RTL/2
Language specification

RTL/2 Reference : 1
Version : 2

Contents

0 Introduction

RTL/2 is a high-level programming language developed at the Corporate Laboratory of Imperial Chemical Industries Limited. It is designed for use in real-time computing and is especially suited for the programming of on-line data acquisition, communication and control systems. Structurally a simple language, it incorporates features important for the effective programming of real-time applications including the ability to program tasks which the computer performs in parallel (or at least appears to do so as far as an outside observer is concerned). It was considered particularly important to be able to produce programs of high integrity which are nevertheless efficient enough in speed and storage requirements to run on small on-line computers.

This manual is intended to provide a semi-formal definition of RTL/2. It is not a training manual but a reference manual

See also

Introduction to RTL/2	RTL/2 Reference : 2
RTL/2 training manual	RTL/2 Reference : 3
RTL/2 Systems standards	RTL/2 Reference : 4
RTL/2 Standard stream I/O	RTL/2 Reference : 5

0.1

It is virtually impossible to define precisely a high-level language such that programs will be executed equally efficiently by all types of computer. It has therefore been necessary to leave certain areas of the definition of RTL/2 deliberately vague so that implementation can take the best advantage of characteristics of particular computers. The most important of these areas are (i) accuracy and range of real values (ii) number of bits in an integer word, (iii) behaviour on arithmetic overflow; they are indicated in this manual by phrases such as '... is implementation dependent'. On the other hand it should be noted that the representation of integer values has been explicitly specified to be in 2's complement form; thus implementations of RTL/2 on computers which do not use this representation will be less efficient.

0.2 Syntax Notation

The notation used to describe the syntax of the language is as follows:

The terminal symbols of the language stand for 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.

The metasymbol ::= denotes 'is' and the metasymbol | denotes 'or'.

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

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

Example:

```
name ::= letter [ letter | digit ] ...
 digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
letter ::= A | B | C ..... Y | Z
```

Thus a 'name' is a sequence of letters and digits of which the first must be 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.

In the text the names of classes are enclosed in quotes when formal correctness is to be emphasised. In less formal passages the quotes are omitted.

0.3 Notes on Examples

This manual contains many examples in which, for compactness, identifiers are used without being 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
```

1 Basic Elements

1.1 Text

The text of an RTL/2 program is a sequence of characters drawn from the 'language subset' of ISO7. This set is tabulated in Appendix 3.

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

The items are of various types:

(i) A name
(ii) An arithmetic constant
(iii) A string
(iv) A comment
(v) A title
(vi) An option
(vii) A code sequence
(viii) A separator

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

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

1.2 Names

A name consists of a sequence of letters and digits of which the first is a letter. There is no limit to the number of characters in a name and all the characters will be significant.

In particular, names which are significant externally (via EXT and ENT; see 6) will be presented in full to the external system; this system may of course impose restrictions on the number of characters allowed and hence on the user's choice of such names.

Syntax:

```
      name ::= letter [ letter | digit ] ...
    letter ::= A | B | C | D ..... X | Y | Z
     digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
identifier ::= name
```

Names are used for two purposes:

(i) to denote keywords in the language. Keywords are names such as IF and REAL which have a predefined meaning. All the keywords are listed in Appendix 4.

(ii) to denote user identifiers. Identifiers are used for a variety of purposes such as naming variables, arrays of variables or pieces of code. A name which denotes a keyword cannot be used as an identifier.

Examples of 'identifier':

```
J
GEORGE
V17X
NO2
```

Note that 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 the keyword GOTO must not be written as GO TO.

1.3 Arithmetic Constants

There are four types of numerical data in RTL/2. These are real, integer, fraction and byte and are defined in detail in section 3.1.

Arithmetic constants (numbers) are used to denote literal and initial values of these types but it should be noted that the form 'integer' is used to denote constants of both type integer and byte. See 4.1.1.

Syntax:

```
    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
```

Examples of 'number':

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

The following are not legal numbers:

27.	E9	.001
'£'	'''	3B–4

Notes:

(i) A stringchar is any character of the language subset of ISO7 except for

" # £ $ HT LF

It can be an actual space.

(ii) The numerical value of a constant denoted by a stringchar in single quotes is the ISO7 value of that character as listed in Appendix 3.

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

(iv) The fraction constant includes a binary scale factor following B. Thus 17.6B–5 is stored as 17.6×2^{-5}.

(v) A real constant may include a decimal exponent following E. Thus 0.1E7 is stored as 0.1×10^{7}.

1.4 Strings

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 3.5.2, 3.5.3, 4.4, 5.9).

A string is basically a sequence of stringchars enclosed in double quotes (") and denotes the set of values formed by taking the ISO7 values of those characters.

As a consequence of this rule space characters are allowed in strings (and stand for themselves) whereas newline and tab are not. Thus a string must be on one line; this restriction prevents subsequent lines of program text being treated as part of a string if the closing quote is inadvertently omitted. Strings that are adjacent apart from layout characters will be concatenated as a single string and so a long string can be written as several stringparts on successive lines.

The character # (which is not a stringchar) has a special significance. A sequence enclosed within a pair of # characters within a string (a stringinsert) will be interpreted as part of an array initial value as described in 3.5.3; thus it will consist of a sequence of integer constants in the range 0—255 (possibly followed by replication factors) separated by commas. Such sequences behave like normal program text and so tab and space characters, comments and LET replacements (see 1.11) are allowed in the normal way. Newline characters are, however, not allowed since the # sequence is formally part of the string.

Syntax:

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

Notes:

(i) There is no direct way of representing a newline in a string. It must be treated like any other character which is not a stringchar and inserted in a # sequence using the LET facility for clarity if so desired (see 1.11).

(ii) The characters $ or £ may be used as alternatives to # but they must occur as matched pairs.

(iii) For two or more adjacent strings to be treated as one string they must occur explicitly as strings in the text and not implicitly via LET replacements.

Examples of 'string':

Assuming that

```
LET NL = 10;
LET TAB = 9;
```

have been set, then valid examples 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 are not legal strings

```
"PRICE=£"
" " "
```

1.5 Comments

All characters starting from and including the character % up to and including the next % are treated as comment and ignored. The layout characters space and tab are allowed in comments but a newline is not. This restriction prevents subsequent lines of program text being treated as comment if a % character is inadvertently omitted.

Example:

```
% THIS IS A COMMENT %
```

1.6 Titles

A title provides a means of labelling the object code or a listing of the source code in whole or in part. The details will depend upon the implementation. It consists of the keyword TITLE followed by any sequence of characters not including a semicolon (;).

Example:

```
TITLE DEBUG ROUTINE 13 MAY 1971
```

1.7 Options

An option provides a means of informing the compiler of any special actions which may be required. An option is valid from its point of occurrence until another option or the end of the program text. Each option completely overrides any previous option and actions not explicitly reset will take a default value.

Syntax:

```
option ::= OPTION (digitlist) [opitem [,opitem] ...]
opitem ::= opchar...
opchar ::= letter | digit
```

The 'digitlist' numbers the option and allows the compiler to be informed of temporary alterations to particular options without modification of the source text.

Example of 'option':

```
OPTION(3) CM, BC
```

1.8 Code

A sequence starting with the keyword CODE will be interpreted as a machine code sequence.

Syntax:

