BRITISH STANDARD

BS 5904:1980

Incorporating Amendment Nos. 1 and 2

Specification for

Computer programming language RTL/2

UDC 681.3.06RTL/2

Cooperating organizations

The Data Processing Standards Committee, under whose direction this British Standard was prepared, consists of representatives from the following Government departments and scientific and industrial organizations:

British Computer Society Ltd.*
British Equipment Trade Association*
British Paper and Board Industry Federation (PIF)
British Printing Industries Federation
Central Computer Agency (Civil Service Department)*
Committee of London Clearing Bankers on behalf of the Committee of Scottish Clearing Bankers, Cooperative Bank, Central Trustee Savings Bank and Yorkshire Bank
Department of Industry (Computers Systems and Electronics)
Department of Industry (National Physical Laboratory)*
Electricity Supply Industry in England and Wales*
Government Communications Headquarters
HM Customs and Excise
Institute of Cost and Management Accountants
Institute of Purchasing and Supply
Institution of Electrical Engineers
Institution of Mechanical Engineers
Inter-university Committee on Computing
London Transport Executive
Ministry of Defence*
National Computer Users' Forum
National Computing Centre Ltd.*
National Research Development Corporation
Post Office*
Society of British Aerospace Companies Limited

The organizations with an asterisk in the above list, together with the following, were directly represented on the committee entrusted with the preparation of this British Standard:

Association for Literary and Linguistic Computing
Association of Computer Units in Colleges of Higher Education (ACUCHE)
British Gas Corporation
Computing Services Association
Control and Automation Manufacturers' Association (BEAMA)
Edinburgh Regional Computing Centre
Engineering Equipment Users' Association
Hatfield Polytechnic
University of London

This British Standard, having been prepared under the direction of the Data Processing Standards Committee, was published under the authority of the Executive Board and comes into effect on 30 September 1980

The following BSI references relate to the work on this standard:
Committee reference DPS/13
Draft for comment 78/64581 DC

ISBN 0 580 11441 4

Amendments issued since publication

Amd. No.	Date of issue	Comments
3722	July 1981	
8233	October 1994	Indicated by a sideline in the margin

Contents

Foreword

This standard has been prepared under the direction of the Data Processing Standards Committee and extracts in this standard from "RTL/2 Language Specification", Version 2, ICI, 1974 are reproduced by permission of Imperial Chemical Industries Limited.

In drafting this standard the continued stability of RTL/2 has been a prime objective. However, apart from changes to clarify or correct the specification, a few minor alterations have seemed advisable. These affect the scope of record component names, strings, LET definitions, equality of label values and the character set. These alterations have been carefully designed to correct features of the original specification that with hindsight have proved to be inappropriate and yet do not alter the semantics of valid programs.

RTL/2 provides a method for writing both application and system programs for use in real-time computing and is especially suited for on-line data acquisition, communication and control systems.

The overall objectives of the language are:

a) to reduce the direct cost of the development and maintenance of software;

b) to encourage the creation of more reliable systems;

c) to increase the mobility of programming staff;

d) to provide continuity of method and flexibility in choice of equipment by ensuring the portability of application programs.

RTL/2 is designed to be as machine-independent as practicable even at the expense of certain (probably obsolescent) machine architectures. RTL/2 recognizes, however, that operating systems need a degree of flexibility that is incompatible with the security needs of application programs. Accordingly, RTL/2 comprises two languages, the full language being the system language, and a secure subset (see **7.6**) being the application language.

It is virtually impossible to define precisely a high-level language so that programs will be executed equally efficiently by all types of computer. It has therefore been necessary to leave certain areas of RTL/2 deliberately unspecified so that implementations can take the best advantage of characteristics of particular computers. The most, important of these areas are:

a) accuracy and range of real values;

b) number of bits in an integer word;

c) behaviour on arithmetic overflow.

Such areas are indicated in this standard by the use of phrases such as "implementation-dependent". On the other hand, it should be noted that the representation of integer values has been explicity specified to be in twos complement form; thus implementations of RTL/2 on computers that do not use this representation will be less efficient.

Program structure. A computer that is not specifically designed to execute RTL/2 code directly would need to be enhanced with various control routines to create an RTL/2 machine. In a simple single-state machine, the rest of the software, if derived from RTL/2, could be of items of software of similar status and this group of items would be known as a "program complex". More usually, however, and of necessity in a two-state machine, the remaining software has a hierarchical structure. The top level, usually known as the "supervisor", is itself a program complex and the individual groups of lower levels can themselves be considered to be program complexes, but with an environment different from that of the supervisor complex.

An RTL/2 complex consists of a collection of items known as bricks, of which there are the following types:

a) procedure brick;

b) data brick;

c) stack brick.

A procedure brick consists of the declaration of a single literal procedure. A procedure is a read-only piece of code describing an executable process, and may have parameters and local variables, but the latter are restricted to be scalars. The entry mechanism and implementation of local variables is reentrant. The coding of a procedure may directly access variables in a data brick, but not the local variables or parameters of another procedure. A procedure may not include internal procedures.

A data brick is a named static collection of scalars, arrays and records.

A stack brick consists of the declaration of a single literal stack. A stack is an area used for the storage of links, dynamic (i.e. local) variables and other housekeeping items.

Several bricks may be grouped together to form a module that is the unit of compilation. A program complex is the result of linking together one or more such modules.

The compiler needs to be informed of the environment of a module for satisfactory compilation to take place. This environment, in general, consists of two parts; first there is the environment of the complex as a whole and secondly there are interfaces with other modules in the complex.

The environment of the complex as a whole is, in the case of the supervisor, simply the enhanced machine, whereas in the case of a complex running under the supervisor in a two-state machine, it also consists of the environment provided by that supervisor. This environment comprises a set of procedures accessed as supervisor calls (SVC procedures) plus a set of data bricks private to each task and nominally housekept by the supervisor (SVC data). A module needs to include descriptions of SVC procedures and data bricks that it accesses.

To reference a brick within another module in the complex, a description of the brick needs to be included in the referring module, and the brick referred to in the other module needs to have been specified as an external entry in that module.

The various cross-references implied by these environment descriptions are satisfied by the linker program (see **7.5**).

Multitasking. A program complex may represent a simple program with a definite start and finish. It could however consist of the code and data for several processes running concurrently, and in order to describe such a complex the term “task” is introduced.

A task, broadly speaking, is an identifiable execution of a logically coherent set of instructions by a (pseudo-) processor.

A conventional computer with a single processor can be considered to be obeying at all times one unique task. It is, however, usually more convenient to consider the execution of each logically distinct process as a task and to convert the one actual processor into several psuedo-processors by a scheduling algorithm within a supervisor.

As a language, RTL/2 imposes few constraints on the design of multitasking systems and the actual facilities of such systems are outside the scope of this standard. It is, however, possible to describe in outline the intended relationships between the various bricks in a multitasking system.

The creation, control and elimination of tasks will, in general, be performed by a supervisor, and the supervisor call (see **7.5**) will provide the channel for the communication of task and control requests.

Whenever a new task is created, a stack will be nominated as workspace and a procedure will be nominated as the coding to be obeyed. Later operations on the task may be made by reference to its stack. Each stack can, of course, be used only by one task at a time, whereas procedures and data bricks may be used by several concurrent tasks. The procedure nominated as coding can call other procedures and access variables in data bricks as required.

Communication between tasks can be via supervisor calls and any message scheme provided by the supervisor, or simply via data bricks with the suitable use of semaphores.

An SVC data brick is private to each task and may be used in a reentrant manner. It will be housekept on task changes and this might well be implemented by mapping it on to part of the stack.

Editorial note. It is normal convention in British Standards to use italic type for algebraic quantities. Since the status of such quantities contained in this standard may or may not directly represent true variable quantities, this convention has not been adopted in this standard.

Summary of pages

This document comprises a front cover, an inside front cover, pages i to iv, pages 1 to 44, an inside back cover and a back cover.

This standard has been updated (see copyright date) and may have had amendments incorporated. This will be indicated in the amendment table on the inside front cover.

1 Scope

This British Standard specifies the semantics and syntax of the computer programming language RTL/2 by specifying requirements for a compiler and for a conforming program.

NOTE The specification requirements have been drafted so that all features of RTL/2 are either explicity defined, or explicity stated to have been left intentionally undefined.

Appendix A contains information regarding input/output. Appendix B contains information on error recovery. Appendix C makes a recommendation on compiler limits.

2 Reference

The title of the publication referred to in this standard is given on the inside back cover.

3 Definitions

For the purposes of this British Standard the definitions given in BS 3527 apply.

4 Syntax

4.1 Syntactic metalanguage[1]. The notation used in this standard to specify the syntax of the language is as follows.

a) The terminal symbols of the language denote themselves. The names of classes are in lower case letters whereas the language alphabet is represented by the upper case letters. Items are separated from each other by spaces where necessary.

b) The metasymbol ::= denotes "is", and the metasymbol | denotes "or".

c) The brackets [and] are used to denote that the items enclosed within are optional.

d) A sequence of three dots ... denotes that the immediately preceding item may be repeated many times.

4.2 Example. The following example denotes that a "name" is a sequence of letters and digits of which the first is a letter. A "digit" is one of 0, 1 ..... 9 and a "letter" is one of A, B ..... Z. The individual digits and letters are terminal symbols and cannot be decomposed further.

name ::= letter [letter | digit] ...

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

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

In the text, the names of classes are enclosed in quotes when formal correctness is emphasized. In less formal passages the quotes are omitted.

4.3 Syntax requirements. The syntax requirements, with references to relevant text in clause **7**, are specified in Table 1.

NOTE The syntax requirements listed in Table 1 should be interpreted in a synthetic rather than analytic way. It is intended primarily for the user of this standard to determine the form in which a program is to be written and reference to this standard by a compiler, to determine whether a form is valid, is a secondary consideration. In some cases involving the class "identifier", the syntax is ambiguous (i.e. it allows two parses to some forms), and semantic information (the mode of the identifier) is required to determine which parse should be used. These cases are as follows.

a) In the production of "arrayelement", both "variable" and "array" can be resolved to "identifier".

b) In the production of "recordcomponent", both "variable" and "record" can be resolved to "identifier".

c) In the production of "constant", both "identifier" and "variable" can be resolved to "identifier".

d) In the production of "primary", both "functioncall" and "variable" can be resolved to "identifier ([expn [, expn] ...])".

The syntax of RTL/2 is presented at two levels. At the higher level the root is "module" and the terminal symbols are the items; at this level the layout characters may be freely inserted to increase legibility. At the lower level the root is "item" and the terminal symbols are characters; at this level the layout characters are generally significant and cannot be arbitrarily inserted. The class "item" is not used other than as a descriptive convenience.

[1] A British Standard syntactic metalanguage is at an early stage of preparation.

Table 1 — Alphabetical lists of syntax requirements

Table 1A — Item syntax

Requirement		Reference
bindigit	::= 0 | 1	**7.1.3**
bindigitlist	::= bindigit ...	**7.1.3**
codeheading	::= CODE digitlist, digitlist;	**7.1.8**
codeitem	::= character-other-than-trip1-or-trip2 | trip1 letitem | trip2 name	**7.1.8**
codeseq	::= codeheading codeitem ...	**7.1.8**
comment	::= % sequence-of-RTL/2-characters-excluding-%-and-newline %	**7.1.5**
compoundseparator	::= := | :/ | // | <= | >= | : = : | : # :	**7.1.9**
digit	::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	**7.1.2**
digitlist	::= digit ...	**7.1.3**
exponent	::= E sign digitlist	**7.1.3**
fraction	::= real B sign digitlist	**7.1.3**
hexdigit	::= digit | A | B | C | D | E | F	**7.1.3**
hexdigitlist	::= hexdigit ...	**7.1.3**
integer	::= digitlist | BIN bindigitlist | OCT octdigitlist | HEX hexdigitlist | "stringchar"	**7.1.3**
item	::= letitem | title | option | codeseq	**7.1.1**
letdefinition	::= LET name = [letitem-other-than-semicolon] ...	**7.1.11**
letitem	::= name | number | string | comment | separator	**7.1.11**
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	**7.1.2**
name	::= letter [letter | digit | $ | _] ...	**7.1.2**
number	::= real | integer | fraction	**7.1.3**
octdigit	::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7	**7.1.3**
octdigitlist	::= octdigit ...	**7.1.3**
opchar	::= letter | digit	**7.1.7**
opitem	::= opchar...	**7.1.7**
real	::= digitlist, digitlist [exponent] | digitlist exponent	**7.1.3**
separator	::= compoundseparator | simpleseparator	**7.1.9**
sign	::= [+ | –]	**7.1.3**
simpleseparator	::= (|) | * | + | – | , | . | / | : | ; | < | = | > | #	**7.1.9**
string	::= " [stringchar ... | stringinsert] ..."	**7.1.4**
stringchar	::= RTL/2-character-other-than-"- #-HT-LF	**7.1.3**
stringinsert	::= # [stringitem [, stringitem] ...] #	**7.1.4**
stringitem	::= integer [(length)]	**7.1.4**
title	::= TITLE sequence-of-RTL/2-characters-excluding-semicolon	**7.1.6**
trip1	::= first-code-trip-character	**7.1.8**
trip2	::= second-code-trip-character	**7.1.8**

