of TRUE causes a FALSE to be stored and inhibits both
-- the set operation and normal operation. The calling
-- sequence is:
--
--    y:= BISTBL(IC,reset,set,clock,x)
--
-- where:
--
--    IC is the logical initial condition;
--    reset resets the bistable to a logical FALSE output;
--    set sets the bistable to a logical TRUE output;
--    clock is normally a logical pulse train; as it becomes
--          TRUE (edge triggering), the logical input (x) is
--          stored in the bistable memory;
--    x is the logical 'data' input variable.
--
--    y is given a value such that:
--          y = FALSE, if reset is TRUE;
--          y = TRUE, if set is TRUE and reset is FALSE;
--          y = x, when clock being TRUE provided set and reset
--                are FALSE.
--
-- The output is a memory variable.
```

3.6 CMPXPL

```
SUBMODEL CMPXPL(REAL:y := CONSTANT REAL:IC1,IC2; REAL:Zeta,Wn,x);

-- Represents a second order system with a damping ratio Zeta and
-- undamped natural frequency Wn.
-- The calling sequence is:
--
--    y:= CMPXPL(IC1,IC2,Zeta,Wn,x)
--
-- where:
--    IC1 and IC2 are the initial conditions of Y and Y';
--    Ztea and Wn are the damping ratio and natural frequency;
--    x is the input variable.
--
-- The differential equation is given by:
--
--    y'' = wn**2*x - 2*Zeta*Wn*y' - Wn**2*y = x
--
-- and the equivalent Transfer function is:
--
--    y(s)              wn**2
--    ---- = -------------------------
--    x(s)   s**2 + 2*Zeta*Wn*s + Wn**2
```

```
--
-- The output is a memory variable.
```

3.7 COMPAR

```
SUBMODEL COMPAR(LOGICAL:y := REAL:x1,x2);

-- Sets a LOGICAL value from the comparison of the
-- amplitudes of two variables. The calling sequence is:
--
--    y:= COMPAR(x1,x2)
--
-- where:
--    x1 and x2 are input variables;
--    y is given a value such that:
--          y = TRUE, if x1 >= x2;
--          y = FALSE, if x1 < x2.
--
-- The output is a memory variable.
```

3.8 COMPB

```
SUBMODEL COMPB(LOGICAL:y := CONSTANT LOGICAL:IC;
                                CONSTANT REAL:LL,UL; REAL:x);

-- Comparator with a backlash. The calling sequence is:
--
--    y:= COMPB(IC,LL,UL,x)
--
-- where:
--    IC is the logical initial condition;
--    LL is the lower limit;
--    UL is the lower limit;
--    x is the input variable.
--
--    y is given a value such that:
--          y = TRUE, when x >= UL becomes TRUE;
--          y = FALSE, when x < LL becomes TRUE.
--
-- Note the inputs LL and UL must be UL > LL, and are assumed
-- constant throughout a run. The output is a memory variable.
```

3.9 COULOMB

```
SUBMODEL COULOMB(REAL:friction := CONSTANT REAL:Fs,Fc;
                                      REAL:velocity,force);

-- Computes the coulomb friction force which results from the
-- movement of sliding surfaces given their relative velocity
-- and the effective applied force. The remaining available
-- force is given by:
--    available force = applied force - frictional force
--
-- The calling sequence is:
--
--    friction:= COULOMB(Fs,Fc,velocity,force)
--
-- where:
--    friction   is the frictional force;
--    Fs         is the static limiting friction value;
--    Fc         is the coulomb friction value;
--    velocity   is the relative velocity of sliding surfaces;
--    force      is the applied force (causing sliding motion).
-- Note the inputs Fs, Fc are assumed constant throughout a run.
-- The output is an algebraic variable.
```

3.10 CPXPL

```
SUBMODEL CPXPL(REAL:y := CONSTANT REAL:IC1,IC2,Z,W; REAL:x);

-- Represents a second order transfer function with fixed complex
-- poles at (-Z - jW) and (-Z + jW), i.e.:
--
--    Y(s)           1
--    ---- = ------------------- .
--    X(s)   (s + Z+jW)(s + Z-jW)
--
-- The corresponding differential equation which is solved is:
--
--    y'' = x - 2*Z*y' - (Z**2 + W**2)*y .
--
-- The calling sequence is:
--
--    Y:=CPXPL(IC1,IC2,Z,W,X)
--
-- where:
--    IC1 and IC2 are the initial values of Y and Y' respectively;
--    Z and W are the real and imaginary parts of the poles;
--    X is the input variable.
--
-- Setting Z=0 gives imaginary poles at +jW and -JW, and
-- setting W=0 gives a double real pole at -Z.
--
-- Note the inputs Z and W are assumed constant throughout a run.
-- The output is a memory variable.
```

3.11 DEADSP

```
SUBMODEL DEADSP(REAL:y := CONSTANT REAL:LL,UL; REAL:x);

-- Simulates the effect of a 'deadspace'. The calling
-- sequence is:
--
--    y:= DEADSP(LL,UL,x)
--
-- where:
--    LL is the lower limit;
--    UL is the upper limit;
--    x is the input variable;
--    y is given a value such that:
--          y = 0.0, if LL < x < UL;
--          y = x-UL, if x >= UL;
--          y = x-LL, if x <= LL.
--
-- Note the inputs LL, UL must be UL > LL, and are assumed constant
-- throughout a run. The output is an algebraic variable.
```

3.12 DELAY

```
SUBMODEL DELAY(REAL:y := CONSTANT INTEGER:N; CONSTANT REAL:WAIT;
                                                          REAL:x);

-- Samples the input periodically and delays these sampled
-- values for a specified time period to produce a delayed
-- output. The calling sequence is:
--
--    y:= DELAY(N,WAIT,x)
--
-- where:
--    N is the number of samples during period WAIT;
--    WAIT is the time of the delay;
--    x is the input variable;
--    y is the sampled input delayed by a period WAIT.
--
-- The sampling period is:
--    per = WAIT/N
--
-- Note the inputs N and WAIT are assumed constant throughout a
-- run. The output is a memory variable.
```

3.13 DERIV

