occam® 2.1 reference manual

SGS-THOMSON Microelectronics Limited

May 12, 1995

occam 2.1 REFERENCE MANUAL

SGS-THOMSON Microelectronics Limited

First published 1988 by Prentice Hall International (UK) Ltd as the occam 2 Reference Manual.

Document number: 72 occ 45 03

Contents

Contents overview

Preliminaries

The chapters

Appendices

Preface

The occam programming language is designed to express concurrent algorithms and their implementation on a network of processing components.

The occam 2.1 *Reference Manual* serves to provide a single reference and definition of the occam 2.1 language. The manual describes each aspect of the language, starting with the most primitive components of an occam program, and moving on to cover the whole language in detail. The manual is addressed to the wider audience, including not only the computer scientist, software engineer and programmer, but also the electronics engineer and system designer. This manual describes the extended version of the language called occam 2.1 which was defined in 1994; it has undergone extensive revision since the original occam 2 edition of the book published in 1988.

occam enables an application to be described as a collection of *processes*, where the processes execute concurrently, and communicate with each other through *channels*. Each process in such an application describes the behaviour of a particular aspect of the implementation, and each channel describes a connection between two processes. This approach has two important consequences. Firstly, it gives the program a clearly defined and simple structure. Secondly, it allows the application to exploit the performance of a system which consists of many parts.

Concurrency and communication are the prime concepts of the occam model. occam captures the hierarchical structure of a system by allowing an interconnected set of processes to be regarded as a unified, single process. At any level of detail, the programmer is only concerned with a small, manageable set of processes.

occam is an ideal introduction to a number of key methodologies in modern computer science. occam programs can provide a degree of security unknown in conventional programming languages such as C, FORTRAN or Pascal. occam semantics simplify the task of program verification, by allowing application of mathematical proof techniques to prove the correctness of programs. Transformations, which convert a process from one form to a directly equivalent form, can be applied to the source of an occam program to improve its efficiency in any particular environment. occam makes an ideal language for specification and behavioural description. occam programs are easily configured onto the hardware of a system or indeed, may specify the hardware of a system.

occam has a minimalist approach which avoids unnecessary duplication of language mechanism, and is named after the 14th century philosopher William of Occam who proposed that invented entities should not be duplicated beyond necessity. This proposition has become known as “Occam’s razor”.

The occam programming language arises from the concepts founded by David May in EPL (Experimental Programming Language) and Tony Hoare in CSP (Communicating Sequential Processes). Since its conception in 1982 occam has been, and continues to be, under development at INMOS Limited, now SGS-THOMSON Microelectronics Limited in the United Kingdom. The development of the INMOS transputer, a family of devices which place one or more microcomputers on a single chip, has been closely related to occam, its design and implementation. The transputer reflects the occam architectural model, and may be considered an occam machine. However, this manual does not make any assumptions about the hardware implementation of the language or the target system.

Introduction

This manual describes the programming language occam 2.1. occam 2.1 is an extension of occam 2 with new constructs to support named and structured data types. In particular, occam 2.1 has mechanisms for the definition of such data types, for efficient access to their components and for the construction of literal values of these types. Many other minor improvements to the occam 2 language described in earlier editions of this manual have also been made in response to the comments of a wide community of users in many countries. New features are not explicilty noted as such in the body of the manual, but are summarised in appendix P.

The first edition of this manual was completed during 1986 and 1987 in parallel with the development of the occam 2 language at the INMOS Microcomputer Centre, Bristol, UK.

This revised edition was prepared in 1994 to support the work on language extensions to define occam 2.1 by INMOS Limited, now SGS-THOMSON Microelectronics Group Limited, at Bristol.

In the rest of this manual the name occam should be understood to mean the extended occam 2.1 language. When necessary the previous version of the language will be explicitly referred to as occam 2.

Using this manual

This book is designed primarily to be used as a reference text for the programming language occam 2.1. However, the manual should also serve as an introduction to the language for someone with a reasonable understanding of programming languages. The primitive aspects of the language are presented at the start of the manual, with as few forward references as possible. It is therefore possible to read the manual from cover to cover, giving the reader an insight into the language as a whole. The manual is cross referenced throughout, and a glossary of terms, a bibliography and a comprehensive index are provided at the end of the manual.

Keywords and example program fragments appear in a **bold program font** throughout, for example:

```
-- example program fragment
IF
  occam
    programming := easy
```

Words which appear in *italic* often indicate a syntactic object, but may also serve to emphasise a need to cross reference and encourage referral to the index. Mathematical symbols and names referring to mathematical values use a $roman\ italic\ font$.

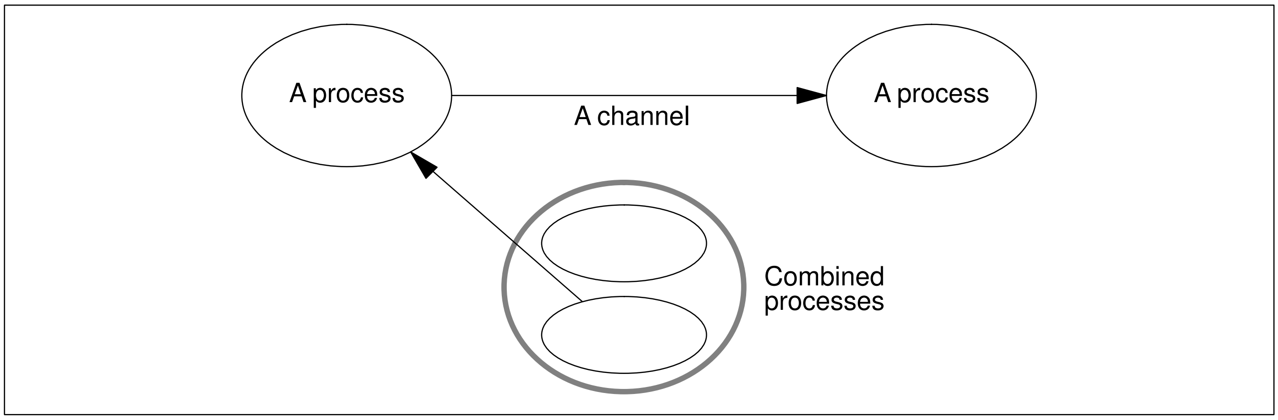

Figure 0.1 Figure conventions

Figures are used in the chapter on constructed processes to illustrate examples. They use the following conventions: an arrowed line represents a *channel*, a round cornered box represents a *process* (referred to here as a *process box*), and a lighter coloured process box combines a number of smaller processes. The conventions are illustrated in figure 0.1.