Table 1 — Alphabetical lists of syntax requirements

Table 1B — Module syntax

Requirement		Reference
amode	::= simplemode | recmode	**7.2.4.3**
arraydec	::= ARRAY (length [, length] ...) amode initidlist	**7.2.4.3**
array	::= identifier | recordcomponent	**7.3.2.3**
arrayelement	::= variable (subscriptlist) | array (subscriptlist)	**7.3.2.3**
arrayinititem	::= datainitvalue [(length)]	**7.2.6.5**
arrayinitvalue	::= ([arrayinititem [, arrayinititem] ...]) | string	**7.2.6.5**
arraymode	::= ARRAY [([,]...)] amode	**7.2.4.3**
arraysimplespec	::= ARRAY (length [, length] ...) simplemode idlist	**7.2.5.1**
arrayspec	::= ARRAY (length [, length] ...) amode idlist	**7.5.1**
assignmentst	::= destination : = [destination : =] ... expn	**7.4.4**
block	::= BLOCK blockbody ENDBLOCK	**7.4.3**
blockbody	::= [simpledec;] ... sequence	**7.2.7.1**
brick	::= [ENT] datadec | [ENT] procdec | [ENT] stackdec	**7.5.1**
codest	::= codeseq	**7.4.13**
comparator	::= = | # | < | <= | > | >= | :=: | :#:	**7.3.4**
comparison	::= expn comparator expn	**7.3.4**
condexpn	::= IF condition THEN expn [ELSEIF condition THEN expn] ... ELSE expn END	**7.3.2.5**
condition	::= subcondition [OR subcondition] ...	**7.3.4**
constant	::= number | identifier | variable | structure | string	**7.3.2.2**
datadec	::= DATA identifier; declaration [; declaration] ... ENDDATA	**7.2.11**
datadescn	::= DATA identifier; dspec [; dspec] ... ENDDATA	**7.5.1**
datainitvalue	::= priminitvalue | refinitvalue | recinitvalue | arrayinitvalue	**7.2.6.5**
declaration	::= [simpledec | arraydec | recorddec]	**7.2.11**
destination	::= variable | VAL variable	**7.4.4**
dspec	::= [simplespec | arrayspec | recordspec]	**7.5.1**
dyadicop	::= SLL | SRL | SHL | SLA | SRA | SHA | * | / | // | :/ | MOD | LAND | LOR | NEV | + | –	**7.3.3.4**
dummyst	::= [ ]	**7.4.12**
environdescn	::= EXT stackdescn | EXT procdescn | EXT datadescn | SVC procdescn | SVC datadescn	**7.5.1**
expn	::= term [dyadicop term]...	**7.3.3.4**
forst	::= [FOR identifier: = expn [BY expn]] TO expn DO blockbody REP	**7.4.8**
functioncall	::= variable paralist | identifier paralist	**7.3.2.4**
gotost	::= GOTO expn	**7.4.5**
identifier	::= name	**7.1.2**
idlist	::= identifier [,identifier] ...	**7.2.5.1**
ifst	::= IF condition THEN sequence [ELSEIF condition THEN sequence] ... [ELSE sequence] END	**7.4.7**
initidlist	::= inititem [, inititem] ...	**7.2.3**
inititem	::= identifier [: = [identifier: =] ... initvalue]	**7.2.3**
initvalue	::= datainitvalue | localinitvalue	**7.2.6.5**
labellist	::= identifier [, identifier] ...	**7.4.6**
labels	::= [identifier:] ...	**7.4.2**
length	::= staticintexpn	**7.2.4.2**
localinitvalue	::= expn	**7.2.6.5**
module	::= moduleitem [; moduleitem] ...	**7.5.1**
moduleitem	::= [environdescn | recmodedef | letdefinition | title | brick]	**7.5.1**
monadicop	::= + | – | ABS | NOT | REAL | INT | FRAC | BYTE | LENGTH	**7.3.3.3**

Table 1 — Alphabetical lists of syntax requirements

Table 1B — Module syntax

Requirement		Reference
paradescription	::= [pspec [, pspec] ...]	**7.2.7.1**
paralist	::= ([expn [, expn] ...])	**7.3.2.4**
plainmode	::= REAL | INT | FRAC | BYTE	**7.2.3**
primary	::= constant | variable | functioncall | condexpn | (expn)	**7.3.3.1**
priminitvalue	::= sign | number | identifier	**7.2.6.2**
primmode	::= plainmode | progmode	**7.2.3**
procdec	::= PROC identifier (paradescription) resultmode; blockbody ENDPROC	**7.2.7.1**
procdescn	::= PROC procdescriptor idlist	**7.5.1**
procdescriptor	::= ([simplemode [, simplemode] ...]) resultmode	**7.2.7.3**
procst	::= variable paralist | identifier | paralist	**7.4.10**
progmode	::= LABEL | STACK | PROC | procdescriptor	**7.2.3**
pspec	::= simplespec	**7.2.7.1**
recinitvalue	::= (datainitvalue [, datainitvalue] ...)	**7.2.6.5**
recmode	::= recmodeident	**7.2.5.1**
recmodedef	::= MODE recmodeident (rspec [, rspec] ...)	**7.2.5.1**
recmodeident	::= identifier	**7.2.5.1**
record	::= identifier | arrayelement	**7.3.2.3**
recordcomponent	::= variable, selector | record.selector	**7.3.2.3**
recorddec	::= recmode initidlist	**7.2.5.2**
recordspec	::= recmode idlist	**7.5.1**
refinitvalue	::= variable | structure | string	**7.2.6.3**
resultmode	::= [simplemode]	**7.2.7.2**
returnst	::= RETURN | RETURN (expn)	**7.4.11**
rspec	::= simplespec | arraysimplespec	**7.2.5.1**
selector	::= identifier	**7.3.2.3**
sequence	::= statement [; statement] ...	**7.4.1**
simpledec	::= simplemode initidlist	**7.2.3**
simplemode	::= [REF] primmode | REF arraymode | REF recmode	**7.2.3**
simplespec	::= simplemode idlist	**7.2.5.1**
stackdec	::= STACK identifier length	**7.2.8**
stackdescn	::= STACK idlist	**7.5.1**
statement	::= labels unlabelledst	**7.4.1**
staticdyadicop	::= + | − | * | /	**7.3.7**
staticintexpn	::= staticterm [staticdyadicop staticterm]	**7.3.7**
staticmonadicop	::= + | −	**7.3.7**
staticprimary	::= integer | (staticintexpn)	**7.3.7**
staticterm	::= [staticmonadicop]...staticprimary	**7.3.7**
structure	::= array | record	**7.3.2.3**
subcondition	::= comparison [AND comparison] ...	**7.3.4**
subscriptlist	::= expn [, expn] ...	**7.3.2.3**
switchst	::= SWITCH expn OF labellist	**7.4.6**
term	::= [monadicop] ... primary	**7.3.3.3**
unlabelledst	::= block | assignmentst | gotost | switchst | ifst | forst | whilest | procst | returnst | dummyst | codest	**7.4.1**
variable	::= identifier | arrayelement | recordcomponent	**7.3.2.3**
whilest	::= WHILE condition DO sequence REP	**7.4.9**

NOTE The class names "npconstant", "npexpn" and "vectordec" have been omitted from Table 1 because they are merely subclasses of "constant", "expn" and "arraydec" respectively; these subclasses are introduced in this standard only as an explanatory aid.

5 Examples

This standard contains many examples in which, for compactness, identifiers are used without having been formally declared. In all such cases the identifiers should be considered to be declared as follows.

```
MODE COMPLEX (REAL RL, IM);
MODE LIST (INT HD, REF LIST TL);
MODE PERSON (INT AGE, ARRAY (8) BYTE NAME, REF ARRAY BYTE ADDRESS, REF PERSON
MOTHER, FATHER, REF ARRAY PERSON CHILDREN, SIBLINGS, BYTE SEX);
EXT PROC () REAL TIME;
EXT PROC (REAL) REAL LOG, EXP, SIN, COS;
EXT PROC (INT, INT, INT) INT F;
EXT PROC (REF ARRAY BYTE) TWRT;
EXT PROC (INT) IWRT;
DATA EXAMPLES;
   REAL X, Y, Z;
   REF REAL XX, YY, ZZ;
   INT I,J, K, L;
   REF INT JJ, KK, LL;
   BYTE M, N;
   FRAC P, Q, R;
   LABEL RESTART;
   COMPLEX U, V, W;
   LIST CELLA, CELLB;
   REF LIST NEXTCELL;
   PERSON JOHN, JIM, JANE;
   REF PERSON WHO;
   ARRAY (7)INT G;
   ARRAY (10) REAL A, B, C;
   ARRAY (5) LIST CELLS;
   ARRAY (100) PERSON PEOPLE;
   ARRAY (5, 10) REAL A2, B2, C2;
   ARRAY (6) LABEL S;
   REF ARRAY REAL AA, BB;
   REF ARRAY (,) REAL AA2, BB2;
   PROC () ROUTINE;
   PROC (REAL) REAL FN;
ENDDATA
```

6 Compliance

6.1 Implementations

6.1.1 An implementation of RTL/2 that can accept all of the features of the language specified in clause **7** and with the meanings defined therein complies with the requirements of this standard.

6.1.2 An implementation that omits any of the features of the language specified in clause **7**, or that includes such a feature but with a meaning altered in any way from that specified in clause **7**, does not comply with the requirements of this standard.

6.1.3 An implementation that includes features additional to those specified in clause **7** (i.e. extensions) complies with the requirements of this standard provided that:

a) the extensions do not singly or collectively alter in any way the meanings of the features specified in clause **7**; *and*

b) the extensions are clearly detailed in all descriptions of the implementation as being "extensions to RTL/2 as specified by BS 5904"; *and*

c) the compiler is capable of issuing a warning message at each use of such an extension.

6.2 Programs

6.2.1 A program written in RTL/2 complies with the requirements of this standard provided that the program uses only those features of the language specified in clause **7**, together with any desired sequences of machine code included in the manner specified in clause **7**.

6.2.2 Any program that includes any extension to the language (see **6.1.3**), or includes machine code in a manner other than as allowed by the requirements of clause **7** does not comply with the requirements of this standard.

7 Requirements

7.1 Basic elements

7.1.1 *Items*. The text of an RTL/2 program shall be a sequence of characters drawn from the character set specified in Table 2. No other character shall appear in the text of an RTL/2 program, except within a code sequence (see **7.1.8**). The three characters #, £ and $ shall be interchangeable and all shall mean the same thing (except possibly within a code sequence).

NOTE 1 For simplicity, only the character # is shown in the syntax requirements in this standard.

NOTE 2 The characters are grouped together into items of various sorts and the program text is best considered as a sequence of these items. **7.1** describes this grouping which should be considered to occur before any further analysis takes place.

The characters shall be grouped into items. Each item shall be one of the following:

a) name;

b) arithmetic constant;

c) string;

d) comment;

e) title;

f) code sequence;

g) separator.

An item shall be terminated by any character that cannot be interpreted as being part of that item.

NOTE 3 As a consequence of this general requirement, the layout characters space, newline or tab terminate most items. Layout characters are otherwise not significant outside items and may be freely used to improve the legibility of program text.

Items can be divided into two categories according to whether the requirements of **7.1.11** allow them to appear in the replacement sequence of a LET definition.

The syntax shall be:

item ::= letitem | title | option | codeseq

Table 2 — RTL/2 character set

Character	Decimal value	Language use	Reference
HT	9	layout: horizontal tab	**7.1.1**
LF	10	layout: newline	**7.1.1**
SP	32	layout: space	**7.1.1**
!	33	not used	
″	34	string quote	**7.1.4**
#	35	not equals, strings	**7.1.4** and **7.3.4**
$	36	names	**7.1.2**
%	37	comments	**7.1.5**
&	38	not used	
‘	39	byte quote	**7.1.3**
(	40	open bracket	
)	41	close bracket	
*	42	multiply	**7.3.3.4**
+	43	add	**7.3.3.4**
’	44	comma	
–	45	minus	**7.3.3.4**
.	46	constants, records	**7.1.3** and **7.3.2.3**
/	47	divide	**7.3.3.4**
0 to 9	48 to 57	numbers	**7.1.3**
:	58	labels, assignment, etc.	**7.4.2**, **7.4.4** and **7.3.4**
;	59	statement, declaration separator	
<	60	less than	**7.3.4**
=	61	assignment, equals	**7.4.4** and **7.3.4**
>	62	greater than	**7.3.4**
?	63	not used	
@	64	not used	
A to Z	65 to 90	names, numbers	**7.1.2** and **7.1.3**
[	91	not used	
\	92	not used	
]	93	not used	
↑	94	not used	
—	95	names	
‘	96	not used	
a to z	97 to 122	names	
{	123	not used	
|	124	not used	
}	125	not used	
~	126	not used	
	160 to 255	not used	

NOTE The characters designated as not used have no particular purpose in the language but they may occur in strings, comments and titles.

7.1.2 *Names*. A name shall consist of a sequence of letters, digits, underscore and dollar characters of which the first is a letter. There shall be no limit to the number of characters in a name and all the characters shall be significant.

Names of EXT, ENT and SVC bricks (see **7.5**) shall not be specially restricted and shall be fully significant within a module.

NOTE 1 However, the linkage system may impose restrictions on the number of characters that are externally significant and hence on the user's choice of such names.

The syntax shall be:

	name	::= letter [letter | digit | $ | _] ...
	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
	identifier	::= name

Lower-case letters may be used in names. Names that differ only in their use of upper-case and lower-case letters are considered identical.

Names shall be used for two purposes.

a) To denote the language keywords, i.e. names such as IF and REAL, that have a predefined meaning. All the keywords are listed in Table 3.

b) To denote user identifiers, used for a variety of purposes such as naming variables, arrays of variables or pieces of code. A name that denotes a keyword shall not be used as an identifier.

Examples of "identifier" are:

J

GEORGE

V17X

NO2

NOTE 2 As a consequence of the general rules regarding the termination of items, a layout character will terminate a name. Thus PROC FRED consists of two adjacent names whereas PROCFRED is a single name. Similarly, it is essential that the keyword GOTO should not be written as GO TO.

7.1.3 *Arithmetic constants*. The four types of numerical data in RTL/2 shall be "real", "integer", "fraction" and "byte"; these are specified in detail in **7.2.2**.

Arithmetic constants (numbers) shall be used to denote literal and initial values of these types.

NOTE 1 The form "integer" may be used to denote constants of both type integer and byte (see **7.3.2.2**).

The syntax shall be:

	number	::= real | integer | fraction
	sign	::= [+ | –]
	exponent	::= E sign digitlist
	real	::= digitlist. digitlist [exponent] | digitlist exponent
	integer	::= digitlist | BIN bindigitlist | OCT octdigitlist | HEX hexdigitlist | "stringchar"
	fraction	::= real B sign digitlist
	digitlist	::= digit ...
	bindigitlist	::= bindigit ...
	octdigitlist	::= octdigit ...
	hexdigitlist	::= hexdigit ...
	bindigit	::= 0 | 1
	octdigit	::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
	hexdigit	::= digit | A | B | C | D | E | F
	stringchar	::= RTL/2-character-other-than-" -#-HT-LF

Lower-case letters e, b and a – f may be used in the forms for exponent, fraction and hexdigit respectively.

NOTE 2 A stringchar may be an actual space.

The numerical value of a constant denoted by a stringchar in single quotes shall be the decimal value of that character specified in Table 2.