```
SUBMODEL DERIV(REAL:y := CONSTANT REAL:IC; REAL:x);

-- Outputs an approximation to the first derivative value of a
-- real input. The calling sequence is:
--
--    y:= DERIV(IC,x)
--
-- where:
--    IC is the initial condition;
--    x is the input variable;
--    y is given a value such that:
--          y = IC, initially;
--          y = (x-xlast)/(T-Tlast), elsewhere.
--
-- The output is an algebraic variable.
```

3.14 FG3D

```
PROCEDURE FG3D(REAL:TABLE(*),x,y,z) RETURN REAL;
-----------------------------------------------------------------
-- The function is called as:
--
--      REAL: f,x,y,z;
--      REAL: TABLE(xx)/....../;
--
--      f:= FG3D(TABLE,x,y,z);
--
-- The "TABLE" array may define one, two or three dimension
-- generation. When one dimension generation is defined the
-- values of x and z in the call to FG3D are ignored, similarly
-- the value of z is ignored in two dimension generation.
--
-- The "TABLE" data comprises:
--
--      no of independent variables,  dimensions,  function data.
--
-- where:
--
```

```
--      no of independent variables must be 1.0, 2.0 or 3.0.
--
--      dimensions: for each independent variable the number
--      of data points provided. A dimension of one is illegal.
--
--      function data the values of independent variables are listed
--      first. The number of these points must equal the sum of the
--      dimensions. Then the dependent values are listed for the
--      first independent variable changing fastest. The number of
--      data points must equal the product of the dimensions.
--
-- All data points, breakpoints, for each independent variable must
-- be in monotonically increasing order. Intermediate values may be
-- equal, but a breakpoint must never be less than the preceeding value.
--
-- Linear extrapolation is used when breakpoints are outside the
-- specified range, linear interpolation is used for data between
-- breakpoints
```

3.15 FGEN0

```
SUBMODEL FGEN0(REAL:y:= CONSTANT REAL:TABLE(2,*); REAL:x);

-- Searches a table of x-y coordinate values (ascending x order)
-- and returns a zero order interpolation corresponding to input
-- x. That is the y-table-value corresponding to the largest
-- x-table-value which is less than or equal to x. If x is less
-- than the first x-table-value then the first y-table-value is
-- returned. The calling sequence is:
--
--    y:= FGEN0(TABLE,x)
--
-- where:
--    TABLE is the table of values which represents a two row
--          n column matrix, row 1 represents the input values
--          which must be in ascending order, and row 2 represents
--          the corresponding function values;
--    x  is the input value;
--    y is returned value.
--
-- The output is a memory variable.
```

3.16 FGEN1

```
SUBMODEL FGEN1(REAL:y:= CONSTANT REAL:TABLE(2,*); REAL:x);

-- Searches a table of x-y coordinate values and finds
-- which values span the input value of x and performs a
-- first order interpolation to obtain a value for y. The
-- calling sequence is:
--
--    y:= FGEN1(TABLE,x)
--
-- where:
--    TABLE is the table of values which represents a two row
--          n column matrix, row 1 represents the input values
--          which must be in ascending order, and row 2 represents
--          the corresponding function values;
--    x  is the input value;
--    y is returned value.
--
-- y is calculated by use of the equation:
--
```

```
--    y = ycl+(x-xcl)*grad
--
-- where grad = (ycu-ycl)/(xcu-xcl), and
--
--    ycu - TABLE(2,i) array upper segment value;
--    ycl - TABLE(2,i-1) array lower segment value;
--    xcu - TABLE(1,i) array upper segment value;
--    xcl - TABLE(1,i-1) array lower segment value.
--
-- The output is an algebraic variable.
```

3.17 FGEN2

```
SUBMODEL FGEN2(REAL:y:= CONSTANT REAL:TABLE(2,*); REAL:x);

-- Searches a table of x-y coordinate values and finds
-- which values span the input of x and performs a second
-- order interpolation to obtain a value for y.
-- Over any segment, a number of simple polynomial fits
-- is found and the interpolated function, y, is a
-- weighted average over two of these; the weighting
-- being a function of the independent variable.
-- The weighting function has a zero slope at the points
-- where the interpolating polynomials switch over. In
-- the range TABLE(1,i) to TABLE(1,i+1), two quadratics
-- are used; Qn fits TABLE(2,i-1), TABLE(2,i) and
-- TABLE(2,i+1) and Q1n fits TABLE(2,i),
-- TABLE(2,i+1) and TABLE(2,i+2). Qn and Q1n overlap in the
-- range TABLE(1,i) to TABLE(1,i+1). In this range:
--
--    y = w*Qn+(1-w)*Q1n
--
-- where:
--    w = 1-3*s**2+2*s**3, is the weighting function,
--    s = (x-TABLE(1,i))/(TABLE(1,i+1)-TABLE(1,i)),
--        the independent variable,
--
--    s has a normalised range [0,1] and,
--    w(0) = w(1) = w'(0) = w'(1) = 0.0.
--
-- The calling sequence is:
--
--    y:= FGEN2(TABLE,x)
--
-- where:
--    TABLE is the table of values which represents a two row
--          n column matrix, row 1 represents the input values
--          which must be in ascending order, and row 2 represents
--          the corresponding function values;
--    x  is the input value;
--    y is returned value.
--
-- The output is an algebraic variable.
```

3.18 FHOLD

```
SUBMODEL FHOLD(REAL:y := CONSTANT REAL:IC,per; REAL:x);

-- Provides a first order hold function. It periodically
-- samples and holds the value of an input; the output is
-- 'held' value modified by the slope resulting from the
-- previous two samples. The calling sequence is:
--
```

```
--    y:= FHOLD(IC,per,x)
--
-- where:
--    IC is the initial condition, (y at sampling point,
--          TSTART-per);
--    per is the sampling period, and is treated as constant
--          through a run;
--    x is the input variable;
--    y is given a value such that;
--    y = last sample + slope*(T-start)
--          where slope is the gradient joining the two
--          previously sampled input values, and start is
--          the time at the start of the present period.
--
-- The output is an algebraic variable.
```

3.19 FOURINT

