ASM

This keyword is used in the transputer implementations of occam to introduce, on following indented lines, sequences of target processor instructions in a simple assembly language. Similar extensions designed for other target processor types should allow reference to names declared in the surrounding occam scope, and should follow all the usual occam conventions concerning the indentation of source text lines.

A.3 Target word size

Target processors for which occam programs will be compiled differ in such matters as word length and the addressing of bytes within words. Documentation accompanying an implementation must fully define the mapping of data types onto the words in the target processor's memory, and specify any requirement for alignment of values with respect to word boundaries and/or need for padding bytes within records or arrays.

In order to optimise computations on integers, the occam programmer is given a choice between explicit specification of the length of all integer variables as **INT16**, **INT32** or **INT64** which will maximise portability of code between targets, or declaring all integers as **INT** which will allow the compiler to choose the length of integer which is natural in the target.

There are, however, contexts in occam where type **INT** is always assumed. These include the values returned by timers, the sizes of arrays and the bases and counts of replicators and segments. Programmers must be aware that if the target word length is only 16 bits, then there will be classes of program that they might reasonably wish to write whose portability is compromised by the limited size of integers.

Particular care is necessary when designing code that is to be distributed across a network of processors, not all having the same word length. The protocols of channels connecting dissimilar processors should not include components of type **INT**.

An occam compiler is required to perform any arithmetic on constants that are known at compile time using the word length of the target processor. Any such computations that would overflow on the target will produce a compile time error message.

A.4 Endianness

The word *endianness* has been adopted by computer engineers to define the relationship between the addressing of bytes within words and the positional significance of these bytes in the representation of integer values.

A processor architecture may be *little-endian* implying that bytes with lower addresses have least significance in integers, or *big-endian* implying that bytes with lower addresses have most significance.

An occam program which does not include any of the keywords **ASM**, **RETYPES**, **BYTESIN**, **OFFSETOF**, or **PACKED** is capable of compilation to target code whose behaviour will be identical whether the processor is little-endian or big-endian.

If an occam program is to be ported from a little-endian target processor, such as a transputer, to a big-endian one, then it will be necessary to inspect all uses of these keywords and either to confirm that the bit patterns mapped will be treated identically by the two implementations, or to make the necessary adjustments to the code for the new target processor.

B Configuration

This appendix describes the aspects of occam which specify the *configuration* of an occam program. Configuration associates the components of an occam program with a set of physical resources. During configuration the processes which make up an occam program are distributed over the number of interconnected processing devices available in the environment in which the program will execute. The processes which execute on a single processor may be given a priority of execution, and the channels which interconnect the distributed processes may be mapped onto the physical communication links between processing devices. Configuration does not affect the logical behaviour of a program.

An implementation may extend the syntax of configuration to take advantage of the features of particular target computer and network architectures. Readers are advised to consult documentation accompanying a particular implementation for the details of such extensions, which may introduce additional reserved keywords to the language. A single processor implementation may omit these language features altogether. An implementation may choose to arrange for the configuration of programs by means outside the language.

B.1 Execution on multiple processors

The component processes of a parallel may each be executed on an individual processor. This can be specified by a *placed parallel* which assigns a process for execution on a specified processor. Consider the following example:

```
PLACED PAR
  PROCESSOR 1
    terminal (term.in, term.out)
  PROCESSOR 2
    editor (term.in, term.out, files.in, files.out)
  PROCESSOR 3
    network (files.in, files.out)
```

In this example, the processes **terminal**, **editor** and **network**, are placed on three individual processors numbered **1**, **2** and **3**. Each process is executed on the assigned processor; each process uses local memory, and communicates with the other processes via channels.

The syntax for a placed par is:

placedpar	=	**PLACED PAR** { *placedpar* }
	|	**PLACED PAR** *replicator* *placedpar*
	|	**PROCESSOR** *expression* *process*
parallel	=	*placedpar*

The keywords **PLACED PAR** are followed by zero or more processor allocations. A processor allocation is the keyword **PROCESSOR**, and an expression of type **INT** which serves to identify the processor on which the associated process is to be placed. As for normal parallels (page 16), the placed parallel may be replicated. An implementation may extend this syntax to identify the type of processor on which the process is placed. All variables and timers used within the placement must be declared within it.

B.2 Execution priority on a single processor

B.2.1 Priority parallel

The component processes of a parallel (page 15) executing on a single processor may be assigned a priority of execution. Consider the following example:

```
PRI PAR
  terminal (term.in, term.out)
  editor   (term.in, term.out)
```

This process will always execute the process **terminal** in preference to the process **editor**. Each process executes at a separate priority, the first process is the highest priority, the last is the lowest. Lower priority processes may only continue when all higher priority processes are unable to. The process may also be replicated, as shown in the following example:

```
PRI PAR i = 0 FOR 8
  users (term.in[i], term.out[i])
```

The process with the highest index is executed at the lowest priority.

The syntax for priority execution is:

parallel	=	**PRI PAR** { *process* }
	|	**PRI PAR** *replicator* *process*

The keywords **PRI PAR** are followed by zero or more processes at an indentation of two spaces. As for parallels detailed in the main body of the manual (page 16), the process may be replicated.

An implementation for a target processor without hardware support for process priority may omit this extension or may implement it as though the keyword **PRI** were not present. If hardware only implements a limited number of priorities an implementation may impose restrictions on the nesting of **PRI PAR** constructs or may not permit replicated **PRI PAR**s.

B.2.2 Priority alternation

Priority alternations have been discussed with other alternations above (page 23). They are not concerned with process priority and are not dependent on hardware support for priority.

B.3 Allocation to memory

This section explains how a *variable*, *channel*, *timer* or *array* may be placed at an absolute location in the memory of a target processor. occam presents a consistent view of a processor's memory map. Memory is considered to be an array of type **INT**; each address in memory is considered a subscript into that array. The location used an *allocation* refers to the lowest address occupied by the variable, etc, in memory. It is therefore the location of first byte of a scakar or the first byte of the zeroth component of an array or the first field defined in a packed record type.

Consider the following example:

```
PLACE term.in AT link1in :
```

This allocation places **term.in** at the location specified by **link1in**. Here are some further examples:

```
[80]INT buffer :
PLACE buffer AT #0400 :

[5]REAL32 points :
PLACE points AT #0800 :

CHAN OF INT term.out :
PLACE term.out AT 3 :
```

The syntax for allocation is:

process	=	*allocation* *process*
allocation	=	**PLACE** *name* **AT** *expression* **:**

An allocation begins with the keyword **PLACE**, followed by the name of the variable, channel, timer or array to be placed. This in turn is followed by an expression of type **INT** which indicates the absolute location in memory.

Although not explicitly determined by the syntax, it is usual for an implementation to require that an allocation appear immediately after the declaration of the variable whose place it defines. A compiler may reject an allocation which appears at a position violating this guideline, especially after code may have been generated referring to the variable.

An allocation must place a channel, timer or variable at a compatible location. That is, a timer should be placed at a location which acts as a timer, and a channel should be placed at the location which implements a channel. Also, arrays must not be placed so that the components of an array overlap other allocations.

An implementation may extend the concept of allocation to give the user further control over the placement of variables in a way meaningful only in that particular implementation.

C Ports

This appendix describes how memory mapped devices may be addressed in occam. A process may communicate with external devices which are mapped into the processor's memory map, using a special input or output in a way similar to communication on channels. A special type declares a *port* which must then be placed using an allocation (page 94). Consider the following example:

```
PORT OF INT16 status :
PLACE status AT uart.status :
SEQ
  ...
  status ? state
  status ! reset
  ...
```

This example declares a port which is then allocated to a location **uart.status** in memory. The following sequence includes an input which reads the value of the port, and also an output which writes a value **reset** to the port location. Consider the following examples of port declarations:

PORT OF [8]INT uart :	one port of type **[8]INT**
[8]PORT OF BYTE transducer :	eight ports of byte type

A port declaration is similar to a channel declaration, and must obey the same rules of scope (page 39); that is, a port may not be used for input or output in more than one component process in a parallel. It is, however, possible to use a port for both input and output in sequence, as in the example above.

The syntax for ports and the necessary extensions to other syntax are:

port.type	=	**PORT OF** *data.type*
	|	**[** *expression* **]** *port.type*
declaration	=	*port.type* $\{_1$, *name* $\}$ **:**
port	=	*name*
	|	*port* **[** *expression* **]**
	|	**[** *port* **FROM** *base* **FOR** *count* **]**
	|	**[** *port* **FROM** *base* **]**
	|	**[** *port* **FOR** *count* **]**
input	=	*port* **?** *variable*
output	=	*port* **!** *expression*

A port is declared in the same way as a channel. Instead of a defined *protocol* (page 47) the port definition specifies a data type as the type for communication. A declaration of a port must be followed by an allocation (see page 94) defining its address in the memory of the target processor.

Although port input and output are treated syntactically as communications, in practice they behave more like assignments from and to variables mapped onto the processor's address space. They do not cause a process to wait for input or output.

Ports may be abbreviated, and therefore passed as parameters to procedures. The necessary syntax extensions are:

specifier	=	*port.type*
abbreviation	=	*name* **IS** *port* **:**
	|	*specifier name* **IS** *port* **:**
actual	=	*port*

A specifier used in a port abbreviation must be a port type.

Ports may be retyped and arrays of ports reshaped. If a port is retyped the types on the left and right must be of the same size. An array of ports may be retyped from or to a scalar port or another array of the same total size. The extensions to the syntax are:

definition	=	*specifier name* **RETYPES** *port* **:**
	|	*specifier name* **RESHAPES** *port* **:**

The specifier in a port retyping or reshaping definition must be a port type. A specifier which is an array port type with an empty dimension is allowed only in a retyping or reshaping definition.

The following code fragment illustrates port retyping and reshaping by example:

```
PORT OF INT32 pi :
PLACE pi AT #2000 :              -- at word #2000 of memory map
[4]PORT OF BYTE p4b RETYPES pi :
SEQ
  pba IS p4b[1] :
  pba ! 'z'
  [2]PORT OF INT16 pia16 RETYPES p4b :
  pia16[0] ! 32767
  PORT OF [4]BYTE pir RETYPES p4b :
  pir ! "dead"
  [][2]PORT OF BYTE pirs RESHAPES p4b :
  pirs[0][1] ! 'g'
```

D Rounding errors

Earlier sections of this manual have discussed rounding and the possibility of rounding errors. These occur because the types **REAL32** and **REAL64** only contain a subset of the real numbers. This is because it is not possible to store all the possible real values in the format for real numbers available on a machine. Rounding takes a value, which is considered infinitely precise and, if necessary, modifies it to a value which is representable by the type. By default, values are rounded to the nearest value of the type; if the nearest greater value and the nearest smaller value are equally near, then the result which has the least significant bit zero is chosen. Other modes of rounding are selectable using the **IEEEOPnn** library routines. These modes round values toward plus infinity, minus infinity or toward zero. A value rounded to plus infinity is the value nearest to and not less than the value to be represented; a value rounded to minus infinity is the value nearest to and not greater than the value to be represented; a value rounded toward zero is the value no greater in magnitude than the value to be represented.