Table 3 — Keywords

Keyword	Reference	Keyword	Reference	Keyword	Reference
ABS	**7.3.3.3**	FRAC	**7.2.3** and **7.3.3.3**	REAL	**7.2.3** and **7.3.3.3**
AND	**7.3.4**	GOTO	**7.4.5**	REF	**7.2.3**
ARRAY	**7.2.4**	HEX	**7.1.3**	REP	**7.4.8** and **7.4.9**
BIN	**7.1.3**	IF	**7.3.2.5** and **7.4.7**	RETURN	**7.4.11**
BLOCK	**7.4.3**	INT	**7.2.3** and **7.3.3.3**	RTL	**7.1.8**
BY	**7.4.8**	LABEL	**7.2.3**	SHA	**7.3.3.4**
BYTE	**7.2.3** and **7.3.3.3**	LAND	**7.3.3.4**	SHL	**7.3.3.4**
CODE	**7.1.8**	LENGTH	**7.3.3.3**	SLA	**7.3.3.4**
DATA	**7.2.11**	LET	**7.1.11**	SLL	**7.3.3.4**
DO	**7.4.8** and **7.4.9**	LOR	**7.3.3.4**	SRA	**7.3.3.4**
ELSE	**7.3.2.5** and **7.4.7**	MOD	**7.3.3.4**	SRL	**7.3.3.4**
ELSEIF	**7.3.2.5** and **7.4.7**	MODE	**7.2.5.1**	STACK	**7.2.3**, **7.2.8** and **7.5.1**
END	**7.3.2.5** and **7.4.7**	NEV	**7.3.3.4**	SVC	**7.5.1**
ENDBLOCK	**7.4.3**	NOT	**7.3.3.3**	SWITCH	**7.4.6**
ENDDATA	**7.2.11**	OCT	**7.1.3**	THEN	**7.3.2.5** and **7.4.7**
ENDPROC	**7.2.7.1**	OF	**7.4.6**	TITLE	**7.1.6**
ENT	**7.5.1**	OPTION	**7.1.7**	TO	**7.4.8**
EXT	**7.5.1**	OR	**7.3.4**	VAL	**7.4.4**
FOR	**7.4.8**	PROC	**7.2.3**, **7.2.7.1** and **7.5.1**	WHILE	**7.4.9**

NOTE The following additional keywords are reserved for use in extensions (see **6.1.3**): CASE ENUM EXITDO MODE_CONVERT PRAGMA WHEN

In the case of the binary, octal and hexadecimal forms for integers, the keyword BIN, OCT or HEX shall be followed by one or more of the layout characters space, newline or tab. All other occurrences of a layout character shall terminate an arithmetic constant.

The fraction constant shall include a binary scale factor following B, e.g. 17.6B – 5 is stored as 17.6×2^{-5}.

NOTE 3 A real constant may include a decimal exponent following E, e.g. 0.1 E7 is stored as 0.1×10^{7}.

The following are examples of "number":

3.47	0.1E7	76E–4	are real;
999	BIN 101	HEX F7	are integer;
"A"	""	""	are also integer;
0.493 B1	17.6B–5	1E10B–36	are fraction.

The following examples of numbers do not comply with the requirements of this standard:

27	E9	.001
"£"	""	3B – 4

7.1.4 *Strings*

NOTE 1 A string provides a convenient way of representing a set of byte constants such as a byte array parameter or the initial value of a byte array (see **7.2.6.3**, **7.2.6.4**, **7.3.5** and **7.4.10**).

A string shall be basically a sequence of stringchars enclosed in double quotes (") and shall denote the set of values formed by taking the decimal values of those characters as specified in Table 2. As a consequence of this requirement, space characters may occur in strings (to stand for themselves) whereas newline and tab shall not occur.

The character #, which is not a stringchar, shall have a special significance; a sequence enclosed within a pair of # characters within a string (a stringinsert) shall be interpreted as part of an array initial value as specified in **7.2.6.4**. Thus it shall consist of a sequence of integer constants in the range 0 to 255 (possibly followed by replication factors) separated by commas.

NOTE 2 Such sequences behave like normal program text and so newline, tab and space characters, comments and LET replacements (see **7.1.11**) may occur in the normal way.

The syntax shall be:

string ::= " [stringchar ... | stringinsert] ..."
stringinsert ::= # [stringitem [, stringitem]...] #
stringitem ::= integer [(length)]

NOTE 3 A string that is too long to fit on one line may be spread over several lines by using stringinserts containing newline characters. There shall be no direct way of representing a newline in a string. It shall be treated like any other character that is not a stringchar and shall be inserted in a # sequence, using the LET facility for clarity if so desired (see **7.1.11**).

If the characters $ or £ are used as alternatives to #, they shall occur as matched pairs.

Assuming that

LET N L = 10;
LET TAB = 9;

have been set, valid examples of "string" are:

"THIS IS A STRING"	is a simple example;
" # NL #"	is a single newline;
"DAY # TAB # MONTH # TAB # YEAR"	inserting tabs;
" # NL(4), TAB # ALARM"	four newlines, tab and "ALARM";
""	a null string.

The following examples do not comply with the requirements of this standard:

"PRICE=£"
""""

7.1.5 *Comments*. All characters starting from and including the character % up to and including the next % character shall be treated as comment and ignored by the compiler. The layout character newline shall not be used in comments; this requirement prevents subsequent lines of program text being treated as comment if a % character is inadvertently omitted.

NOTE The layout characters space and tab may occur in comments.

The syntax shall be:

comment ::= % sequence-of-RTL/2-characters-excluding-%-and-newline %

An example is:

% THIS IS A COMMENT %

7.1.6 *Titles*. A title shall provide a means of labelling the object code or a listing of the source code in whole or in part. The details shall depend upon the implementation. A title shall consist of the keyword TITLE followed by any sequence of characters not including a semicolon (;).

The syntax shall be:

title ::= TITLE sequence-of-RTL/2-characters-excluding-semicolon

An example is:

TITLE DEBUG ROUTINE 13 MAY 1980

7.1.7 *Options*. The provision of an option statement is deprecated.

NOTE For compatibility with option statements included prior to the implementation of amendment No. 2, the keyword "OPTION" remains reserved.

7.1.8 *Code*. A sequence starting with the keyword CODE shall be interpreted as a machine code sequence. The syntax shall be:

codeseq	::=codeheading codeitem ...
codeheading	::=CODE digitlist, digitlist;
codeitem	::=character-other-than-trip1-or-trip2 | trip1 letitem | trip2 name
trip1	::=first-code-trip-character
trip2	::=second-code-trip-character

The two values denoted by "digitlist" in the heading shall be interpreted in a machine-dependent manner and may be used to indicate resources in terms of code size and stack workspace required by the code sequence. Within the codeheading, layout characters are permitted in the normal way.

The sequence of characters following the codeheading shall constitute the actual code, and within this sequence layout characters shall stand for themselves. The characters in the sequence shall not be restricted to being characters listed in Table 2.

Within the sequence, a letitem (see **7.1.11**) may be denoted by immediately prefixing the item by the machine-dependent character "trip 1". Thus a name, number, comment, string or separator may be referred to by this means and in particular a LET name shall be replaced by its corresponding item sequence. Also, a literal label may be declared within a code sequence. The action performed by the compiler and the code inserted for each item shall be implementation-dependent.

Whenever any variable name in a data brick, or selector name of a MODE, is referenced, that name shall be followed by the name of the host brick, or mode, itself preceded by the second (different) character "trip2".

NOTE This construction allows the compiler to check the validity of the reference and will minimize errors resulting from incorrectly accessing names if their definition is changed.

The characters "trip 1" and "trip2" themselves may be denoted within the sequence by prefixing them by "trip 1".

The code sequence shall be terminated by the last "codeitem", which shall be the keyword RTL preceded by the character "trip1".

7.1.9 *Separators*. Separators shall be used as punctuation symbols and operators. A separator either shall be compound and constructed of several characters, or shall be a single character. A compound separator shall not contain a layout character.

The syntax shall be:

separator	::= compoundseparator | simpleseparator
compoundseparator	::= := | :/ | // | <= | >= | :=: | :#:
simpleseparator	::= (|) | * | + | – | , | . | / | : | ; | < | = | > | #

7.1.10 *Item hierarchy*. Except for items in a # sequence in a string, and letitems in a CODE sequence, all items shall be of the same lexicographic priority.

Thus within a string the characters % and ' shall have no particular significance, and similarly the characters "and" shall have no particular significance within comments.

7.1.11 *LET replacement*. The language shall include a simple non-parameterised replacement facility that enables the user to give a name to a sequence of items and to use this name instead of the sequence.

NOTE The use of the facility to name constants within a program is strongly recommended (see the examples given in **7.1.4**).

The syntax shall be:

letdefinition	::= LET name = [letitem-other-than-semicolon] ...
letitem	::= name | number | string | comment | separator

The "name" following LET shall not be a keyword and the letitems shall not be the name following LET, nor one of the following keywords:

LET, TITLE, OPTION, CODE, MODE, RTL.

A "letdefinition" shall occur only where a moduleitem is valid and is terminated by the semicolon that separates it from the succeeding moduleitem (see **7.5**).

The definition shall be valid from its point of occurrence in the program text until the end of the text or redefinition. Occurrences of the name, including occurrences within # sequences within strings, during its validity shall be replaced by the defined sequence. Replacement shall not occur in titles, comments, code sequences (except following a trip 1 character) or when the name and = following LET is expected. Replacement shall occur following the = in a LET definition, but the replaced sequence shall not include the name being defined.

Examples are:

LET NL = 10

LET RAB = REF ARRAY BYTE

LET ATMOS = 14.7

7.2 Declarations

7.2.1 *Preliminary*. Declarations shall define the properties of the various identifiers used within a module of program. Identifiers are used for many purposes and the relevant requirements are given in **7.2.2** to **7.2.10** inclusive.

7.2.2 ***Modes***

7.2.2.1 An RTL/2 program is ultimately concerned with manipulating numerical data of four plain modes. These plain modes and their corresponding requirements shall be as follows.

a) *real* A value of mode REAL shall be represented by a floating point number. The range and accuracy shall be implementation-dependent.

b) *integer*. A value of mode INT shall be a signed integer. The range shall be implementation-dependent but is usually that provided by the natural word of a word machine. In the remainder of this standard the term "word" is used in the sense of the space occupied by an integer. At least 16 bits shall always be used; it is assumed that integer values are stored in twos complement form (this latter restriction is necessary to ensure that logical operations such as LAND are well defined). (See also **7.3.3.2** for double forms of the integer mode.)

c) *fraction*. A value of mode FRAC shall be a value in the range [–1, + 1), i.e. including – 1 but not + 1. The accuracy shall be implementation-dependent but a fraction value shall occupy a word. Fractions shall be provided so that machine-independent fixed point arithmetic operations can be written for computers that do not have floating point hardware. (See also **7.3.3.2** for double forms of the fraction mode.)

d) *byte*. A value of mode BYTE shall be an integer in the range [0, 255].

There are also three other primitive modes for which the requirements shall be as follows.

e) *label A* value of mode LABEL shall be a level-address pair. The level shall indicate a stack pointer position identifying an execution of a procedure and the address shall be that of a literal label in that procedure to which control shall be transferred if the label value is used in a GOTO statement (see **7.4.5**).

f) *procedure*. A value of mode PROC shall be a pointer to a piece of executable code, i.e. a procedure.

g) *stack*. A value of mode STACK shall be a pointer to an area for dynamic workspace, i.e. a stack.

7.2.2.2 In addition to the primitive modes specified in **7.2.2.1**, there shall be reference forms of these modes, values of which shall be references to instances of values of the primitive modes themselves.

NOTE 1 Identifiers may be used to denote locations that contain values of these primitive modes, in the traditional manner.

NOTE 2 Identifiers may be used to denote locations that contain references to instances of values of the primitive modes themselves. Thus there will be normal variables and address variables.

NOTE 3 The programmer may define new composite modes known as "records" and may associate an identifier with such a new mode (this is tantamount to introducing a new keyword such as REAL). Identifiers may be used to denote actual instances of records. References to records may be manipulated and identifiers may be used to denote variables containing such references.

NOTE 4 Arrays of values may also be denoted by identifiers. References to arrays may also be manipulated and identifiers may be used to denote variables containing such references.

NOTE 5 Identifiers may be used to denote literal instances of labels, stacks and procedures.

NOTE 6 It has not been felt necessary to allow identifiers to denote literal instances of plain values because the effect may be achieved by the use of the LET facility.

7.2.2.3 In order to clarify the distinction between literal, normal and reference modes, the following two examples are useful.

7.2.2.3.1 *Example 1*

a) In the literal form, an identifier would be used to denote an actual integer constant; this does not comply with the requirements of this standard, nevertheless

LET NL = 10;

has a very similar effect. The identifier NL denotes the constant 10 and whenever NL is used it is as if the constant 10 had been used.

NOTE There is no actual location called NL. The subsequent statement NL := 20 is not permitted because this is equivalent to the nonsense 10 := 20.

b) In the value form, an identifier is used to denote a location that contains integer values. Thus the declaration INT J defines such a variable and a subsequent statement such as

J := 20

is valid.

c) In the reference form, an identifier is used to denote a location that contains the address of a location such as J above. Thus the declaration

REF INT JJ

defines such a variable, and a subsequent statement such as

JJ := J

d) is valid, and assigns the address of J to JJ.

7.2.2.3.2 *Example 2*

a) In the literal form, an identifier is used to denote an actual piece of coding, thus

```
PROC SIN (REAL X) REAL;
.
.
.
.
ENDPROC
```

declares a literal procedure called SIN.

The association between the coding and the identifier SIN is permanent and there is no variable location called SIN whose value could be changed.

b) In the value form, an identifier is used to denote a location whose value is a pointer to an actual piece of coding.

Thus

PROC (REAL) REAL FN

declares a variable FN which could be used in a sequence such as

```
FN := SIN;
X := FN(Y);
FN := COS;
Z := FN(Y);
```

where the result would be to assign the value of SIN(Y) to X and COS(Y) to Z.

c) In the reference form, an identifier is used to denote a location that contains the address of a location such as FN above. The use of such variables is likely to be rare.

7.2.3 *Simple declarations*. A simple declaration shall serve to declare certain identifiers to represent simple variables of a given mode.

NOTE 1 A simple declaration may also assign initial values to the variables.

A simple declaration shall consist of a simple mode description followed by a list of identifiers (with optional initial values) separated by commas.

The simple mode description shall be one of the following:

REAL	denoting a real variable;
INT	denoting an integer variable;
FRAC	denoting a fraction variable;
BYTE	denoting a byte variable;
LABEL	denoting a label variable;
PROC procdescriptor	denoting a procedure variable;
STACK	denoting a stack variable;

with the keyword REF prefixed if a reference mode is required; the mode description may also be one of the following:

REF recmode	reference to a record;
REF arraymode	reference to an array.

NOTE 2 For "procdescriptor" see **7.2.7.3**, for "recmode" see **7.2.5.1** and for "arraymode" see **7.2.4.3**.

The syntax shall be:

simpledec ::= simplemode initidlist
simplemode ::= [REF] primmode | REF arraymode | REF recmode
primmode ::= plainmode | progmode
plainmode ::= REAL | INT | FRAC | BYTE
progmode ::= LABEL | STACK | PROC procdescriptor
initidlist ::= inititem [, inititem] ...
inititem ::= identifier [:= [identifier :=] ... initvalue]

NOTE 3 For the syntax of initial values and examples see **7.2.6**.