```
SUBMODEL FOURINT(REAL:mag,angle := CONSTANT INTEGER: n;
                                  CONSTANT REAL:Tperiod; REAL:x);

-- Calculates the rms magnitude and angle of a specified
-- harmonic component of the Fourier series for an input
-- signal. The submodel detects when the signal becomes
-- greater than zero, times the period of the signal and
-- computes the Fourier coefficients. The main output is
-- the statistics printed at the end of each cycle. They
-- give values of the computed coefficients and the size of
-- changes which have occurred during the last cycle. Small
-- changes indicate that a steady-state has been achieved
-- and the results can be treated with some confidence.
-- The calling sequence is:
--
--    mag,angle:= FOURINT(n,Tperiod,x)
--
-- where:
--    mag and angle are the computed Fourier coefficient rms
--          magnitude and angle (degrees);
--    n is the harmonic number: 1, 2, 3, etc.;
--    Tperiod is the approximate period specified by the user,
--          it need not be accurate but must be non-zero;
--    x is the object waveform whose coefficients are to be
--          computed.
--
-- The output is a memory variable.
```

3.20 HSTRSS

```
SUBMODEL HSTRSS(REAL:y := CONSTANT REAL:IC,LL,UL; REAL:x);

-- Simulates a pure hysteresis or backlash function. The
-- calling sequence is:
--
--    y:= HSTRS(IC,LL,UL,x)
--
-- where:
--    IC is the initial condition for y;
--    (x - LL) is the lower limit line on the y/x graph;
--    (x - UL) is the upper limit line on the y/x graph;
--    x is the input function.
--    Note UL > LL.
--
--    y is given a value such that:
```

```
--
--          initially y = x-UL, if (x-UL) >= IC,
--                      = x-LL, if (x-LL) <= IC,
--                      = IC otherwise;
--          y = x-UL, if (x-UL) >= Ylast;
--          y = x-LL, if (x-LL) <= Ylast;
--          y = Ylast, otherwise.
--
-- Ylast is the previous value of y.
--
-- Note UL > LL, and UL and LL are considered constant throughout
-- run. The output is an algebraic variable.
```

3.21 IMPUL

```
SUBMODEL IMPUL(LOGICAL:y := CONSTANT REAL:Td,per);

-- Generates a periodic logical train of impulses (TRUE for
-- zero time) following an initial time delay. The calling
-- sequence:
--
--    y:= IMPUL(Td,per)
--
-- where:
--    Td is the time delay before first pulse (at TSTART+Td);
--          if Td zero, or negative, the first pulse will occur
--          at (TSTART+per),
--    per is the interval (period) between pulses.
--
-- Note the time delay (Td) or period (per) cannot be changed
-- once simulation has started. The output is a memory variable.
```

3.22 INTEG

```
SUBMODEL INTEG(REAL:y := CONSTANT REAL:IC; REAL:x);

-- Standard integrator, ordinary style CSSL type. The
-- calling sequence is:
--
--    y:= INTEG(IC,x)
--
-- where:
--    IC is the initial condition;
--    x is the input variable.
--
-- The output is a memory variable.
```

3.23 INTX

```
SUBMODEL INTX(INTEGER:y := REAL:x);

-- Outputs the integer value of a real input. This is the
-- modelling version of the standard function which treats
-- changes in output as discontinuities. The calling
-- sequence is:
--
--    y:= INTX(x)
--
-- where:
--    x is the input variable;
--    y is given a value that is equal to INT(x).
```

```
--
-- Note, with the sign removed, x is truncated to the
-- largest integer less than or equal to x and then the
-- sign of x is added to the integer result. The output
-- is a memory variable.
```

3.24 LEDLAG

```
SUBMODEL LEDLAG(REAL:y := CONSTANT REAL:YIC,P1,P2; REAL:x);

-- Generates a lead-lag transfer function. The calling
-- sequence is:
--
--    y:= LEDLAG(YIC,P1,P2,x)
--
-- where:
--    YIC is the initial condition for y;
--    P1 and P2 are constants which must be non-zero;
--    x is the input variable.
--
-- The differential equation is given by:
--
--    P2*y'+y = P1*x'+x
--
-- and the equivalent Laplace Transform function is:
--
--    y(s)   P1*s + 1
--    ---- = -------- .
--    x(s)   P2*s + 1
--
-- Note the inputs P1, P2 are assumed constant throughout a run.
-- The output is an algebraic variable.
```

3.25 LIMINT

```
SUBMODEL LIMINT(REAL:y := CONSTANT REAL:IC,LL,UL; REAL:x);

-- A limited  function, which holds the integrator at a
-- limit as long as the derivative is of such a sign to
-- drive it further into limit. When the derivative
-- reverses sign, the integrator will immediately come off
-- limit. The calling sequence is:
--
--    y:= LIMINT(IC,LL,UL,x)
--
-- where:
--    IC is the initial condition;
--    LL is the lower limit on y;
--    UL is the upper limit on y;
--    x is the expression for the derivative.
--    y is given a value such that:
--          y = INTGL(x), with y(TSTART) = IC, and
--          y' = 0, if y > UL and x >= 0.0,
--          y' = 0, if y < LL and x <= 0.0,
--          y' = x, if LL <= y <= UL.
--
-- Note the inputs LL, UL must be UL > LL, and are assumed
-- constant throughout a run. The output is a memory variable.
```

3.26 LIMIT

```
SUBMODEL LIMIT(REAL:y := CONSTANT REAL:LL,UL; REAL:x);

-- A limiter sets lower and upper limits on the amplitude
-- of an input variable. The calling sequence is:
--
--    y:= LIMIT(LL,UL,x)
--
-- where:
--    LL is the lower limit;
--    UL is the upper limit;
--    x is the input variable.
--    y is given a value such that:
--          y = x, if LL < x < UL,
--          y = UL, if x >= UL,
--          y = LL, if x <= LL.
--
-- Note the inputs LL, UL must be UL > LL, and are assumed
-- constant throughout a run. The output is an algebraic
-- variable.
```

3.27 LOGINT

```
SUBMODEL LOGINT(REAL:y := LOGICAL:reset,integ; CONSTANT REAL:IC;
                                                          REAL:x);