Syntax and program format

Syntactic notation

The syntax of occam programs is described in a simple metalanguage close to that known as Backus-Naur Form (BNF) from its introduction by those authors in the definition of Algol60. Syntax is described in terms of a set of *productions* each defining the meaning of a syntactic category (or *non-terminal symbol*) in terms of a sequence composed of literal language elements (*terminal symbols*) and further syntactic categories. The names used for syntactic categories are made up of lower case letters and dots and are always printed in an *italic* font. As an example, the following shows the syntax of the syntactic category *assignment*, discussed on page 5:

assignment	=	*variable* **`:=`** *expression*

This *production* means "An assignment is a *variable* followed by the symbol **`:=`**, followed by an *expression*". Where two or more characters in **`program font`** appear adjacent in a production, e.g. **`:=`** or **`SEQ`**, they comprise a *symbol* or a *keyword*. Symbols and keywords are lexical units and may not contain embedded spaces (see appendix G).

A vertical bar (**|**) means "or", so for example:

specification	=	*definition*
	|	*declaration*
	|	*abbreviation*

is the same as:

specification	=	*definition*
specification	=	*declaration*
specification	=	*abbreviation*

The meaning of this syntax is "A specification is a *definition*, a *declaration*, or an *abbreviation*".

The written structure of occam programs is specified by the syntax. Each statement in an occam program normally occupies a single line, and the indentation of each statement forms an intrinsic part of the syntax of the language. The following example shows the syntax for *sequence* discussed on page 9:

```
sequence        =   SEQ
                      { process }
```

The syntax here means "A sequence is the keyword **`SEQ`** followed by zero or more processes, each on a separate line, and indented two spaces beyond **`SEQ`**". Instead of BNF's recursive definitions, curly brackets { and } are used to indicate that a syntactic object may occur a number of times. There are two styles. { *process* } means, "zero or more processes, each on a separate line". $\{_0$ **`,`** *expression* }, means "A list of zero or more expressions, separated by commas", and $\{_1$ **`,`** *expression* }, means "A list of one or more expressions, separated by commas".

It is important to note that spaces at the start of a line are significant as they define the indentation which determines program structure. After the first visible lexical unit, arbitrary numbers of additional spaces may be added between units to improve readability. Leading spaces in groups of 8 may be replaced by TAB characters.

Syntactic rules must always be considered in conjunction with relevant semantic rules which are given informally in words following each group of productions. For example the production above defining *assignment* is qualified by a semantic rule stating that the data types of the expression being assigned and of the variable must be the same.

A complete summary of the lexical units and syntax of the language is given in appendices (starting on page 102). This summary refers back to the pages where supporting semantic rules are given.

The syntax of occam presented in this manual includes definitions of some syntactic categories that may be more conveniently handled by lexical analysis. An implementation is free to choose its own distinction between lexical and syntactic analysis as long as the overall effect is to accept all *legal* occam programs specified by this manual and to reject all illegal ones with appropriate error messages.

Continuation lines

A long statement may be broken immediately after one of the following:

an operator	e.g. **+, −, *, /, REM,** etc.. (see page 58)
a comma	,
a semi-colon	;
assignment	:=
one of the keywords	**FROM**, **FOR**, **IS**, **RETYPES** or **RESHAPES**

A statement can be broken over several lines providing the continuation is indented at least as much as the first line of the statement.

The annotation of occam programs

As the format of occam programs is significant, there are a number of rules concerning how programs are annotated. A comment is introduced by a double dash symbol (--), and extends to the end of the line. Consider the following sequence:

```
SEQ
  -- This example illustrates the use of comments
  -- A comment may not be indented less than
     --      the following statement
  ...
  SEQ        -- A sequence
    ...
```

Comments may not be indented less than a legal following statement, i.e. a statement conforming the syntax and the supporting semantic rules.

A sequence of three dots ... is used in this manual to indicate any occam source code whose contents are not of importance to the reader at this point.

Names and keywords used in occam programs

Names used in occam programs must begin with an alphabetic character. Names consist of a sequence of alphanumeric characters and dots. There is no length restriction. occam is sensitive to the case of names, e.g. **Say** is considered different from **say**. With the exception of the names of channel protocols and user-defined types, names in the examples presented in this manual are usually all lower case. However, the following are all legal names in occam:

PACKETS	**transputer**
vector6	**terminal.in**
LinkOut	**terminalOut**
NOT.A.NUMBER	**dotty..**

All keywords are upper case (e.g. **SEQ**), possibly with digits (e.g. **REAL64**). All keywords are reserved, and thus may not be used by the programmer. A full list of the keywords appears on page 102. The names of library routines are given in the appendix starting on page 116; these are not reserved but should only be reused if new routines for the same purpose are being defined.

1 Primitive processes

1.1 Assignment

occam programs are built from processes. The simplest process in an occam program is an *action*. An action is either an *assignment*, an *input* or an *output*. Consider the following example:

```
x := y + 2
```

This simple example is an *assignment*, which assigns the value of the expression **y + 2** to the variable **x**. The syntax of an assignment is:

assignment	=	*variable* **:=** *expression*

The *variable* on the left of the assignment symbol (**:=**) is assigned the value of the *expression* on the right of the symbol. The value of the expression must be of the same *data type* as the variable to which it is to be assigned; otherwise the assignment is not legal.

Variables are discussed on page 35, data types are discussed on page 25, and expressions on page 57.

A *multiple assignment* assigns values to several variables, as illustrated in the following example:

```
a, b, c := x, y + 1, z + 2
```

This assignment assigns the values of **x**, **y + 1** and **z + 2** to the variables **a**, **b** and **c** respectively. The expressions on the right of the assignment are evaluated, and the assignments are then performed in parallel. Consider the following example:

```
x, y := y, x
```

The effect of this multiple assignment is to swap the values of the variables **x** and **y**.

The syntax of multiple assignment extends the syntax for assignment:

assignment	=	*variable.list* **:=** *expression.list*
variable.list	=	$\{_1$ **,** *variable* $\}$
expression.list	=	$\{_1$ **,** *expression* $\}$

A list of expressions appearing to the right of the assignment symbol (**:=**) is evaluated in parallel, and then each value is assigned (in parallel) to the corresponding variable of the list to the left of the symbol.

The expression on the right of the assignment symbol (**:=**) may be a function instance, or a value process in parentheses.

An instance of a function, or a value process, with multiple results can also be an expression list in a multiple assignment. It may not be any combination of these three. Value processes and functions are discussed in chapter 8 starting on page 75, where examples of such multiple assignments are given.

The rules which govern the names used in the variable list of a multiple assignment follow from those for names used in parallel constructions (see page 16). Practically, this means that no variable may appear twice on the left hand side of a multiple assignment, nor may any variable in a such a variable list also appear in an expression (page 37) which selects a component from an array or defines a segment of an array used in the variable list.

1.2 Communication

Communication is an essential part of occam programming. Values are passed between concurrent processes by communication on *channels*. Each channel provides unbuffered, unidirectional point-to-point communication between two concurrent processes. The format and *type* of communication on a channel is specified by a *channel protocol* referenced in the *declaration* of the channel. Channel protocols are discussed in section 5.4, and channel declarations are discussed on page 45.

Two *actions* exist in occam which perform communication on a channel. They are: *input* and *output*.

1.2.1 Input

An *input* receives a value from a *channel* and assigns the received value to a *variable*. Consider the following example:

```
keyboard ? char
```

This simple example receives a value from the channel named **keyboard** and assigns the value to the variable **char**. The input waits until a value is received.