Examples of "simpledec" without initial values are:

REAL X, Y, Z
BYTE M, N
INT I,J, K
FRAC P, Q
LABEL RESTART
PROC () ROUTINE
REF LIST NEXTCELL
REF ARRAY BYTE S, T
REF ARRAY REF LIST D

NOTE 4 Simple variables may be declared in a data brick or as variables local to a procedure.

7.2.4 *Array declarations*

7.2.4.1 *General*. An array declaration shall declare one or more identifiers each representing an array of array elements.

NOTE 1 The array may have one or more dimensions.

The declaration shall give the bounds of the subscripts and the mode of the elements.

NOTE 2 The declaration may also assign initial values.

Arrays shall be declared only in a data brick.

The elements of an array shall all be of the same mode and shall be of any mode except arrays themselves.

NOTE 3 The elements of an array may be references to arrays, records, references to records or primitive values.

Multidimensional arrays shall be implemented as vectors of references to other vectors (see **7.2.4.2**).

7.2.4.2 *Vector declarations*. A vector (one-dimensional array) shall be a data structure consisting of an indexable set of elements of the same mode. The structure shall have a length attribute and the index shall range from 1 to the length.

The syntax of a vector declaration shall be:

vectordec ::= ARRAY (length) amode initidlist
length ::= staticintexpn
amode ::= simplemode | recmode

Examples are;

ARRAY (10) REAL A, B, C

ARRAY (5) LIST CELLS

The first example declares three vectors A, B, C of real variables, each vector being of length 10. The second example declares a vector of five records of mode LIST called CELLS (see **7.2.5**).

The length of an array is given by a static integer expression and is evaluated at compilation time (see **7.3.7**).

NOTE 1 The length of an array may be zero, in which case no elements actually exist, e.g. a null string " ".

Elements of a vector shall be accessed by appending an integer expression in brackets to the identifier of the array.

Examples are:

A(7)

CELLS (I + J)

It shall be possible to declare variables whose values are references to vectors as specified in **7.2.3**, e.g.

REF ARRAY REAL AA, BB

declares two simple variables AA and BB whose values are references to arrays of reals such as A, B and C above.

Supposing that the value of AA is in fact a reference to the array A, then the elements of A can be accessed indirectly by appending an integer expression in subscripts to the variable AA; thus AA (7) would in this instance access A (7) itself.

NOTE 2 This automatic indirect reference is an instance of automatic dereferencing specified in **7.3.1**.

NOTE 3 REF ARRAY REAL is itself a simple mode and so an array of references to other arrays may be declared thus:

ARRAY (10) REF ARRAY REAL AAA

or even a reference to such an array thus:

REF ARRAY REF ARRAY REAL AAAA

NOTE 4 The full syntax of "arraymode" has been deferred to **7.2.4.3** to avoid introducing unnecessary syntactic classes. This full syntax contains a mechanism for simplifying the description of the items AAA and AAAA above.

7.2.4.3 *Multidimensional array declarations*. Multidimensional arrays shall be treated as vectors of references to vectors and although, in general, the mechanism described in **7.2.4.2** could be used for their creation, a more convenient form shall be provided. The full syntax of array declarations shall be:

arraydec ::= ARRAY (length [, length] ...) amode initidlist
arraymode ::= ARRAY [([,] ...)] amode
length ::= staticintexpn
amode ::= simplemode | recmode

In an "arraydec", the number of dimensions of an array shall be equal to the number of lengths. Thus

ARRAY (5, 10) REAL A2

declares a two-dimensional array A2. This array is a vector of length 5, each element of which is a reference to a vector of length 10.

The elements of a multidimensional array shall be accessible by expressions of the form

A2(I, J) *or* A2 (I)(J)

, which are equivalent.

NOTE 1 The form A2 (I) may stand alone to access the reference to the subarray.

In an "arraymode", the dimension of the mode shall be either the number of commas in round brackets plus one or just one if the brackets are omitted. Thus

REF ARRAY (,) REAL AA2

declares a simple variable AA2 whose value is a reference to a two-dimensional array of reals.

NOTE 2 Because of the way in which multidimensional arrays are created from vectors, the modes
REF ARRAY (,) REAL
and
REF ARRAY REF ARRAY REAL
are equivalent.

NOTE 3 Note carefully the difference between "amode", which is the thing of which one can have arrays, and "arraymode", which is the arrays of such things.

Examples of "arraydec" without initial values are:

ARRAY (5, 10) REAL A2, B2, C2

ARRAY (7) INT G

ARRAY (120) BYTE BUFF 1, BUFF2

ARRAY (3, 9, 2) REF LIST TABLE

ARRAY (7) REF ARRAY BYTE DAYS

ARRAY (3) REF ARRAY (,) PROC (INT) H

The following examples do not comply with the requirements of this standard:

ARRAY (2) ARRAY (3) REAL X

ARRAY (I, J) REAL X

7.2.5 *Record declarations*

7.2.5.1 A record shall be a data structure consisting of one or more components. A record shall belong to a record class defined by a MODE definition that indicates the modes of the individual components and the identifiers by which the components may be selected.

Records shall be declared only in a data brick.

The syntax shall be:

recmodedef	::= MODE recmodeident (rspec [, rspec] ...)
recmodeident	::= identifier
rspec	::= simplespec | arraysimplespec
simplespec	::= simplemode idlist
idlist	::= identifier [, identifier] ...
arraysimplespec	::= ARRAY (length [, length] ...) simplemode idlist
recmode	::= recmodeident

A component of a record shall only be a simple item or an array of simple items. It shall not be a record or an array of records.

NOTE 1 A component of a record may be a reference to a record or an array of such references.

Initialization shall not occur in the mode definition.

Examples of "recmodedef" are:

```
MODE COMPLEX (REAL RL IM)
MODE LIST (INT HD, REF LIST TL)
MODE XLINK (INT XP, XQ, REF LIST XT)
MODE PERSON (INT AGE, ARRAY (8) BYTE NAME,
             REF ARRAY BYTE ADDRESS,
             REF PERSON MOTHER, FATHER,
             REF ARRAY PERSON CHILDREN, SIBLINGS,
             BYTE SEX)
```

The first example defines a mode known as COMPLEX that has two components of mode REAL known as RL and IM respectively. The second example defines typical list processing cells where the component HD contains an integer value whilst the component TL points to another such cell.

NOTE 2 In the second example, the mode definition is recursive; mode definitions may also be mutually recursive.

The following examples do not comply with the requirements of this standard:

MODE NUT (LIST KERNEL)

MODE CAT (ARRAY (4) CAT KITTEN)

7.2.5.2 Having defined the mode of a record class, actual records can be declared in a manner analogous to simple declarations.

The syntax shall be:

recorddec ::= recmode initidlist

Examples of "recorddec" are:

COMPLEX U, V, W

LIST CELLA, CELLB,

PERSON JOHN, JIM, JANE

Arrays of records (see **7.2.4**) or references to records (see **7.2.3**) shall be declared thus:

ARRAY (5) LIST CELLS

REF LIST NEXTCELL

ARRAY (100) PERSON PEOPLE

REF PERSON WHO

Individual components of a record shall be accessed by suffixing a . (point) and the component name to the identifier of the record thus:

U.RL W.IM

CELLA.HD CELLB.TL

Automatic dereferencing (see **7.3.1**) shall occur if there is a need to access the components of a record referenced by a REF variable, e.g.

NEXTCELL. HD

7.2.6 *Initialization of data*

7.2.6.1 *General*. If data in a data brick is initialized, the initial value shall be a constant or constant address. If data local to a procedure is initialized, the initial value shall be any suitable expression that could occur in an assignment statement, except as restricted by the requirements of **7.4.3**.

In the case of the application subset of RTL/2, all variables other than those of plain modes and LABEL shall be initialized (see **7.6**).

Initialization shall be indicated by following the identifier by := and the appropriate initial value.

NOTE 1 Multiple initialization may be performed in a manner analogous to an assignment statement (see **7.2.3** and **7.4.4**).

Examples of initializations of local data are:

INT I := J + K

REF INT JJ := G(K)

REALX := Y := 3.9 + Z

LABEL L:= RESTART

If a local variable is not initialized, its initial value shall be undefined.

NOTE 2 **7.2.6.2** to **7.2.6.5** specify requirements for the initialization of data in a data brick.

7.2.6.2 *Primitive modes*. The initial value of a variable of mode REAL, INT or FRAC shall be a signed "real", "integer" or "fraction" respectively, whereas in the case of mode BYTE it shall be an (unsigned) "integer" in the range [0, 255]. If no initial value is given, a default value of zero, of appropriate mode, shall be applied.

The range of values for a signed number shall be as follows.

If the number is "real", "fraction", or an "integer" of the form "digitlist", the normal arithmetic meaning of the signed number shall be taken and shall lie within the range appropriate to the mode as specified in **7.2.2.1** and **7.3.3.2**. If the value expressed cannot be precisely represented in the implementation, in the case of a fraction it shall be taken to be the highest value algebraically less than the expressed value that can be represented, i.e. the expressed value shall be truncated towards – 1.0B0. In the case of a real, an approximate but undefined value shall be taken.

If the number is one of the binary, octal or hexadecimal forms of integers, the pattern shall be such that all bits specified as a 1 shall lie within the word. Leading zeros shall be ignored. If the number is preceded by a minus sign, this shall be applied, at compile time, to the pattern interpreted as a signed integer. Any notional overflow caused by negating the most negative integer shall be ignored; the result being the same most negative integer.

The initial value of a variable of mode procedure or stack shall be (the identifier of) a literal procedure or stack (but not SVC procedure, see **7.5.3**), possibly external to the module. The variables of mode label shall not be initialized. There shall be no default value for variables of mode label, procedure or stack.

The syntax shall be:

priminitvalue ::= sign number | identifier

Examples are:

INT J := 37, L, I := K := –11;

PROC (REAL) REAL FN := SIN

In this example, L is initialized to zero by default.

7.2.6.3 *Reference modes*. The initial value of a reference variable, i.e. of mode commencing REF, shall be denoted by a variable of appropriate primitive mode, or, in the case of a ref array or ref record variable, by a structure denoting an appropriate, actual array or record.

This variable or structure shall not be required to be in the module being compiled (but it shall not be in an SVC data brick). Subscripts occurring in this variable or structure shall be constants of form integer and no automatic dereferencing (see **7.3.1**) shall be involved in determining its value. These requirements ensure that the value can be determined at compile time without appeal to the contents of other initialized variables. A form such as A2(4, 6) shall not be used as an initial value because dereferencing (albeit implicit) is involved.

NOTE 1 This language feature constitutes a requirement for the link-loader as well as for the compiler.

In the case of REF ARRAY BYTE variables, a special form of initial value shall be allowed in addition to the normal structure denoting a specific array of bytes. This special form shall have the form of a "string"; the array of bytes denoted by this string shall be located in a pool of strings and the REF ARRAY BYTE variable shall be initialized to the address of this string. (See **7.3.5** for other uses of the string pool.)

NOTE 2 Default values for uninitialized reference variables are not specified in this standard.

The syntax shall be:

refinitvalue ::= variable | structure | string

NOTE 3 For "variable" and "structure" see **7.3.2.3**.

Examples are:

REF INT JJ := K, KK := G(3), LL := CELLA.HD

REF ARRAY REAL AA := A

REF LIST NEXTCELL := CELLS(4)

REF PERSON WHO := JIM

REF ARRAY BYTE P := "PIG"

The following examples do not comply with the requirements of this standard:

REF LIST NEXTCELL := CELLA.TL

REF INT JJ := G(J)

REF ARRAY REAL AA := A2(2)

7.2.6.4 *Arrays and records.* In the case of an array or record, the initial value shall be denoted by a list of the initial values of the elements or components, separated by commas and enclosed in brackets.

NOTE A repetition factor in brackets may follow an initial value to denote repetition of that value over successive elements of an array. The repetition factor takes the same form as the length of an array; it is a static integer expression and is evaluated at compilation time; it may be zero. The initial value of a byte array may also be denoted by a string.

If an initial value is provided, an appropriate value shall be given for every element or component. If no initial value is provided, default values shall be applied, appropriate to the mode, as for scalars.

Examples are:

COMPLEX U := (2.0, 4.0)

ARRAY (10) INT T := (9, 6, 4, 1, 2, 3, 0, 0, 0, –1)

or

ARRAY (10) INT T := (9, 6, 4, 1, 2, 3, 0(3), –1)

ARRAY (3) BYTE PP := (“P”, “I”, “G”)

or

ARRAY (3) BYTE PP := “PIG”

ARRAY (9) REF ARRAY BYTE PLANETS :=
(“MERCURY”, “VENUS”, “EARTH”, “MARS”,
“JUPITER”, “SATURN”, “URANUS”,
“NEPTUNE”, “PLUTO”)

The last example declares an array of length 9 in a data brick, and initializes each element to one of nine (in this case, distinct), entries in the string pool.

If multidimensional arrays and arrays of records are initialized the initial value shall be expressed as a list of lists.

Examples are:

ARRAY (2, 2) REAL UNIT := ((1.0, 0.0), (0.0, 1.0))

ARRAY (5) LIST CELLS := ((0, CELLA) (5))

7.2.6.5 *Syntax.* The syntax of initial values shall be:

initvalue ::= datainitvalue | localinitvalue
datainitvalue ::= priminitvalue | refinitvalue | recinitvalue | arrayinitvalue
localinitvalue ::= expn
priminitvalue ::= sign number | identifier
refinitvalue ::= variable | structure | string
recinitvalue ::= (datainitvalue [, datainitvalue] ...)
arrayinitvalue ::= ([arrayinititem [, arrayinititem] ...]) string
arrayinititem ::= datainitvalue [(length)]

7.2.7 *Procedure declarations*

7.2.7.1 Normal literal procedure declarations in which an identifier is associated with a piece of code shall be declared as follows. The word PROC shall be followed by the identifier of the procedure, a description of its parameters and result, a semicolon, a body describing the action of the procedure, and the word ENDPROC. If the declaration is preceded by ENT, it shall be externally accessible (see **7.5**).

The syntax shall be:

procdec	::= PROC identifier (paradescription) resultmode; blockbody ENDPROC
resultmode	::= [simplemode]
paradescription	::= [pspec [, pspec] ...]
pspec	::= simplespec
blockbody	::= [simpledec;] ... sequence

NOTE For “simplespec”, see **7.2.5.1**. For “sequence”, see **7.4.1**.

The parameters shall be of any mode except actual records and arrays. The formal parameters shall behave exactly like normal declared variables in the body of the procedure.

If the procedure declaration defines a function call, the mode of the result shall be indicated.

Execution of the statements that form the body of the procedure shall be normally initiated as a consequence of calling the procedure either from a procedure statement (see **7.4.10**) or as a function call (see **7.3.2.4**).

Control shall be transferred out of a procedure by *either*:

a) obeying a generalized GOTO statement; *or*

b) calling a procedure, which itself directly transfers control; *or*

c) obeying a RETURN statement; *or*

d) encountering the final ENDPROC, which implies RETURN.