-- Simulates a logically controlled integrator. The
-- calling sequence is:
--
--    y:= LOGINT(reset,integ,IC,x)
--
-- where:
--    reset is a logical input. When reset becomes TRUE,
--          the integrator is set to its initial condition
--          (edge-triggering);
--    integ is the logical input; if TRUE then integration
--          takes place, else the current integrator output
--          is held;
--    IC is the initial condition;
--    x is the input variable.
--    y is given a value such that:
--          y' = x, if integ is TRUE,
--          y' = 0, if integ is FALSE.
--
-- The output is a memory variable.
```

3.28 MODULT

```
SUBMODEL MODULT(LOGICAL:Y := CONSTANT REAL:Td; REAL:sig;
                                           CONSTANT REAL:per);

-- Logical pulse width modulator which generates a logical
-- pulse train with specified period and a mark-space
-- ratio. An initial delay is permitted, and the initial
-- output may be specified as TRUE or FALSE. The calling
-- sequence is:
--
--    y:= MODULT(Td,sig,per)
--
-- where:
--    Td is the time at which the pulse train starts. If
--          Td >= 0.0, y will remain FALSE for Td seconds. If
--          Td < 0.0, pulse train will remain TRUE for
--          (-Td) seconds.
--    sig is the modulating signal in the range (0..1).
--    per is the period of the pulse train in units of T.
--
-- Note that Td and per are regarded as constant during a run.
-- The output is a memory variable.
```

3.29 MONO

```
SUBMODEL MONO(LOGICAL:y := REAL:w,x);

-- A monostable function sets the output TRUE for a
-- specified time period when the input becomes positive,
-- it will remain TRUE if the input also remains TRUE.
-- The calling sequence is:
--
--    y:= MONO(w,x)
--
-- where:
--    w is the width of the pulse;
--    x is the input variable;
--    y is given a value such that:
--          y = TRUE, when x >= 0.0 becomes TRUE and stays TRUE
--                for at least w units of T, or until x
--                becomes negative;
--          y = FALSE, otherwise.
--
-- The output is a memory variable.
```

3.30 PICONT

```
SUBMODEL PICONT(REAL:y := CONSTANT REAL:IC; REAL:TC,K,x);

-- This submodel defines a proportional plus integral (PI)
-- controller. The calling sequence is:
--
--    y:= PICONT(IC,TC,K,x)
--
-- where:
--    IC is the integrator initial condition, z(TSTART) = IC;
--    TC is the time constant of the integrator;
--    K is the proportional gain;
--    x is the input variable.
--
-- The differential equations are given by:
--
--    z' = x/TC
--
--    y = K*(x+z)
--
-- and the equivalent Laplace Transform function is:
--
--    y(s)        K
--     --- = K + ----
--    x(s)       s*TC
--
-- The output is an algebraic variable.
```

3.31 PIDCONT

```
SUBMODEL PIDCONT(REAL: u :=  CONSTANT REAL: ic, prop_band, Ti, Td;  REAL:
e);
-------------------------------------------------------------------------
-- Three Term Controller with limited output and integral anti-windup
--
-- inputs:
--          ic         initial value of integral term
--          prop_band  proportional gain
--          Ti         integral action time constant
--          Td         derivative action time constant
--          e          error signal
-- output:
--          u          control actuation signal
--
-- The output is calculated as:
--
--          u = 100/prop_band*(e + 1/Ti*integral(e) + Td*derivative(e))
--
-- The output, u, is limited to the range -1.0 to + 1.0.
--
-- Integral action is inhibited if the output is in limit and the input
-- is such as to force the output further into limit i.e.
--
--          if u >= 1.0 and e > 0.0 or u <= -1 and e < 0.0
```

3.32 PIDCONT1

```
SUBMODEL PIDCONT1(REAL: u, e :=
                  CONSTANT REAL: ic, prop_band, dead_band, Ti, Td;
                  REAL: sp, um, pv);
-------------------------------------------------------------------------
-- Three Term Controller with limited output, anti-windup and dead-space
--
-- inputs (constants):
--          ic         initial value of integral term
--          prop_band  proportional gain (%)
--          dead_band variation in error for no change in output (%)
--          Ti         integral action time constant
--          Td         derivative action time constant
--
-- inputs (variables):
--          sp         set point
--          pv         process variable
--          um         offset - to make error zero when control action
--                              is proportional only
-- outputs:
--          u          control actuation signal
--          e          error signal (provided for monitoring)
--
-- The output is calculated as:
--
--      u = 100/prop_band*(e + 1/Ti*integral(e) + Td*derivative(e)) + um
--
-- Inputs are assumed to be normalized i.e. in range -1 to +1.
--
-- The output, u, is limited to the range -1.0 to + 1.0.
--
-- Integral action is inhibited if the output is in limit and the input
-- is such as to force the output further into limit i.e.
--
--          if u >= 1.0 and e > 0.0 or u <= -1 and e < 0.0
--
-- The error has no effect if within the dead band i.e.
```

```
--
--       effective error = if e < -dead_band/100 then e + dead_band/100
--                      else_if e >  dead_band/100 then e - dead_band/100
--                      else 0.0;
```

3.33 POLYROOTS

```
PROCEDURE POLYROOTS(REAL:p(*), A(*,*), Y(*,*); INTEGER:n) RETURN REAL;
------------------------------------------------------------------------
-- Finds complex roots of a polynomial using Eigenvalue method
--
--     Y := POLLYROOTS(p, A, Y, n);
--
-- p   Vector (size n+1) of polynomial coefficients in ascending order
-- A   nxn work matrix;
-- Y   nx2 matrix returning complex roots
-- n   Order of polynomial
```

3.34 POLYVAL

```
PROCEDURE POLYVAL(REAL:p(*),x; INTEGER:n) RETURN REAL;
------------------------------------------------------------------------
-- Evaluates a polynomial value using nested multiplication
--
--           y := POLYVAL(p, x, n);
--
-- p   Vector (size n+1) of polynomial coefficients in ascending order
-- x   Real value at which polynomial is to be evaluated
-- n   Order of polynomial
```

3.35 PULSE