A value is rounded to the precision of its type. A value of type **REAL32** is equivalent to IEEE single precision, and a value of type **REAL64** is equivalent to IEEE double precision.

Values in the **REAL32** and **REAL64** formats are stored in the following formats

s	*exp*	*frac*

where *s* is the sign bit, *exp* is the exponent and *frac* is the fraction. For the **REAL32** type *s* is 1 bit wide, *exp* is 8 bits wide and *frac* is 23 bits wide. For the **REAL64** type *s* is 1 bit wide, *exp* is 11 bits wide and *frac* is 52 bits wide. Whenever the *exp* field is not 0 the actual fraction of the number represented has an “implied” 1 placed on the left of the *frac* value.

The value of finite **REAL**s is given by

$$\mathit{val}\ \boxed{s \mid exp \mid frac} = \begin{cases} (-1)^s \times 1.frac \times 2^{exp-bias}, & \text{if } exp \neq 0; \\ (-1)^s \times 0.frac \times 2^{1-bias}, & \text{if } exp = 0; \end{cases}$$

where *bias* is 127 for **REAL32** and 1023 for **REAL64**.

In the **REAL32** type the value 1.0 is represented by an unset sign bit *s*, an *exp* equal to 127, and a *frac* of 0. The next larger number has an unset sign bit, *exp* of 127 and a *frac* of 1. This has the value 1.000000119209…. Hence any number lying between 1.0 and this value cannot be exactly represented in the **REAL32** type – such values have to be *rounded* to one of these values. Now consider the assignment:

```
X := 1.0(REAL32) + 1.0E-7(REAL32)
```

The previous sections show that the result of this operation cannot be exactly represented by the type **REAL32**. The exact result has to be rounded to “fit” the type. Here the exact result will be rounded to the nearest **REAL32** value 1.000000119209….

Other rounding modes – Round to Zero (truncation), Round to Plus infinity and Round to Minus infinity – can be obtained through the use of the **IEEEOP** function. Because of the presence of rounding, programmers should be wary of using equality tests on real types. Consider the following example:

```
SEQ
  X := 1.0(REAL32)
  WHILE X <> 1.000001(REAL32)
    X := X + 0.0000005(REAL32)
```

never terminates as rounding errors cause `1.000001` and `1.0 + 0.0000005 + 0.0000005` to differ.

The nearest unique value of a conversion of a literal of type **REAL32** can be determined from the first 9 significant digits, and from the first 17 significant digits of a literal of type **REAL64**. Complete details of the IEEE Standard for Binary Floating-Point Arithmetic can be found in the published ANSI/IEEE Std 754-1985 standard.

E Usage rules check list

This appendix summarises the rules which govern the use of variables, channels, timers, ports (page 96) and arrays in parallel constructions, and the rules which govern abbreviations and parameters. These rules are discussed in context throughout the manual, and are gathered here as a check list. Programs violating these rules are deemed *illegal* as alias and usage checking can usually be performed at compile time. If the compiler has to generate run-time checks then a violation is treated as an invalid process.

E.1 Usage in parallel

The purpose of these rules is to prevent parallel processes from sharing variables, to ensure that each channel connects only two parallel processes, and to ensure that the connection of channels is unidirectional. The rules allow most of the checking for valid usage to be performed by a compiler, thus reducing runtime overheads.

- A channel implements a point-to-point communication between two parallel processes. The name of a channel may only be used in one component of a parallel for input, and in one other component of the parallel for output.
- A timer may be used for input by any number of components of a parallel.
- A variable or component of an array variable, which is assigned to in a component of a parallel, may not appear in any other component of the parallel.
- Components of an array variable may be assigned to in more than one component of a parallel, if and only if the used subscripts are compile-time constants and select distinct components of the array variable. However, an implementation may use the following more general rule: components of an array may be assigned to in parallel, if and only if it can be determined at compile time that the used subscripts select distinct components of the array.
- Several abbreviations can decompose an array into non-overlapping disjoint parts; components of these parts may then be selected using variable subscripts.
- A port may be used in only one component of a parallel.

E.2 The rules for abbreviations

The purpose of these rules is to ensure that each name identifies a unique object, and that the substitution semantics are maintained.

- All reference to an abbreviated element must be via the abbreviation only, with the exception that array elements may be further abbreviated providing the later abbreviations do not include components of the array already abbreviated.
- Variables used in an abbreviated expression may not receive new values by input or assignment within the scope of the abbreviation.
- The abbreviated expression must be valid, i.e. in range and not subject to overflow, and all subscript expressions must be in range.
- All subscript expressions used in an abbreviation must be valid, i.e. not subject to overflow and in range.
- All references to a *retyped* or *reshaped* variable must be via the new name only, with the exception that array variables may be further retyped providing the later retyping conversions do not include components of the array already retyped.
- Variables used in a retyping or reshaping conversion may not receive new values by input or assignment within the scope of the new name.

E.3 The rules for procedures

- The rules for procedure parameters follow from those for abbreviations, but in addition a channel parameter or free channel may not be used for both input and output in a procedure.

E.4 The rules for value processes and functions

- Functions may only have value parameters.
- Only variables declared within the scope of a value process may be assigned to. Free names may be used in expressions.
- A value process may not contain inputs, outputs parallels or alternations.
- The body of a procedure used within a function must also obey these rules. Such a procedure may include assignments to variables declared in the enclosing function.

F Invalid processes

Processes which become invalid during program execution may behave in one of three ways, determined by a compiler option or otherwise. An invalid process may behave in one of these ways: the process may stop, the system may halt, or the behaviour of the process may be undefined.

The three *error modes* in detail are:

Stop process mode In this mode, processes which become invalid behave like the primitive process `STOP`, thus allowing other processes to continue. The invalid process stops, and in particular does not make erroneous outputs to other processes. Other processes continue until they become dependent upon communication with the stopped process. In this mode it is therefore possible to write communications which will timeout to warn of a stopped process, and to construct a system with redundancy in which a number of processes performing the same task may be used to enable the system to continue after one of the processes has failed.

Halt system mode In this mode an invalid process may cause the whole system to halt, and is useful for the development of programs, particularly when debugging concurrent systems. In this mode the primitive process `STOP` will also cause the whole system to halt.

Undefined mode In this mode, an invalid process may have an arbitrary effect, and is only useful for allowing a compiler to optimise programs known to be correct!

G Lexical program components

The lexical components of an occam program are the keywords, symbols, names and literals. The distinction between lexical and syntactic components is not rigid and may vary between different compilers. For example, in the following appendix H the categories *integer* and it real are defined by syntactic productions, whereas a compiler will probably recognise these by lexical analysis.

The sections of this appendix are a table of occam keywords, a table of symbols, informally classified into groups, and a table of the ASCII character set for strings and byte literals. The final section of this appendix defines in words the syntactic categories for names and literals that are not defined in the syntax summarised in appendix H.

G.1 Keywords

The occam keywords are:

AFTER	later than operator	**OFFSETOF**	record layout
ALT	alternation	**OR**	boolean or operator
AND	boolean and operator	**PACKED**	record type qualifier
ANY	anarchic protocol	**PAR**	parallel
ASM	assembly language insertion	**PLACE**	allocation
AT	at *location*	**PLACED**	placed processes
BITAND	bitwise and operator	**PLUS**	modulo addition operator
BITNOT	bitwise not operator	**PORT OF**	port type
BITOR	bitwise or operator	**PRI**	prioritised construction
BOOL	boolean type	**PROC**	procedure
BYTE	byte type	**PROCESSOR**	processor allocation
BYTESIN	width operator	**PROTOCOL**	protocol definition
CASE	selection, variant protocol, case input	**REAL32**	32bit real type
CHAN OF	channel type	**REAL64**	64bit real type
DATA	data type definition	**RECORD**	record type
ELSE	default selection	**REM**	remainder operator
FALSE	boolean constant	**RESHAPES**	reshaping conversion
FOR	count	**RESULT**	value process result
FROM	base	**RETYPES**	retyping conversion
FUNCTION	function definition	**ROUND**	rounding operator
IF	conditional	**SEQ**	sequence
INLINE	inline code indicator	**SIZE**	array size operator
INT	integer type	**SKIP**	skip process
INT16	16bit integer type	**STOP**	stop process
INT32	32bit integer type	**TIMER**	timer type
INT64	64bit integer type	**TIMES**	modulo multiplication operator
IS	specification introduction	**TRUE**	boolean constant
MINUS	modulo subtraction/negation operator	**TRUNC**	truncation operator
MOSTNEG	most negative	**TYPE**	type definition
MOSTPOS	most positive	**VAL**	value
NOT	boolean not operator	**VALOF**	value process
		WHILE	loop

If an implementation adds further reserved words, then the names used will not include lower case letters. Programmers should consult documentation supplied with a particular compiler to see if there are other words which cannot be used as occam names.

G.2 Symbols

The symbols of occam each composed of one or two ASCII characters are tabulated here according to a simple classification. They all appear explicitly in one or more syntactic productions in appendix H.