```
codeseq ::= codeheading codeitem...
codeheading ::= CODE digitlist, digitlist;
codeitem ::= ISO7–character–other–than–trip1–or–trip2 |
             trip1 letitem | trip2 name
```

The two values denoted by 'digitlist' in the heading will 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 constitutes the actual code and within this sequence layout characters stand for themselves.

Within the sequence an RTL/2 letitem (see 1.11) may be denoted by immediately prefixing the item by the machine dependent character 'trip1'. Thus a name, number, comment, string or separator may be referred to by this means and in particular a LET name will be replaced by its corresponding item sequence. Also note that it is possible to declare an RTL/2 literal label within a code sequence. The action performed by the compiler and the code inserted for each item will be implementation dependent.

Whenever any variable name in a data brick or selector name of a MODE is referenced then that name must be followed by the name of the host brick or mode itself preceeded by the second (different) character 'trip2'. This construction allows the compiler to check the validity of the reference and will minimise errors due to incorrectly accessing names if their definition is changed.

The characters 'trip1' and 'trip2' may themselves be denoted within the sequence by prefixing them by 'trip1'.

The code sequence is terminated by the last 'codeitem' being the keyword RTL preceeded by the character 'trip1'.

1.9 Separators

The remaining items are separators which are used as punctuation symbols and operators. Separators are either compound and constructed of several characters, or else just a single character.

The compound separators are:

:= :/ // <= >= :=: :#:

with :$: and :£: being alternative forms for :#:. A compound separator may not contain a layout character.

The single character separators are:

( ) * + – , . / : ; < = > #

with $ and £ being alternative forms for #.

1.10 Item Hierarchy

It is emphasised that the various items are all of the same lexicographic priority, apart from the case of items in a # sequence in a string and the case of letitems in a CODE sequence.

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

Note, however, that there may be an end-of-module character which has overall significance and thus may only be used to mark the end of a file of program text.

1.11 LET Replacement

RTL/2 includes a simple non-parameterised replacement facility. This enables the user to give a name to a sequence of items and use this name instead of the sequence. The use of the facility to name constants within a program is strongly recommended (see 1.4 Examples).

Syntax:

letdefinition ::= LET name = [letitem] ...;
letitem ::= name | number | string | comment | separator

The 'name' following LET may not be a keyword and the letitems may not be one of the keywords;

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

A 'letdefinition' may only occur where a moduleitem is valid. See 6.

The definition is valid from its point of occurrence in the program text until the end of the text or redefinition. All occurrences of the name during its validity will be replaced by the defined sequence, this includes occurrences within # sequences within strings.

Examples:

```
LET NL = 10;
LET RAB = REF ARRAY BYTE;
LET ATMOS = 14.7;
```

2 Program Structure

A computer which is not specifically designed to execute RTL/2 code directly will 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 will have a hierarchical structure. The top level, usually known as the supervisor will itself be a program complex and the individual groups of lower levels can themselves be considered to be program complexes but with a different environment to the supervisor complex.

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

(i) Procedure brick
(ii) Data brick
(iii) 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. It may have parameters and local variables but the latter are restricted to be scalars. The entry mechanism and implementation of local variables is re-entrant. 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 (ie, local) variables and other housekeeping items.

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

The compiler must be informed of the environment of a module for satisfactory compilation to take place. This environment will, in general, consist of two parts; firstly 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 will in the case of the supervisor be simply the enhanced machine, whereas in the case of a complex running under the supervisor in a two-state machine, it will also consist of the environment provided by that supervisor. This will comprise 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 must include descriptions of SVC procedures and data bricks which it accesses.

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

The various cross references implied by these environment descriptions will be satisfied by the linker program. See 6.

2.1 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 manual. 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 6) will provide the channel for the communication of task control requests.

Whenever a new task is created a stack will be nominated as workspace and a procedure as the coding to be obeyed. Later operations on the task may be made by reference to its stack. Each stack can, of course, only be used 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 onto part of the stack.

3 Declarations

Declarations serve to define the properties of the various identifiers used within a module of program. Identifiers are used for many purposes and these are classified below.

3.1 Modes

An RTL/2 program is ultimately concerned with manipulating numerical data of four plain modes. These plain modes are

(i) Real
A value of mode REAL is represented by a floating point number. The range and accuracy are implementation dependent.

(ii) Integer
A value of mode INT is a signed integer. The range is implementation dependent but will usually be that provided by the natural word of a word machine. In the remainder of this manual the term word will be used in the sense of the space occupied by an integer. It is anticipated that at least 16 bits will always be used; it is also assumed that integer values are stored in 2's complement form (this latter restriction is necessary to ensure that logical operations such as LAND are well defined). See also 4.2.2 for double forms of the integer mode.

(iii) Fraction
A value of mode FRAC is a value in the range [−1, +1) (that is including −1 but not +1). The accuracy is implementation dependent but a fraction value will occupy a word as defined for integers. Fractions are provided so that machine independent fixed point arithmetic operations can be written for computers that do not have floating point hardware. See also 4.2.2 for double forms of the fraction mode.

(iv) Byte
A value of mode BYTE is an integer in the range [0,255] .

There are also three other primitive modes:

(v) Label
A value of mode LABEL is a level-address pair. The level indicates a stack pointer position identifying an execution of a procedure and the address is that of a literal label in that procedure to which control will be transferred if the label value be used in a GOTO statement (see 5.4).

(vi) Procedure
A value of mode PROC is a pointer to a piece of executable code (ie, a procedure).

(vii) Stack
A value of mode STACK is a pointer to an area for dynamic workspace (ie, a stack).

Identifiers may be used to denote locations which contain values of the above modes in the traditional manner. In addition to these primitive modes there are reference forms of these modes, values of which are references to instances of values of the primitive modes themselves; identifiers may be used to denote locations which contain these references. (Thus we have 'normal' variables and 'address' variables).

The programmer may also define new composite modes known as records and 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 used to denote variables containing such references.

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

Finally identifiers may be used to denote literal instances of labels, stacks and procedures. It has not been felt necessary to allow identifiers to denote literal instances of plain values since the effect may be achieved by the use of the LET facility.

In order to clarify the distinction between literal, normal and reference modes two examples will be considered.

Firstly consider the case of mode integer.

(a) Literal form; in this case an identifier would be used to denote an actual integer constant. Strictly speaking this is not allowed in RTL/2 but

```
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 carefully that there is no location called NL. The subsequent statement NL := 20 is not legal since this is equivalent to the nonsence 10 := 20.

(b) Value form; in this case an identifier is used to denote a location which contains integer values. Thus the declaration INT J will define such a variable and a subsequent statement such as

```
J:=20
```

is valid.

(c) Reference form; in this case an identifier is used to denote a location which contains the address of a location such as J above. Thus the declaration

```
REF INT JJ
```

will define such a variable and a subsequent statement such as

```
JJ:=J
```

will be valid and assign the address of J to JJ.

Secondly consider the case of mode procedure (see 3.6).

(a) Literal form; in this case 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) Value form; in this case 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. The use of variables such as FN as formal parameters is familiar in languages such as Algol and PL/1.

(c) Reference form; in this case an identifier is used to denote a location which contains the address of a location such as FN above. The use of such variables is likely to be rare.

3.2 Simple Declarations

A simple declaration serves to declare certain identifiers to represent simple variables of a given mode. It may also assign initial values to the variables.

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

The simple mode description is one of

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 be required; also the mode description could be

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

For 'procdescriptor' see 3.6, for 'recmode' see 3.4 and for 'arraymode' see 3.3.2.

Syntax:

```
   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]
```

For the syntax of initial values and examples see 3.5.

Examples of 'simpledec' without initial values:

```
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
```

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

3.3 Array Declarations

An array declaration declares one or more identifiers each representing an array of subscripted variables. The array may have one or more dimensions. The declaration gives the bounds of the subscripts and the modes of the variables. It may also assign initial values.

Arrays may only be declared in a data brick.

The elements of an array may be of any mode except arrays themselves; they may be references to arrays, records, references to records or primitive values. The elements must all be of the same mode.

Multidimensional arrays are implemented as vectors of references to other vectors and it is thus convenient to consider first the simple case of a vector.

3.3.1 Vector declarations

A vector (one dimensional array) is a data structure consisting of an indexable set of components of the same mode. The structure has a length attribute and the index ranges from 1 to the length.

The syntax of a vector declaration is as follows:

Syntax:

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

Examples:

```
ARRAY (10) REAL A,B,C
ARRAY (5) LIST CELLS
```

The first example declares three vectors A, B, C of real variables each of which is of length 10. The second example declares a vector of 5 records of mode LIST called CELLS. See 3.4.