The syntax of an input is:

input = *channel* **?** *variable*

An input receives a value from the channel on the left of the input symbol (**?**), and assigns that value to the variable on the right of the symbol. The value input must conform to the channel protocol and be of the same *data type* as the variable to which it is assigned, otherwise the input is not legal. Variables are discussed on page 35, and data types are discussed on page 25.

1.2.2 Output

An *output* transmits the value of an *expression* to a *channel*. Consider the following example:

```
screen ! char
```

This simple example transmits the value of the variable **char** to the channel named **screen**. The output waits until the value has been received by a corresponding input on the same channel.

The syntax of an output is:

output = *channel* **!** *expression*

An output transmits the value of the expression on the right of the output symbol (**!**) to the channel named on the left of the symbol. The value output must conform to the channel protocol, otherwise the output is not valid.

Variables are discussed on page 35 and expressions on page 57.

1.3 SKIP and STOP

The primitive process **SKIP** starts, performs no action and terminates.

The primitive process **STOP** starts, performs no action and never terminates.

To explain how **SKIP** behaves, consider the following *sequence* (sequences are introduced on page 9):

```
SEQ
  keyboard ? char
  SKIP
  screen   ! char
```

This sequence executes the input **keyboard ? char**, then executes **SKIP**, which performs no action. The sequence continues, and the output **screen ! char** is executed.

The behaviour of **STOP** is illustrated by the following sequence:

```
SEQ
  keyboard ? char
  STOP
  screen   ! char
```

This sequence performs the input **keyboard ? char** as before, then executes **STOP**, which starts but does not terminate and so does not allow the sequence to continue. The output **screen ! char** is never executed.

occam programs which are syntactically legal (page 4) but for some reason or under some circumstances are semantically *invalid* are said to behave like the process **STOP** (see also appendix F). The word *invalid* should normally be assumed to have this meaning in this manual. Note that all valid constructs are legal, but that all legal constructs are not valid.

2 Constructed processes

occam programs are built from processes. Primitive processes are described in the previous chapter. Larger processes are built by combining smaller processes in a *construction*. A construction builds a process of one of the following kinds:

SEQ	sequence
IF	conditional
CASE	selection
WHILE	loop
PAR	parallel
ALT	alternation

A sequential process is built by combining processes in a sequence, conditional or selection construction. A loop is built by combining processes in a **WHILE** loop. Concurrent processes are built with parallel and alternation constructions, and communicate by inputs and outputs using channels.

The constructions **SEQ**, **IF**, **PAR** and **ALT** can all be *replicated*. A replicated construction *replicates* the constructed *process*, *choice* or *alternative* a specified number of times. The application of replication to each of these constructions is given in the following sections.

2.1 Sequence

A sequence combines processes into a construction in which one process follows another. Consider the following example:

```
SEQ
  keyboard ? char
  screen   ! char
```

This process combines two processes which are performed sequentially. The input **keyboard ? char** receives a value which is assigned to the variable **char**, then the following output **screen ! char** is performed.

Programs are built by constructing larger processes from smaller ones. Thus a construction may contain other constructions, as shown in the following example:

```
SEQ
  SEQ
    screen ! '?'
    keyboard ? char
  SEQ
    screen ! char
    screen ! cr
    screen ! lf
```

This simple example combines five processes, and suggests how embedded sequences may be used to show the hierarchical structure of a program. Embedding constructions of the same kind has no effect on the behaviour of the process. This example is equivalent to the following:

```
SEQ
  screen ! '?'
  keyboard ? char
  screen ! char
  screen ! cr
  screen ! lf
```

The syntax for a sequence is:

```
sequence          =  SEQ
                       { process }
```

The keyword **SEQ** is followed by zero or more processes at an indentation of two spaces.

2.1.1 Replicated sequence

A sequence can be *replicated* to produce a number of similar processes which are performed in sequence. A replicated sequence is used for a counted loop in which the number of repetitions is known at the start of the loop. This type of loop should be contrasted with the **WHILE** loop described below (page 14) which is terminated only when a certain condition becomes false. Consider the following:

```
SEQ i = 0 FOR array.size
  stream ! data.array[i]
```

This example uses an *array*; arrays are explained later in the manual (page 30). The process performs the output **stream ! data.array[i]** the number of times specified by the value of **array.size**. The initial value of the *replication index* **i** is the value of the base (in this case 0). The replication index is incremented by 1 after each execution of the output. If **array.size** has the value 2, the example can be expanded to show the effect of the replication as follows:

```
SEQ
  stream ! data.array[0]
  stream ! data.array[1]
```

Consider the following example in which the base value is 14:

```
SEQ i = 14 FOR 2
  stream ! data.array[i]
```

This example may also be expanded to show the value of the index for each replication, as follows:

```
SEQ
  stream ! data.array[14]
  stream ! data.array[15]
```

Arrays may also be communicated in a single output (see page 47).

The syntax for a replicated sequence extends the syntax for sequences:

sequence	=	**SEQ** *replicator* *process*
replicator	=	*name* **=** *base* **FOR** *count*
base	=	*expression*
count	=	*expression*

The keyword **SEQ** and a replicator are followed by a process which is indented two spaces. The replicator appears to the right of the keyword **SEQ**. The replicator specifies a name for the index (i.e. the name does not need to be declared elsewhere). The value of the index for the first replication is the value of the *base* expression, and the number of times the process is replicated is the value of the *count* expression at the start of the sequence.

The index may be used in expressions but cannot be changed by an input or assignment. The index has a value of *type* **INT**. The base and count expressions must also be of data type **INT**. Data types (page 25) are explained later in the the manual. If the count expression has a negative value the process is *invalid*. See appendix F, page 101 for an explanation of how *invalid processes* behave. If the value of the count expression is zero, the replicated sequence behaves like the primitive process **SKIP** (page 7) and does nothing.

2.2 Conditional

A conditional combines a number of processes each of which is guarded by a boolean expression. The conditional evaluates the boolean expressions in sequence; if a boolean expression is found to be true the associated process is performed, and the conditional terminates. If none of the boolean expressions is true the conditional behaves like the primitive process **STOP** (page 7), for example:

```
IF
  x < y
    x := x + 1
  x >= y
    SKIP
```

Consider this example in detail: if `x < y` is true, the associated process `x := x + 1` is performed, however if the expression `x < y` is false, the next boolean expression `x >= y` is evaluated. If `x >= y` is true, then the associated process **SKIP** is performed. In this example, one of the boolean expressions must be true. However, consider the next example:

```
IF
  x < y
    x := x + 1
```

This conditional has a single component. If the expression `x < y` is false then the conditional will behave like the primitive process **STOP** (page 7). It is often convenient to use a form of conditional where the final choice is guaranteed to be performed, as illustrated by the following example:

```
IF
  x > y
    order := gt
  x < y
    order := lt
  TRUE
    order := eq
```

The expressions `x > y` and `x < y` will each be either true or false. The final expression uses the boolean constant **TRUE** which is always true, and acts as a catch-all which causes the associated process to be performed if none of the previous boolean expressions is true. In this context **TRUE** may be read as "otherwise".