7.2.7.2 Examples of “procdec” are:

a)
```
PROC TRACE (REF ARRAY (,) REAL A) REAL;
        REAL T := 0.0;
        FOR I := 1 TO LENGTH A DO T := T + A ( I, I ) REP;
        RETURN (T);
    ENDPROC
```

b)
```
PROC F (REAL X, LABEL L);
        IF X < = 0.0 THEN GOTO L END;
        P := LOG(X); Q := SQRT(X);
    ENDPROC
```

c)
```
PROC FAIL (INT N, REF ARRAY BYTE S);
                    TWRT ( “FAILURE” ); IWRT (N);
                    TWRT (“BECAUSE OF”); TWRT (S);
                    GOTO RESTART;
    ENDPROC
```

d)
```
PROC CALL (INT N, M, PROC (INT)INT P) INT;
                    TO N DO M := P(M) REP;
                    RETURN (M);
    ENDPROC
```

In example d), the third parameter P is a procedure and is described in the way in which a variable of mode procedure is declared (see **7.2.7.3**) just as all parameters are described in the same way as corresponding declarations.

7.2.7.3 A procedure variable is a variable that can take as value (a pointer to) a particular “literal procedure”. Thus, in example d) of **7.2.7.2**, the variable P points to the particular piece of code that is handed over as parameter.

NOTE 1 In RTL/2 this sort of variable may be declared in a manner analogous to a variable of any other mode (see **7.2.3**).

However, the type of parameters and result, if any, shall be specified by a “procdescriptor”.

The syntax shall be:

procdescriptor ::= ([simplemode [, simplemode] ...]) resultmode

Thus the types of parameters shall be specified by a list of modes in brackets and the result type shall be appended.

NOTE 2 This is very like the description of the parameters of a literal procedure, except that no identifiers are associated with the parameters since there is no need to cross-refer to them.

Examples of "procdescriptor" are:

PROC(INT, INT)INT	describes a procedure with two integer parameters and integer result;
PROC(INT, INT, PROC(INT)INT)INT	describes a procedure like the literal procedure CALL in example d) of **7.2.7.2**.

7.2.8 *Stack declarations*. Normal literal stack declarations in which an identifier is associated with an area of store to be used as a stack shall be as follows. The word STACK shall be followed by the identifier of the stack and its length in machine-dependent units. If the declaration is preceded by ENT, it shall be externally accessible (see **7.5**).

The syntax shall be:

stackdec ::= STACK identifier length

An example is:

STACK JOB 150

A stack variable shall be a variable that can take as value (a pointer to) a particular "literal stack" and can be declared in a similar manner to procedure variables (see **7.2.7.3**).

NOTE Stack variables are likely to be of most use as formal parameters of supervisor calls controlling multitasking. Corresponding actual parameters are usually the identifiers of literal stacks that are in use by various tasks. For example, there might exist procedures described thus:

EXT PROC (STACK) STOP, START;

which are used in statements such as

STOP (JOB); START (OTHERJOB);

where JOB and OTHERJOB are the identifiers of literal stacks. In practice, STOP and START might be supervisor calls. (See **7.5**).

7.2.9 *Label declarations*

NOTE 1 Normal literal label declarations in which an identifier is associated with a particular statement in a procedure are specified in **7.4.2**.

A label variable shall be a variable that can take as value a level-address pair as described in **7.2.2.1**.

NOTE 2 A label variable may be declared in the usual way (see **7.2.3**). The use of label variables is specified in **7.4.5**.

7.2.10 *Scopes*. All identifiers shall be in scope throughout a complete module except for the following:

a) parameters and other variables local to a procedure;
b) literal labels in a procedure;
c) identifiers defining LET sequences;
d) record component names.

NOTE 1 The scope of LET identifiers is specified in **7.2.11**. A name used as a LET identifier may also be used for other purposes because LET replacement is a preprocessing activity and uses a separate name space.

The scope of record component names shall be the sequence in brackets in the mode definition plus selectors following variables or records referring to that mode.

NOTE 2 In this way, the same name may be used in several different mode definitions and elsewhere without confusion.

In the cases a) and b), the scope shall be the block in which the name is declared plus inner blocks, unless the name is redeclared therein (see **7.4.3**).

NOTE 3 It is a consequence of the scope rules that

REF M M;

does not comply with the requirements of this standard because the redeclaration of M hides the M used in its declaration. Similarly,

REF M X;

followed by a later declaration of M in the same block (e.g. as a literal label) also does not comply with the requirements of this standard.

7.2.11 *Data bricks*. A data brick shall be a named collection of scalars, arrays and records. The word DATA shall be followed by the identifier of the brick, a semicolon, a series of declarations separated by semicolons and the word ENDDATA. If the brick is preceded by ENT, it shall be externally accessible (see **7.5**).

The syntax shall be:

datadec ::= DATA identifier; declaration [; declaration] ... ENDDATA

declaration ::= (simpledec | arraydec | recorddec)

All declarations of variables other than simple variables local to a procedure shall be in a data brick.

An example of "datadec" is:

```
DATA LOCAL;
        INT I, J, K;
        ARRAY (3) REAL A, B := (1.0, 2.0, 3.0);
        PERSON JOHN;
ENDDATA
```

7.3 Expressions

7.3.1 *General* Expressions are rules for computing values and they shall define the operations to be performed on the components of the expression.

NOTE 1 The meaning and value of an expression may depend upon the environment in which it is to be evaluated.

The environment of an expression shall be determined by:

a) the mode of the destinations of an assignment statement; *or*

b) the mode required in some statement parameter (e.g. after TO); *or*

c) the mode of operand required for a monadic or dyadic operator.

NOTE 2 In deciding upon the meaning and validity of an expression in a given environment, certain automatic changes of mode may occur. These automatic changes are of two kinds; first, there are the familiar transfers between the various arithmetic modes; secondly, the mechanism known as dereferencing. The latter is familiar in practice but is not normally treated as a formal mechanism and so needs describing in some detail.

Broadly speaking, dereferencing is "taking the contents of". It has been observed above that identifiers can be used for various levels of reference, e.g. as normal variables as in REAL X, Y, and as reference variables as in R EF REAL XX, YY. When an identifier occurs in an expression it is first considered at its face value, either as the name of a place such as X, or the name of a place such as XX. If this does not suit the environment, the identifier is reconsidered as standing for its contents; this process can be repeated in the case of reference variables.

As an illustration, it is useful to consider the assignment statement in which the destinations dictate the mode required.(see **7.4.4**).

X := 3.14	X is the name of a place that holds real values and so the right hand side has to generate such a real value. However, the right hand side is a value (3.14) already and no dereferencing is required.
X := Y	Here Y at face value is the name of a location; however, X needs a value and so the contents of Y are taken. One level of dereferencing is required.
XX := Y	In this case XX requires an address and Y does stand for such an address. The address of Y is assigned to XX and no dereferencing is required.
X := YY	In this case two levels of dereferencing are required to access the value in the location pointed to by the value in the location YY.
XX := 3.1 4	This is not a statement complying with the requirements of this standard.

Note that dereferencing is also applied in array and record access in the case of a reference variable (see **7.2.4.2** and **7.2.5**).

Occasionally it will be found that dereferencing needs to be forced on the left hand side. In this case the operator VAL can be used (see **7.4.4**).

Automatic transfers between arithmetic modes shall be restricted to those in which no information is lost, e.g.:

BYTE → INT → REAL

FRAC → REAL.

NOTE 3 These types of transfer are known as "widening".

To obtain transfers in which information is lost, the appropriate monadic operator (BYTE, INT or FRAC) shall be applied explicitly (see **7.3.3.3**).

NOTE 4 These types of transfer are known as "narrowing". See **7.3.3.2** for the details of transfers between double and single forms of integer and fraction modes.

7.3.2 ***Expression components***

7.3.2.1 *General*. Expressions shall be built up from four kinds of component:

a) constants (see **7.3.2.2**);

b) variables (see **7.3.2.3**);

c) function calls (see **7.3.2.4**);

d) conditional expressions (see **7.3.2.5**).

7.3.2.2 *Constants*. Constants shall denote literal values that remain unchanged throughout the execution of a program complex, Constants shall be *either:*

a) actual expressed numbers denoting fixed plain values; *or*

b) identifiers denoting literal stacks, labels or procedures; *or*

c) fixed reference values represented by appropriate variables, structures or strings.

The syntax shall be:

Constant ::= number | identifier | variable | structure | string
number ::= real | fraction | integer

The mode of a constant of form "real" shall be REAL and the mode of a constant of form "fraction" shall be FRAC. The mode of a constant of form "integer" shall be BYTE if its value lies in the range [0,255], and otherwise shall be INT.

NOTE See **7.1.3** for these forms of "number". Non-plain constants are fully specified in **7.3.5**.

Examples of "constant" are:

3.75

"p"

"PIGLETS"

SIN

7.3.2.3 *Variables and structures*. Variables shall be single places whereas structures shall be complete arrays or records. Variables and structures are denoted by similar syntactic forms. A structure shall not occur as the destination of an assignment statement.

NOTE 1 A variable may occur as the destination of an assignment statement.

Variables and structures either shall be simple, or shall be array elements or record components (or both), as follows.

a) A simple variable, or simple structure shall be represented by an identifier. The modes of the value of a simple variable or the elements or components of a simple structure shall be defined by the declaration of the identifier.

b) Array elements shall be denoted by the use of a subscript list. The array shall be indicated either directly by a structure denoting the array or indirectly by a ref array variable, in which case dereferencing shall be applied. The subscript list shall follow and shall consist of one or more arithmetic expressions separated by commas and enclosed in brackets. The particular element referred to shall be specified by the actual numerical values of the subscript expressions that are evaluated as mode integer. The array and its subscripts shall be evaluated from left to right.

c) Record components shall be denoted by the use of a selector. The record shall be indicated either directly by a structure denoting the record or indirectly by a ref record variable, in which case dereferencing shall be applied. This shall be followed by a . (point) and by a selector denoting the component concerned. The selector shall be an identifier.

The syntax shall be:

variable ::= identifier | arrayelement | recordcomponent
structure ::= array | record
array ::= identifier | recordcomponent
record ::= identifier | arrayelement
arrayelement ::= variable (subscriptlist) | array (subscriptlist)
recordcomponent ::= variable. selector | record. selector
subscriptlist ::= expn [, expn] ...

selector ::= identifier

NOTE 2 Since elements of arrays and records may be reference variables to other arrays and records, the consequent dereferencing means that the syntactic form of variables and structures may include many instances of component selection and element indexing. NOTE that the syntax allows A2(I)(J) as an alternative form for A2(I, J).

Examples of "variable" without dereferencing are:

XX

A(1)

RESTART

CELLA.HD

PEOPLE(27).NAME(3)

A(G(J))

Examples of "variable" with dereferencing are:

BB(J)

NEXTCELL.HD

WHO.AGE

CELLA.TL.TL.TL.HD

JIM.MOTHER.NAME(8)

PEOPLE(99).FATHER.SIBLINGS(1).SEX

Examples of "structure" are:

G

CELLA

JIM.NAME

WHO.NAME

PEOPLE(K)

7.3.2.4 *Function calls*. A function call shall define a single value that results through the application of statements defined by a procedure declaration to a set of parameters. A function call shall be of the same form as a procedure statement (see **7.4.10**) and shall consist of the identifier of a literal procedure (or SVC procedure, see **7.5**) or the identifier of a procedure variable, followed by a list of parameters separated by commas and enclosed in round brackets. The brackets shall not be omitted even if there are no parameters. The form of parameter in each position in the parameter list shall be an expression of mode determined by the corresponding procedure declaration (see also **7.4.10**).

The syntax shall be:

functioncall ::= variable paralist | identifier paralist

paralist ::= ([expn [, expn] ...])

Examples are:

LOG(X)

TIME()

F(J, K, L – 3)

EXP (EXP(EXP(X)))

NOTE It is possible to have an array of procedure variables, e.g.:

ARRAY(10)PROC(REAL)INT QQ

and to call a selected one of these by

QQ(9)(4.7)

which means call the procedure pointed to by element 9 of array QQ with parameter value 4.7.

7.3.2.5 *Conditional expressions*. A conditional expression shall comprise a rule for choosing one of two or more expressions according to the values of specified conditions.

The syntax shall be:

condexpn ::= IF condition THEN expn [ELSEIF condition THEN expn] ... ELSE expn END

The value shall be determined as follows. The condition following IF shall be evaluated; if this is true, the expression following THEN shall be evaluated and this shall be the result. If the condition is not true, the conditions following the optional keywords ELSEIF shall be evaluated in turn until one of value true is encountered; the expression following the corresponding THEN shall be evaluated and this shall be the result. If there are no optional ELSEIF conditions or they are all false, the expression following ELSE shall be evaluated and this shall be the result.

NOTE For "condition" see **7.3.4**.

All the alternative expressions in a conditional expression shall be of the same mode, or of modes that can be dereferenced and/or widened to a common mode. In the case of non-plain modes, only dereferencing is involved and the mode of the expression shall be that common mode entailing the least dereferencing. In the case of procedure, ref array and ref record expressions, all alternatives shall have the same specifications, array modes or record modes respectively.

In the case of plain modes the situation is more complex, and regard shall be paid to the double forms of integer and fraction modes (see **7.3.3.2**). If all the alternative modes can be reduced to a common mode by dereferencing alone, the resultant mode shall be that common mode entailing the least dereferencing. Otherwise, all the alternatives shall be dereferenced to plain modes and the resultant mode shall be the lowest point on the tree shown in Figure 1 that can be reached by all these plain modes. However, the resultant mode shall not be of double form, but shall always be automatically converted to the corresponding normal form. Nevertheless, if the final mode is real and one or more alternatives give rise to a double form, the conversions shall take place directly and not through the normal form.

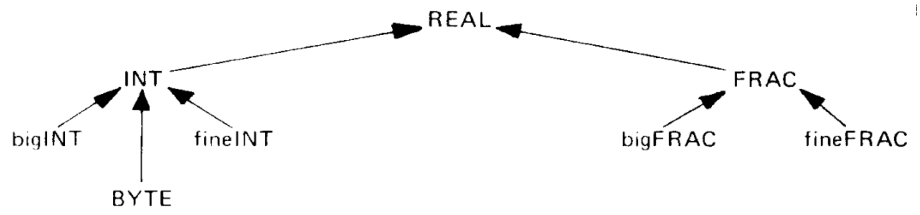


Figure 1 — Numerical mode conversion in conditional expressions

Examples are:

IF I = 3 THEN 7 ELSE 10 END

IF I = 0 THEN IF J = 0 THEN 0 ELSE K END ELSE L + I END

IF I = 0 THEN I ELSEIF I = J THEN K ELSE L END

IF I = J THEN XX ELSE K END

In the last example, XX is dereferenced twice to give a real value, whereas K is dereferenced once to give an integer value and then widened to real.