The length of an array could be zero in which case no elements will actually exist (eg, a null string "").

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

Examples:

```
A (7)
CELLS (I+J)
```

It is possible to declare variables whose values are references to vectors as described in 3.2. Thus

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.

This automatic indirect reference is an instance of automatic dereferencing which is described in 4.

Note that REF ARRAY REAL is itself a simple mode and so one can declare an array of references to other arrays; thus

ARRAY (10) REF ARRAY REAL AAA

or even a reference to such an array

REF ARRAY REF ARRAY REAL AAAA

The full syntax of 'arraymode' has been deferred to the next section to avoid introducing unnecessary syntactic classes. This full syntax contains a mechanism for simplifying the description of the items AAA and AAAA above.

3.3.2 Multidimensional array declarations

Multidimensional arrays are treated as vectors of references to vectors and although in general the mechanism described above could be used for their creation a more convenient form is provided. The full syntax of array declarations is as follows:

Syntax:

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

In an 'arraydec' the number of dimensions of an array is equal to the number of lengths. Thus

ARRAY (5, 10) REAL A2

will declare a two dimensional array A2. This array is in fact a vector of length 5 each element of which is a reference to a vector of length 10. The elements of a multidimensional array may be accessed by expressions of the form

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

which are equivalent. The form A2(1) may stand alone to access the reference to the sub array.

In an 'arraymode' the dimension of the mode is the number of commas in round brackets plus one (or just one if the brackets be omitted). Thus

REF ARRAY (,) REAL AA2

will declare a simple variable AA2 whose value will be a reference to a two dimensional array of reals.

It will be noted that 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 carefully the difference between 'amode' which is the sort of thing of which one can have arrays and 'arraymode' which is the arrays of these things.

Examples of 'arraydec' without initial values:

```
ARRAY (5, 10) REAL A2, B2, C2
ARRAY (7) INT G
ARRAY (120) BYTE BUFF1, BUFF2
ARRAY (3, 9, 2) REF LIST TABLE
ARRAY (7) REF ARRAY BYTE DAYS
ARRAY (3) REF ARRAY (,) PROC (INT) H
```

The following are illegal

```
ARRAY (2) ARRAY (3) REAL X
ARRAY (I, J) REAL X
```

3.4 Record Declarations

A record is a data structure consisting of several components. A record will belong to a record class defined by a MODE definition which indicates the modes of the individual components and the identifiers by which the components may be selected.

Records may only be declared in a data brick.

Syntax:

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

Note that a component of a record may only be a simple item or an array of simple items. It may not be a record or an array of records but can be a reference to a record or an array of such references. Note also that initialisation is not allowed in the mode definition.

Examples of 'recmodedef':

```
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 which 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. This mode definition is recursive; mode definitions may also be mutually recursive.

The following are illegal.

```
MODE NUT (LIST KERNEL)
MODE CAT (ARRAY (4) CAT KITTEN)
```

Having defined the mode of a record class it is possible to declare actual records in a manner analogous to simple declarations.

Syntax:

recorddec ::= recmode initidlist

Examples of 'recorddec':

```
COMPLEX U, V, W
LIST CELLA, CELLB
PERSON JOHN, JIM, JANE
```

It is also possible to declare arrays of records (see 3.3) or references to records (see 3.2) thus:

```
ARRAY (5) LIST CELLS
REF LIST NEXTCELL
ARRAY (100) PERSON PEOPLE
REF PERSON WHO
```

Individual components of a record are 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 4) occurs if it is wished to access the components of a record referenced by a REF variable thus

```
NEXTCELL.HD
```

3.5 Initialisation of Data

Data in a data brick or local to a procedure may be initialised. In the former case the initial value must be a constant or constant address whereas in the latter case the initial value may be any suitable expression such as could occur in an assignment statement. (In the case of the application subset of RTL/2 all variables other than those of plain modes and LABEL must be initialised. See 7)

Initialisation is indicated by following the identifier by := and the appropriate initial value. Multiple initialisation is allowed in a manner analogous to the left part list of an assignment statement. See 3.2, 5.3.

Examples of initialisations of local data:

```
INT I := J + K
REF INT JJ:= G(K)
REAL X := Y := 3.9+Z
LABEL L := RESTART
```

If a local variable is not initialised then its initial value is undefined.

The remainder of this section will be devoted to the description of the initialisation of data in a data brick.

3.5.1 Primitive Modes

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

The initial value of a variable of mode procedure or stack must be (the identifier of) a literal procedure or stack, possibly external to the module. It is not possible to initialise variables of mode label. There is no default value for variables of mode label, procedure or stack.

Syntax:

simpleinitvalue ::= sign number | identifier

Examples:

```
INT J := 37, L, I := K := –11;
PROC (REAL)REAL FN := SIN
```

Note that L above is initialised to zero by default.

3.5.2 Reference Modes

The initial value of a reference variable (that is of mode commencing REF) must be denoted by a variable of appropriate primitive mode, or, in the case of a ref array or ref record variable, a structure denoting an appropriate actual array or record. This variable or structure need not be in the module being compiled, but subscripts occurring in it must be constants of form integer and no automatic dereferencing must be involved in determining its value. These restrictions ensure that the value can be determined at compile time without appeal to the contents of other initialised variables. Note that a form such as A2(4,6) is not allowed as an initial value since dereferencing (albeit implicit) is involved.

In the case of REF ARRAY BYTE variables a special form of initial value is allowed in addition to the normal structure denoting a specific array of bytes. This has the form of a 'string' and the array of bytes denoted by the string is located in a pool of strings and the REF ARRAY BYTE variable is initialised to the address of this string. See 4.4 for other uses of the string pool.

Default values for uninitialised reference variables are not defined.

Syntax:

refinitvalue ::= variable | structure | string

For 'variable' and 'structure' see 4.1.2.

Examples:

```
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 are illegal

```
REF LIST NEXTCELL := CELLA.TL
REF INT JJ := G(J)
REF ARRAY REAL AA := A2(2)
```

3.5.3 Arrays and records

In the case of an array or record the initial value is denoted by a list of the initial values of the components separated by commas and enclosed in brackets. 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 may be zero.) The initial value of a byte array can also be denoted by a string.

Examples:

```
    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 initialises each element to one of 9 (in this case distinct) entries in the string pool.

Multidimensional arrays and arrays of records may also be initialised and in this case the initial value is expressed as a list of lists.

Examples:

```
ARRAY (2,2)REAL UNIT := ((1.0,0.0),(0.0,1.0))
ARRAY (5) LIST CELLS := ((0,CELLA) (5))
```

3.5.4 Syntax:

The full syntax of initial values is as follows:

```
          initvalue ::= simpleinitvalue | refinitvalue |
                        recinitvalue | arrayinitvalue
  simpleinitvalue ::= sign number | identifier
     refinitvalue ::= variable | structure | string
     recinitvalue ::= ( initvalue [ , initvalue] ... )
   arrayinitvalue ::= ( [ arrayinititem [ , arrayinititem] ...] ) | string
    arrayinititem ::= initvalue [ ( integer ) ]
```

3.6 Procedure Declarations

Normal literal procedure declarations in which an identifier is associated with a piece of code are declared as follows:

The word PROC is 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 preceeded by ENT then it will be externally accessible. See 6.

Syntax:

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

For 'simplespec' see 3.4 and for 'sequence' see 5.

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

If the procedure declaration defines a function call then the mode of the result must be indicated.

Execution of the statements which form the body of the procedure is normally initiated as a consequence of calling the procedure either from a procedure statement (see 5.9) or as a function call (see 4.1.3).

Control may be transferred out of a procedure by

(i) obeying a generalised GOTO statement
(ii) calling a procedure which itself directly transfers control
(iii) obeying a RETURN statement
(iv) encountering the final ENDPROC which implies RETURN

Examples of 'procdec':

```
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

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

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

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

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

A procedure variable is a variable which can take as value (a pointer to) a particular 'literal procedure'. Thus in the above example the variable P will point to the particular piece of code which is handed over as parameter. In RTL/2 this sort of variable can be declared in a manner analogous to a variable of any other mode (see 3.2). It is necessary however to specify the type of parameters and result if any. This is done by a 'procdescriptor'.