The syntax for a conditional is:

conditional	=	**IF** { *choice* }
choice	=	*guarded.choice*
	|	*conditional*
guarded.choice	=	*boolean* *process*
boolean	=	*expression*

The keyword **IF** is followed by zero or more choices, indented two spaces. A choice is either a *guarded* choice or another conditional. A guarded choice is a boolean expression followed by a process, indented two spaces.

A choice which is itself a conditional has the same behaviour if "expanded" in a similar way to the embedded sequences shown earlier (page 9). Consider the following example:

```
IF
  IF
    x > y
      x := x + 1
  TRUE
    SKIP
```

This has the same effect as:

```
IF
  x > y
    x := x + 1
  TRUE
    SKIP
```

Boolean expressions (page 63) are discussed later in the manual.

2.2.1 Replicated conditional

A conditional may also be replicated, just as a sequence may (page 10). A replicated conditional constructs a number of similar choices. In a replicated conditional each choice may be guarded by a boolean expression involving the replication index. The following example compares the two strings **string** and **object**:

```
IF
  IF i = 1 FOR length
    string[i] <> object[i]
      found := FALSE
  TRUE
    found := TRUE
```

The first choice in this example is a replicated conditional. This has created a number of similar choices each guarded by a boolean expression comparing components of the array **string** and the array **object**. The replication may be expanded to show its meaning. If **length** has the value 2, this example has the same effect as:

```
IF
  IF
    string[1] <> object[1]
      found := FALSE
    string[2] <> object[2]
      found := FALSE
  TRUE
    found := TRUE
```

or

```
IF
  string[1] <> object[1]
    found := FALSE
  string[2] <> object[2]
    found := FALSE
  TRUE
    found := TRUE
```

The syntax for the replicated conditional is:

conditional	=	**IF** *replicator* *choice*
replicator	=	*name* = *base* **FOR** *count*
base	=	*expression*
count	=	*expression*

The keyword **IF** and a replicator are followed by a choice which is indented two spaces. The replicator appears to the right of the keyword **IF**. The replicator specifies a name for the index. The value of the index for the first replication is the value of the *base* expression, and the number of times the choice is replicated is the value of the *count* expression.

The index may be used in expressions but cannot be changed by an input or assignment. The index has a value of *type* **INT**. The base and count expressions must also be of data type **INT**. Data types (page 25) are explained later in the manual. If the count expression has a negative value the process is *invalid*. See appendix F, page 101 for an explanation of how *invalid processes* behave. If the value of the count expression is zero, the replicated conditional behaves like a conditional with no true conditions.

2.3 Selection

A selection combines a number of *options*, one of which is selected by matching the value of a *selector* with the value of a constant expression (called a *case expression*) associated with the option. Consider the following example:

```
CASE direction
  up
    x := x + 1
  down
    x := x - 1
```

In this example the value of **direction** is compared to the value of the case expressions **up** and **down**. If **direction** has a value equal to **up** then **x := x + 1** is performed; if **direction** has a value equal to **down** then **x := x - 1** is performed; however if no match is found, the selection behaves like the primitive process **STOP** (page 7). Several case expressions may be associated with a single option, for example:

```
CASE letter
  'a', 'e', 'i', 'o', 'u'
    vowel := TRUE
```

If **letter** has the value **'a'**, **'e'**, **'i'**, **'o'**, or **'u'**, then the variable **vowel** is assigned the value **TRUE**, otherwise the selection behaves like the primitive process **STOP**. Here it is useful to use a special form of selection where one of the *options* is guaranteed to be performed, as illustrated below:

```
CASE letter
  'a', 'e', 'i', 'o', 'u'
    vowel := TRUE
  ELSE
    vowel := FALSE
```

The process associated with **ELSE** in a selection will be performed if none of the case expressions match the selector.

The syntax for a selection is:

selection	=	**CASE** *selector* { *option* }
option	=	$\{_1$ **,** *case.expression* $\}$ *process*
	|	**ELSE** *process*
selector	=	*expression*
case.expression	=	*expression*

The keyword **CASE** is followed by a *selector* expression and then by zero or more *options*, indented two spaces. An option starts with either a list of case expressions or the keyword **ELSE**. This is followed by a process, indented two spaces. All case expressions used in a selection must have distinct constant values (i.e. each must be a different value from the other expressions used). The selector and the case expressions must be the same data type, which may be either an integer, byte or boolean data type. A selection can have only one **ELSE** option.

Constant case expressions may be given a name in an *abbreviation* (page 43). Data types (page 25) and expressions (page 57) are also discussed later.

2.4 **WHILE** loop

A **WHILE** loop repeats a process while an associated *boolean expression* is true. Consider the following example:

```
WHILE buffer <> eof
  SEQ
    in  ? buffer
    out ! buffer
```

This loop repeatedly copies a value from the channel **in** to the channel **out**. The copying continues while the boolean expression **buffer <> eof** is true. The sequence is not performed if the boolean expression is initially false.

To further illustrate how processes combine, consider the following process:

```
SEQ
  -- initialise variables
  pointer  := 0
  finished := FALSE
  found    := FALSE
  -- search until found or end of string
  WHILE NOT finished
    IF
      string[pointer] <> char
        IF
          pointer < end.of.string
            pointer := pointer + 1
          pointer = end.of.string
            finished := TRUE
      string[pointer] = char
        SEQ
          found    := TRUE
          finished := TRUE
```

This example searches the array **string** for a character (**char**). Note how the process is built from primitive processes and constructions. In fact this kind of bounded search is better written using a replicated conditional (page 12) as follows:

```
IF
  IF i = 0 FOR string.size
    string[i] = char
      found := TRUE
  TRUE
    found := FALSE
```

The syntax for a loop is:

loop	=	**WHILE** *boolean* *process*
boolean	=	*expression*

The keyword **WHILE** and a boolean expression are followed by a process which is indented two spaces. The boolean expression appears to the right of the keyword **WHILE**.

2.5 Parallel

A parallel combines a number of processes which are performed concurrently. Consider the following example:

```
PAR
  keyboard (kbd.to.ed)
  editor   (kbd.to.ed, ed.to.screen)
  screen   (ed.to.screen)
```

This parallel combines instances of three named processes (known as procedures, page 69), which are performed together. They start together and the parallel terminates when all three combined processes have terminated. The editor and keyboard process communicate using channel **kbd.to.ed**; the screen and editor communicate using channel **ed.to.screen**.