```
SUBMODEL PULSE(REAL:Y := CONSTANT REAL:T_ON,T_OFF);
-----------------------------------------------------------------
-- Generates a unit pulse of specified duration
--
--    y:= PULSE(T_ON, T_OFF)
--
-- where:
--          T_ON -    pulse start time
--          T_OFF -   pulse finish time
--
-- The output is an algebraic variable.
```

3.36 QNTZR

```
SUBMODEL QNTZR(REAL:y := CONSTANT REAL:P; REAL:x);

-- Quantizes the input variable x (with quantization
-- interval P) so that the output is the largest value of
-- n*P < x where n is an integer. The calling sequence is:
--
--    y:= QNTZR(P,x)
--
-- where:
--    P is a constant;
--    x is the input variable;
--    y is given a value such that:
--          y = i*P
--          where i is the largest integer such that, i*p <= x.
--
-- Note the input P is assumed constant throughout a run.
-- The output is a memory variable.
```

3.37 RAMP

```
SUBMODEL RAMP(REAL:y := REAL:Tgo);

-- Generates a linear ramp of unit slope starting at a
-- specified time. The calling sequence is:
--
--    y:= RAMP(Tgo)
--
-- where:
--    Tgo is the start time of the ramp;
--    y is given a value such that:
--          y = 0.0, if T < Tgo,
--          y = T-Tgo, if T >= Tgo.
--
-- The output is an algebraic variable.
```

3.38 REALPL

```
SUBMODEL REALPL(REAL:y := CONSTANT REAL:IC,P; REAL:x);

-- Generates a first order real pole transfer function.
-- The calling sequence is:
--
--    y:= REALPL(IC,P,x)
--
-- where:
--    IC is the initial condition, y(TSTART) = IC;
--    P is a constant;
--    x is the input variable.
--
-- The differential equation is given by:
--
--    P*y'+y = x
--
-- and the equivalent Laplace Transform function is:
--
--    y(s)       1
--    ---- = ------- .
--    x(s)    P*s + 1
--
-- Note the input P is assumed constant throughout a run.
-- The output is a memory variable.
```

3.39 RECNS0

```
SUBMODEL RECNS0(REAL: y:= CONSTANT REAL: XIC,TAU; REAL:x);

-- Outputs the reconstruction of the data values which are
-- input arguments to a segment, or output arguments from a
-- segment. The calling sequence is:
--
--    y:= RECNS0(XIC,TAU,x);
-- where:
--    XIC corresponds to the value of x at (TSTART-TAU),
--          and is also the initial output value for y.
--    TAU must be a multiple of CINT, and equal to the
--          communication interval of the segment. That is,
--          it is the time period between updates in x.
--    x is the input variable, which is updated after each
--          interval of TAU, a multiple of CINT.
--    y is the output which is computed by:
--          y = xlast + dx * (T-Tlast)
--              xlast is the previous value of x at (Tlast-TAU),
--              dx is the most recent derivative of x.
-- Note that y is a first-order interpolation of the input x, it
-- contains no jumps, or steps. That is, no high frequency
-- components are introduced, but a phase error of TAU results.
-- Note TAU is considered constant throughout a simulation run.
-- The output is an algebraic variable.
```

3.40 RECNS1

```
SUBMODEL RECNS1(REAL: y:= CONSTANT REAL: XIC,TAU; REAL:x);

-- Outputs the reconstruction of the data values which are
-- input arguments to a segment, or output arguments from a
-- segment. The calling sequence is:
--
--    y:= RECNS1(XIC,TAU,x);
--
-- where:
--    XIC corresponds to the value of x at (TSTART-TAU),
--          and is also the initial output value for y.
--    TAU must be a multiple of CINT, and equal to the
--          communication interval of the segment. That is,
--          it is the time period between updates in x.
--    x is the input variable, which is updated after each
--          interval of TAU, a multiple of CINT.
--    y is the output which is computed by:
--          y = xlast + dx * (T-Tlast)
--              xlast is the current value of x,
--              Tlast is point when x last changed,
--              dx is the most recent approx derivative of x;
--
-- Note that y is a first-order prediction of the input x, it
-- contains jumps, or steps, caused by the approximate derivative.
-- That is, high frequency components are introduced, but a phase
-- error is minimal. Note TAU is considered constant throughout a
-- simulation run. The output is an algebraic variable.
```

3.41 RECT

```
SUBMODEL RECT(LOGICAL:cond := REAL:i,v; LOGICAL: gate;
                                             CONSTANT LOGICAL:ic);

-- Determines whether the rectifier is conducting. The
-- calling sequence is:
--
--    cond:= RECT(i,v,gate)
--
-- where:
--    cond is a logical memory variable set TRUE if conducting;
--    i is the current through rectifier;
--    v is the voltage across rectifier, a positive value
--          indicates rectifier could conduct if a gate pulse
--          is also present;
--    gate is a logical TRUE if gate pulse is present;
--    ic is true if rectifier initially conducting, else false.
--
-- It is assumed that the calling code will set the voltage
-- to zero during periods of conduction, and the current is
-- held at zero or slightly negative during non-conduction.
-- The output is a memory variable.
```

3.42 SAMHLD

```
SUBMODEL SAMHLD(REAL:y := CONSTANT REAL:per; REAL:x);
-- Samples and holds the value of an input variable.
-- Samples are taken periodically and the output is the
-- value of the last sample taken. The calling sequence is:
--
--    y:= SAMHLD(per,x)
--
-- where:
--    per is the sampling period;
--    x is the input variable;
--    y is given a value such that:
--          y = x, initially,
--          y = x, at the last sampling period.
--
-- Note per is assumed constant throughout a simulation run.
-- The output is a memory variable.
```

3.43 SQRTX

```
PROCEDURE SQRTX(REAL:x)RETURN REAL;

-- Outputs the square root value of a real input. The
-- calling sequence is:
--
--    y:= SQRTX(x)
--
-- where:
--    x is the input variable;
--    y is given a value such that:
--          y = if x is positive then SQRT(x)
--              else -SQRT(-x).
--
-- This modelling version of the standard function is
-- necessary because due to minor errors, or during
-- integration interpolation passes, it is possible that
-- x may become negative. This formulation avoids failures
-- due to this cause. The output is an algebraic variable.
```

3.44 STEPP