Syntax:

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

Thus the type of the parameter is specified by a list of modes in brackets and the result type appended. 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':

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 the example above.

3.7 Stack Declarations

Normal literal stack declarations in which an identifier is associated with an area of store to be used as a stack are as follows. The word STACK is followed by the identifier of the stack and its length in machine dependent units. If the declaration is preceeded by ENT then it will be externally accessible. See 6.

Syntax:

stackdec ::= STACK identifier integer

Example:

```
STACK JOB 150
```

A stack variable is a variable which can take as value (a pointer to) a particular literal stack and can be declared in a similar manner to procedure variables (see 3.6). Stack variables are likely to be of most use as formal parameters of supervisor calls controlling multitasking. Corresponding actual parameters will usually be the identifiers of literal stacks which 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 6.

3.8 Label Declarations

Normal literal label declarations in which an identifier is associated with a particular statement in a procedure are described in 5.1.

A label variable is a variable which can take as value a level-address pair as described in 3.1. They may be declared in the usual way (see 3.2). The use of label variables is described in 5.4.

3.9 Scopes

All identifiers are in scope throughout a complete module except for

(i) Parameters and other variables local to a procedure
(ii) Literal labels in a procedure
(iii) Identifiers defining LET sequences.

The scope of LET identifiers is described in 1.11. In the other cases the scope is the block in which they are declared plus inner blocks unless the name is redefined therein; see 5.2.

3.10 Data Bricks

A data brick is a named collection of scalars, arrays and records. The word DATA is 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 then it will be externally accessible. See 6.

Syntax:

```
datadec ::= DATA identifier;
                declaration [ ; declaration] ... ENDDATA
declaration ::= [ simpledec | arraydec | recorddec]
```

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

Example of 'datadec':

```
DATA LOCAL;
        INT I,J,K;
        ARRAY (100) REAL A, B := (1,2,3);
        PERSON JOHN;
ENDDATA
```

4 Expressions

Expressions are rules for computing values and define the operations to be performed on the components of the expression. The meaning and value of an expression may depend upon the environment in which it is to be evaluated.

The environment of an expression will be determined by

(i) The mode of the destinations of an assignment statement.
(ii) The mode required in some statement parameter (eg, after TO).
(iii) The mode of operand required for a monadic or diadic operator.

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; firstly there are the familiar transfers between the various arithmetic modes and 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, for example, as normal variables as in REAL X, Y, and as reference variables as in REF 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 then the identifier is reconsidered as standing for its contents; this process can be repeated in the case of reference variables.

As an illustration we will consider the assignment statement in which the destinations dictate the mode required. (See 5.3).

X := 3.14	X is the name of a place which holds real values and so the right hand side must 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 we take the contents of Y. 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 := 3.14	This is illegal.

Note that dereferencing is also applied in array and record access in the case of a reference variable, see 3.3.1 and 3.4.

Occasionally it will be found that dereferencing must be forced on the left hand side. In this case the operator VAL can be used. See 5.3.

Automatic transfers between arithmetic modes are restricted to those in which no information is lost. Thus

BYTE ⟶ INT ⟶ REAL
FRAC ⟶ REAL.

These types of transfer are known as widening.

To obtain transfers in which information is lost the appropriate monadic operator (BYTE, INT or FRAC) must be applied explicitly. See 4.2.3. These types of transfer are known as narrowing. See 4.2.2 for the details of transfers between double and single forms of integer and fraction modes.

4.1 Expression Components

Expressions are built up from four kinds of component:

(i) Constants
(ii) Variables
(iii) Function calls
(iv) Conditional expressions

4.1.1 Constants

Constants denote literal values which remain unchanged throughout the execution of a program complex. Constants are either actual expressed numbers denoting fixed plain values or are identifiers denoting literal stacks, labels or procedures or are fixed reference values represented by appropriate variables, structures or strings.

Syntax:

```
constant ::= number | identifier | variable | structure | string
number ::= real | fraction | integer
```

The mode of a constant of form 'real' is REAL and of form 'fraction' is FRAC. The mode of a constant of form 'integer' is BYTE if its value lies in the range [0,255] and is otherwise INT. See 1.3 for these forms of 'number'. Non plain constants are fully described in 4.4.

Examples of 'constant':

```
3.75
'P'
"PIGLETS"
SIN
```

4.1.2 Variables and structures

Variables are single places whereas structures are complete arrays or records. They are denoted by similar syntactic forms. A variable may occur as the destination of an assignment statement whereas a structure may not.

Variables and structures are either simple or are array elements or record components (or both).

(i) A simple variable or simple structure is represented by an identifier. The modes of the value of a simple variable or the elements or components of a simple structure are defined by the declaration of the identifier.

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

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

Syntax:

```
          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
```

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:

```
XX
A(1)
RESTART
CELLA.HD
PEOPLE(27).NAME(3)
A(G(J))
```

Examples of 'variable' with dereferencing:

```
BB(J)
NEXTCELL.HD
WHO.AGE
CELLA.TL.TL.TL.HD
JIM.MOTHER.NAME(8)
PEOPLE(99).FATHER.SIBLINGS(1).SEX
```

Examples of 'structure':

```
G
CELLA
JIM.NAME
WHO.NAME
PEOPLE(K)
```

4.1.3 Function calls

A function call defines a single value which results through the application of statements defined by a procedure declaration to a set of parameters. The form of a function call is the same as for a procedure statement. It consists of the identifier of a literal procedure (or SVC procedure see 6) or the identifier of a procedure variable followed by a list of parameters separated by commas and enclosed in round brackets. The brackets must not be omitted even if there are no parameters. The allowed form of parameter in each position in the parameter list is an expression of mode determined by the corresponding procedure declaration. For further details see 5.9.

Syntax:

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

Examples:

```
LOG(X)
TIME ()
F(J,K,L-3)
EXP (EXP(EXP(X)))
```

Note also that one can have for example an array of procedure variables

```
ARRAY(10)PROC(REAL)INT QQ
```

and 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.

4.1.4 Conditional expressions

A conditional expression is a rule for choosing one of two or more expressions according to the values of specified conditions.

Syntax:

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

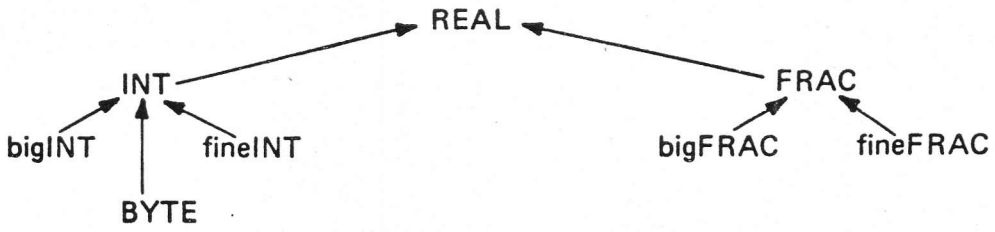
The value is determined as follows:

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

For 'condition' see 4.3.

All the alternative expressions in a conditional expression must be of the same mode or of modes which 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 will be that common mode entailing the least dereferencing. Note that in the case of procedure, ref array and ref record expressions all alternatives must have the same specifications, array modes or record modes respectively.

In the case of plain modes the situation is more complex and due regard must be paid to the double forms of integer and fraction modes. See 4.2.2. If all the alternative modes can be reduced to a common mode by dereferencing alone then the resultant mode will be that common mode entailing the least dereferencing. Otherwise all the alternatives will be dereferenced to plain modes and the resultant mode will be the lowest point on the tree shown below which can be reached by all these plain modes. There is, however, also the restriction that the resultant mode must not be of double form but will always be automatically converted to the corresponding single form.