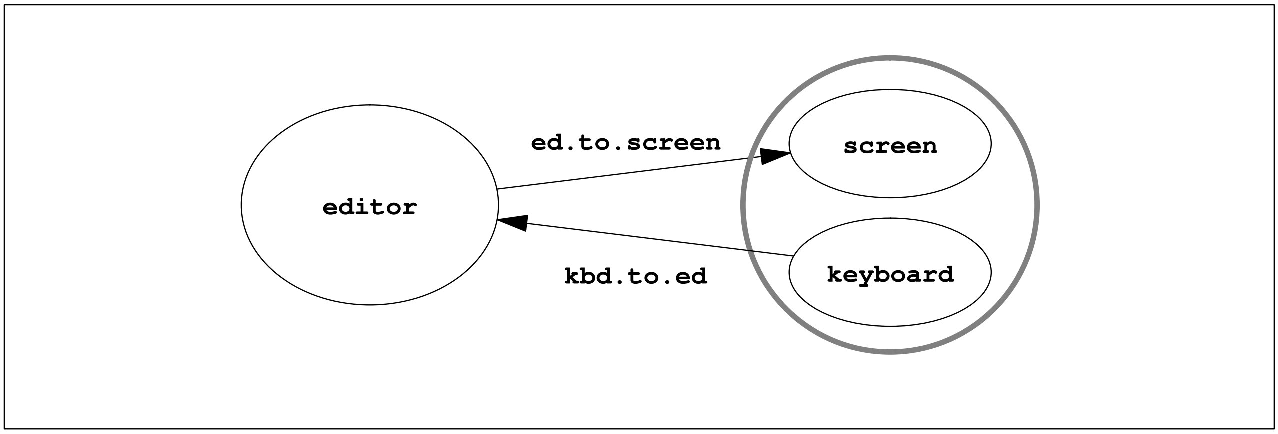

Figure 2.1 Communicating concurrent processes

Values are passed between concurrent processes by communication on *channels* (page 45) using input and output (page 6). Each channel provides unbuffered unidirectional point-to-point communication between two concurrent processes. Figure 2.1 illustrates the channels connecting the three processes in the above example.

The example above shows the parallel being used to tie together the major components of a system. However, a parallel may also be used simply to allow communication and computation to proceed together, as in the following example:

```
WHILE next <> eof
  SEQ
    x := next
    PAR
      in  ? next
      out ! x * x
```

The parallel in this example inputs the next value to be processed from one channel while the previous value is being processed and used in an output on another.

The syntax of a parallel is similar to that of a sequence:

parallel	=	**PAR**
		{ *process* }

The keyword **PAR** is followed by zero or more processes at an indentation of two spaces.

Note that changing the order of the processes combined in a parallel does not change the effect of that parallel. Parallels may be nested to form the hierarchical structure of a program. The behaviour of the following process is the same as the earlier example, and is reflected in the use of nested boxes in Figure 2.1:

```
PAR
  editor (kbd.to.ed, ed.to.screen)
  PAR
    keyboard (kbd.to.ed)
    screen   (ed.to.screen)
```

Writing a parallel like this helps later in program development when a program must be *configured* to its environment (when its processes are allocated to physical processors). See appendix B.

A parallel construction which specifies a priority of execution on a single processing device able to perform several tasks (i.e. a multi-tasking processor) is described in appendix B.2.1, page 94.

2.5.1 Restrictions on parallel use of variables and channels

In occam variables and channels used in parallels are subject to usage rules which prevent them from being accidentally shared between processes in potentially dangerous ways.

Parallel processes which share channels (page 45) and variables (page 35) can be subtly dependent on the way in which parallel composition is implemented. For instance, a variable which is written by one process and read by another depends upon the scheduling of the processes to ensure that the variable is not read before it has been written. The scheduling can be affected by events outside the control of the processes and can differ between implementations. This means that errors in the program can become apparent on rare occasions and are therefore difficult to repeat. The following usage rules ensure that such errors cannot happen.

Variables which are changed by input or assignment in one of the processes of a parallel may not be used in expressions or for assignment in any other process in the parallel. A variable may appear in expressions in any number of components of a parallel so long as it is not assigned in any parallel component. The following process, for example, is INVALID:

```
PAR                 -- this parallel is INVALID!
  SEQ
    mice := 42      -- the variable mice is assigned . .
    c ! 42
  c ? mice          -- . . in more than one parallel component
```

This process is invalid because it assigns to the variable `mice` in the assignment `mice := 42` in the first component of the parallel and also in the input `c ? mice` in the second component.

A channel may not be used for input in more than one component of a parallel and may not be used for output in more than one other component of the parallel. The following process, for example, is INVALID:

```
PAR                 -- this parallel is INVALID!
  c ! 0             -- the channel c is used for output . .
  SEQ
    c ? x
    c ? y
  c ! 1             -- . . in more than one parallel component
```

This process is invalid because it uses the channel `c` for output in more than one parallel component.

A check list of the usage rules which apply to parallel processes is given in appendix E.

2.5.2 Replicated parallel

A parallel can be replicated, in the same way as sequences and conditionals described earlier. A replicated parallel constructs a number of similar concurrent processes, as shown in the following example:

```
PAR i = 3 FOR 4
  user[i] ! message
```

This replication performs the four outputs concurrently, and is equivalent to

```
PAR
  user[3] ! message
  user[4] ! message
  user[5] ! message
  user[6] ! message
```

Now consider the following example:

```
PAR
  farmer ()
  PAR i = 0 FOR 4
    worker (i)
```

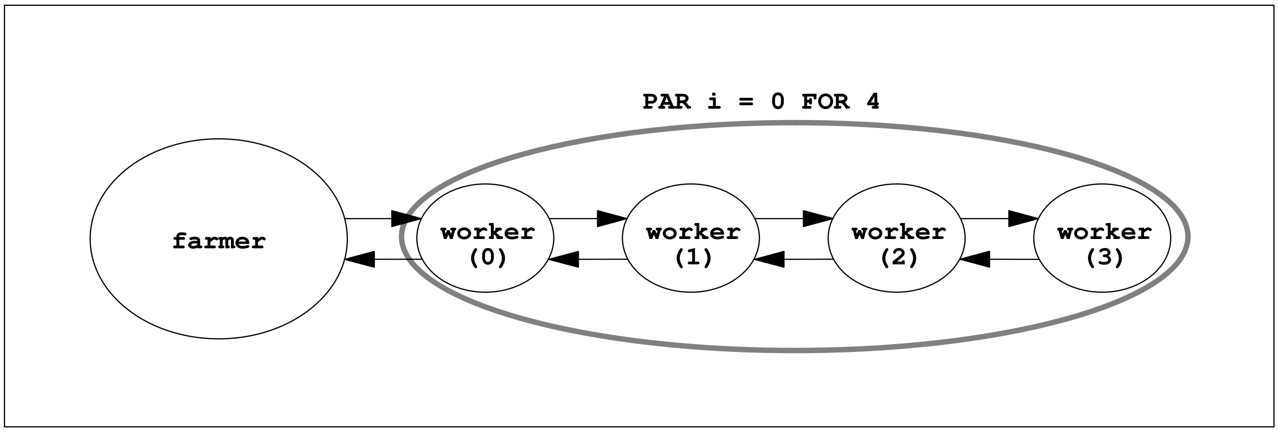

Figure 2.2 A farm of parallel processes

The replicated parallel in this example starts 4 processes, each an instance of the procedure **worker**, and terminates when all four processes are finished. Figure 2.2 shows the structure of this process, which is elaborated upon in the following section. Implementations of occam usually require that, unlike in sequence and conditional replications, the *count* value (here 4) must be constant. The procedure **worker** takes a single *parameter* (page 69), for each *instance* (page 69) of the procedure the value of the index **i** is passed. Expanding the replication shows that the above example is equivalent to the following:

```
PAR
  farmer ()
  PAR
    worker (0)
    worker (1)
    worker (2)
    worker (3)
```

The syntax of a replicated parallel is similar to that of the replicated sequence shown earlier in the manual:

parallel	=	**PAR** *replicator* *process*
replicator	=	*name* **=** *base* **FOR** *count*
base	=	*expression*
count	=	*expression*

The keyword **PAR** and a replicator are followed by a process, indented two spaces. The replicator appears to the right of the keyword **PAR**. The replicator specifies a name for the index. The value of the index for the first replication is the value of the base expression, and the number of times the process is replicated is the value of the count expression.