As a further example, note the difference between

IF I = 0 THEN P ELSEIF J = 0 THEN I ELSE M END

and

IF I = 0 THEN P ELSE IF J = 0 THEN I ELSE M END END

In both these cases the final mode is real. In the first case the three alternatives are each directly converted to mode real. In the second case the inner expression has mode integer and so the conversion of M is performed in two stages.

As a final example, note the difference between

R := IF I = 0 THEN I * J ELSE J END

and

R := IF I = 0 THEN REAL (I * J) ELSE J END

In the first case, the mode of the conditional expression is integer and widening to real occurs immediately before the assignment. In the second case, the intermediate conversion of the big integer to integer is avoided by the use of the operator REAL, which ensures that the mode of the conditional expression is real (see **7.3.3.3**).

7.3.3 *Arithmetic expressions*

NOTE There is no particular distinction between expressions that deliver a plain mode and expressions that deliver some other mode. It is a fact that most operators apply to plain modes and for this reason expressions of plain mode are dealt with first in this standard.

During the evaluation of arithmetic expressions an overflow condition arises if the operands are such that the potential result lies outside the range of values that can be represented by the mode concerned. The behaviour of the program under these conditions is not specified in this standard, but each implementation should provide a means of detecting an overflow condition.

7.3.3.1 *Primaries*. Expressions shall be built up from the primary components specified in **7.3.2** and expressions in round brackets.

The syntax shall be:

primary ::= constant | variable | functioncall | condexpn | (expn)

Examples of "primary" are:

X

28.3

LOG (P + Q)

"0"

IF K = 0 THEN M ELSE N END

(I : / J + 3)

7.3.3.2 *Integers and fractions*

NOTE Most computers can perform a limited range of double length operations. For most calculations involving only integers, these operations are not explicity needed, but for calculations with fractions it is often necessary to use these operations and to be aware of their explicit behaviour. **7.3.3.2** specifies the detailed behaviour of the operations in terms of two double length forms of both integer and fraction values in addition to the usual single length or normal form of each.

Normal values shall occupy a single word (see **7.2.2**) and the values contained in all fraction and integer variables shall thus be in normal form. The other two forms (double modes) shall occupy two words and so shall only be taken by values arising as intermediate results in expressions and shall not be stored in variables (these double forms are the big form and the fine form which are defined by the following text).

Suppose that a word has length w + 1 (where $w \geqslant 15$) and let $M = 2^w$; the forms shall then take values as follows.

a) *normal*. A normal integer shall take any integral value in the range

— $M \leqslant$ integer $< M$

and a normal fraction shall take any value that is a multiple of 1/M in the range

— $1 \leqslant$ fraction < 1

b) *big*. In the big form, the normal part of the value shall occupy the less significant word and so the range of big values shall be M times that of normal values, but with the same precision. A big integer shall take any integral value in the range

— $M^2 \leqslant$ big integer $< M^2$

and a big fraction shall take any value that is a multiple of 1/M in the range

— $M \leqslant$ big fraction $< M$

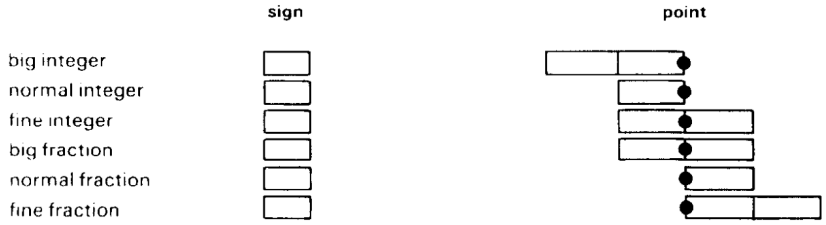
c) *fine*. In the fine form, the normal part of the value shall occupy the more significant word and so the range of fine values shall be the same as that of normal values, but their precision shall be M times greater. A fine integer shall take any value that is a multiple of 1/M in the range

— $M \leqslant$ fine integer $< M$

and a fine fraction shall take any value that is a multiple of $1/M^2$ in the range

— $1 \leqslant$ fine fraction < 1

NOTE 1 The six possible modes arising from the combinations of form and type are shown as follows with their points aligned:



In some implementations, the double length forms may have an extra bit whose use will depend on the implementation; this is invisible to the user.

It should be noted that a value is represented by a fine integer in the same way as by a big fraction. The reason for introducing the distinction between these two modes lies in their behaviour under truncation as illustrated in the example of the use of the multiplication operator in **7.3.3.4**.

For all operators except arithmetic shifts, the operands shall be in specific forms. Where these specific forms differ from normal form they are listed in Table 4 and Table 5. Any arguments not in the specific form shall be converted, according to the scheme shown in Figure 2, by the most direct route.

NOTE 2 If a big value is converted to normal form, an overflow condition may arise.

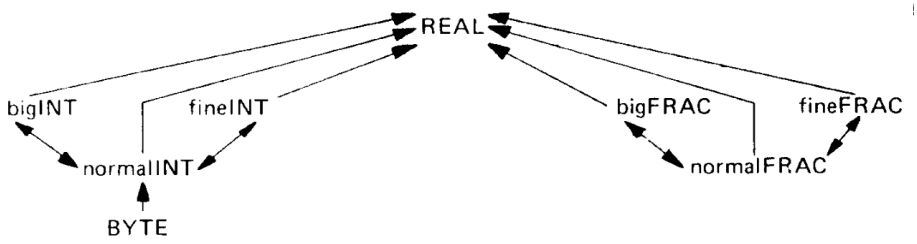


Figure 2 — Numerical mode conversion for operands

If a fine value is converted to normal form, it shall be rounded to the normal value nearest to the original fine value; if the original value lies midway between two normal values, the algebraically greater normal value shall be taken.

NOTE 3 An overflow condition may arise.

When an integer or fraction value in a double form is converted to a real value, it shall be converted directly without first being converted to the single form.

7.3.3.3 *Monadic operators.* A term shall be a primary component optionally preceded by one or more monadic operators. These operators shall be applied from right to left.

The syntax shall be:

term ::= [monadicop] ... primary

monadicop ::= + | − | ABS | NOT | REAL | INT | FRAC | BYTE | LENGTH

Table 4 — Monadic operators, operands and results

Operator	Operand	Result
+	BYTE, INT, FRAC, REAL	same as operand
–	INT, FRAC, REAL	same as operand
ABS	BYTE, INT, FRAC, REAL	same as operand
NOT	INT	INT
REAL	REAL	REAL
INT	big FRAC	fine INT
	INT, REAL	INT
FRAC	fine INT	big FRAC
	FRAC, REAL	FRAC
BYTE	BYTE, INT, REAL	BYTE
LENGTH	array	INT

In the case of operators that have more than one entry in Table 4, each successive entry shall be considered in turn. If the operand in a given instance is of the appropriate mode, or can be converted to that mode, that entry shall be taken, otherwise the next entry shall be considered.

The effect of the operators listed in Table 4 shall be as follows.

Operator **Effect**

+ Shall have no effect.

– Negation; shall change the sign of the operand. If the operand is a byte value, it shall first be converted to mode integer.

If the operand has the form of a number, the operation shall be performed at compile time and the range of values of the consequent signed number shall be as specified in **7.2.6.2**.

NOTE 1 An overflow condition arises if the most negative value of an integer or fraction form is negated during program execution. Observe also that

F := – 1.0B0;

complies with the requirements of this standard, whereas

F := F – 1.0B0;

does not.

ABS Absolute value; shall leave unchanged a positive operand but shall negate a negative operand. See foregoing note on overflow.

NOT Logical not; shall treat an integer value as a bit pattern and shall change the value of each bit.

REAL Shall force widening to a real value.

NOTE 2 This operator may be useful in conditional expressions (see **7.3.2.5**).

INT Shall convert a big fraction to a fine integer without changing its value. The operator shall convert a real value to an integer value by rounding to the integer value nearest to the original real value; if the original value is midway between two integer values, the algebraically greater integer value shall be taken. Thus – 3.5 converts to – 3 and 3.5 converts to 4.

NOTE 3 An overflow condition may arise.

FRAC Shall convert a fine integer to a big fraction without changing its value. The operator shall convert a real value to a fraction value by rounding to the fraction value nearest to the original real value; if the original value lies midway between two fraction values, the algebraically greater fraction value shall be taken.

NOTE 4 An overflow condition may arise.

BYTE Shall convert a real or integer value to a byte. If the operand is real, it shall be first converted to an integer value as for INT. The integer value shall then be converted to a byte by masking, hence producing a value differing from the integer value by a multiple of 256. Thus BYTE 259 has value 3, and BYTE – 0.7 has value 255.

LENGTH Shall apply to an array expression and shall return the length of the array. If the array is multidimensional, the length shall be the range of the first subscript.

Examples of "term" are:

–4

NOT (I NEV J)

–ABS X

LENGTH A

LENGTH A2(I)

NOTE 5 The operands of the operators REAL, INT, FRAC and BYTE may be of the result mode itself. This ensures that any redundant use of the operators for widening does not generate unnecessary coding.

7.3.3.4 *Dyadic operators*. An expression shall consist of one or more terms separated by dyadic operators which shall be applied according to the precedence indicated in Table 5. Operations of the same precedence shall be applied from left to right.

The left operand of a dyadic operator shall be evaluated before the right operand.

The syntax shall be:

expn ::= term [dyadicop term] ...

dyadicop ::= SLL | SRL | SHL | SLA | SRA | SHA | * | :/ | // | / | MOD | LAND | LOR | NEV | + | –

Table 5 — Dyadic operators, operands, results and precedence

Operator	Precedence	Operand types		Result
SLL, SRL, SHL	6	INT	INT	INT
SLA, SRA, SHA	6	see below	INT	see below
*	5	INT	INT	big INT
		INT	FRAC	big FRAC
		FRAC	INT	big FRAC
		FRAC	FRAC	fine FRAC
		REAL	REAL	REAL
:/	5	big INT	INT	INT
		fine INT	FRAC	INT
		big FRAC	FRAC	INT
//	5	fine INT	INT	FRAC
		big FRAC	INT	FRAC
		fine FRAC	FRAC	FRAC
/	5	REAL	REAL	REAL
MOD	5	big INT	INT	INT
		fine INT	FRAC	FRAC
		big FRAC	FRAC	FRAC
LAND	4	BYTE	BYTE	BYTE
		INT	INT	INT
LOR	3	BYTE	BYTE	BYTE
		INT	INT	INT
NEV	2	BYTE	BYTE	BYTE
		INT	INT	INT
+, –	1	INT	INT	INT
		FRAC	FRAC	FRAC
		REAL	REAL	REAL

In the case of operators that have more than one entry in Table 5 each successive entry shall be considered in turn. If the operands in a given instance are of the appropriate mode or can be converted to that mode, that entry shall be taken, Otherwise the next entry shall be considered.

The first operand in an arithmetic shift shall be either an integer or a fraction. The second operand shall be a normal integer. The result shall be the same type as the first operand and its form shall be determined by the form of the first operand and the operator as shown in Table 6.

Table 6 — Form of result of use of shift operators

Form of first operand	Form of result according to operator		
	SLA	SHA	SRA
big	big	big	big
normal	big	normal	fine
fine	fine	fine	fine

The effect of the operators listed in Table 5 shall be as follows.

SLL Shift left logical; shall shift the first operand left the number of places specified by the second operand. If the second operand is negative or greater than the number of bits in the integer representation, the action is not specified.

SRL Shift right logical; shall shift the first operand right the number of places specified by the second operand. If the second operand is negative or greater than the number of bits in the integer representation, the action is not specified.

SHL Shift logical; shall shift the first operand by the number of places specified by the second operand. If the second operand is positive, the behaviour shall be as SLL and otherwise it shall be as SRL.

SLA Shift left arithmetic. If the first operand is single length, it shall be converted first to the big form. The double length operand shall then be shifted left arithmetically by the number of places specified by the second operand. If the second operand is negative or greater than the number of bits in the integer representation, the action is not specified.

NOTE 1 An overflow condition may arise.

SRA Shift right arithmetic. If the first operand is single length, it shall be converted first to the fine form. The double length operand shall then be shifted right arithmetically by the number of places specified by the second operand. If the second operand is negative or greater than the number of bits in the integer representation, the action is not specified. The sign bit shall be propagated.

SHA Shift arithmetic. The first operand shall be shifted without altering its type or form. If the second operand is positive, a left shift shall be performed; if negative, a right shift shall be performed. The number of places shifted shall be the absolute value of the second operand and the action is not specified if this exceeds the number of bits in the integer representation. A right shift shall propagate the sign bit.

NOTE 2 A left shift may give rise to an overflow condition.

* Multiplication. This shall be either a real operation on real operands or a fixed point operation on integer and fraction operands.

NOTE 3 In the former case, an overflow condition may arise in the usual way whereas in the latter case it only arises if both operands have their most negative values. Any subsequent conversion to normal form may itself give rise to an overflow condition.

NOTE 4 There is a difference between

J * INT (K * P)

which produces an intermediate fine integer and then a big integer result, and

J * (K * P)

which produces an intermediate big fraction and then a big fraction result.

:/ Integer division. This shall always produce an integer result with truncation towards zero.

NOTE 5 An overflow condition may arise.

// Fraction division. This shall always produce a fraction result with truncation towards zero.

NOTE 6 An overflow condition may arise.

/ Real division.

NOTE 7 An overflow condition may arise.

MOD Modulo. The result shall be the remainder on dividing the first operand by the second. The sign of the result shall be the same as that of the first operand.

NOTE 8 An overflow condition may arise.

NOTE 9 J MOD K has the same value as J – J :/ K * K.

LAND Logical and. This shall treat both operands as bit patterns. A bit in the result shall be 1 only if the corresponding bits of both operands are 1.

LOR Logical inclusive or. This shall treat both operands as bit patterns. A bit in the result shall be 0 only if corresponding bits in both operands are 0.

NEV Not equivalent. This shall treat both operands as bit patterns. A bit in the result shall be 1 only it the corresponding bits in the operands differ from each other.

+, – Addition, subtraction. This shall be either a real operation on real operands or a fixed point operation on single length integer or fraction operands. Integer ± fraction shall convert to real.

NOTE 10 These operations may give rise to an overflow condition.

Examples of “expn” are:

X + Y

J SLL 3

X/Y/–Z

K MOD L

J LAND OCT 77 NEV K

P * Q // FRAC X

7.3.4 *Conditions*. A condition shall be made up of comparisons connected by the words OR and AND, of which AND shall be the more tightly binding.