Examples:

```
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.

4.2 Arithmetic Expressions

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

During the evaluation of arithmetic expressions an overflow condition will arise if the operands are such that the potential result lies outside the range of values which can be represented by the mode concerned. The behaviour of the program under these conditions is undefined but each implementation should provide a means of detecting an overflow condition.

4.2.1 Primaries

Expressions are built up from the primary components described in 4.1 and expressions in round brackets.

Syntax:

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

Examples of 'primary':

```
X
28.3
LOG (P + Q)
'0'
IF K = 0 THEN M ELSE N END
(I:/J + 3)
```

4.2.2 Integers and fractions

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. This section describes 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 occupy a single word (see 3.1) and the values contained in all fraction and integer variables are thus in normal form. The other two forms (double modes) occupy two words and so can only be taken by values arising as intermediate results in expressions and cannot be stored in variables. These double forms are the big form and the fine form.

Suppose that a word has length w + 1 and let $M = 2^w$; the forms can then take values as follows.

(i) Normal: A normal integer can take any integral value in the range

$$-M \leqslant \text{integer} < M$$

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

$$-1 \leqslant \text{fraction} < 1$$

(ii) Big: In the big form the normal part of the value occupies the less significant word and so the range of big values is M times that of normal values but with the same precision. A big integer can take any integral value in the range.

$$-M^2 \leqslant \text{big integer} < M^2$$

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

$$-M \leqslant \text{big fraction} < M$$

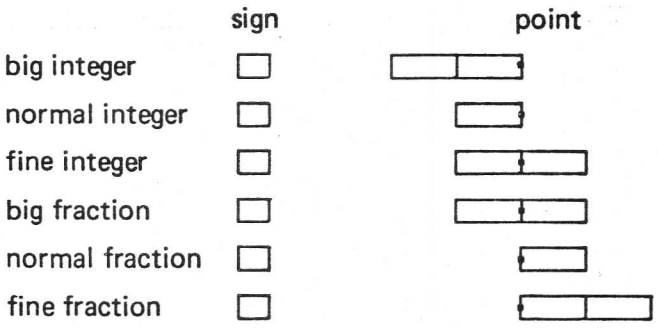
(iii) Fine: In the fine form the normal part of the value occupies the more significant word and so the range of fine values is the same as that of normal values but their precision is M times greater. A fine integer can take any value which is a multiple of 1/M in the range

$$-M \leqslant \text{fine integer} < M$$

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

$$-1 \leqslant \text{fine fraction} < 1$$

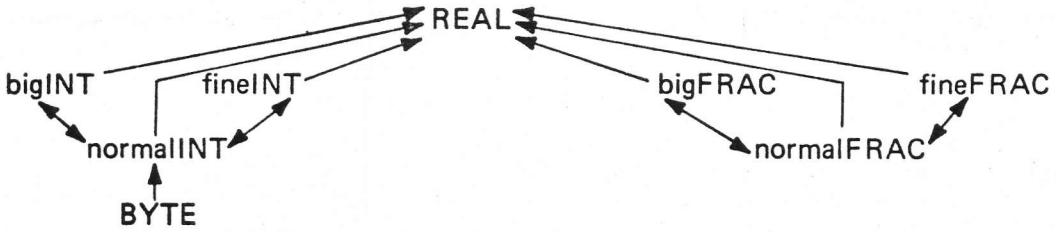
The six possible modes arising from the combinations of form and type are shown below with their points aligned.



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

It will 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 multiply operator in 4.2.4.

All operators except for arithmetic shifts require their operands to be in specific forms. Where these differ from normal form they are listed in the tables of operators in 4.2.3 and 4.2.4. Any arguments not in the required form will be converted according to the table below by the most direct route.



If a big value is converted to normal form then an overflow condition may arise.

If a fine value is converted to normal form then it will be rounded to the normal value nearest to the original fine value; if the original value lies midway between two normal values then the algebraically greater normal value will be taken. An overflow condition may arise.

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

4.2.3 Monadic operators

A term is a primary component optionally preceded by one or more monadic operators. These operators are applied from right to left.

Syntax:

```
        term ::= [monadicop] ...primary
   monadicop ::= + | - | ABS | NOT | REAL |
                 INT | FRAC | BYTE | LENGTH
```

The modes of the operand and result are as follows:

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

Notes:

+ This has no effect

– Negation; this changes the sign of the operand. If the operand is a byte value then it will first be converted to mode integer. An overflow condition will arise if the most negative value of an integer or fraction form is negated.

ABS Absolute value; this leaves unchanged a positive operand but negates a negative operand. See above for note on overflow.

NOT Logical not; this treats an integer value as a bit pattern and changes the value of each bit.

INT Converts a big fraction to a fine integer without changing its value. Converts 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 then the algebraically greater integer value will be taken. Thus –3.5 converts to –3 and 3.5 converts to 4. An overflow condition may arise.

FRAC — Converts a fine integer to a big fraction without changing its value. Converts 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 then the algebraically greater fraction value will be taken. An overflow condition may arise.

BYTE — Converts a real or integer value to a byte. If the operand is real then it is first converted to an integer value as for INT. The integer value is then 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.

REAL — Converts the operand to a real value. This operator may be useful in conditional expressions. See 4.1.4.

LENGTH — This applies to an array expression and returns the length of the array. If the array is multidimensional then the length is the range of the first subscript.

Examples of 'term':

```
–4
NOT (I NEV J)
–ABS X
LENGTH A
LENGTH A2(I)
```

Note that 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.

4.2.4 Diadic operators

An expression consists of one or more terms separated by diadic operators which are applied according to the precedence given below. Operations of the same precedence are applied from left to right. The order of the evaluation of operands is not defined.

Syntax:

```
    expn ::= term [ diadicop term ] ...
diadicop ::= SLL | SRL | SHL | SLA | SRA | SHA | * | :/ |
             // | / | MOD | LAND | LOR | NEV | + | –
```

The modes of the operands and result and the precedence are as follows

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 which have more than one entry in the above table each successive entry should be considered in turn. If the operands in a given instance are of the appropriate mode or can be converted to that mode then that entry is taken, otherwise the next entry is considered.

The first operand in an arithmetic shift must be either an integer or a fraction. The second operand must be a normal integer. The result is the same type as the first operand and its form is determined by the form of the first operand and the operator as given below.

First operand/Operator	SLA	SHA	SRA
big	big	big	big
normal	big	normal	fine
fine	fine	fine	fine

Notes:

SLL Shift left logical; the first operand is shifted left the number of places specified by the second. If the second operand is negative or greater than the number of bits in the integer representation, then the action is undefined.

SRL Shift right logical; the first operand is shifted right the number of places specified by the second. If the second operand is negative or greater than the number of bits in the integer representation, then the action is undefined.

SHL Shift logical; the first operand is shifted by the number of places specified by the second. If the second operand is positive then the behaviour is as SLL and otherwise as SRL.

SLA Shift left arithmetic; if the first operand is single length it is converted first to the big form. The double length operand is then 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 then the action is undefined. An overflow condition may arise.

SRA Shift right arithmetic; if the first operand is single length it is converted first to the fine form. The double length operand is then 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 then the action is undefined. The sign bit is propagated.

SHA — Shift arithmetic; the first operand is shifted without altering its type or form. If the second operand is positive then a left shift is performed, if negative a right shift is performed. The number of places shifted is the absolute value of the second operand and the action is undefined if this exceeds the number of bits in the integer representation. A left shift may give rise to an overflow condition. A right shift will propagate the sign bit.

* — Multiply; this is either a real operation on real operands or a fixed point operation on integer and fraction operands. In the former case an overflow condition may arise in the usual way whereas in the latter case it will only arise if both operands have their most negative value. (Any subsequent conversion to normal form may itself give rise to an overflow condition).

Note the 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 always produces an integer result with truncation towards zero. An overflow condition may arise.

// — Fraction division; this always produces a fraction result with truncation towards zero. An overflow condition may arise.

/ — Real division; an overflow condition may arise.

MOD — Modulo; the result is the remainder on dividing the first operand by the second. The sign of the result is the same as that of the first operand. An overflow condition may arise. Note that J MOD K has the same value as J — J :/ K * K.

LAND — Logical and; this treats both operands as bit patterns. A bit in the result will be 1 only if the corresponding bits in both operands are 1.

LOR — Logical inclusive or; this treats both operands as bit patterns. A bit in the result will be 0 only if corresponding bits in both operands are 0.

NEV — Not equivalent; this treats both operands as bit patterns. A bit in the result will be 1 only if the corresponding bits in the operands differ from each other.

+,— — Addition, subtraction; this is either a real operation on real operands or a fixed point operation on single length integer or fraction operands. Note that integer ± fraction converts to real. These operations may give rise to an overflow condition.

Examples of 'expn':