The index may be used in expressions but cannot be changed by an input or assignment. The index has a value of *type* **INT**. The base and count expressions must also be of data type **INT**. Data types (page 25) are explained later in the manual. If the count expression has a negative value the process is *invalid*. See appendix F, page 101, for an explanation of how *invalid processes* behave. If the value of the count expression is zero, the replicated parallel behaves like the primitive process **SKIP** (page 7) and does nothing.

2.6 Alternation

An alternation combines a number of processes, only one of which is executed. Each of the combined processes is guarded by a guard which may or may not be ready to proceed. Examples of such guards are inputs on channels and delayed inputs on timer channels (page 82). The alternation performs the process associated with a guard which is ready. Consider the following example:

```
ALT
  left  ? packet
    stream ! packet
  right ? packet
    stream ! packet
```

The effect of this process, if used in a loop, would be to merge the input from the two channels **left** and **right**, on to the channel **stream**. The alternation (illustrated in figure 2.3) receives an input from either channel **left** or channel **right**. A ready input is selected, and the associated process is performed. An input is ready if a process running in parallel has executed the corresponding output on the same channel and is unable to proceed until this communication has completed. Consider this example in detail. If the channel **left** is ready, and the channel **right** is not ready, then the input **left ? packet** is selected. If the channel **right** is ready, and the channel **left** is not ready, then the input **right ? packet** is selected. If neither channel is ready then the alternation waits until an input becomes ready. If both inputs are ready, only one of the inputs and its associated process are performed; the language does not define which one.

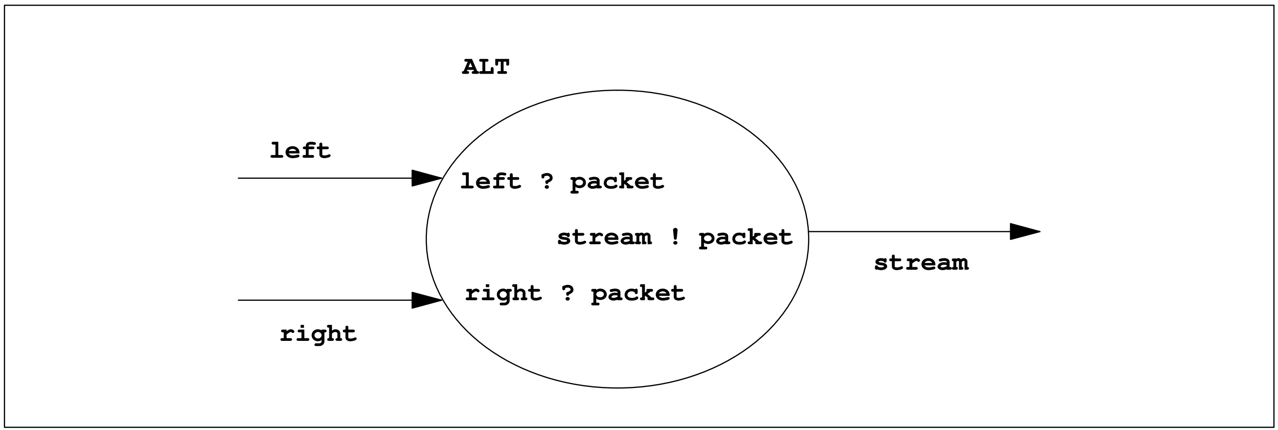

Figure 2.3 Merging the flow of data

A boolean expression may be included in a guard of an alternation to selectively exclude guards from being considered ready, as shown in the following example:

```
ALT
  left.enabled & left  ? packet
    stream ! packet
  right ? packet
    stream ! packet
```

This alternation places the *boolean variable* (page 35) **left.enabled** before the first input using the **&** operator. If **left.enabled** is true, the input is included for consideration by the alternation. If **left.enabled** is false, the input is excluded. To clarify this behaviour, consider the following example:

```
--  Regulator:
--     regulate flow of work into a networked farm
SEQ
  idle := processors
  WHILE running
    ALT
      from.workers ? result
        SEQ
          reg.to.gen ! result
          idle := idle + 1
      (idle >= 1) & gen.to.reg ? packet
        SEQ
          to.workers ! packet
          idle := idle - 1
```