Arithmetic operators	
+	plus
-	minus
*	times
/	divide
\	remainder

Bit operators	
/\	and
\/	or
><	exclusive or
~	not
<<	left shift
>>	right shift

Relational operators	
=	equal
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
<>	not equal

Communication symbols	
?	Input
!	Output

Other symbols	
#	Hexadecimal or compiler directive
&	Ampersand: used in a guard
(	Parentheses: used to delimit expressions,
)	the type of literals and a parameter list
[	Square brackets: used to delimit array subscripts,
]	and to construct segments and tables
[]	Array type specifier
::	Counted array communication
:=	Assignment symbol
"	Double quote: used to construct a string byte table
'	Single quote: used to delimit character byte literal
,	Comma: separator for specifications, parameters,
	expression or variable lists and tables
;	Sequential protocol separator
:	Specification terminator
--	Comment introduction

G.3 Character set

Characters in occam are represented according to the American Standard Code for Information Interchange (ASCII). Where the full character set is not available occam guarantees the following subset:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789
!"#&'()*+,-./:;<=>?[]
```

For reference, here is a table of all printable ASCII characters, and their values:

ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex
SPACE	32	20	`@`	64	40	`` ` ``	96	60
`!`	33	21	`A`	65	41	`a`	97	61
`"`	34	22	`B`	66	42	`b`	98	62
`#`	35	23	`C`	67	43	`c`	99	63
`$`	36	24	`D`	68	44	`d`	100	64
`%`	37	25	`E`	69	45	`e`	101	65
`&`	38	26	`F`	70	46	`f`	102	66
`'`	39	27	`G`	71	47	`g`	103	67
`(`	40	28	`H`	72	48	`h`	104	68
`)`	41	29	`I`	73	49	`i`	105	69
`*`	42	2A	`J`	74	4A	`j`	106	6A
`+`	43	2B	`K`	75	4B	`k`	107	6B
`,`	44	2C	`L`	76	4C	`l`	108	6C
`-`	45	2D	`M`	77	4D	`m`	109	6D
`.`	46	2E	`N`	78	4E	`n`	110	6E
`/`	47	2F	`O`	79	4F	`o`	111	6F
`0`	48	30	`P`	80	50	`p`	112	70
`1`	49	31	`Q`	81	51	`q`	113	71
`2`	50	32	`R`	82	52	`r`	114	72
`3`	51	33	`S`	83	53	`s`	115	73
`4`	52	34	`T`	84	54	`t`	116	74
`5`	53	35	`U`	85	55	`u`	117	75
`6`	54	36	`V`	86	56	`v`	118	76
`7`	55	37	`W`	87	57	`w`	119	77
`8`	56	38	`X`	88	58	`x`	120	78
`9`	57	39	`Y`	89	59	`y`	121	79
`:`	58	3A	`Z`	90	5A	`z`	122	7A
`;`	59	3B	`[`	91	5B	`{`	123	7B
`<`	60	3C	`\`	92	5C	`|`	124	7C
`=`	61	3D	`]`	93	5D	`}`	125	7D
`>`	62	3E	`^`	94	5E	`~`	126	7E
`?`	63	3F	`_`	95	5F		127	7F

The characters `*`, `'` and `"` may not be used directly in strings or as character constants. These and non-printable characters (such as carriage return, tab *&c.*) can be included in strings, or used as character constants, in the following form:

`*c`	`*C`	carriage return	= *#0D
`*n`	`*N`	newline	= *#0A
`*t`	`*T`	tab	= *#09
`*s`	`*S`	space	= *#20
`*'`		quotation mark	
`*"`		double quotation mark	
`**`		asterisk	

In addition, any byte value can be represented in a byte literal or within a string by `*#` followed by two hexadecimal digits, for example:

`soh := '*#01'`	`'*#01'` is a byte constant.

An implementation may accept national variants of the ASCII character set both in comments and in strings and character constants, subject only to the restriction that characters are all held in 8-bit bytes. Programs using such extended character sets may not be portable between implementations.

G.4 Names and literals

The tables of occam syntax in appendix H leave 5 categories undefined as it is assumed, but not required, that these will be recognised by lexical analysis. Semi-formal definitions of these (*name, digits, hex.digits, character* and *string*) are given below:

An occam *name* must begin with an alphabetic character. Names consist of a sequence of one or more upper or lower case alphabetic characters, digits or dots.

The syntactic category *digits* represents a sequence of one or more of the characters **`0123456789`** and *hex.digits* a sequence of one or more of the characters **`0123456789ABCDEF`**.

A *character* is either an explicit ASCII character, a special character escape introduced by **`*`**, or a hexadecimal coded character introduced by **`*#`**. For further details see page 103.

A *string* is a sequence of *character*s between a pair of double quotes (**`"`**). A string may be broken over several lines by terminating broken lines with an asterisk, and starting the continuation on the following line with another asterisk at an indentation no less than the line where the string starts. An additional character escape (**`*l`** or **`*L`**) may be defined for use as first character of a string of less than 256 bytes. The value of this byte is the length in bytes of the string following this byte.

H Ordered syntax of occam

The following tables present the syntax of occam, with each syntactic category placed in alphabetical order. In this tabulation all productions for each category appear together, including those introduced in appendices B and C which may not be implemented by all occam compilers.

The metalanguage of these descriptions is introduced on page 3. Note that these rules must be read in conjunction with the semantic rules given informally in the text of the definition, as it is possible to construct programs which obey the syntax rules but are not legal occam.

Opposite the productions for each syntactic category is given a list of page numbers where these productions first appear together with the qualifying semantic rules which restrict the class of legal occam programs.

Certain syntactic categories are not defined in this tabulations. These are *digits, hex.digits* (see page 28), *character* (see page 29), *name* (see page 4) and *string* (see page 58). The definitions of these categories were repeated in appendix G above.

abbreviation	=	*name* **IS** *variable* **:**	(43)
	|	*specifier name* **IS** *variable* **:**	(43)
	|	**VAL** *name* **IS** *expression* **:**	(44)
	|	**VAL** *specifier name* **IS** *expression* **:**	(44)
	|	*name* **IS** *channel* **:**	(54)
	|	*specifier name* **IS** *channel* **:**	(54)
	|	*name* **IS** **[**$\{_1$ **,** *channel* $\}$**]** **:**	(54)
	|	*specifier name* **IS** **[**$\{_1$ **,** *channel* $\}$**]** **:**	(54)
	|	*name* **IS** *timer* **:**	(83)
	|	*specifier name* **IS** *timer* **:**	(83)
	|	*name* **IS** *port* **:**	(96)
	|	*specifier name* **IS** *port* **:**	(96)

actual	=	*variable*	(73)
	|	*channel*	(73)
	|	*timer*	(73)
	|	*port*	(96)
	|	*expression*	(73)

allocation	=	**PLACE** *name* **AT** *expression* **:**	(95)

alternation	=	**ALT** { *alternative* }	(21)
	|	**ALT** *replicator* *alternative*	(23)
	|	**PRI ALT** { *alternative* }	(24)
	|	**PRI ALT** *replicator* *alternative*	(24)

alternative	=	*guarded.alternative*	(21)
	|	*alternation*	(21)
	|	*channel* **? CASE**	(52)
		{ *variant* }	
	|	*boolean* **&** *channel* **? CASE**	(52)
		{ *variant* }	
	|	*specification*	
		alternative	(41)
assignment	=	*variable.list* **:=** *expression.list*	(5)
base	=	*expression*	(10,12,17,23)
boolean	=	*expression*	(11)
byte	=	**'** *character* **'**	(28)
case.expression	=	*expression*	(13)
case.input	=	*channel* **? CASE**	(52)
		{ *variant* }	
channel	=	*name*	(45)
	|	*channel* **[** *expression* **]**	(46)
	|	**[** *channel* **FROM** *base* **FOR** *count* **]**	(46)
	|	**[** *channel* **FROM** *base* **]**	(46)
	|	**[** *channel* **FOR** *count* **]**	(46)
channel.type	=	**CHAN OF** *protocol*	(45)
	|	**[** *expression* **]** *channel.type*	(46)
choice	=	*guarded.choice*	(11)
	|	*conditional*	(11)
	|	*specification*	(41)
		choice	

conditional	=	**IF**	(11)
		{ *choice* }	
	|	**IF** *replicator*	(12)
		choice	

conversion	=	*data.type operand*	(67)
	|	*data.type* **ROUND** *operand*	(67)
	|	*data.type* **TRUNC** *operand*	(67)

count	=	*expression*	(10,12,17,23)

data.type	=	**BOOL**	(26)
	|	**BYTE**	(26)
	|	**INT**	(26)
	|	**INT16**	(26)
	|	**INT32**	(26)
	|	**INT64**	(26)
	|	**REAL32**	(26)
	|	**REAL64**	(26)
	|	*name*	(27)
	|	**[** *expression* **]** *data.type*	(31)

declaration	=	*data.type* $\{_1$ **,** *name* $\}$ **:**	(35)
	|	*channel.type* $\{_1$ **,** *name* $\}$ **:**	(45)
	|	*timer.type* $\{_1$ **,** *name* $\}$ **:**	(81)
	|	*port.type* $\{_1$ **,** *name* $\}$ **:**	(96)

```
definition        =  DATA TYPE name IS data.type :                          (27)
                  |  DATA TYPE name                                         (31)
                       structured.type
                     :
                  |  PROTOCOL name IS simple.protocol :                     (48)
                  |  PROTOCOL name IS sequential.protocol :                 (48)
                  |  PROTOCOL name                                          (50)
                       CASE
                         { tagged.protocol }
                     :
                  |  PROC name ( {0 , formal } )                            (73)
                       process
                     :
                  |  {1 , data.type } function.header                       (78)
                       value.process
                     :
                  |  {1 , data.type } function.header IS expression.list :  (78)
                  |  specifier name RETYPES variable :                      (85)
                  |  VAL specifier name RETYPES expression :                (85)
                  |  specifier name RETYPES channel :                       (87)
                  |  specifier name RETYPES port :                          (97)
                  |  specifier name RESHAPES variable :                     (88)
                  |  VAL specifier name RESHAPES expression :               (88)
                  |  specifier name RESHAPES channel :                      (88)
                  |  specifier name RESHAPES port :                         (97)

delayed.input     =  timer ? AFTER expression                               (82)

dyadic.operator   =  + | - | * | / | \ | REM | PLUS | MINUS | TIMES         (57)
                  |  /\ | \/ | >< | BITAND | BITOR | AND | OR
                  |  = | <> | < | > | >= | <= | AFTER

exponent          =  +digits                                                (28)
                  |  -digits                                                (28)