```
X + Y
J SLL 3
X/Y/—Z
K MOD L
J LAND OCT 77 NEV K
P * Q / / FRAC X
```

4.3 Conditions

A condition is made up of comparisons connected by the words OR and AND of which AND is the more tightly binding.

Syntax:

```
    condition ::= subcondition [ OR subcondition] ...
  subcondition ::= comparison [ AND comparison] ...
   comparison ::= expn comparator expn
   comparator ::= = | # | < | <= | > | >= | :=: | :#:
```

The comparators =, # operate on the irreducible level of the seven primitive modes. The comparators <, <=, >, >= operate on the three plain modes REAL, FRAC and INT. The comparators :=:, :#: operate on the nine REF modes (ie, they compare addresses).

In all cases dereferencing will be applied to the operands where necessary to extract a value of appropriate mode. In the case of the comparisons of plain modes, mode conversion (widening) will occur as for subtraction: Note carefully that the operators <, <=, >, >= do not apply to bytes; such operands will first be widened to mode integer.

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

Note that the characters $ and £ are alternatives to # both alone and in :#:

Examples of 'condition':

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

4.4 Non-Arithmetic Expressions

Expressions of modes other than plain modes are of the same syntactic form as described above in 4.2. However, the absence of operators which deliver non-plain modes means that in practice the syntax can be simplified as follows:

Syntax:

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

Non-plain constants are as follows:

Procedure and stack constants are represented by the identifiers of literal procedures or stacks which could possibly be external to the module (but not SVC procedures; see 6).

Label constants are represented by the identifiers of literal labels. If a label constant occurs in an expression then the label value is obtained by taking as level the current stack pointer which identifies the execution of the current procedure and as address the address of the label in the code. (See 3.1).

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.

In the particular case of mode REF ARRAY BYTE a string may also be used. Storage for the array of bytes denoted by the string is located in a pool of strings

and the value represented by the string is the address of the string in the pool. Note that the compiler may choose to share the storage for identical strings; they should be treated as read-only. See 3.5.2 for other uses of the string pool.

Variables, function calls and conditional expressions are just as for arithmetic expressions. Thus for example one can have ref array conditional expressions; these can be useful as parameters of procedure calls such as

```
TWRT (IF J = 0 THEN JIM.NAME ELSE "NOBODY" END)
```

Examples of 'npexpn':

```
RESTART
JOHN
WHO.NAME
"FRED"
IF AA :#: BB THEN JIM.MOTHER ELSE JANE END
```

4.5 Byte Arithmetic

This section summarises the various distinctions between integers and bytes which on a first reading might not be clear.

The modes BYTE and INT are quite distinct and variables of these modes can only hold appropriate values. 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) will be interpreted according to its context. In an expression (dynamically evaluated) the mode is BYTE if the value lies in [0,255] and INT otherwise. As a repetition factor in an array initialisation, the length of an array or the length of a stack, the question of mode does not arise; it is merely a value to be heeded by the compiler. As an initial value for a BYTE or INT variable (and consequently as a value in # sequences in strings) the mode is determined by that of the variable being initialised.

The only effective operators defined on BYTE values are LAND, LOR, NEV, = and #. Care should be taken when using operators such as +, −, >, < (which take INT operands) that unnecessary run time widening of values does not take place.

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

```
M := 7
```

is valid whereas, because + is not defined between bytes,

```
M := M + 1
```

is not; to increment M one must write

```
M := BYTE (M + 1)
```

5 Statements

Statements define the actual operations to be performed. The statements are obeyed sequentially unless stated otherwise. All statements may be labelled. A series of statements separated from each other by semicolons is known as a sequence.

The possible forms of statement are

(i) Block
(ii) Assignment statement
(iii) Goto statement
(iv) Switch statement
(v) If statement
(vi) For and to statements
(vii) While statement
(viii) Procedure statement
(ix) Return statement
(x) Dummy statement
(xi) Code statement.

Syntax:

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

5.1 Labels

A label has the form of an identifier. A statement is labelled by preceding it by a label followed by a colon. A statement may have many labels.

Syntax:

```
labels ::= [ identifier :] ..
```

For the scope of a label see 3.9.

5.2 Blocks

A block introduces a new level of nomenclature and access for local variables. A block consists of the word BLOCK followed by declarations of simple variables, statements and the word ENDBLOCK.

Syntax:

```
     block ::= BLOCK blockbody ENDBLOCK
 blockbody ::= [simpledec ;] ... sequence
```

Note that since a block is a statement the nesting of blocks can 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 controlled variable is considered to be declared in that block.

The scope of a local variable is the block in which it is declared including any inner blocks unless the identifier is redeclared therein. Note however that in the declarations in a block, the contents of a local variable must not be used in an initial value until after the declaration of the variable itself. Thus the following is illegal.

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

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

5.3 Assignment Statements

An assignment statement consists of a list of destinations followed by := (the left part) followed by an expression.

The destinations each consist of a variable (possibly preceded by VAL to indicate dereferencing) and all the destinations must be of the same mode. The expression is evaluated and converted to that mode and assigned to each destination in turn. Note that the expression on the right is evaluated before the destinations on the left are evaluated and the evaluation of the destinations and assignment to them occurs from right to left. For mode conversions see 4.

Syntax:

```
assignmentst ::= leftpartlist expn
  leftpartlist ::= destination := [ destination := ] ...
 destination ::= variable | VAL variable
```

Examples of 'assignmentst':

```
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 the 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 must denote a single location and not an actual array or record. Thus

```
U := V
```

and

```
JIM.NAME := "JIM"
```

are illegal.

5.4 Goto Statements

A goto statement causes an explicit transfer of control.

It consists of the keyword GOTO followed by a label expression.

Syntax:

```
gotost ::= GOTO expn
```

Examples of 'gotost':

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

The normal case of a label expression will be a literal label, when the effect of the goto statement will be 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 in fact formally a switch; it is a jump to one of an array of labels. See 5.5 for the switch statement.

A goto statement may not lead into a block (this includes the body of a for statement).

Jumps other than to local literal labels could be performed as follows.

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 which was current when the label value was created. If the two levels are equal then 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 then 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 was not in scope and an unrecoverable error will occur. See 7.

In short, control will be 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 will be terminated.

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

Example:

```
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;
```

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 turn 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 perhaps many calls of ONE and TWO: the mechanism performs no work until recovery is necessary.

5.5 Switch Statements

A switch statement transfers control to one of a number of literal labels according to the value of an expression of mode integer.

Syntax:

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

Note carefully that the identifiers must 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 then control is not transferred but passes to the next statement. This provides a convenient and efficient means of monitoring an illegal value of the integer expression.

Example:

```
SWITCH K OF P1,P2,P3,P4,P5;
FAIL ("K OUT OF RANGE")
```

Note that a switch statement is a statement and so can be labelled and hence jumped to in the usual way.