```
SUBMODEL STEPP(LOGICAL:y := REAL:Tgo);

-- Produces a change from FALSE to TRUE at a specified step
-- time (Tgo). The calling sequence is:
--
--    y:= STEPP(Tgo)
--
-- where:
--    Tgo is the start time of the step;
--    y is given a value such that:
--          y = FALSE, if T < Tgo,
--          y = TRUE, if T >= Tgo.
--
-- The output is a memory variable.
```

3.45 TIMER

```
SUBMODEL TIMER(REAL:elapsed := LOGICAL:reset);

-- Returns the elapsed time since the last reset. In the
-- time returned is that time since the previous reset
-- point, and the following invocations will return the
-- time from the most recent reset point. The calling
-- sequence is:
--
--    elapsed:= TIMER(reset)
--
-- where:
--    elapsed is the simulated time since the last reset;
--    reset is a logical expression which resets the timing
--          mechanism when it becomes true, note that the
--          reset operation only takes place at the instant
--          it changes from false to true. During the
--          invocation in which the reset occurs the elapsed
--          time returned is for the period from the previous
--          reset
-- The output is an algebraic variable.
```

3.46 ZHOLD

```
SUBMODEL ZHOLD(REAL:y := CONSTANT REAL:IC; LOGICAL:hold; REAL:x);

-- Simulates the effect of a zero-order hold with logical
-- input control. The calling sequence is:
--
--    y:= ZHOLD(IC,hold,x)
--
-- where:
--    hold and x are input variables;
--    y is given a value such that:
--          y = last value of output, if hold is TRUE,
--          y = x, elsewhere.
--
-- The output is an algebraic variable.
```

CHAPTER 4

ESL Syntax

4.1 Syntax Summary

This section provides a full MBNF syntax definition of the ESL language, with a page references to each syntax statement, and then an alphabetical list of the "terminal Symbols" and keywords.

```
upper_case_letter =
                    "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
                    "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
                    "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z".
lower_case_letter =
                    "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
                    "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
                    "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z".
digit =
                    "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
special_character =
                    """" | "(" | ")" | "*" | "+" | "," | "-" | "." | "/" |
                    ":"  | ";" | "<" | "=" | ">" | "_" | "[" | "]".
space_character =
                    " ".
other_special_character =
                    "!" | "'" | "£" | "$" | "%" | "&" | "?" | "@" | "\" |
                    "^" | "{" | "|" | "}" | "~".
identifier =
                    letter {letter | digit | "_"}.
letter =
                    upper_case_letter | lower_case_letter.
variable =
                    identifier ["(" subscript {"," subscript} ")" ].
subscript =
                    expression [".." expression ].
derivative_identifier =
                    identifier "'" { "'" }.
derivative_variable =
                    derivative_identifier
                    ["(" subscript {"," subscript} ")" ].
character_string =
                    """" character {character} """" |
                    "%"  character {character} "%".
```

```
character =
                    letter | digit | space_character |
                    special_character |
                    other_special_character.
number=
                    real_number | integer.
real_number =
                    integer ("." (integer [exponent] | exponent) | exponent ).
exponent =
                    ("E"|"e"|"D"|"d") ["+"|"-"] integer.
integer =
                    digit {digit}.
library_declaration =
                    "--" "LIBRARY" file_identifier {"," file_identifier}.
file_identifier =
                    character {character}.
include_statement =
                    "INCLUDE" character_string ["-" ("L"|"l")] ";".
program =
                    study_program | remote_program |
                    embedded_program | non_program.
study_program =
                    "STUDY"
                    {program_unit}
                    experiment
                    "END_STUDY".
remote_program =
                    "REMOTE"
                    {package_specification |
                    procedure_subprogram |
                    function_subprogram |
                    submodel_subprogram }
                    segment_subprogram.
embedded_program =
                    "EMBEDDED"
                    {package_specification |
                    procedure_subprogram |
                    function_subprogram |
                    submodel_subprogram }
                    segment_subprogram.
non_program =
                    {program_unit}.
```

```
program_unit =
                    package_specification |
                    procedure_subprogram |
                    function_subprogram |
                    model_subprogram |
                    submodel_subprogram |
                    segment_subprogram |
                    external_segment_declaration.
model_subprogram =
                    "MODEL" identifier argument_specification ";"
                    declarations
                    model_body
                    "END" [identifier] ";".
model_body =
                    ["INITIAL" statements]
                    "DYNAMIC" dynamic_region_code
                    ["TERMINAL" statements]
                    ["ANALYSIS" statements].
dynamic_region_code =
                    {model_statement ";"}
                    ["STEP" statements]
                    ["COMMUNICATION" statements].
submodel_subprogram =
                    "SUBMODEL" identifier argument_specification ";"
                    declarations
                    submodel_body
                    "END" [identifier] ";".
submodel_body =
                    ["INITIAL" statements]
                    "DYNAMIC" dynamic_region_code.
submodel_call_statement =
                    output_arguments ":="
                    identifier "(" input_arguments ")" ";".
segment_subprogram =
                    "SEGMENT" identifier argument_specification ";"
                    declarations
                    segment_body
                    "END" [identifier] ";".
segment_body =
                    ["INITIAL" statements]
                    "DYNAMIC" dynamic_region_code.
```