expression        =  operand                                                (57)
                  |  monadic.operator operand                               (57)
                  |  operand dyadic.operator operand                        (57)
                  |  MOSTPOS data.type                                      (65)
                  |  MOSTNEG data.type                                      (65)
                  |  SIZE data.type                                         (65)
                  |  conversion                                             (57)
```

expression.list	=	$\{_1$ **,** *expression* $\}$	(5)
	|	*name* **(** $\{_0$ **,** *expression* $\}$ **)**	(78)
	|	**(** *value.process* **)**	(75)
field.name	=	*name*	(31)
formal	=	*specifier* $\{_1$ **,** *name* $\}$	(73)
	|	**VAL** *specifier* $\{_1$ **,** *name* $\}$	(73)
function.header	=	**FUNCTION** *name* **(** $\{_0$ **,** *formal* $\}$**)**	(78)
guard	=	*input*	(21)
	|	*boolean* **&** *input*	(21)
	|	*boolean* **&** **SKIP**	(21)
guarded.alternative	=	*guard* *process*	(21)
guarded.choice	=	*boolean* *process*	(11)
input	=	*channel* **?** $\{_1$ **;** *input.item* $\}$	(6,48,49)
	|	*channel* **?** **CASE** *tagged.list*	(52)
	|	*timer.input*	(82)
	|	*delayed.input*	(82)
	|	*port* **?** *variable*	(96)
input.item	=	*variable*	(48)
	|	*variable* **::** *variable*	(48)
integer	=	*digits*	(28)
	|	**#***hex.digits*	(28)

literal	=	*integer*	(28)
	|	*byte*	(28)
	|	*real*	(28)
	|	*integer* **(** *data.type* **)**	(28)
	|	*byte* **(** *data.type* **)**	(28)
	|	*real* **(** *data.type* **)**	(28)
	|	**TRUE**	(28)
	|	**FALSE**	(28)
loop	=	**WHILE** *boolean* *process*	(14)
monadic.operator	=	**-** | **MINUS** | **~** | **BITNOT** | **NOT** | **SIZE**	(57)
operand	=	*variable*	(57)
	|	*literal*	(57)
	|	*table*	(57)
	|	**(** *expression* **)**	(57)
	|	**(** *value.process* **)**	(75)
	|	*name* **(** $\{_0$ **,** *expression* $\}$ **)**	(78)
	|	*operand* **[** *expression* **]**	(78)
	|	**BYTESIN** **(** *operand* **)**	(66)
	|	**BYTESIN** **(** *data.type* **)**	(66)
	|	**OFFSETOF** **(** *name* **,** *field.name* **)**	(66)
option	=	$\{_1$ **,** *case.expression* $\}$ *process*	(13)
	|	**ELSE** *process*	(13)
	|	*specification* *option*	(41)
output	=	*channel* **!** $\{_1$ **;** *output.item* $\}$	(6,48,49)
	|	*channel* **!** *tag*	(50)
	|	*channel* **!** *tag* **;** $\{_1$ **;** *output.item* $\}$	(50)
	|	*port* **!** *expression*	(96)
output.item	=	*expression*	(48)
	|	*expression* **::** *expression*	(48)

parallel	=	**PAR** { *process* }	(15)
	|	**PAR** *replicator* *process*	(17)
	|	**PRI PAR** { *process* }	(94)
	|	**PRI PAR** *replicator* *process*	(94)
	|	*placedpar*	(93)
placedpar	=	**PLACED PAR** { *placedpar* }	(93)
	|	**PLACED PAR** *replicator* *placedpar*	(93)
	|	**PROCESSOR** *expression* *process*	(93)
port	=	*name*	(96)
	|	*port* **[** *expression* **]**	(96)
	|	**[** *port* **FROM** *base* **FOR** *count* **]**	(96)
	|	**[** *port* **FROM** *base* **]**	(96)
	|	**[** *port* **FOR** *count* **]**	(96)
port.type	=	**PORT OF** *data.type*	(96)
	|	**[** *expression* **]** *port.type*	(96)
proc.instance	=	*name* **(** $\{_0$ **,** *actual* } **)**	(73)
process	=	*assignment*	(24)
	|	*input*	(24)
	|	*output*	(24)
	|	**SKIP**	(24)
	|	**STOP**	(24)
	|	*sequence*	(24)
	|	*conditional*	(24)
	|	*selection*	(24)
	|	*loop*	(24)
	|	*parallel*	(24)
	|	*alternation*	(24)
	|	*case.input*	(52)
	|	*proc.instance*	(73)
	|	*specification* *process*	(41)
	|	*allocation* *process*	(95)

protocol	=	*name*	(48)
	|	*simple.protocol*	(48)
real	=	*digits* **.** *digits*	(28)
	|	*digits* **.** *digits* **E** *exponent*	(28)
replicator	=	*name* **=** *base* **FOR** *count*	(10,12,17,23)
selection	=	**CASE** *selector* { *option* }	(13)
selector	=	*expression*	(13)
sequence	=	**SEQ** { *process* }	(9)
	|	**SEQ** *replicator* *process*	(10)
sequential.protocol	=	$\{_1$ **;** *simple.protocol* $\}$	(49)
simple.protocol	=	*data.type*	(48)
	|	**ANY**	(54)
	|	*data.type* **::[]** *data.type*	(48)
specification	=	*declaration*	(40)
	|	*abbreviation*	(40)
	|	*definition*	(40)
specifier	=	*data.type*	(43)
	|	*channel.type*	(54)
	|	*timer.type*	(83)
	|	*port.type*	(96)
	|	**[]** *specifier*	(43,54,83)
	|	**[** *expression* **]** *specifier*	(43,54,83)
structured.type	=	**RECORD** { *data.type* $\{_1$ **,** *field.name* $\}$ **:** }	(31)
	|	**PACKED RECORD** { *data.type* $\{_1$ **,** *field.name* $\}$ **:** }	(31)

table	=	*string*	(59)
	|	*string* **(** *name* **)**	(59)
	|	**[** {1 **,** *expression* } **]**	(32,59)
	|	*table* **[** *expression* **]**	(32,59)
	|	**[** *table* **FROM** *base* **FOR** *count* **]**	(59)
	|	**[** *table* **FROM** *base* **]**	(59)
	|	**[** *table* **FOR** *count* **]**	(59)
tag	=	*name*	(50)
tagged.list	=	*tag*	(52)
	|	*tag* **;** {1 **;** *input.item* }	(52)
tagged.protocol	=	*tag*	(50)
	|	*tag* **;** *sequential.protocol*	(50)
timer.input	=	*timer* **?** *variable*	(82)
timer	=	*name*	(81)
	|	*timer* **[** *expression* **]**	(81)
	|	**[** *timer* **FROM** *base* **FOR** *count* **]**	(81)
	|	**[** *timer* **FROM** *base* **]**	(81)
	|	**[** *timer* **FOR** *count* **]**	(81)
timer.type	=	**TIMER**	(81)
	|	**[** *expression* **]** *timer.type*	(81)

value.process	=	**VALOF**
		  process
		  RESULT *expression.list*
			(75)

	|	*specification* *value.process*	(75)
variable	=	*name*	(35)
	|	*variable* **[** *expression* **]**	(37)
	|	**[** *variable* **FROM** *base* **FOR** *count* **]**	(37)
	|	**[** *variable* **FROM** *base* **]**	(37)
	|	**[** *variable* **FOR** *count* **]**	(37)

variable.list	=	$\{_1$, *variable* }	(5)
variant	=	*tagged.list* *process*	(52)
	|	*specification* *variant*	(41,52)

I Library procedures and functions

This appendix provides a list of the library routines that an implementation should provide.

The behaviour of routines is described in greater detail in the following appendices, but this may be amplified or modified in further documentation accompanying an implementation.

It may be necessary to use compiler directives such as **#USE** to incorporate specifications of these routines into an occam program at an appropriate point (see page 91).

Some library routines (typically the most primitive routines) may be predefined in an implementation, that is, they may be known to the compiler and so not need to be explicitly referenced by the programmer. The names of such routines do not become reserved words as the programmer is free to redefine them in local specifications. An implementation may define additional predefined names not mentioned here. This is typically done to provide convenient access to special purpose instructions available in one or more target processor types.

Other libraries may need to be explicitly referenced by the programmer, and the names used in their specifications behave like any other names and may be redeclared with different meaning in local scopes. However, programmers are discouraged from using the names of any library routine for any purpose other than that of naming the routine in question. The following tables include the name of the routine, and a specifier which indicates the type of each of the parameters to the routine.

The libraries described here are not intended to be exhaustive, and a large group of additional routines have been provided with most existing occam implementations and have become established by common usage. None of the routines described here make explicit use of extensions to the language beyond occam2.

I.1 Multiple length integer arithmetic functions

The arithmetic functions provide arithmetic shifts, word rotations and the primitives to construct multiple length arithmetic and multiple length shift operations.

```
INT         FUNCTION LONGADD      (VAL INT left, right, carry.in)
INT         FUNCTION LONGSUB      (VAL INT left, right, borrow.in)
INT         FUNCTION ASHIFTRIGHT  (VAL INT argument, places)
INT         FUNCTION ASHIFTLEFT   (VAL INT argument, places)
INT         FUNCTION ROTATERIGHT  (VAL INT argument, places)
INT         FUNCTION ROTATELEFT   (VAL INT argument, places)
INT,INT     FUNCTION LONGSUM      (VAL INT left, right, carry.in)
INT,INT     FUNCTION LONGDIFF     (VAL INT left, right, borrow.in)
INT,INT     FUNCTION LONGPROD     (VAL INT left, right, carry.in)
INT,INT     FUNCTION LONGDIV      (VAL INT dividend.hi, dividend.lo, divisor)
INT,INT     FUNCTION SHIFTLEFT    (VAL INT hi.in, lo.in, places)
INT,INT     FUNCTION SHIFTRIGHT   (VAL INT hi.in, lo.in, places)
INT,INT,INT FUNCTION NORMALISE    (VAL INT hi.in, lo.in)
```

I.2 Floating point functions

The floating point functions provide the list of facilities suggested by the ANSI/IEEE standard 754-1985.

```
REAL32             FUNCTION ABS                (VAL REAL32 X)
REAL64             FUNCTION DABS               (VAL REAL64 X)
REAL32             FUNCTION SCALEB             (VAL REAL32 X,VAL INT n)
REAL64             FUNCTION DSCALEB            (VAL REAL64 X,VAL INT n)
REAL32             FUNCTION COPYSIGN           (VAL REAL32 X, Y)
REAL64             FUNCTION DCOPYSIGN          (VAL REAL64 X, Y)
REAL32             FUNCTION SQRT               (VAL REAL32 X)
REAL64             FUNCTION DSQRT              (VAL REAL64 X)
REAL32             FUNCTION MINUSX             (VAL REAL32 X)
REAL64             FUNCTION DMINUSX            (VAL REAL64 X)
REAL32             FUNCTION NEXTAFTER          (VAL REAL32 X, Y)
REAL64             FUNCTION DNEXTAFTER         (VAL REAL64 X, Y)
REAL32             FUNCTION MULBY2             (VAL REAL32 X)
REAL64             FUNCTION DMULBY2            (VAL REAL64 X)
REAL32             FUNCTION DIVBY2             (VAL REAL32 X)
REAL64             FUNCTION DDIVBY2            (VAL REAL64 X)
REAL32             FUNCTION LOGB               (VAL REAL32 X)
REAL64             FUNCTION DLOGB              (VAL REAL64 X)
BOOL               FUNCTION ISNAN              (VAL REAL32 X)
BOOL               FUNCTION DISNAN             (VAL REAL64 X)
BOOL               FUNCTION NOTFINITE          (VAL REAL32 X)
BOOL               FUNCTION DNOTFINITE         (VAL REAL64 X)
BOOL               FUNCTION ORDERED            (VAL REAL32 X, Y)
BOOL               FUNCTION DORDERED           (VAL REAL64 X, Y)
INT,REAL32         FUNCTION FLOATING.UNPACK    (VAL REAL32 X)
INT,REAL64         FUNCTION DFLOATING.UNPACK   (VAL REAL64 X)
BOOL,INT32,REAL32  FUNCTION ARGUMENT.REDUCE    (VAL REAL32 X, Y, Y.err)
BOOL,INT32,REAL64  FUNCTION DARGUMENT.REDUCE   (VAL REAL64 X, Y, Y.err)
REAL32             FUNCTION FPINT              (VAL REAL32 X)
REAL64             FUNCTION DFPINT             (VAL REAL64 X)
```

I.3 Full IEEE arithmetic functions