5.6 Conditional Statements

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

Syntax:

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

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

There are no restrictions on the statements in the sequences. They may be labelled and control passed to them directly by a goto or switch statement. If control is transferred in this way then the behaviour after the execution of the sequence will be as if control had entered via the conditions.

Examples of 'ifst':

```
IF X = 0 THEN P := Q END

IF Y > 1 THEN GOTO STOP END

IF Y = Z AND J = K THEN
        I := I + 1; L := K := INT U.RL;
        F(I, J, K);
END

IF X = 0 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 THEN
        XX := ZZ; J := L;
END
```

Note that 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.

5.7 For and To Statements

A for statement causes a sequence of statements to be repeatedly executed zero or more times and in addition performs a series of assignments to a control variable.

Syntax:

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

Examples of 'forst':

```
FOR I := 1 TO 10 DO A(I) := 0 REP

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
```

Notes:

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

(ii) If the 'BY expn' is omitted then an increment of 1 is assumed.

(iii) The controlled sequence behaves as a block (hence declarations are allowed) and the controlled variable is considered to be declared to be of mode INT in that block. Thus on exit from the for statement the controlled variable is inaccessible.

(iv) The control variable is read-only. Thus it cannot occur explicitly as the destination of an assignment statement nor can its address be evaluated (eg, handed over to a REF parameter of a procedure) since an indirect assignment might then occur.

(v) A goto statement may not lead into the blockbody controlled by a for statement.

(vi) 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, name :=e1;
lab: IF inc > 0 AND name <= limit
        OR inc < 0 AND name >= limit THEN
        seq;
        name := name + inc; GOTO lab;
        END;
ENDBLOCK
```

If the increment is zero then the behaviour is not defined.

(vii) An abbreviated form of for statement is allowed when the control variable is not used. In this case the statement merely takes the form

```
TO expn DO blockbody REP
```

and the body is executed 'expn' times. Note that if the 'expn' is non positive then the body will not be executed at all. Note also that the body still behaves as a block.

5.8 While Statements

A while statement causes a sequence of statements to be repeatedly executed while a specified condition remains true.

Syntax:

whilest ::= WHILE condition DO sequence REP

The statement

WHILE e DO seq REP

is essentially equivalent to

lab : IF e THEN seq ; GOTO lab END

5.9 Procedure Statements

A procedure statement invokes 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 is the same as that of a function call. (See 4.1.3). It consists of the identifier of a literal procedure (or SVC procedure see 6) or the identifier of a procedure variable followed by a list of parameters separated by commas and enclosed in round brackets. The brackets must not be omitted even if there are no parameters.

Syntax:

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

The correspondence between the parameters is 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 behave just as if they were variables declared in the block which is the body of the procedure. The parameters are evaluated from left to right before the procedure itself is evaluated.

Note that a function call may be used as a procedure statement for its side effects; the result will be discarded.

Example:

Consider the procedure:

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

Then the statement

ACTION (P, Q * R, 1, "NOGO" , FAILURE)

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. In fact 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
```

5.10 Return Statements

A return statement may be used to leave a procedure and return control to the calling procedure in the usual way. In the case of a procedure which is not a function procedure the final ENDPROC will behave as if preceded by a return statement. In the case of a function procedure, control must be explicitly returned and so the final ENDPROC must be preceded by a return statement or a goto statement.

Syntax:

returnst ::= RETURN | RETURN (expn)

The 'expn' is only present in the case of a function procedure and is evaluated to give the result.

5.11 Dummy Statements

The dummy statement is simply a void whose execution has no effect. It is formally necessary in order to allow a label to be placed before keywords such as END, REP etc.

Syntax:

dummyst ::= [ ]

For example there is a dummy statement after the colon in

```
; LAB : REP
```

5.12 Code Statements

See 1.8.

6 Modules

The unit of compilation is the module. It consists of

(i) environment descriptions

(ii) record mode definitions

(iii) titles and options

(iv) LET definitions

(v) the bricks to be compiled

An environment description describes

(i) the format of another separately compiled brick

(ii) the format of an available supervisor call

(iii) it may also (redundantly) describe a brick to be compiled in the present module.

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

Syntax:

```
        module ::= moduleitem [ ; moduleitem ] ... eom
           eom ::= end-of-module-character
    moduleitem ::= [ environdescn | recmodedec | letdefinition |
                     title | option | brick ]
  environdescn ::= EXT stackdescn | EXT procdescn | EXT datadescn |
                   SVC procdescn | SVC datadescn
     stackdescn ::= STACK idlist
      procdescn ::= PROC procdescriptor idlist
      datadescn ::= datadec-without-initial-values
         brick ::= [ ENT ] datadec | [ ENT ] procdec | [ ENT ] stackdec
```

Example of 'module':

```
OPTION (1) BC;

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;
```

The external descriptions inform the compiler of the characteristics of bricks which are external to the module so that reference to these bricks may be made in the module. The stack description merely lists the identifiers of stacks. The data description is effectively the same as the declaration of the data brick except that initial values cannot be set. The procedure description describes the parameters and results in the same style as for procedure variables.

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

7 Integrity

The language presented in this manual is the full RTL/2 language. This full language is quite 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 illegal activities. If the language is constrained so that illegal references cannot be manipulated then much system programming will 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 is seen as comprising two languages with the full language being the system language and a secure subset being the application language.

The application language is the full language with the following restrictions.

(i) All variables of modes other than plain modes and LABEL must be initialised.

(ii) The addresses of local variables must not be assigned to data brick reference variables, returned as the result of a procedure, or manipulated in any other way which might result in their passing out of scope.

(iii) Code statements are forbidden.

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

In the system language

(i) a stack check on procedure entry is optional

(ii) array bound checks are optional

(iii) monitoring of general goto statements is optional

In the application language

(i) a stack check on procedure entry is mandatory

(ii) array bound checks are optional for fetching from plain arrays and otherwise mandatory

(iii) monitoring of general goto statements is mandatory

For an outline of a recommended mechanism for handling the errors which arise when these and other checks fail, see Appendix 2.

Appendix 1

Standard Input-Output

The RTL/2 language as such 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 standard package for character streaming has been defined. This is briefly outlined below and described in detail in the manuals 'RTL/2 System standards' and 'RTL/2 Standard stream I/O' which should be consulted for details of formats, errors, etc.