The syntax shall be:

condition ::= subcondition [OR subcondition] ...

subcondition ::= comparison [AND comparison] ...

comparison ::= expn comparator expn

comparator ::= =|#|<|<=|>|>=|:=:|:#:

The comparators =, # shall operate on the irreducible level of the six primitive modes BYTE, INT, FRAC, REAL, PROC and STACK, but not on the LABEL mode. The comparators <, <=, >, >= shall operate on the three plain modes REAL, FRAC and INT. The comparators :=:, :#: shall operate on the nine REF modes, i.e., they shall compare addresses.

In all cases dereferencing and widening shall be applied to the operands, where necessary, to extract a value of appropriate mode. In the case of the comparisons between plain modes, mode conversion (widening) shall occur as for the alternatives in a conditional expression. The comparisons <, <=, >, >= shall not apply to bytes; such operands shall first be widened to mode integer.

Conditions shall be evaluated from left to right only as far as is necessary to determine their truth or falsity.

NOTE The characters $ and £ are alternatives to #, both alone and in :#:

Examples of “condition” are:

X = Y AND P < 0

X = 7

XX = YY true if the values in the locations pointed to by XX and YY are the same value;

XX :=: YY true if the locations pointed to by XX and YY are the same locations;

AA :#: B true if the array pointed to by AA is not the array B;

NEXTCELL : = : CELLA true if NEXTCELL is pointing at CELLA.

7.3.5 *Non-arithmetic expressions*. Expressions of modes other than plain modes shall be of the same syntactic form as specified in **7.3.3**. However, the absence of operators that deliver non-plain modes means that in practice the syntax can be simplified as follows:

npexpn ::= npconstant | variable | functioncall | condexpn | (npexpn)
npconstant ::= identifier | string | structure

Non-plain constants shall be as follows.

a) Procedure and stack constants shall be represented by the identifiers of literal procedures or stacks that could possibly be external to the module (but not SVC procedures; see **7.5**).

b) Label constants shall be represented by the identifiers of literal labels. If a label constant occurs in an expression, the label value shall be obtained by taking as level the current stack pointer that identifies the execution of the current procedure and by taking as address the address of the label in the code (see **7.2.2**).

c) In the case of expressions of mode ref array or ref record, the address of complete arrays or records may be denoted by a structure of appropriate mode. In the case of other reference modes, the address of a variable of the corresponding primitive mode may be denoted by that variable.

d) In the particular case of mode REF ARRAY BYTE a string may also be used and storage for the array of bytes denoted by the string shall be located in a pool of strings and the value represented by the string shall be the address of the string in the pool.

NOTE 1 The compiler may choose to share the storage for identical strings; they should be treated as read-only.
NOTE 2 See **7.2.6.3** for other uses of the string pool.

Requirements for non-plain variables, function calls and conditional expressions shall be as for arithmetic modes.

Thus, for example, there may be ref array conditional expressions; these can be useful as parameters of procedure calls such as

TWRT(IF J = O THEN JIM.NAME ELSE “NOBODY” END)

Examples of “npexpn” are:

RESTART

JOHN

WHO.NAME

“FRED”

IF AA :#: BB THEN JIM.MOTHER ELSE JANE END

7.3.6 ***Byte arithmetic***

This clause summarises the various specified distinctions between integers and bytes.

The modes BYTE and INT are quite distinct, and variables of these modes shall only hold appropriate values.

NOTE 1 The fact that the value 3 (say) is a permissible value of both modes should cause no more confusion than the fact that the value 0.5 is a permissible value of both FRAC and REAL modes. The internal representations are distinct in both cases.

The syntactic form “integer” (of which 3 is an example) shall be interpreted according to its context. In an expression (dynamically evaluated), the mode shall be BYTE if the value lies in the range [0, 255] and shall be INT otherwise. As an initial value for a BYTE or INT variable, and consequently as a value in # sequences in strings, the mode shall be determined by that of the variable being initialized.

The only effective operators defined on BYTE values shall be LAND, LOR, NEV, = and #.

NOTE 2 When using operators such as +, –, <, >, which take INT operands, care is needed to ensure that unnecessary run time widening of values does not take place.

As a consequence of these requirements, if M has been declared to be of mode BYTE, the statement

M := 7

complies with the requirements of this standard whereas, because + is not defined between bytes,

M := M + 1

does not comply with the requirements of this standard; to increment M,

M := BYTE (M + 1)

shall be written.

7.3.7 *Static integer expressions*. The syntactic form "length" is used as the length of an array, the length of a stack and as a repetition factor in an array initialization. It is a static integer expression and is evaluated at compilation time.

The syntax shall be:

length	::= staticintexpn
staticintexpn	::= staticterm [staticdyadicop staticterm]
staticterm	::= [staticmonadicop]...staticprimary
staticmonadicop	::= + | –
staticdyadicop	::= + | – | * | /
staticprimary	::= integer | (staticintexpn)

The operators + – * have their normal meaning of addition, subtraction and multiplication respectively. The operator / denotes integer division giving an integer result with truncation towards zero. They take the precedence given in Table 5. Operations of the same precedence are applied from left to right.

Any intermediate result shall lie in some range $[-2^n, +2^n - 1]$ and the final result shall lie in some range $[0, +2^n - 1]$ where n is dependent upon the implementation and shall be not less than 15.

7.4 Statements

7.4.1 *General*. Statements shall define the actual operations to be performed. The statements shall be obeyed sequentially unless stated otherwise. A series of statements separated from each other by semicolons shall be known as a sequence.

NOTE All statements may be labelled.

The possible forms of statement shall be:

a) block;

b) assignment statement;

c) goto statement;

d) switch statement;

e) if statement;

f) for and to statements;

g) while statement;

h) procedure statement;

i) return statement;

j) dummy statement;

k) code statement.

The syntax shall be:

statement	::= labels unlabelledst
unlabelledst	::= block | assignmentst | gotost | switchst | ifst | forst | whilest | procst | returnst | dummyst | codest
sequence	::= statement [; statement] ...

7.4.2 *Labels*. A label shall have the form of an identifier. A statement shall be labelled by preceding it by a label followed by a colon.

NOTE 1 A statement may have many labels.

The syntax shall be:

labels	::= [identifier :] ...

NOTE 2 For the scope of a label see **7.2.10**.

7.4.3 *Blocks*. A block shall introduce a new level of nomenclature and access for local variables. A block shall consist of the word BLOCK followed by declarations of simple variables, statements and the word ENDBLOCK.

The syntax shall be:

block ::= BLOCK blockbody ENDBLOCK

blockbody ::= [simpledec;] ... sequence

NOTE Because a block is a statement, the nesting of blocks may be continued indefinitely. The body of a procedure behaves like a block and the parameters of the procedure are considered to be declared in that block. The body of a for statement also behaves like a block, and the control variable is considered to be declared in that block.

The scope of a local variable shall be the block in which it is declared, including any inner blocks, unless the identifier is redeclared therein. In the declarations in a block, the contents of a local variable shall not be used in an initial value until after the declaration of the variable itself. Thus the following example does not comply with the requirements of this standard.

```
BLOCK
    REF INT JJ := KK;
    REF INT KK := J;
ENDBLOCK
```

The storage space for all inner blocks shall be allocated on entry to the procedure concerned and there shall be no time penalty involved in entering inner blocks. The storage for variables in blocks on the same level shall be shared.

7.4.4 *Assignment statements*. An assignment statement shall consist of a list of destinations followed by := (the left part) followed by an expression.

The destinations shall each consist of a variable (possibly preceded by VAL to indicate dereferencing) and all the destinations shall be of the same mode. The expression shall be evaluated and shall be converted to that mode and shall be assigned to each destination in turn. The expression on the right shall be evaluated before the destinations on the left are evaluated and the evaluation of the destinations and assignment to them shall occur from right to left. (For mode conversions see **7.3**.)

The syntax shall be:

assignmentst ::= destination := [destination :=] ... expn

destination ::= variable | VAL variable

Examples of “assignmentst” are:

```
I := J := 7 + K : / L
A(I) := B(I) := X := J + 1
CELLA.HD := 37
U.RL := V.RL * W.RL – V.IM * W.IM
N := LENGTH BB2(J)
JJ := G(7)
JIM.ADDRESS := JANE.ADDRESS
```

NOTE There is a distinction between

```
XX := X
```

which assigns the address of X to XX, and

```
VAL XX := X
```

which assigns the value in X to the location currently pointed to by XX.

The destination shall denote a single location and not an actual array or record. Thus

```
U := V
```

and

```
JIM.NAME := “JIM”
```

do not comply with the requirements of this standard.

7.4.5 *Goto statements.* A goto statement shall cause an explicit transfer of control and shall consist of the keyword GOTO followed by a label expression.

The syntax shall be:

gotost ::= GOTO expn

Examples of "gotost" are;

GOTO FINISH
GOTO RESTART
GOTO ENDOFPROGRAM
GOTO S(J)
GOTO IF P < Q THEN L1 ELSE L2 END

NOTE The normal case of a label expression is a literal label, when the effect of the goto statement is to transfer control within the current procedure. The more general situation involving label variables is likely to prove of most value in controlling error recovery.

Note that GOTO S(J) is not formally a switch; it is a jump to one of an array of labels. See **7.4.6** for the switch statement.

A goto statement shall not lead into a block or the body of a for statement.

An obvious method by which a jump other than to a local literal label may be performed shall be by the following procedure.

The current level denoted by the current position of the stack pointer is compared with the level component of the label value; this component identifies the execution of the procedure that was current when the label value was created. If the two levels are equal, control is passed to the address in the code given by the address component of the label value and the statement is complete. If the two levels are unequal, the current level is terminated and the comparison performed again with the calling level as current level. This process is repeated until either the levels agree, when transfer will occur, or the root level has been reached. In the latter case the label value is not in scope and an unrecoverable error occurs (see **7.6**).

In short, control is passed to the label address in the relevant activation of the procedure containing the label and any procedure calls descendant from that activation and still active shall be terminated.

The following example illustrates the typical use of a label variable for error recovery:

```
DATA.....
        LABEL RESTART;                          % THIS IS A LABEL VARIABLE %
        .
        .
        .
        .
ENDDATA;
PROC MAIN ();
        .
        .
        .
        .
        RESTART := L2;                          % ASSIGN L2 TO THE VARIABLE %
L2;
        .
        .
        .
        ONE ();
        .
        .
ENDPROC;
```

```
PROC ONE ();
        .
        .
        .
        TWO ():
        .
        .
        .
ENDPROC;
PROC TWO ():
        .
        .
        .
        .
        GOTO RESTART;                    %RETURN TO L2%
        .
        .
        .
        .
ENDPROC;
```

In this example, the label variable RESTART is in a data brick and is thus accessible throughout the module. The procedure MAIN assigns the value of its literal label L2 and current level to the variable RESTART. Procedure MAIN now calls ONE which in turns calls TWO. In some circumstances TWO can fail and the statement GOTO RESTART in TWO will terminate the activations of ONE and TWO and return control to L2 in MAIN. Any relevant error action can now be taken in the coding at L2. The advantage of this technique is that no parameters need be set up or tested at the many possible calls of ONE and TWO; the mechanism performs no work until recovery is necessary.

7.4.6 *Switch statements*. A switch statement shall transfer control to one of a number of literal labels according to the value of an expression that is evaluated as if the value were assigned to a variable of mode integer.

The syntax shall be;

```
switchst    ::= SWITCH expn OF labellist
labellist   ::= identifier [ , identifier ]...
```

The identifiers shall be those of literal labels in the procedure containing the statement. If the value of the integer expression is zero, negative or greater than the number of labels in the list, control shall not be transferred but shall be passed to the next statement, thus providing a convenient and efficient means of monitoring the value of the integer expression.

An example is;

```
SWITCH K OF P1, P2, P3, P4, PE;
FAIL ("K OUT OF RANGE")
```

NOTE A switch statement is a statement and so can be labelled and therefore jumped to in the usual way.

7.4.7 *Conditional statements.* Conditional statements shall cause certain statements to be executed or skipped according to the running values of specified conditions.

The syntax shall be:

ifst ::= IF condition THEN sequence [ELSEIF condition THEN sequence] ... [ELSE sequence] END

The behaviour shall be as follows. The condition following IF shall be evaluated. If this is true, the sequence following THEN shall be obeyed and this completes the statement. If the condition is not true, the conditions following the optional keywords ELSEIF shall be evaluated until one of value true is encountered; the corresponding sequence shall then be obeyed and this shall complete the statement. If there are no optional ELSEIF conditions, or if they are false, the sequence following ELSE, if present, shall be obeyed and this shall complete the statement. If there is no ELSE part, the statement shall be considered to be completed.

There shall be no restrictions on the statements in the sequences. If the statements are labelled and control passed to them directly by a goto or switch statement, the behaviour after the execution of the sequence shall be as if control had entered via the conditions.

Examples of "ifst" are:

```
IF X = O THEN P := Q END
IF Y > 1 THEN GOTO STOP END
IF = Z AND J = K THEN
      I := I + 1; L := K := INT U.RL;
      F(I, J, K);
END
IF X = O THEN P := Q ELSE Q := P END
IF X = 1 THEN I := 0 ELSEIF X = 2 THEN J := 0 ELSE K := 0 END
IF X < Y THEN
      XX := YY; J := K;
ELSEIF X > Y
      XX := ZZ; J := L;
END
```

The last example is equivalent to:

```
IF X < Y THEN
      XX := YY; J := K;
ELSE
      IF X > Y THEN
XX := ZZ; J := L;
      END;
END
```

The use of ELSEIF saves an END.

7.4.8 *For and to statements.* A for statement shall cause a sequence of statements to be repeatedly executed zero or more times and in addition shall perform a series of assignments to a control variable.

The syntax shall be:

forst ::= [FOR identifier := expn [BY expn]] TO expn DO blockbody REP

Examples of "forst" are:

```
FOR I := 1 TO 10 DO A(I) := 0REP
FOR T := 10 BY – 1 TO K — L DO
        IF G(T) < 1 THEN GOTO SKIP END;
REP
FOR J := – K BY 2 TO L DO
        A(J) := B(J) := C(J) := 1.0;
REP
TO 15 DO TWRT ("*/") REP
```

The initial value, increment and limit shall be evaluated once only at the start of the for statement in the following order: increment, limit, initial value. The evaluations shall occur as if the values were assigned to variables of mode integer.

If the "BY expn" is omitted, an increment of 1 shall be assumed.

The controlled sequence shall behave as a block (hence declarations shall be permitted) and the control variable shall be considered to be declared to be of mode INT in that block. Thus on exit from the for statement, the control variable shall be inaccessible.

The control variable shall be read-only. Thus it shall not occur explicitly as the destination of an assignment statement nor shall its address be evaluated (e.g. handed over to a REF parameter of a procedure) since an indirect assignment might then occur.

A goto statement shall not lead into the blockbody controlled by a for statement.