```
REAL32       FUNCTION REAL32OP       (VAL REAL32 X, VAL INT Op, VAL REAL32 Y)
REAL64       FUNCTION REAL64OP       (VAL REAL64 X, VAL INT Op, VAL REAL64 Y)
BOOL,REAL32  FUNCTION IEEE32OP       (VAL REAL32 X, VAL INT Rm, VAL INT Op,
                                                                VAL REAL32 Y)
BOOL,REAL64  FUNCTION IEEE64OP       (VAL REAL64 X, VAL INT Rm, VAL INT Op,
                                                                VAL REAL64 Y)
REAL32       FUNCTION REAL32REM      (VAL REAL32 X, Y)
REAL64       FUNCTION REAL64REM      (VAL REAL64 X, Y)
BOOL,REAL32  FUNCTION IEEE32REM      (VAL REAL32 X, Y)
BOOL,REAL64  FUNCTION IEEE64REM      (VAL REAL64 X, Y)
BOOL         FUNCTION REAL32EQ       (VAL REAL32 X, Y)
BOOL         FUNCTION REAL64EQ       (VAL REAL64 X, Y)
BOOL         FUNCTION REAL32GT       (VAL REAL32 X, Y)
BOOL         FUNCTION REAL64GT       (VAL REAL64 X, Y)
INT          FUNCTION IEEECOMPARE    (VAL REAL32 X, Y)
INT          FUNCTION DIEEECOMPARE   (VAL REAL64 X, Y)
```

I.4 Elementary function library

```
REAL32         FUNCTION ALOG      (VAL REAL32 X)
REAL64         FUNCTION DALOG     (VAL REAL64 X)
REAL32         FUNCTION ALOG10    (VAL REAL32 X)
REAL64         FUNCTION DALOG10   (VAL REAL64 X)
REAL32         FUNCTION EXP       (VAL REAL32 X)
REAL64         FUNCTION DEXP      (VAL REAL64 X)
REAL32         FUNCTION TAN       (VAL REAL32 X)
REAL64         FUNCTION DTAN      (VAL REAL64 X)
REAL32         FUNCTION SIN       (VAL REAL32 X)
REAL64         FUNCTION DSIN      (VAL REAL64 X)
REAL32         FUNCTION ASIN      (VAL REAL32 X)
REAL64         FUNCTION DASIN     (VAL REAL64 X)
REAL32         FUNCTION COS       (VAL REAL32 X)
REAL64         FUNCTION DCOS      (VAL REAL64 X)
REAL32         FUNCTION ACOS      (VAL REAL32 X)
REAL64         FUNCTION DACOS     (VAL REAL64 X)
REAL32         FUNCTION SINH      (VAL REAL32 X)
REAL64         FUNCTION DSINH     (VAL REAL64 X)
REAL32         FUNCTION COSH      (VAL REAL32 X)
REAL64         FUNCTION DCOSH     (VAL REAL64 X)
REAL32         FUNCTION TANH      (VAL REAL32 X)
REAL64         FUNCTION DTANH     (VAL REAL64 X)
REAL32         FUNCTION ATAN      (VAL REAL32 X)
REAL64         FUNCTION DATAN     (VAL REAL64 X)
REAL32         FUNCTION ATAN2     (VAL REAL32 X, Y)
REAL64         FUNCTION DATAN2    (VAL REAL64 X, Y)
REAL32, INT32  FUNCTION RAN       (VAL INT32 N)
REAL64, INT64  FUNCTION DRAN      (VAL INT64 N)
REAL32         FUNCTION POWER     (VAL REAL32 X, Y)
REAL64         FUNCTION DPOWER    (VAL REAL64 X, Y)
```

I.5 Value, string conversion procedures

The library provides primitive procedures to convert a value to and from decimal or hexadecimal representations.

```
PROC INTTOSTRING     (INT len, []BYTE string, VAL INT n)
PROC INT16TOSTRING   (INT len, []BYTE string, VAL INT16 n)
PROC INT32TOSTRING   (INT len, []BYTE string, VAL INT32 n)
PROC INT64TOSTRING   (INT len, []BYTE string, VAL INT64 n)
PROC STRINGTOINT     (BOOL error, INT n, VAL []BYTE string)
PROC STRINGTOINT16   (BOOL error, INT16 n, VAL []BYTE string)
PROC STRINGTOINT32   (BOOL error, INT32 n, VAL []BYTE string)
PROC STRINGTOINT64   (BOOL error, INT64 n, VAL []BYTE string)
PROC HEXTOSTRING     (INT len, []BYTE string, VAL INT n)
PROC HEX16TOSTRING   (INT len, []BYTE string, VAL INT16 n)
PROC HEX32TOSTRING   (INT len, []BYTE string, VAL INT32 n)
PROC HEX64TOSTRING   (INT len, []BYTE string, VAL INT64 n)
PROC STRINGTOHEX     (BOOL error, INT n, VAL []BYTE string)
PROC STRINGTOHEX16   (BOOL error, INT16 n, VAL []BYTE string)
PROC STRINGTOHEX32   (BOOL error, INT32 n, VAL []BYTE string)
PROC STRINGTOHEX64   (BOOL error, INT64 n, VAL []BYTE string)
PROC STRINGTOREAL32  (BOOL error, REAL32 r, VAL []BYTE string)
PROC STRINGTOREAL64  (BOOL error, REAL64 r, VAL []BYTE string)
PROC REAL32TOSTRING  (INT, []BYTE, VAL REAL32, VAL INT)
PROC REAL64TOSTRING  (INT, []BYTE, VAL REAL64, VAL INT)
PROC STRINGTOBOOL    (BOOL error, b, VAL []BYTE string)
PROC BOOLTOSTRING    (INT len, []BYTE string, VAL BOOL b)
```

I.6 Programming support routines

The library provides procedures which provide some useful facilities for the occam programmer.

```
PROC RESCHEDULE  ()
PROC ASSERT      (VAL BOOL assertion)
```

J Multiple length integer arithmetic functions

The following arithmetic functions provide arithmetic shifts, word rotations and the primitives to construct multiple length integer arithmetic and multiple length shift operations.

LONGADD	signed addition with a carry in.
LONGSUM	unsigned addition with a carry in and a carry out.
LONGSUB	signed subtraction with a borrow in.
LONGDIFF	unsigned subtraction with a borrow in and a borrow out.
LONGPROD	unsigned multiplication with a carry in, producing a double length result.
LONGDIV	unsigned division of a double length number, producing a single length result.
SHIFTRIGHT	right shift on a double length quantity.
SHIFTLEFT	left shift on a double length quantity.
NORMALISE	normalise a double length quantity.
ASHIFTRIGHT	arithmetic right shift on a double length quantity.
ASHIFTLEFT	arithmetic left shift on a double length quantity.
ROTATERIGHT	rotate a word right.
ROTATELEFT	rotate a word left.

For the purpose of explanation imagine a new type *INTEGER*, and the associated conversion. This imaginary type is capable of representing the complete set of integers and is presumed to be represented as an infinite bit two's complement number. With this one exception the following are occam descriptions of the various arithmetic functions.

```
-- constants used in the following description
VAL bitsperword  IS machine.wordsize(INTEGER) :
VAL range        IS storeable.values(INTEGER) :
                 -- range = 2^bitsperword
VAL maxint       IS INTEGER (MOSTPOS INT) :
                 -- maxint = (range/2 - 1)
VAL minint       IS INTEGER (MOSTNEG INT) :
                 -- minint = -(range/2)
-- INTEGER literals
VAL one          IS 1(INTEGER) :
VAL two          IS 2(INTEGER) :
-- mask
VAL wordmask     IS range - one :
```

In occam, values are considered to be signed. However, in these functions the concern is with other interpretations. In the construction of multiple length arithmetic the need is to interpret words as containing both signed and unsigned integers. In the following the new *INTEGER* type is used to manipulate these values, and other values which may require more than a single word to store.

The sign conversion of a value is defined in the functions **unsign** and **sign**. These are used in the description following but they are NOT functions themselves.

```
INTEGER FUNCTION unsign (VAL INT operand)

  -- Returns the value of operand as an unsigned integer value.
  -- for example, on a 32 bit word machine :
  -- unsign ( 1 ) = 1
  -- unsign (−1 ) = 2^32 − 1

  INTEGER  operand.i
  VALOF
    IF
      operand < 0
        operand.i := (INTEGER  operand) + range
      operand >= 0
        operand.i := INTEGER  operand
    RESULT operand.i
:

INT FUNCTION sign (VAL INTEGER  result.i)

  -- Takes the INTEGER result.i and returns the signed type INT.
  -- for example, on a 32 bit word machine :
  --  2^31 − 1 becomes 2^31 − 1
  --  2^31 becomes −2^31

  INT result :
  VALOF
    IF
      (result.i > maxint) AND (result.i < range)
        result := INT (result.i - range)
      TRUE
        result := INT result.i
    RESULT result
:
```

J.1 The integer arithmetic functions

LONGADD performs the addition of signed quantities with a carry in. The function is invalid if arithmetic overflow occurs.

The action of the function is defined as follows:

```
INT FUNCTION LONGADD (VAL INT left, right, carry.in)

  -- Adds (signed) left word to right word with least significant bit of carry.in.

  INTEGER sum.i, carry.i, left.i, right.i :
  VALOF
    SEQ
      carry.i := INTEGER (carry.in /\ 1)
      left.i  := INTEGER left
      right.i := INTEGER right
      sum.i   := (left.i + right.i) + carry.i
    -- overflow may occur in the following conversion
    -- resulting in an invalid process
    RESULT INT sum.i
:
```

LONGSUM performs the addition of unsigned quantities with a carry in and a carry out. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGSUM (VAL INT left, right, carry.in)

  -- Adds (unsigned) left word to right word with the least significant bit of carry.in.
  -- Returns two results, the first value is one if a carry occurs, zero otherwise,
  -- the second result is the sum.

  INT carry.out :
  INTEGER sum.i, left.i, right.i :
  VALOF
    SEQ
      left.i := unsign (left)
      right.i := unsign (right)
      sum.i := (left.i + right.i) + INTEGER (carry.in /\ 1)
      IF                    -- assign carry
        sum.i >= range
          SEQ
            sum.i := sum.i - range
            carry.out := 1
        TRUE
          carry.out := 0
    RESULT carry.out, sign (sum.i)
:
```

LONGSUB performs the subtraction of signed quantities with a borrow in. The function is invalid if arithmetic overflow occurs.