A1.1 Streaming Mechanism

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

```
DATA RRSIO;
  PROC()BYTE IN;
  PROC(BYTE) OUT;
ENDDATA;
```

and

```
DATA RRSED;
  BYTE TERMCH;
  BYTE IOFLAG;
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.

A1.2 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.

A1.3 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.

Appendix 2

Standard Error Recovery

In order to aid transportability of programs between systems a recommended standard mechanism for error recovery has been defined. This is outlined below and described in detail in the manual 'RTL/2 System standards'.

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. (Examples include 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 the unrecoverable error number and ERP 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 ERL. 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 wishes 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 3

RTL/2 Language Subset of ISO7

Character	Decimal value	Language use	Ref
HT	9	layout – horizontal tab	1.1
LF	10	layout – newline	1.1
SP	32	layout – space	1.1
"	34	string quote	1.4
# £ $	35, 36, 92	not equals, strings	1.4, 4.3
%	37	comments	1.5
&	38	not used	
'	39	byte quote	1.3
(	40	open bracket	
)	41	close bracket	
*	42	multiply	4.2.3
+	43	add	4.2
,	44	comma	
–	45	minus	4.2
.	46	constants, records	1.3, 4.1.2
/	47	divide	4.2.3
0–9	48–57	numbers	1.3
:	58	labels, assignment etc.	5.1, 5.3, 4.3
;	59	statement, declaration separator	
<	60	less than	4.3
=	61	assignment, equals	5.3, 4.3
>	62	greater than	4.3
?	63	not used	
@	64	not used	
A–Z	65–90	names, numbers	1.2, 1.3

Notes:

(i) The characters &, ? and @ are not used for any particular purpose in the language but they may occur in strings, comments and titles.

(ii) Because of lack of uniformity in manufacturers' treatment of #, £ and $ it should be made clear that they are considered interchangeable and all mean the same thing. The intention is that on any preparation equipment the key marked # may be used with confidence. Note that no confusion can arise as to the internal value as far as representing these characters in strings is concerned because they cannot stand for themselves.

Appendix 4

Keywords

Appendix 5

Syntax Rules

amode	::= simplemode | recmode	3.3.2
arraydec	::= ARRAY (length [, length] ...) amode initidlist	3.3.2
array	::= identifier | recordcomponent	4.1.2
arrayelement	::= variable (subscriptlist) | array (subscriptlist)	4.1.2
arrayinititem	::= initvalue [(integer)]	3.5.4
arrayinitvalue	::= ([arrayinititem [, arrayinititem] ...]) | string	3.5.4
arraymode	::= ARRAY [([,] ...)] amode	3.3.2
arrayspec	::= ARRAY (length [, length] ...) simplemode idlist	3.4
assignmentst	::= leftpartlist expn	5.3
bindigit	::= 0 | 1	1.3
bindigitlist	::= bindigit...	1.3
block	::= BLOCK blockbody ENDBLOCK	5.2
blockbody	::= [simpledec ;] ... sequence	3.6
brick	::= [ENT] datadec | [ENT] procdec | [ENT] stackdec	6
codeheading	::= CODE digitlist, digitlist;	1.8
codeitem	::= ISO7-character-other-than-trip1-or-trip2 | trip1 letitem | trip2 name	1.8
codest	::= codeheading codeitem...	1.8
comment	::= % sequence-of-characters -excluding-%-and-newline %	1.5
comparator	::= = | # | < | <= | > | >= | :=: | :#:	4.3
comparison	::= expn comparator expn	4.3
condexpn	::= IF condition THEN expn [ELSEIF condition THEN expn] ... ELSE expn END	4.1.4
condition	::= subcondition [OR subcondition] ...	4.3
constant	::= number | identifier | variable | structure | string	4.1.1
datadec	::= DATA identifier ; declaration [; declaration] ... ENDDATA	3.10
datadescn	::= datadec-without-initial-values	6
declaration	::= [simpledec | arraydec | recorddec]	3.10
destination	::= variable | VAL variable	5.3
diadicop	::= SLL | SRL | SHL | SLA | SRA | SHA | * | / | // | :/ | MOD | + | – | NEV | LAND | LOR	4.2.4
digit	::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9	1.2
digitlist	::= digit...	1.3
dummyst	::= [ ]	5.11
environdescn	::= EXT stackdescn | EXT datadescn | EXT procdescn | SVC procdescn | SVC datadescn	6
eom	::= end-of-module-character	6
exponent	::= E sign digitlist	1.3
expn	::= term [diadicop term] ...	4.2.4

forst	::= [FOR identifier := expn [BY expn]] TO expn DO blockbody REP	5.7
fraction	::= real B sign digitlist	1.3
functioncall	::= variable paralist | identifier paralist	4.1.3
gotost	::= GOTO expn	5.4
hexdigit	::= digit | A | B | C | D | E | F	1.3
hexdigitlist	::= hexdigit...	1.3
identifier	::= name	1.2
idlist	::= identifier [, identifier] ...	3.4
ifst	::= IF condition THEN sequence [ELSEIF condition THEN sequence] ... [ELSE sequence] END	5.6
initidlist	::= inititem [, inititem] ...	3.2
inititem	::= identifier [:=[identifier :=] ... initvalue]	3.2
initvalue	::= simpleinitvalue | refinitvalue | recinitvalue | arrayinitvalue	3.5.4
integer	::= digitlist | BIN bindigitlist | OCT octdigitlist | HEX hexidigitlist | 'stringchar'	1.3
item	::= name | number | string | comment | title | option | separator	1.1
labellist	::= identifier [, identifier] ...	5.5
labels	::= [identifier :] ...	5.1
leftpartlist	::= destination := [destination :=] ...	5.3
length	::= integer	3.3.2
letdefinition	::= LET name = [letitem] ...;	1.11
letitem	::= name | number | string | comment | separator	1.11
letter	::= A | B | C | D ..... X | Y | Z	1.2
module	::= moduleitem [; moduleitem] ... eom	6
moduleitem	::= [environdescn | recmodedef | letdefinition | title | option | brick]	6
monadicop	::= + | − | ABS | NOT | REAL | INT | FRAC | BYTE | LENGTH	4.2.3
name	::= letter [letter | digit] ...	1.2
number	::= real | integer | fraction	1.3
octdigit	::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7	1.3
octdigitlist	::= octdigit...	1.3
opchar	::= letter | digit	1.7
opitem	::= opchar...	1.7
option	::= OPTION (digitlist) [opitem [, opitem] ...]	1.7
paradescription	::= [pspec [, pspec] ...]	3.6
paralist	::= ([expn [, expn] ...])	4.1.3
plainmode	::= REAL | INT | FRAC | BYTE	3.2
primary	::= constant | variable | functioncall | condexpn | (expn)	4.2.1

primmode	::= plainmode | progmode	3.2
procdec	::= PROC identifier (paradescription) resultmode ; blockbody ENDPROC	3.6
procdescn	::= PROC procdescriptor idlist	6
procdescriptor	::= ([simplemode [, simplemode] ...]) resultmode	3.6
procst	::= variable paralist | identifier paralist	5.9
progmode	::= LABEL | STACK | PROC procdescriptor	3.2
pspec	::= simplespec	3.6
real	::= digitlist . digitlist [exponent] | digitlist exponent	1.3
recinitvalue	::= (initvalue [,initvalue] ...)	3.5.4
recmode	::= recmodeident	3.4
recmodedef	::= MODE recmodeident (rspec [, rspec] ...)	3.4
recmodeident	::= identifier	3.4
record	::= identifier | arrayelement	4.1.2
recordcomponent	::= variable . selector | record . selector	4.1.2
recorddec	::= recmode initidlist	3.4
refinitvalue	::= variable | structure | string	3.5.2
resultmode	::= [simplemode]	3.6
returnst	::= RETURN | RETURN (expn)	5.10
rspec	::= simplespec | arrayspec	3.4
selector	::= identifier	4.1.2
separator	::= see-section-1.9	1.9
sequence	::= statement [; statement] ...	5
sign	::= [+ | –]	1.3
simpledec	::= simplemode initidlist	3.2
simpleinitvalue	::= sign number | identifier	3.5.1
simplemode	::= [REF] primmode | REF arraymode | REF recmode	3.2
simplespec	::= simplemode idlist	3.4
stackdec	::= STACK identifier integer	3.7
stackdescn	::= STACK idlist	6
statement	::= labels unlabelledst	5
string	::= stringpart...	4.1.2
stringpart	::= " [stringchar... | stringinsert] ... "	1.4
stringchar	::= see-section-1.3	1.3
stringinsert	::= # [stringitem [, stringitem] ...] #	1.4
stringitem	::= integer [(integer)]	1.4
structure	::= array | record	4.1.2
subcondition	::= comparison [AND comparision] ...	4.3
subscriptlist	::= expn [, expn] ...	4.1.2
switchst	::= SWITCH expn OF labellist	5.5
term	::= [monadicop] ... primary	4.2.3
title	::= TITLE sequence-of-characters-excluding-semicolons	1.6
unlabelledst	::= block | assignment | gotost | switchst | ifst | forst | whilest | procst | returnst | dummyst | codest	5

variable	::= identifier | arrayelement | recordcomponent	4.1.2
whilest	::= WHILE condition DO sequence REP	5.8

The class names 'npconstant', 'npexpn' and 'vectordec' have been ommitted from the above list because they are merely subclasses of 'constant', 'expn' and 'arraydec', respectively; these classes are introduced in the body of this manual only as an explanatory aid.