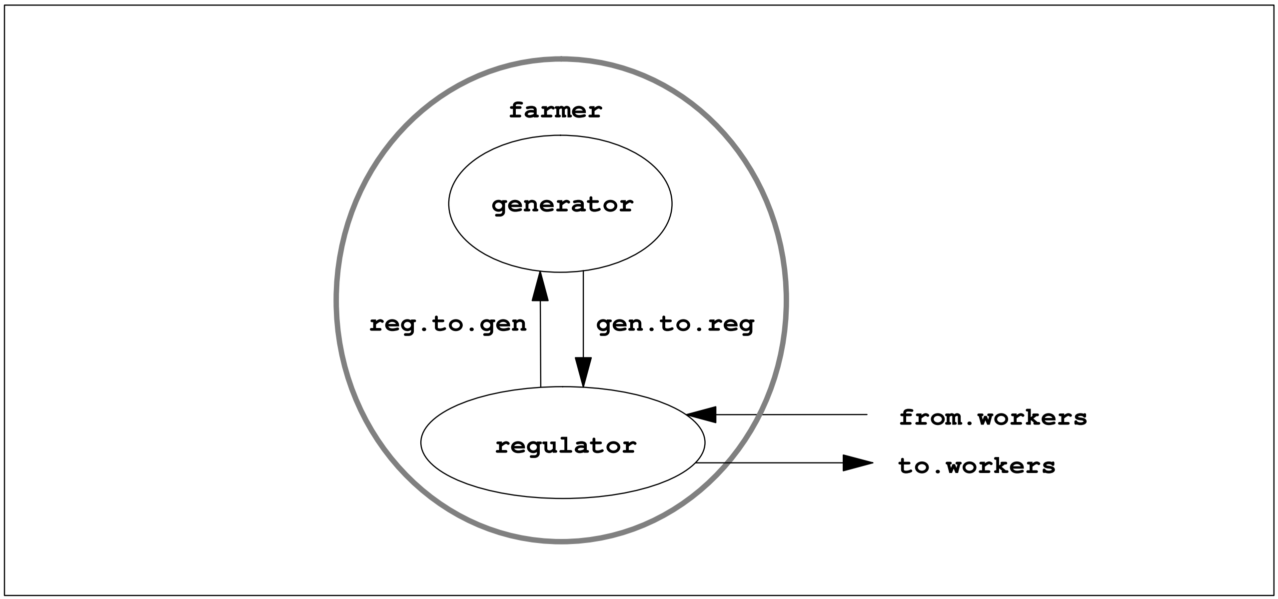

Figure 2.4 Regulating the flow of data

This is an example (part of the farmer process first illustrated in figure 2.2 and fully illustrated in figure 2.4) of a process which regulates the flow of work into a processor *farm*. A processor farm can be thought of as a number of machines (*worker processes*), each able to perform some task and output a result. The above example controls the amount of work (as packets of data) given to a farm which consists of a network of worker processes. Work may be received by the input **`gen.to.reg ? packet`**, but is only accepted if a worker is idle (i.e. **`(idle >= 1)`**). As a packet of work is sent to the farm, the counter **`idle`** is decremented to indicate the number of worker processes which are idle. The worker processes are sent work on the channel **`to.workers`** (see figure 2.4), and the variable **`idle`** is decremented to keep a count of the idle machines in the farm.

The syntax for alternation is:

```
alternation            =  ALT
                            { alternative }
alternative            =  guarded.alternative
                       |  alternation
guarded.alternative    =  guard
                            process
guard                  =  input
                       |  boolean & input
                       |  boolean & SKIP
```

The keyword **ALT** is followed by zero or more *alternatives*, indented two spaces. An alternative is either a *guarded* alternative or another alternation. A guarded alternative is a *guard* followed by a process on the following line undented two spaces. A guard is an input, or a boolean expression to the left of an ampersand (**&**) with an input or **SKIP** on the right. **SKIP** can take the place of an input in a guard which includes a boolean expression, as shown in the following example:

```
ALT
  in ? data
    out ! data
  monday & SKIP
    out ! no.data
```

If the boolean **monday** is true then **SKIP** is treated as though it where a ready input, and may be selected immediately. If the input **in ? data** is also ready, only one of the processes is performed; which process will be performed is undefined.

An alternation with no component alternatives behaves as **STOP**.

Alternation with priority selection is explained on page 23. *Delayed inputs*, explained on page 82, will delay before they become ready, and may be used in guards wherever an input may be used.

Inputs (page 6) and **SKIP** (page 7) are discussed in chapter 1. Expressions (page 57) are discussed later in the manual. Details of boolean expressions are given on page 63.

2.6.1 Replicated alternation

An alternation can be replicated in the same way as sequences, conditionals and parallels described earlier in the manual. A replicated alternation constructs a number of similar alternatives. The alternation performs a single process which is associated with a ready guard. Consider the following example:

```
ALT
  ALT i = 0 FOR number.of.workers
    free.worker[i] & gen.to.reg ? packet
      SEQ
        to.workers[i] ! packet
        free.worker[i] := FALSE

  ALT i = 0 FOR number.of.workers
    from.workers[i] ? result
      SEQ
        reg.to.gen ! result
        free.worker[i] := TRUE
```

This example presents an alternate version of the process **farmer** discussed in the previous section and is illustrated in figure 2.5. This version also regulates the flow of work into the farm, but does so by maintaining an array of booleans (**free.worker**) which indicate when a worker is busy. This version of the farmer process is most suitable where several worker processes in the farm are able to input directly from the process.

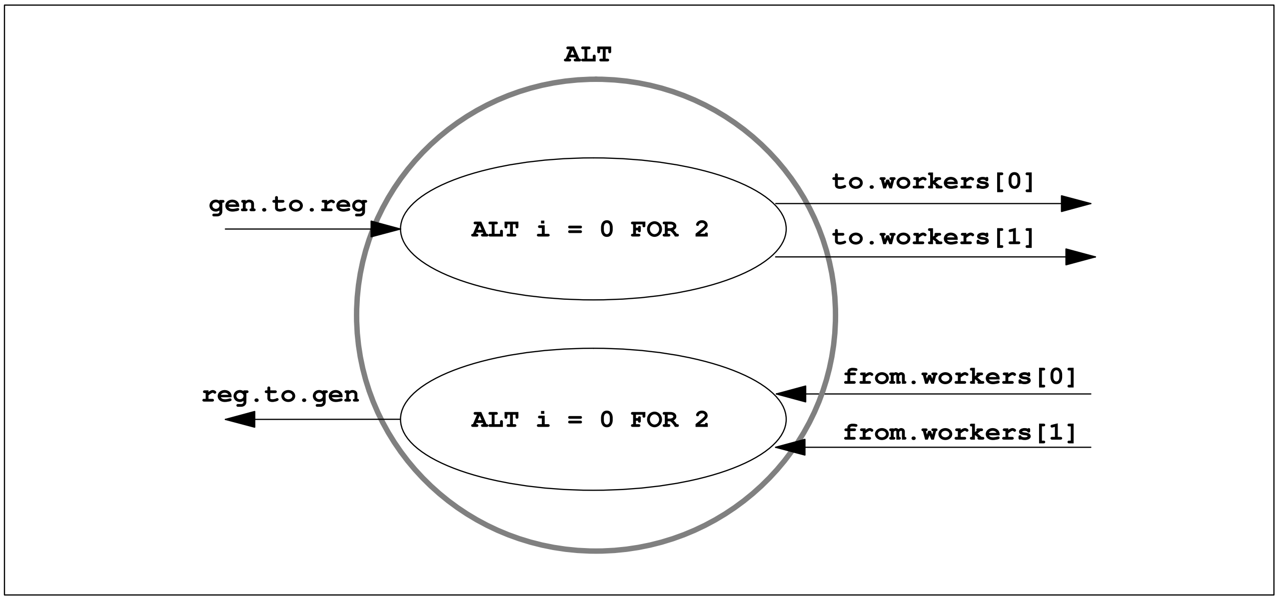

Figure 2.5 A tree structured farm of parallel processes

Work packets are input on the channel **gen.to.reg** and distributed to an array of worker processes. The completed result is returned to the farmer process via the channels **from.workers**. Consider first the upper half of this alternation. Each alternative is guarded by a boolean **free.worker[i]** (which has the value true if the worker process is idle), and an input **gen.to.reg ? packet** which inputs packets of work. A selected component of this replication will, after completing the input of a packet, perform the output **to.workers[i] ! packet** (i.e. pass work to an idle worker process), and then set the boolean **free.worker[i]** to false, indicating the worker is no longer idle.

Now consider the lower half of this example, which handles the results returning from worker processes. Each component of the replication is guarded by an input **from.workers[i] ? result** which receives results from a worker process. A selected component of this replication will, after completing the input from the worker process, perform the output **reg.to.gen ! result** (i.e. pass the result back to the generator process which sent the work), and reset the boolean **free.worker** to true to indicate the worker is now idle.