NOTE 1 The detailed behaviour of the for statement

FOR name := e1 BY e2 TO e3 DO seq REP

is more accurately described by the following statements, which are essentially equivalent:

```
BLOCK INT inc := e2, limit := e3, start := e1;
   BLOCK INT name := start;
lab:  IF inc > 0 AND name <= limit
         OR inc < 0 AND name >= limit THEN
         BLOCK
         seq;
         ENDBLOCK;
         name := name + inc; GOTO lab;
        END;
   ENDBLOCK;
ENDBLOCK
```

Since declarations occurring in the sequence are considered to be part of the sequence, any initializations are repeated on each iteration.

If the increment is zero, the behaviour is not specified in this standard.

When the control variable is not used, the for statement may be abbreviated, in which case the abbreviated form shall be

TO expn DO blockbody REP

and the body shall be executed "expn" times.

NOTE 2 If the "expn" is non-positive, the body is therefore not executed at all. Note also that the body still behaves as a block.

7.4.9 *While statements*. A while statement shall cause a sequence of statements to be repeatedly executed while a specified condition remains true.

The syntax shall be:

whilest ::= WHILE condition DO sequence REP

NOTE The statement
WHILE e DO seq REP
is essentially equivalent to
lab :IF e THEN seq; GOTO lab END

7.4.10 *Procedure statements*. A procedure statement shall invoke the execution of the corresponding procedure body after the correspondence between the actual and formal parameters (if any) has been performed.

The form of a procedure statement shall be the same as that of a function call (see **7.3.2.4**). It shall consist of the identifier of a literal procedure (or SVC procedure, see **7.5**), or the identifier of a procedure variable, followed by a list of parameters separated by commas and enclosed in round brackets. The brackets shall not be omitted even if there are no parameters.

The syntax shall be:

procst ::= variable paralist | identifier paralist
paralist ::= ([expn [, expn] ...])

The correspondence between the parameters shall be established by assigning the actual parameters to the formal variables just as if they were the right and left sides respectively of assignment statements. In the body of the procedure, the formal parameters shall behave just as if they were variables declared in the block that is the body of the procedure. The parameters shall be evaluated from left to right before the procedure itself is evaluated.

If a function call is used as a procedure statement for its side effects, the result shall be discarded.

An example is the procedure:

```
PROC ACTION (REAL A, B, INT N, REF ARRAY BYTE C, LABEL L);
        INT I, J, K;
        .
        .
        .
        .
ENDPROC
```

with the statement

ACTION (P, Q * R, 1, “NOGO”, FAILURE)

which then behaves just as if the statements

```
A  := P; B := Q * R; N := 1;
C  := “NOGO”; L := FAILURE
```

were obeyed immediately on entry to the body of ACTION followed by the body of ACTION. It is as if the procedure statement were replaced by the block

```
BLOCK
        REAL A := P,B := Q * R;
        INT N := 1;
        REF ARRAY BYTE C := “NOGO”;
        LABEL L := FAILURE;
        INT I, J, K;
        .
        .
        .
        .
ENDBLOCK
```

7.4.11 *Return statements*. A return statement can be used to leave a procedure and return control to the calling procedure in the usual way. In the case of a procedure that is not a function procedure, the final ENDPROC shall behave as if preceded by a return statement. In the case of a function procedure, control shall be explicitly returned and so the final ENDPROC shall be immediately preceded by a return statement or a goto statement.

The syntax shall be:

returnst ::= RETURN | RETURN (expn)

The "expn" shall only be present in the case of a function procedure and shall be evaluated to give the result.

7.4.12 *Dummy statements*. The dummy statement shall simply be a void whose execution has no effect. It shall be used when formally necessary in order to allow a label to be placed before keywords such as END, REP, etc.

The syntax shall be;

dummyst ::= [ ]

For example, there is a dummy statement after the colon in

```
;LAB : REP
```

7.4.13 *Code statements*. A code statement shall be represented by a codesequence (see **7.1.8**). Code statements shall not be permitted in the application language (see **7.6**).

The syntax shall be;

codest ::= codeseq

An example is;

```
CODE 6, 0;
        MOV * I/EXAMPLES, * JJ/EXAMPLES
        * RTL
```

This example shows the characters * and / being used as trip1 and trip2 respectively.

7.5 Modules

7.5.1 The unit of compilation shall be the module. It shall consist of none or any of the following:

a) environment descriptions;

b) record mode definitions;

c) titles;

d) LET definitions;

e) the bricks to be compiled.

An environment description shall describe:

f) the format of another separately compiled brick;

g) the format of an available supervisor call;

h) also (optionally and redundantly), an ENT brick to be compiled in the present module.

The bricks to be compiled shall be preceded by the word ENT if their name is to be made accessible to other modules via the linker.

The syntax shall be:

module	::= moduleitem [; moduleitem] ...
moduleitem	::= [environdescn | recmodedef | letdefinition | title | brick]
environdescn	::= EXT stackdescn | EXT procdescn | EXT datadescn | SVC procdescn | SVC datadescn
stackdescn	::= STACK idlist
procdescn	::= PROC procdescriptor idlist
datadescn	::= DATA identifier; dspec [; dspec] ... ENDDATA
dspec	::= [simplespec | arrayspec | recordspec]
arrayspec	::= ARRAY (length [, length] ...) amode idlist

recordspec ::= recmode idlist
brick ::= [ENT] datadec | [ENT] procdec | [ENT] stackdec

7.5.2 An example of "module" is:

```
TITLE
ILLUSTRATION OF MODULE;
LET NL = 10;
EXT PROC (REF ARRAY BYTE) TWRT;
SVC DATA RRERR;
        LABEL ERL;
        INT ERN;
        PROC (INT) ERP;
ENDDATA;
MODE PAIR (INT OLD, NEW);
ENT PROC SEARCH (REF ARRAY PAIR P, INT X) INT;
        % SEARCHES ARRAY P FOR OLD ENTRY X AND %
        % RETURNS CORRESPONDING NEW ENTRY %
        % OUTPUTS MESSAGE AND GOES TO ERL IF FAILS %
        FOR I : = 1 TO LENGTH P DO
                REF PAIR RP := P(I);
                IF RP.OLD = X THEN RETURN (RP.NEW) END;
        REP;
        TWRT ("# NL # SEARCH FAILS");
        GOTO ERL;
ENDPROC;
```

7.5.3 The external descriptions shall inform the compiler of the characteristics of bricks that are external to the module, so that reference to these bricks may be made in the module. The stack description shall merely list the identifiers of stacks. The data description shall be effectively the same as the declaration of the data brick except that initial values shall not be set. The procedure description shall describe the parameters and result in the same style as for procedure variables.

If a procedure description is preceded by SVC rather than EXT, procedure statements or function calls referring to that procedure shall be treated as supervisor calls and a special linkage may be compiled. The name of an SVC procedure shall be used only in a procedure statement or function call and not as a literal value. If a data description is preceded by SVC rather than EXT, the data brick shall be treated as private to the task and a special method of access may be compiled for variables within the brick.

7.6 Integrity

NOTE The language specified in this standard is the full RTL/2 language. This full language is insecure in the sense that reference values could be used as addresses without their containing sensible values, and it is unreasonable to suppose that efficient implementations of the language could, in general, trap such activities. If the language were constrained so that references could not be manipulated, much system programming would be difficult, inefficient or impossible. On the other hand, a secure language is of value for application programs where address manipulation is of less importance. Hence RTL/2 may be seen as comprising two languages, with the full language being the system language and a secure subset being the application language.

7.6.1 The application language shall be the full language with the following restrictions.

a) All variables of modes other than plain modes and LABEL shall be initialized.

b) REF primitive variables shall not be assigned literal values of primitive variables or the contents of other reference variables if, in either case, the source variable is out of scope of the destination variable at the latter's declaration. An expression in a return statement shall be considered to be assigned to a global variable.

c) Code statements shall not be used.

NOTE It is intended that a program complex formed by linking application modules compiled with identical environments will be secure.

7.6.2 In the system language, the following shall be optional:

a) a stack check on procedure entry;

b) array bound checks;

c) monitoring of general goto statements.

7.6.3 In the application language:

a) a stack check on procedure entry shall be included;

b) array bound checks shall be optional for fetching from plain arrays and otherwise shall be included;

c) monitoring of general goto statements shall be included.

NOTE For an outline of a recommended mechanism for handling the errors that arise when these and other checks fail, see Appendix B.

Appendix A Information on input/output

A.1 Introduction

The RTL/2 language specified in this standard contains no specific provision for input and output since to do so might prove an undesirable burden for some systems. However, in order to aid transportability of programs between systems, a recommended package[2)] for character streaming is outlined in this appendix.

A.2 Streaming mechanism

Each task has two SVC data bricks associated with it, namely:

```
DATA RRSIO;                    DATA RRSED;
    PROC()BYTE IN;     and     BYTE TERMCH;
    PROC(BYTE) OUT;            BYTE IOFLAG;
ENDDATA                        ENDDATA
```

The procedure in IN will remove the next character from the current input stream and return it as result. The procedure in OUT will send the character passed as parameter to the current output stream. All streaming of individual characters will be via IN and OUT as appropriate.

TERMCH and IOFLAG are concerned with the standard stream input procedures.

A.3 Input

Individual characters are obtained from the current input stream by calls of the procedure variable IN in data brick RRSIO.

Numbers and text may be read from the current input stream by the following procedures. In each case the last character read and removed (the terminating character) is placed in TERMCH in data brick RRSED.

PROC FREAD ( ) FRAC reads a signed decimal number and returns a truncated fraction value as result.

PROC IREAD ( ) INT reads a signed decimal integer and returns its value as result.

PROC RREAD ( ) REAL reads a signed decimal number with optional exponent and returns its value as result.

PROC TREAD(REF ARRAY BYTE X, T) INT reads characters and places them into successive elements of X. Input is terminated as soon as one of the characters of T is encountered. The number of characters placed in X is returned as result.

A.4 Output

Individual characters are sent to the current output stream by calls of the procedure variable OUT in data brick RRSIO.

Numbers and text may be output to the current output stream by the following procedures.

PROC NLS (INT N) and PROC SPS(INT N) send N newline and space characters respectively.

PROC FWRT(FRAC X) sends the unrounded fraction value X as a signed decimal number in a fixed format dependent upon the implementation.

PROC IWRT(INT X) sends the integer value X as a signed decimal integer with leading zeros suppressed.

PROC RWRT(REAL X) sends the unrounded real value X as a decimal number in a fixed format dependent upon the implementation.

PROC FWRTF(FRAC X, INT N), PROC IWRTF(INT X, M) and PROC RWRTF(REAL X, INT M, N) send the fraction, integer and real values (rounded where appropriate) in formats determined from the values of the additional parameters M and N.

PROC TWRT(REF ARRAY BYTE A) sends the successive elements of the array A as characters.

[2)] For further details on the package for character streaming and the mechanism for error recovery, apply to the Central Enquiries Section, British Standards Institution, 2 Park Street, London W1A 2BS, enclosing a stamped addressed envelope for reply.

Appendix B Information on error recovery

B.1 In order to aid transportability of programs between systems, a recommended mechanism[3)] for error recovery is outlined in this appendix.

B.2 Two types of error are distinguished; unrecoverable and recoverable errors. An unrecoverable error is one such that further processing of the task concerned might destroy the structure of the system, e.g. array bound violations and stack overflow. A recoverable error is one such that further processing of the task can continue without danger to the structure of the system.

The kernel of the standard is an SVC data brick thus:

```
DATA RRERR;
    LABEL ERL;
    INT ERN;
    PROC(INT) ERP;
ENDDATA;
```

ERL contains the unrecoverable error label, ERN
contains the unrecoverable error number and ERP
contains the recoverable error procedure.

On detection of an unrecoverable error by the system, an appropriate error number is assigned to ERN, any monitoring facilities are invoked and control is then passed to the label in ERE. A user task could simulate an unrecoverable error by merely assigning a number to ERN and passing control to ERL. However, this would bypass any monitoring facilities and so the standard includes

```
PROC RRGEL(INT N)
```

which when called assigns N to ERN, invokes the monitoring facilities and finally transfers control to ERL

Detection of a recoverable error results in a call of the procedure contained in ERP with the integer parameter indicating the cause of the error. Recoverable errors may be signalled by the system or the user.

A task can create its own error recovery environment merely by directly assigning appropriate values to ERL, ERN and ERP. A procedure, which needs to set up its own error environment, whilst preserving the existing one, may do this by assigning the existing values of ERL, ERN and ERP to local variables on entry and restoring them on exit.

Appendix C Recommendation on compiler limits

It is recognised that any practical compiler is likely to impose certain limits on the size of program that it will compile. At the least, there will be limits imposed by the finite nature of the compiling computer. In the interests of portability of conforming programs, it would be desirable to set minima which all compilers need to allow in order to comply with the requirements of this standard. Equally, however, to set such limits would require a number of ad hoc decisions that would be hard to justify. This standard, therefore, does not generally set such limits, but a compiler should not impose arbitrary and unnecessarily small limits. In particular it is recommended that identifier names and numbers of up to 72 characters should be acceptable by all compilers.

[3)] For further details on the package for character streaming and the mechanism for error recovery, apply to the Central Enquiries Section, British Standards Institution, 2 Park Street, London W1A 2BS, enclosing a stamped addressed envelope for reply.

Publications referred to

BS 3527, *Glossary of terms used in data processing.*

BSI — British Standards Institution

BSI is the independent national body responsible for preparing British Standards. It presents the UK view on standards in Europe and at the international level. It is incorporated by Royal Charter.

Revisions

British Standards are updated by amendment or revision. Users of British Standards should make sure that they possess the latest amendments or editions.

It is the constant aim of BSI to improve the quality of our products and services. We would be grateful if anyone finding an inaccuracy or ambiguity while using this British Standard would inform the Secretary of the technical committee responsible, the identity of which can be found on the inside front cover. Tel: 020 8996 9000. Fax: 020 8996 7400.

BSI offers members an individual updating service called PLUS which ensures that subscribers automatically receive the latest editions of standards.

Buying standards

Orders for all BSI, international and foreign standards publications should be addressed to Customer Services. Tel: 020 8996 9001. Fax: 020 8996 7001.

In response to orders for international standards, it is BSI policy to supply the BSI implementation of those that have been published as British Standards, unless otherwise requested.

Information on standards

BSI provides a wide range of information on national, European and international standards through its Library and its Technical Help to Exporters Service. Various BSI electronic information services are also available which give details on all its products and services. Contact the Information Centre. Tel: 020 8996 7111. Fax: 020 8996 7048.

Subscribing members of BSI are kept up to date with standards developments and receive substantial discounts on the purchase price of standards. For details of these and other benefits contact Membership Administration. Tel: 020 8996 7002. Fax: 020 8996 7001.

BSI
389 Chiswick High Road
London
W4 4AL