The action of the function is defined as follows:

```
INT FUNCTION LONGSUB (VAL INT left, right, borrow.in)

  -- Subtracts (signed) right word from left word and subtracts borrow.in from the result.

  INTEGER diff.i, borrow.i, left.i, right.i :
  VALOF
    SEQ
      borrow.i := INTEGER (borrow.in /\ 1)
      left.i   := INTEGER left
      right.i  := INTEGER right
      diff.i   := (left.i - right.i) - borrow.i
    -- overflow may occur in the following conversion
    -- resulting in an invalid process
    RESULT INT diff.i
:
```

LONGDIFF performs the subtraction of unsigned quantities with borrow in and borrow out. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGDIFF (VAL INT left, right, borrow.in)

  -- Subtracts (unsigned) right word from left word and subtracts borrow.in from the result.
  -- Returns two results, the first is one if a borrow occurs, zero otherwise,
  -- the second result is the difference.

  INTEGER diff.i, left.i, right.i :
  VALOF
    SEQ
      left.i := unsign (left)
      right.i := unsign (right)
      diff.i := (left.i - right.i) - INTEGER (borrow.in /\ 1)
      IF    -- assign borrow
        diff.i < 0
          SEQ
            diff.i := diff.i + range
            borrow.out := 1
        TRUE
          borrow.out := 0
    RESULT borrow.out, sign (diff.i)
:
```

LONGPROD performs the multiplication of two unsigned quantities, adding in an unsigned carry word. Produces a double length unsigned result. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGPROD (VAL INT left, right, carry.in)

  -- Multiplies (unsigned) left word by right word and adds carry.in.
  -- Returns the result as two integers most significant word first.

  INTEGER prod.i, prod.lo.i, prod.hi.i, left.i, right.i, carry.i :
  VALOF
    SEQ
      carry.i := unsign (carry.in)
      left.i  := unsign (left)
      right.i := unsign (right)
      prod.i    := (left.i * right.i) + carry.i
      prod.lo.i := prod.i REM range
      prod.hi.i := prod.i / range
    RESULT sign (prod.hi.i), sign (prod.lo.i)

:
```

LONGDIV divides an unsigned double length number by an unsigned single length number. The function produces an unsigned single length quotient and an unsigned single length remainder. An overflow will occur if the quotient is not representable as an unsigned single length number. The function becomes invalid if the divisor is equal to zero.

The action of the function is defined as follows:

```
INT, INT FUNCTION LONGDIV (VAL INT dividend.hi, dividend.lo, divisor)

  -- Divides (unsigned) dividend.hi and dividend.lo by divisor.
  -- Returns two results the first is the quotient and the second is the remainder.

  INTEGER divisor.i, dividend.i, hi, lo, quot.i, rem.i :
  VALOF
    SEQ
      hi := unsign (dividend.hi)
      lo := unsign (dividend.lo)
      divisor.i := unsign (divisor)
      dividend.i := (hi * range) + lo
      quot.i := dividend.i / divisor.i
      rem.i  := dividend.i REM divisor.i
    -- overflow may occur in the following conversion of quot.i
    -- resulting in an invalid process
    RESULT sign (quot.i), sign (rem.i)
:
```

SHIFTRIGHT performs a right shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. 0 <= **places** <= 2∗**bitsperword**

The action of the function is defined as follows:

```
INT, INT FUNCTION SHIFTRIGHT (VAL INT hi.in, lo.in, places)

  -- Shifts the value in hi.in and lo.in right by the given number of places.
  -- Bits shifted in are set to zero.
  -- Returns the result as two integers most significant word first.

  INT hi.out, lo.out :
  VALOF
    IF
      (places < 0) OR (places > (two*bitsperword))
        SEQ
          hi.out := 0
          lo.out := 0
      TRUE
        INTEGER operand, result, hi, lo :
        SEQ
          hi := unsign (hi.in)
          lo := unsign (lo.in)
          operand := (hi << bitsperword) + lo
          result  := operand >> places
          lo := result /\ wordmask
          hi := result >> bitsperword
          hi.out := sign (hi)
          lo.out := sign (lo)
    RESULT hi.out, lo.out
:
```

SHIFTLEFT performs a left shift on a double length quantity. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. 0 <= **places** <= 2∗**bitsperword**

The action of the function is defined as follows:

```
INT, INT FUNCTION SHIFTLEFT (VAL INT hi.in, lo.in, places)

  -- Shifts the value in hi.in and lo.in left by the given number of places.
  -- Bits shifted in are set to zero.
  -- Returns the result as two integers most significant word first.

  VALOF
    IF
      (places < 0) OR (places > (two*bitsperword))
        SEQ
          hi.out := 0
          lo.out := 0
      TRUE
        INTEGER operand, result, hi, lo :
        SEQ
          hi := unsign (hi.in)
          lo := unsign (lo.in)
          operand := (hi << bitsperword) + lo
          result  := operand << places
          lo := result /\ wordmask
          hi := result >> bitsperword
          hi.out := sign (hi)
          lo.out := sign (lo)
    RESULT hi.out, lo.out
:
```

NORMALISE normalises a double length quantity. No overflow can occur.

The action of the function is defined as follows:

```
INT, INT, INT FUNCTION NORMALISE (VAL INT hi.in, lo.in)

  -- Shifts the value in hi.in and lo.in left until the highest bit is set.
  -- The function returns three integer results
  -- The first returns the number of places shifted.
  -- The second and third return the result as two integers with the least significant word first;
  -- If the input value was zero, the first result is 2*bitsperword.

  INT places, hi.out, lo.out :
  VALOF
    IF
      (hi.in = 0) AND (lo.in = 0)
        places := INT (two*bitsperword)
      TRUE
        VAL msb IS one << ((two*bitsperword) - one) :
        INTEGER operand, hi, lo :
        SEQ
          lo := unsign (lo.in)
          hi := unsign (hi.in)
          operand := (hi << bitsperword) + lo
          places := 0
          WHILE (operand /\ msb) = 0
            SEQ
              operand := operand << one
              places := places + 1
          hi := operand / range
          lo := operand REM range
          hi.out := sign (hi)
          lo.out := sign (lo)
      RESULT places, hi.out, lo.out
:
```

J.2 Arithmetic shifts

ASHIFTRIGHT performs an arithmetic right shift, shifting in and maintaining the sign bit. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. 0 <= **places** <= **bitsperword**

No overflow can occur.

N.B the result of this function is NOT the same as division by a power of two.

e.g. $-1/2 = 0$
ASHIFTRIGHT (−1, 1**)** = −1

The action of the function is defined as follows:

```
-- Shifts the value in operand right by the given number of places.
-- The status of the high bit is maintained

INT FUNCTION ASHIFTRIGHT (VAL INT operand, places) IS
             INT( INTEGER(operand) >> places ) :
```

ASHIFTLEFT performs an arithmetic left shift. The function is invalid if significant bits are shifted out, signalling an overflow. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. 0 <= **places** <= **bitsperword**

N.B the result of this function is the same as multiplication by a power of two.

The action of the function is defined as follows:

```
INT FUNCTION ASHIFTLEFT (VAL INT argument, places)

  -- Shifts the value in argument left by the given number of places.
  -- Bits shifted in are set to zero.

  INTEGER result.i :
  VALOF
    result.i := INTEGER(argument) << places
    -- overflow may occur in the following conversion
    -- resulting in an invalid process
    RESULT INT result.i
:
```

J.3 Word rotation

ROTATERIGHT rotates a word right. Bits shifted out of the word on the right, re-enter the word on the left. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. 0 <= **places** <= **bitsperword**

No overflow can occur.

The action of the function is defined as follows:

```
INT FUNCTION ROTATERIGHT (VAL INT argument, places)

  -- Rotates the value in argument by the given number of places.

  INTEGER high, low, argument.i :
  VALOF
    SEQ
      argument.i := unsign(argument)
      argument.i := (argument.i * range) >> places
      high := argument.i / range
      low := argument.i REM range
    RESULT INT(high \/ low)
:
```

ROTATELEFT rotates a word left. Bits shifted out of the word on the left, re-enter the word on the right. The function must be called with the number of places in range, otherwise the implementation can produce unexpected effects.

i.e. 0 <= **places** <= **bitsperword**

The action of the function is defined as follows:

```
INT FUNCTION ROTATELEFT (VAL INT argument, places)

  -- Rotates the value in argument by the given number of places.

  INTEGER high, low, argument.i :
  VALOF
    SEQ
      argument.i := unsign(argument)
      argument.i := argument.i << places
      high := argument.i / range
      low := argument.i REM range
    RESULT INT(high \/ low)
:
```

K Floating point functions

Most of the floating point functions described in this appendix conform to the ANSI/IEEE standard 754-1985. However some of the functions act as invalid processes when called with arguments which are not finite (page 131). These behaviours do not conform to the standard and are noted as part of the description of each function.

Each function is specified by a skeletal function declaration, a predicate stating the relationship between the actual parameters after the function call and an informal textual description of the operation. All functions are implemented in a way which allows the same variable to be used as both the input and receiving variable in an assignment. The predicate gives the formal definition of the operation, although for most purposes the text will be an adequate explanation.

NaN and *Inf* are the sets of all Not-a-Numbers and all infinities in the format.

K.1 Not-a-number values

Floating point arithmetic implementations will return the following valued Not-a-Numbers to signify the various errors that can occur in evaluations.

Error	Single length value	Double length value
Divide zero by zero	#7FC00000	#7FF80000 00000000
Divide infinity by infinity	#7FA00000	#7FF40000 00000000
Multiply zero by infinity	#7F900000	#7FF20000 00000000
Addition of opposite signed infinities	#7F880000	#7FF10000 00000000
Subtraction of same signed infinities	#7F880000	#7FF10000 00000000
Negative square root	#7F840000	#7FF08000 00000000
REAL64 to **REAL32** NaN conversion	#7F820000	#7FF04000 00000000
Remainder from infinity	#7F804000	#7FF00800 00000000
Remainder by zero	#7F802000	#7FF00400 00000000

K.2 Absolute

```
REAL32 FUNCTION ABS(VAL REAL32 X)
  ...
:
REAL64 FUNCTION DABS(VAL REAL64 X)
  ...
:
```

ABS(**X**) = |**X**|

This returns the absolute value of **X**. This is implemented clearing the sign bit so that **-0.0** becomes **+0.0**. If **X** is a NaN or infinity, then calls to these functions act as invalid processes.

K.3 Square root

```
REAL32 FUNCTION SQRT(VAL REAL32 X)
  ...
:
REAL64 FUNCTION DSQRT(VAL REAL64 X)
  ...
:
```

$\mathbf{SQRT}(\mathbf{X}) = \sqrt{X}$

This returns the square root of **X**. If **X** is a negative number, NaN or infinity, then calls to these functions act as invalid processes.