```
external_segment_declaration =
                    "SEGMENT" identifier argument_specification
                    "EXTERNAL" ";"
                    [declarations segment_body]
                    "END" [identifier] ";".
procedure_subprogram =
                    "PROCEDURE" identifier
                    ["("[argument_list]")"]";"
                    procedure_specification
                    "END"[identifier]";".
procedure_specification =
                    declarations
                    statements.
function_subprogram =
                    "PROCEDURE" identifier
                    "("[argument_list]")" "RETURN" type ";"
                    procedure_specification
                    "RETURN" expression ";"
                    "END"[identifier]";".
function_call =
                    identifier "(" expression {"," expression} ")".
package_specification =
                    "PACKAGE" identifier";"
                    declarations
                    "END" [identifier]";".
argument_specification =
                    [ "(" [output_argument_list]
                    [ ":=" input_argument_list] ")" ].
output_argument_list =
                    argument_list.
input_argument_list =
                    input_argument_declaration
                    { "," input_argument_declaration}.
input_argument_declaration =
                    ["CONSTANT"] variable_type_declaration | file_declaration.
argument_list =
                    argument_declaration
                    { ";" argument_declaration}.
argument_declaration =
                    variable_type_declaration | file_declaration.
variable_type_declaration =
                    type ":"
                    variable_declaration
                    {"," variable_declaration}.
variable_declaration =
                    identifier ["("(dimension_bounds|"*")
                    {","(dimension_bounds|"*")}")"].
type =
                    "REAL"|"INTEGER"|"LOGICAL"|"CHARACTER".
declarations =
                    { (external_declaration
                    | file_declaration
                    | constant_declaration
                    | parameter_declaration
                    | type_declaration
                    | use_declaration
                    | nosort_declaration) }.
type_declaration =
                    type ":" declaration_variable
                    [("/" | "[") aggregate ("/" | "]")]
                    {"," declaration_variable
                    [("/" | "[") aggregate ("/" | "]")]}.
declaration_variable =
                    identifier
                    [ "(" dimension_bounds {"," dimension_bounds} ")"].
dimension_bounds =
                    ["-"] integer [".." ["-"] integer].
aggregate =
                    aggregate_element {"," aggregate_element }.
aggregate_element =
                    {(identifier | integer) "*"}
                    (identifier | ["+"|"-"] (integer | real_number ) |
                    "FALSE" | "TRUE" | character_string).
constant_declaration =
                    "CONSTANT" type ":" declaration_variable
                    ("/" | "[") aggregate ("/" | "]")
                    {"," declaration_variable
                    ("/" | "[") aggregate ("/" | "]")} ";".
parameter_declaration =
                    "PARAMETER" type ":" declaration_variable
                    ("/" | "[") aggregate ("/" | "]")
                    {"," declaration_variable
                    ("/" | "[") aggregate ("/" | "]")}.
file_declaration =
                "FILE" ":" file_specifier {"," file_specifier} ";".
file_specifier =
                identifier.
external_declaration =
                "EXTERNAL" [type ":"] identifier
                {"," identifier} ";".
use_declaration =
                "USE" identifier {"," identifier} ";".
nosort_declaration =
                "NOSORT" ";".
experiment =
                declarations
                statements.
model_statement =
                model_variable_statement |
                submodel_call_statement |
                procedural_model_block |
                when_statement.
model_variable_statement =
                model_variable ":="
                expression | if_clause | transfer_expression | transfer_matrix_expression.
model_variable =
                identifier | derivative_identifier.
transfer_expression =
                "TRANSFER" "(" ( [gain]
                transfer_factor {transfer_factor} | gain )
                "/"( [pole] transfer_factor {transfer_factor} | pole )
                {"," initial_expression} ")" "*" input_expression ";".
gain =
                [unary_operator] coefficient.
coefficient =
                ["-"|"+"] identifier | number.
transfer_factor =
                "(" [unary_operator] transfer_term
                {adding_operator transfer_term } ")".
transfer_term =
                coefficient ["*" pole] | pole.
pole =
                ("S" | "s") ["**" integer ].

initial_expression =
                expression.
input_expression =
                expression.
```

```
transfer_matrix_expression =
                    "TRANSFER_MATRIX" "(" denominator
                    " [" matrix_row { “;” matrix_row } "]" ")" "*" input_expression ";".

denominator =
                    ([ pole ] transfer_factor { transfer_factor } | pole ).
matrix_row =
                    numerator { "," numerator }.

numerator =
                    (([ gain [ "*" zero ] | zero ]) transfer_factor { transfer_factor } | (gain [ "*" zero ] | zero )).

zero =
                    ( "S" | "s" ) [ "**" integer ].


if_clause =
                    "IF" logic_expression "THEN" expression
                    {"ELSE_IF" logic_expression "THEN" expression}
                    "ELSE" expression.
when_statement =
                    "WHEN" logic_expression "THEN"
                              statements
                    {"WHEN" logic_expression "THEN"
                              statements}
                    "END_WHEN".
logic_expression =
                    expression.
procedural_model_block =
                    "PROCEDURAL" ["(" [ output_list ]
                    [":=" input_list] ")"]";"
                    statements
                    "END_PROCEDURAL" ";".
output_list =
                    identifier {"," identifier}.
input_list =
                    model_variable {"," model_variable}.
statements =
                    {procedural_statement}.
```

```
procedural_statement =
                          procedural_assignment_statement
                         | if_statement
                         | loop_statement
                         | terminate_statement
                         | return_statement
                         | stop_statement
                         | subprogram_call
                         | open_statement
                         | create_statement
                         | rewrite_statement ]
                         | close_statement
                         | delete_statement
                         | print_statement
                         | read_statement
                         | readel_statement
                         | prepare_statement
                         | tabulate_statement
                         | plot_statement
                         | clear_screen_statement
                         | interact_statement
                         | trim_statement
                         | linearize_statement
                         | eigenvalue_statement
                         | optimize_statement
                         | resume_statement
                         | restart_statement
                         | snapshot_statement.
procedural_assignment_statement =
                         variable | derivative_identifier | derivative_variable
                         ":=" expression.
if_statement =
                         "IF" logic_expression "THEN"
                         statements
                         {"ELSE_IF" logic_expression "THEN"
                         statements}
                         ["ELSE"
                         statements]
                         "END_IF" ";".
loop_statement =
                         [iteration_specification] basic_loop.
```