K.4 Test for Not-a-Number

```
BOOL FUNCTION ISNAN(VAL REAL32 X)
  ...
:
BOOL FUNCTION DISNAN(VAL REAL64 X)
  ...
:
```

$\mathbf{ISNAN}(\mathbf{X}) = \mathbf{TRUE} \Leftrightarrow \mathbf{X} \in NaN$

This returns **TRUE** if **X** is a Not-a-Number and **FALSE** otherwise.

K.5 Test for Not-a-Number or infinity

```
BOOL FUNCTION NOTFINITE(VAL REAL32 X)
  ...
:
BOOL FUNCTION DNOTFINITE(VAL REAL64 X)
  ...
:
```

$\mathbf{NOTFINITE}(\mathbf{X}) = \mathbf{TRUE} \Leftrightarrow \mathbf{X} \in NaN \cup Inf$

This returns **TRUE** if **X** is a Not-a-Number or an infinity and **FALSE** otherwise.

K.6 Scale by power of two

```
REAL32 FUNCTION SCALEB(VAL REAL32 X, VAL INT n)
  ...
:
REAL64 FUNCTION DSCALEB(VAL REAL64 X, VAL INT n)
  ...
:
```

$\mathbf{SCALEB}(\mathbf{X}, \mathbf{n}) = \mathbf{X} \times 2^n$

This multiplies **X** by 2^n. Overflow and underflow behaviour is as for normal multiplication under the ANSI/IEEE standard 754-1985. **n** can take any value as the operation will return the correct result even when 2^n cannot be represented in the format. If **X** is a NaN or infinity, then calls to these functions act as invalid processes.

K.7 Return exponent of floating point number

```
REAL32 FUNCTION LOGB(VAL REAL32 X)
  ...
:
REAL64 FUNCTION DLOGB(VAL REAL64 X)
  ...
:
```

LOGB (X) = result
where $\mathbf{X} \notin Inf \cup NaN \wedge \mathbf{X} \neq 0 \Rightarrow result = \mathbf{REAL32}(\mathbf{X.exp} - Bias)$
$\mathbf{X} = 0 \Rightarrow result = -inf$
$\mathbf{X} \in Inf \Rightarrow result = +inf$
$\mathbf{X} \in NaN \Rightarrow result = \mathbf{X}$

This returns the exponent of **X** as an integer valued floating point number; special cases for Infs, NaNs and zero. **NOTE** that all denormalised numbers return the same value – this is not equivalent to rounding the logarithm to an integer value. If **X** is a NaN then it is returned as the result, if **X** is an infinity then the result is plus infinity and if **X** is zero then the result is minus infinity.

K.8 Unpack floating point value

```
INT, REAL32 FUNCTION FLOATING.UNPACK(VAL REAL32 X)
  ...
:
INT, REAL64 FUNCTION DFLOATING.UNPACK(VAL REAL64 X)
  ...
:
```

FLOATING.UNPACK (X) = (n, r)
where $\mathbf{X} = r \times 2^n \wedge r \in [1, 2)$

This "unpacks" **X** into a real (*r*) and an integer (*n*) so that *r* lies between 1 and 2 and that $\mathbf{X} = r \times 2^n$. This is useful for reducing a value to the primary range for "exponential" type functions. If **X** is zero, a NaN or infinity. then calls to these functions act as invalid processes. If one of these values is possible, then it must be tested for explicitly before calling these functions.

K.9 Negate

```
REAL32 FUNCTION MINUSX(VAL REAL32 X)
  ...
:
REAL64 FUNCTION DMINUSX(VAL REAL64 X)
  ...
:
```

MINUSX (X) = result
where $result.sign = toggle\mathbf{X}.sign, result.frac = \mathbf{X}.frac, result.exp = \mathbf{X}.exp$

This returns **X** with the sign bit toggled. This is not the same as (0 – **X**) as it has different behaviour on zero and NaNs. This should not be used as a unary negation where (0 – **X**) should be used. As with **ABS** it does affect the representation of NaNs even though they have no sign in their interpretation.

K.10 Copy sign

```
REAL32 FUNCTION COPYSIGN(VAL REAL32 X, Y)
  ...
:
REAL64 FUNCTION DCOPYSIGN(VAL REAL64 X, Y)
  ...
:
```

COPYSIGN (X,Y) = result
where $result.sign = \mathbf{Y}.sign, result.frac = \mathbf{X}.frac, result.exp = \mathbf{X}.exp$

This returns **X** with the sign bit from **Y**.

K.11 Next representable value

```
REAL32 FUNCTION NEXTAFTER(VAL REAL32 X,Y)
  ...
:
REAL64 FUNCTION DNEXTAFTER(VAL REAL64 X,Y)
  ...
:
```

NEXTAFTER (X,Y) = result
where $\mathbf{X} \in NaN \vee \mathbf{Y} \in NaN \Rightarrow result \in NaN \cap \{X, Y\}$
$\mathbf{X} = \mathbf{Y} \Rightarrow \mathbf{X}$
$\mathbf{X} \neq \mathbf{Y} \Rightarrow$ *"result is next real after* **X** *in the direction of* **Y***"*

This can be specified precisely but as several subsidiary definitions are required first the informal third line of the "predicate" is used for brevity.

This returns the first floating point number from **X** in the direction of **Y**. The major area where this will be used is in interval arithmetic. If either or both of **X** or **Y** is a NaN then a NaN equal to **X** or **Y** is returned. An overflow from a finite **X** to an infinite result is handled in the same way as an arithmetic overflow.

K.12 Test for orderability

```
BOOL FUNCTION ORDERED(VAL REAL32 X,Y)
  ...
:
BOOL FUNCTION DORDERED(VAL REAL64 X,Y)
  ...
:
```

$\mathbf{ORDERED}(\mathbf{X}, \mathbf{Y}) = \mathbf{TRUE} \Leftrightarrow \mathbf{X} \notin NaN \wedge \mathbf{Y} \notin NaN$

This returns **TRUE** if **X** and **Y** are "orderable" as defined by the ANSI/IEEE standard 754-1985. This implements the negation of the *unordered* comparison in ANSI/IEEE 754-1985 §5.7 and enables the full IEEE style comparison to be derived from the standard **<**, **>**, ... comparisons of real types in occam.

K.13 Perform range reduction

```
BOOL,INT32,REAL32 FUNCTION ARGUMENT.REDUCE(VAL REAL32 X, Y, Y.err)
  ...
:
BOOL,INT32,REAL64 FUNCTION DARGUMENT.REDUCE(VAL REAL64 X, Y, Y.err)
  ...
:
```

$\mathbf{ARGUMENT.REDUCE}(\mathbf{X}, \mathbf{Y}, \mathbf{error}) = (b, n, r)$

$$\begin{array}{lll} \textit{where} & \mathbf{X}.exp \leq \mathbf{Y}.exp + maxexpdiff \Rightarrow & b \wedge \mathbf{X} = n \times (\mathbf{Y} + \mathbf{error}) + r \\ & & \wedge(r < (\mathbf{Y} + \mathbf{error})/2 \vee (r = (\mathbf{Y} + \mathbf{error})/2 \wedge n\, MOD\, 2 = 0)) \\ & \mathbf{X}.exp > \mathbf{Y}.exp + maxexpdiff \Rightarrow & \exists m : \mathbf{Z} \\ & & \neg b \wedge \mathbf{X} = m \times \mathbf{Y} + r \\ & & \wedge(r < \mathbf{Y}/2 \vee (r = \mathbf{Y}/2 \wedge m\, MOD\, 2 = 0)) \\ & & \wedge n = undefined \end{array}$$

where $maxexpdiff$ is 20 for **ARGUMENT.REDUCE** and 30 for **DARGUMENT.REDUCE**.

This performs a more accurate remainder **X REM Y** by using an extended precision value for **Y** where possible. It is assumed that **error** is no larger than a last bit error in **Y**. **TRUE** is returned as the boolean result *b* to indicate that the more accurate remainder has been done and the integer result *n* will then be the quotient. If the more accurate remainder cannot be done a normal remainder is performed and the quotient *n* must be calculated separately. This is designed to be used to reduce an argument to the primary range for cyclical functions - such as the trigonometric functions.

K.14 Fast multiply by two

```
REAL32 FUNCTION MULBY2(VAL REAL32 X)
  ...
:
REAL64 FUNCTION DMULBY2(VAL REAL64 X)
  ...
:
```

$\mathbf{MULBY2}(\mathbf{X}) = \mathbf{X} \times 2$

This returns 2 times **X** with overflow handling as defined in the ANSI/IEEE standard 754-1985, except that if **X** is a NaN or infinity, then calls to these functions act as invalid processes.

K.15 Fast divide by two

```
REAL32 FUNCTION DIVBY2(VAL REAL32 X)
  ...
:
REAL64 FUNCTION DDIVBY2(VAL REAL64 X)
  ...
:
```

$\mathbf{DIVBY2}(\mathbf{X}) = \mathbf{X} \div 2$

This returns **X** divided by 2 with underflow handling as defined in the ANSI/IEEE standard 754-1985, except that if **X** is a NaN or infinity, then calls to these functions act as invalid processes.

K.16 Round to floating point integer

```
REAL32 FUNCTION FPINT(VAL REAL32 X)
  ...
:
REAL64 FUNCTION DFPINT(VAL REAL64 X)
  ...
:
```

FPINT (X) = result

where $|\mathbf{X}| \geq 2^{bitsperword} \Rightarrow result = \mathbf{X}$

$|\mathbf{X}| < 2^{bitsperword} \Rightarrow result =$ **REAL32(INT ROUND X)**

This returns **X** rounded to a floating point integer value. If **X** is a NaN or infinity, then calls to these functions act as invalid processes.

L Full IEEE floating point arithmetic

REALOP and **REALREM** are implementations of the ANSI/IEEE 754-1985 floating point arithmetic standard. An implementation should comply to the requirements of the standard in as much as all results returned by them should be correct as defined in the standard. Most programmers will not need to use these functions directly as most occam implementations will compile all real arithmetic as calls to these functions. In some applications, such as interval arithmetic, the rounding modes are needed so the functions will need to be explicitly called in those cases. Also, in some applications, the IEEE standards use of infinities and Not-a-number to handle errors and overflows may be required in preference to the standard occam treatment of them as invalid expressions.

L.1 ANSI/IEEE real arithmetic operations

The functions for **REAL32** operands are

```
REAL32 FUNCTION REAL32OP (VAL REAL32 X, VAL INT Op, VAL REAL32 Y)
  ...
:

REAL32 FUNCTION REAL32REM (VAL REAL32 X, VAL REAL32 Y)
  ...
:
```

REAL32OP (X, Op, Y) evaluates $X\ Op\ Y$ according to the standard without error checking, using the conventional rounding mode. The various operations are coded in **Op** where:

op	= 0	**+**
	= 1	**–**
	= 2	*****
	= 3	**/**

REAL32REM (X, Y) evaluates X **REM** Y according to the standard without error checking.