```
iteration_specification =
                        ("FOR" identifier ":=" expression ".." expression
                        ["STEP" expression]) |
                        ("WHILE" logic_expression).
basic_loop =
                        "LOOP"
                        statements
                        "END_LOOP" ";".
terminate_statement =
                        "TERMINATE" logic_expression ";".
return_statement =
                        "RETURN" [expression] ";".
stop_statement =
                        "STOP" ";".
subprogram_call =
                        identifier["("output_arguments
                        [":="input_arguments]")"]";".
output_arguments =
                        variable | derivative_variable
                        {"," variable | derivative_variable}.
input_arguments =
                        expression {"," expression}.
linearize_statement =
                        "LINEARIZE" a_matrix "," b_matrix ":="
                        "[" state_vector "]" "," "[" input_vector "]"
                        [ c_matrix "," d_matrix ":=" "[" output_vector "]" ] ";".
state_vector =
                        ( identifier | derivative_identifier )
                        {","( identifier | derivative_identifier ) }.
input_vector =
                        identifier {","identifier}.
output_vector =
                        identifier {","identifier}.
a_matrix =
                        identifier.
b_matrix =
                        identifier.
c_matrix =
                        identifier.
d_matrix =
                        identifier.
trim_statement =
                        "TRIM" "[" control_vector "]" ":="
                        "[" derivative_vector "]" ";".
```

```
control_vector =
                    ( identifier | derivative_identifier )
                    {"," ( identifier | derivative_identifier ) }.
derivative_vector =
                    ( identifier | derivative_identifier )
                    {"," ( identifier | derivative_identifier ) }.
eigenvalue_statement =
                    "EIGENVALUE" eigenvalue_array ":="
                    system_matrix ";".
eigenvalue_array =
                    identifier.
system_matrix =
                    identifier.
optimize_statement =
                    "OPTIMIZE" identifier "(" variable ":="
                    variable {variable} ")" ";".
resume_statement =
                    "RESUME" subprogram_call ";".
restart_statement =
                    "RESTART" subprogram_call ";".
snapshot_statement =
                    "SNAPSHOT" [file_name] ";".
interact_statement =
                    "INTERACT" ";".
open_statement =
                    "OPEN" file_specifier "," file_name
                    ["," file_status] ";".
file_name =
                    character_expression.
create_statement =
                    "CREATE" file_specifier "," file_name
                    ["," file_status] ";".
rewrite_statement =
                    "REWRITE" file_specifier "," file_name
                    ["," file_status] ";".
close_statement =
                    "CLOSE" file_specifier ";".
delete_statement =
                    "DELETE" file_name ["," file_status] ";".
file_status =
                    "IOSTAT" "=" integer_identifier.
```

```
print_statement =
                    "PRINT" [ (file_specifier | print_element)
                    {"," print_element }] ";".
print_element =
                    expression [":" print_format_control]
                    | "/" {"/"}
                    | "-/".
print_format_control =
                    ["-"] integer ["." integer].
plot_statement =
                    "PLOT" [ plot_title ","]
                    independent_variable ","
                    dependent_variable
                    ( "," [more_depend_var] | more_depend_var)
                    x_min "," x_max ","
                    y_min "," y_max ";".
independent_variable =
                    model_variable | expression.
more_depend_var =
                    "[" dependent_variable
                    { "," dependent_variable } "]" [","].
dependent_variable =
                    model_variable | expression.
x_min =
                    expression.
        x_max =
                    expression.
        y_min =
                    expression.
        y_max =
                    expression.
clear_screen_statement =
                    "CLEAR_SCREEN" ";".
tabulate_statement =
                    "TABULATE" [ file_specifier |
                    file_name ","]
                    output_value {"," output_value} ";".
output_value =
                    model_variable | expression.
```

```
prepare_statement =
                    "PREPARE" file_name ","
                    [plot_title ","]
                    [plot_subtitle ","]
                    output_value "," [value_title ","]
                    {output_value "," [value_title "," ]} ";".
plot_title =
                    character_expression.
          plot_subtitle =
                    character_expression.
          value_title =
                    character_expression.
read_statement =
                    "READ" [ ( file_specifier | prompt | read_element )
                    {"," read_element}
                    ["," file_status] | file_status ] ";".
prompt =
                    character_string | "(" expression ")".
read_element =
                    variable [":" read_format_control ].
read_format_control =
                    ["-"] integer ["." integer].
readel_statement =
                    "READEL" [read_element {"," read_element}
                    ["," file_status] | file_status] ";".
expression =
                    logical_term {"OR" logical_term}.
logical_term =
                    logical_factor {"AND" logical_factor}.
logical_factor =
                    ["NOT"] logical_primary.
logical_primary =
                    simple_expression
                    [relational_operator  simple_expression].
simple_expression =
                    [unary_operator] term {adding_operator term}.
term =
                    factor {multiplying_operator factor}.
factor =
                    primary {exponentiating_operator primary}.
primary =
                    "(" expression ")" |
                    function_call |
                    variable |
                    derivative_identifier | derivative_variable
                    number |
                    character_string |
                    "FALSE" |
                    "TRUE".
relational_operator =
                    "=" | "/=" | "<" | "<=" | ">" | ">=".
adding_operator =
                    "+" | "-".
unary_operator =
                    "+" | "-".
multiplying_operator =
                    "*" | "/" | dot_product | cross_product.
exponentiating_operator =
                    "**".
dot_product =
                    ".".
cross_product =
                    "^".
```

4.2 Syntax Keywords

The following list all two, or more, character "terminal symbols" used in the syntax specification in alphabetic order.

**
--
-/
..
/=
:=
<=
>=
ANALYSIS
AND
CHARACTER
CLEAR_SCREEN
CLOSE
COMMUNICATION
CONSTANT
CREATE
DELETE
DYNAMIC
EIGENVALUE
ELSE
ELSE_IF
EMBEDDED
END
END_IF
END_LOOP
END_PROCEDURAL
END_STUDY
END_WHEN
EXTERNAL
FALSE
FILE
FOR
IF
INCLUDE
INITIAL
INTEGER
INTERACT
IOSTAT
LIBRARY
LINEARIZE
LOGICAL
LOOP
MODEL
NOSORT
NOT
OPEN
OPTIMIZE
OR
PACKAGE
PARAMETER
PLOT
PREPARE
PRINT
PROCEDURAL
PROCEDURE
READ
READEL
REAL
REMOTE
RESTART
RESUME
RETURN
REWRITE
SEGMENT
SNAPSHOT
STEP
STOP
STUDY
SUBMODEL
TABULATE
TERMINAL
TERMINATE
THEN
TRANSFER
TRANSFER_MATRIX
TRIM
TRUE
USE
WHEN
WHILE