REAL64OP and **REAL64REM** are defined in an similar manner to operate on **REAL64**s.

IEEExxOP (X, Rm, Op, Y) evaluates $X\ Op\ Y$ according to the standard without error checking. The rounding mode to be used is indicated by **Rm** where:

round_mode	= 0	Round to Zero
round_mode	= 1	Round to Nearest
round_mode	= 2	Round to Plus Infinity
round_mode	= 3	Round to Minus Infinity

The functions are:

```
BOOL, REAL32 FUNCTION IEEE32OP (VAL REAL32 X, VAL INT Rm, Op, VAL REAL32 Y)
  ...
:
BOOL, REAL64 FUNCTION IEEE64OP (VAL REAL64 X, VAL INT Rm, Op, VAL REAL64 Y)
  ...
:
```

IEEExxREM (X, Y) evaluates X **REM** Y according to the standard . The functions are:

```
BOOL, REAL32 FUNCTION IEEE32REM (VAL REAL32 X, VAL REAL32 Y)
  ...
:
BOOL, REAL64 FUNCTION IEEE64REM (VAL REAL64 X, VAL REAL64 Y)
  ...
:
```

The functions return a boolean which is true if an error has occurred, and false otherwise, and the result.

L.2 ANSI/IEEE real comparison

The comparisons on the real types provided in the occam language should suffice for most purposes. However, if the comparisons detailed in the ANSI/IEEE 754-1985 standard are required then they can be generated from the set of primitive comparisons.

```
BOOL FUNCTION REAL32EQ (VAL REAL32 X, Y)
  -- result = (X = Y) in the IEEE sense
  ...
:
BOOL FUNCTION REAL32GT (VAL REAL32 X, Y)
  -- result = (X > Y) in the IEEE sense
  ...
:
```

A standard function **IEEECOMPARE** will return a value which indicates which of the relations *less than*, *greater than*, *equals* or *unordered* as defined by IEEE 754 paragraph 5.7. This procedure is

```
INT FUNCTION IEEECOMPARE (VAL REAL32 X, Y)
  INT result :
  VALOF
    IF
      ORDERED (X, Y)
        IF
          REAL32EQ (X, Y)
            result := 0
          REAL32GT (X, Y)
            result := 1
          TRUE
            result := -1
      TRUE
        result := 2
    RESULT result
:
```

Then, if really necessary, any of the 26 varieties of comparison suggested by the IEEE standard can be derived. For instance the $? >=$ predicate could be implemented by

```
BOOL, BOOL FUNCTION IEEE.UGE. (VAL REAL32 X,Y)
  VAL LT IS -1, EQ IS 0, GT IS 1, UN IS 2:
  INT relation :
  VALOF
    relation := IEEECOMPARE (X, Y)
    RESULT FALSE,
           (relation=GT) OR ((relation=EQ) OR (relation=UN))
:
```

Similarly $NOT(<>)$ could be implemented as

```
BOOL, BOOL FUNCTION IEEENOT.LG. (VAL REAL32 X,Y)
  VAL LT IS -1, EQ IS 0, GT IS 1, UN IS 2:
  INT relation :
  VALOF
    relation := IEEECOMPARE (X, Y)
    RESULT (relation=UN), (relation=EQ) OR (relation=UN)
:
```

In either of these cases the value returned in the first boolean is equivalent to the invalid operation flag being set according to the ANSI/IEEE standard 754-1985.

The double length version **DIEEECOMPARE** is defined in a similar manner to **IEEECOMPARE**.

M Elementary functions

The elementary function library provides a set of routines which provide elementary functions compatible with the ANSI/IEEE standard 754-1985 for binary floating-point arithmetic.

All single length functions other than **POWER**, **ATAN2** and **RAN** have one parameter which is a **VAL REAL32** taking the argument of the function. **POWER** and **ATAN2** have two parameters. They are both **VAL REAL32**s which receive the arguments of the function. **RAN** has a single parameter which is a **VAL INT32**. In each case the double-length version is obtained by prefixing a **D** onto the function name, whose parameters are **VAL REAL64** or, in the case of **DRAN**, **VAL INT64**.

Accompanying the description of each function is the specification of the function's *Domain* and *Range*. The *Domain* specifies the range of valid inputs, *ie* those for which the output is a normal or denormal floating-point number. The *Range* specifies the range of outputs produced by all arguments in the *Domain*. The given endpoints are not exceeded. Note that some of the domains specified are implementation dependent.

Ranges are given as intervals, using the convention that a square bracket { [or] } means that the adjacent endpoint is included in the range, whilst a round bracket { (or) } means that it is excluded. Endpoints are given to a few significant figures only. Where the range depends on the floating-point format, single-length is indicated with an S and double-length with a D.

For functions with two arguments the complete range of both arguments is given. This means that for each number in one range, there is at least one (though sometimes only one) number in the other range such that the pair of arguments is valid. Both ranges are shown, linked by an 'x'.

In the specifications, XMAX is the largest representable floating-point number: in single-length it is approximately $3.4 * 10^{38}$, and in double-length it is approximately $1.8 * 10^{308}$. Pi means the closest floating-point representation of the transcendental number π, ln(2) the closest representation of $\mathbf{log_e(2)}$, and so on. In describing the algorithm, X is used generically to designate the argument, and "result" to designate the output.

The routines will accept any value, as specified by the IEEE standard, including special values representing **NaN**s ('Not a Number') and **Inf**s ('Infinity'). **NaN**s are copied directly to the result, whilst **Inf**s may or may not be valid arguments. Valid arguments are those for which the result is a normal (or denormalised) floating-point number.

Arguments outside the domain (apart from **NaNs** which are simply copied to the result) give rise to *exceptional results*, which may be **NaN**, **+Inf**, or −**Inf**. **Inf**s mean that the result is mathematically well-defined but too large to be represented in the floating-point format.

Error conditions are reported by means of three distinct **NaN**s :

undefined.NaN	This means that the function is mathematically undefined for this argument, for example the logarithm of a negative number.
unstable.NaN	This means that a small change in the argument would cause a large change in the value of the function, so any error in the input will render the output meaningless.
inexact.NaN	This means that although the mathematical function is well-defined, its value is in range, and it is stable with respect to input errors at this argument, the limitations of word-length (and reasonable cost of the algorithm) make it impossible to compute the correct value.

Implementations will return the following values for these Not-a-Numbers:

Error	Single length value	Double length value
undefined.NaN	#7F800010	#7FF00002 00000000
unstable.NaN	#7F800008	#7FF00001 00000000
inexact.NaN	#7F800004	#7FF00000 80000000

In all cases, the function returns a **NaN** if given a **NaN**.

M.1 Logarithm

```
REAL32 FUNCTION ALOG (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DALOG (VAL REAL64 X)
  ...
:
```

These compute : result = $\mathbf{log_e}(\mathbf{X})$.

Domain : (0, XMAX]
Range : [MinLog, MaxLog] = $[-103.28, 88.72]S$ = $[-745.2, 709.78]D$

All arguments outside the domain generate an **undefined.NaN**.

M.2 Base 10 logarithm

```
REAL32 FUNCTION ALOG10 (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DALOG10 (VAL REAL64 X)
  ...
:
```

These compute : result = $\mathbf{log_{10}}(\mathbf{X})$

Domain : (0, XMAX]
Range : [MinLog10, MaxLog10] = $[-44.85, 38.53]S$ = $[-323.6, 308.25]D$

All arguments outside the domain generate an **undefined.NaN**.

M.3 Exponential

```
REAL32 FUNCTION EXP (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DEXP (VAL REAL64 X)
  ...
:
```

These compute : result = e^X.

Domain : [−Inf, MaxLog) = $[-Inf, 88.72)S$, = $[-Inf, 709.78)D$
Range : [0, XMAX)

If the result is too large to be represented in the floating-point format, **Inf** is returned.

M.4 X to the power of Y

```
REAL32 FUNCTION POWER (VAL REAL32 X, Y)
  ...
:
REAL64 FUNCTION DPOWER (VAL REAL64 X, Y)
  ...
:
```

These compute : result = X^Y.

Domain : [0, Inf] x [$-$Inf, Inf]
Range : [$-$Inf, Inf]

If the result is too large to be represented in the floating-point format, **Inf** is returned. If X or Y is **NaN**, **NaN** is returned. Other special cases are as follows :

First Input (X)	Second Input (Y)	Result
$X < 0$	any	**undefined.NaN**
0	≤ 0	**undefined.NaN**
0	$0 < Y \leq$ XMAX	0
0	**Inf**	**unstable.NaN**
$0 < X < 1$	**Inf**	0
$0 < X < 1$	$-$**Inf**	**Inf**
1	$-$XMAX $\leq Y \leq$ XMAX	1
1	$\pm$ **Inf**	**unstable.NaN**
$1 < X \leq$ XMAX	**Inf**	**Inf**
$1 < X \leq$ XMAX	$-$**Inf**	0
Inf	$1 \leq Y \leq$ **Inf**	**Inf**
Inf	$-$**Inf**$\leq Y \leq -1$	0
Inf	$-1 < Y < 1$	**undefined.NaN**
otherwise	0	1
otherwise	1	X

M.5 Sine

```
REAL32 FUNCTION SIN (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DSIN (VAL REAL64 X)
  ...
:
```

These compute : result = sine(X) (where X is in radians).

Domain : [$-$Smax, Smax] $= [-12868.0, 12868.0]S, = [-2.1 * 10^8, 2.1 * 10^8]D$
Range : [$-$1.0, 1.0]

All arguments outside the domain generate an **inexact.NaN**. Implementations may provide a larger domain.

M.6 Cosine

```
REAL32 FUNCTION COS (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DCOS (VAL REAL64 X)
  ...
:
```

These compute : result = cosine(X)   (where X is in radians).

Domain : [−Smax, Smax] = $[-12868.0, 12868.0]S$, = $[-2.1 * 10^8, 2.1 * 10^8]D$
Range : [−1.0, 1.0]

All arguments outside the domain generate an **inexact.NaN**. Implementations may provide a larger domain.

M.7 Tangent

```
REAL32 FUNCTION TAN (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DTAN (VAL REAL64 X)
  ...
:
```

These compute : result = tan(X)   (where X is in radians).

Domain : [−Tmax, Tmax] = $[-6434.0, 6434.0]S$, = $[-1.05 * 10^8, 1.05 * 10^8]D$
Range : (−Inf, Inf)

All arguments outside the domain generate an **inexact.NaN**. Implementations may provide a larger domain.

M.8 Arcsine

```
REAL32 FUNCTION ASIN (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DASIN (VAL REAL64 X)
  ...
:
```

These compute : result = sine$^{-1}(X)$ (in radians).

Domain : [−1.0, 1.0]
Range : [−Pi/2, Pi/2]

All arguments outside the domain generate an **undefined.NaN**.