A number of these farmer processes in parallel can form a tree of worker processes (see figure 2.5), enabling large and effective farms to be built.

If **number.of.workers** has the value 2, the example has the same effect as:

```
ALT
  ALT
    free.worker[0] & gen.to.reg ? packet
      SEQ
        to.workers[0] ! packet
        free.worker[0] := FALSE
    free.worker[1] & gen.to.reg ? packet
      SEQ
        to.workers[1] ! packet
        free.worker[1] := FALSE

  ALT
    from.workers[0] ? result
      SEQ
        reg.to.gen ! result
        free.worker[0] := TRUE
    from.workers[1] ? result
      SEQ
        reg.to.gen ! result
        free.worker[1] := TRUE
```

As for the earlier descriptions of replication, the value of the index for the first replication is the value of the base expression, and the number of replications is the value of the count expression. The syntax for the replicated alternation is:

alternation	=	**ALT** *replicator* *alternative*
replicator	=	*name* **=** *base* **FOR** *count*
base	=	*expression*
count	=	*expression*

The keyword **ALT** and a replicator are followed by an alternative which is indented two spaces. The replicator appears to the right of the keyword **ALT**. The replicator specifies a name for the index.

The index may be used in expressions but cannot be changed by an input or assignment. The index has a value of *type* **INT**. The base and count expressions must also be of data type **INT**. Data types (page 25) are explained later in the manual. If the count expression has a negative value the process is *invalid*. See appendix F, page 101 for an explanation of how *invalid processes* behave. If the value of the count expression is zero, the replicated alternation behaves like an alternation with no alternatives that can proceed.

2.6.2 Priority alternation

The inputs which guard alternatives in an alternation may be given a selection priority. Priority is determined by textual order, the alternative appearing first having the highest priority for selection. Consider the following example:

```
PRI ALT
  disk ? block
    d ()
  keyboard ? char
    k ()
```

This priority alternation will input values from the channel **disk** in preference to inputs from the channel **keyboard**. If both channels **disk** and **keyboard** become ready then **disk** will be selected as it has the highest priority.

Consider the following example:

```
PRI ALT
  stream ? data
    P ()
  busy & SKIP
    Q ()
```

This process inputs **data** if an input from **stream** is ready, and performs the process **P**. Otherwise if the boolean **busy** is true the process **Q** is performed.

An alternative guarded by **TRUE & SKIP** is always ready and so can only usefully appear within a priority alternation after all other alternatives.

The syntax for priority alternation is:

alternation	=	**PRI ALT**
		{ *alternative* }
	|	**PRI ALT** *replicator*
		alternative

The keywords **PRI ALT** are followed by zero or more alternatives at an indentation of two spaces. The alternative may be replicated.

2.7 Processes

The constructs introduced in this chapter and the previous chapter are all kinds of process. The following tabulation presents the syntax of an occam process. This syntax is extended in later chapters where further kinds of process are introduced, (see pages 41, 52 and 73):

process	=	*assignment*
	|	*input*
	|	*output*
	|	**SKIP**
	|	**STOP**
	|	*sequence*
	|	*conditional*
	|	*selection*
	|	*loop*
	|	*parallel*
	|	*alternation*

3 Data types

occam programs act upon *variables*, *channels* and *timers*. A variable has a value, and may be assigned a value in an *assignment* or *input*. Channels communicate values. Timers produce a value which represents time.

This chapter describes the *data type* of values and variables and literal representations of known values. Channels are discussed on page 45, and timers are discussed on page 81.

3.1 Primitive data types

Values are classified by their *data type*. A data type determines the set of values that may be taken by objects of that type and the set of operators which may be applied to objects of that type.

These are the primitive data types built into occam:

BOOL	Boolean values true and false. A boolean type.
BYTE	Integer values from 0 to 255. A byte type.
INT	Signed integer values represented in twos complement form using the word size most efficiently provided by the implementation. An integer type.
INT16	Signed integer values in the range -32768 to 32767, represented in twos complement form using 16 bits. An integer type.
INT32	Signed integer values in the range -2^{31} to $(2^{31} - 1)$, represented in twos complement form using 32 bits. An integer type.
INT64	Signed integer values in the range -2^{63} to $(2^{63} - 1)$, represented in twos complement form using 64 bits. An integer type.
REAL32	Floating point numbers stored using a sign bit, 8 bit exponent and 23 bit fraction in ANSI/IEEE Standard 754-1985 representation. The value is positive if the sign bit is 0, negative if the sign bit is 1. A real type. The magnitude of the value is: $\begin{array}{ll} (2^{(exponent-127)}) * 1.fraction & \text{if } 0 < exponent \text{ and } exponent < 255 \\ (2^{-126}) * 0.fraction & \text{if } exponent = 0 \text{ and } fraction \neq 0 \\ 0 & \text{if } exponent = 0 \text{ and } fraction = 0 \end{array}$
REAL64	Floating point numbers stored using a sign bit, 11 bit exponent and 52 bit fraction in ANSI/IEEE Standard 754-1985 representation. The value is positive if the sign bit is 0, negative if the sign bit is 1. A real type. The magnitude of the value is: $\begin{array}{l\|l} (2^{(exponent-1023)}) * 1.fraction & \text{if } 0 < exponent \text{ and } exponent < 2047 \\ (2^{-1022}) * 0.fraction & \text{if } exponent = 0 \text{ and } fraction \neq 0 \\ 0 & \text{if } exponent = 0 \text{ and } fraction = 0 \end{array}$

As the above list shows, all signed integer values are represented in twos complement form using the number of bits indicated by the type. All real values are represented according to the representation specified by the ANSI/IEEE standard 754-1985, for binary floating-point arithmetic.

A primitive type is either a real type, an integer type, a byte type or a boolean type. These properties of types determine which operators may legally be applied to them, except in the cases where an operator is defined to apply to an operand of only one type such as **INT**.

Objects which have values in occam have one of the following forms:

Literals	Textual representation of known values
Constants	Symbolic names which have a constant value
Variables	Symbolic names which have a value, and may receive a new value by input or assignment
Index	Replication index value

A literal is a known value (**1**, **2**, **'H'**, **1.0E+6**, etc). A variable has a value of a specified type, and may receive a new value in an input or assignment. Names with a constant value are specified by an *abbreviation* (page 43). *Expressions* (page 57) and *functions* (page 75) also have a data type and value. The name specified as the index of a replication has a different value for each component of the replication.

The syntax of primitive data types is:

```
data.type  =  BOOL
           |  BYTE
           |  INT
           |  INT16
           |  INT32
           |  INT64
           |  REAL32
           |  REAL64
```

Rounding of real values

An accepted limitation in the use of floating point representations of real values is that only a finite set of all possible real values can be represented, thus any real value will be rounded to produce a result which is the nearest value that can be represented by the type. For example, where the type is **REAL32**, the next representable value after 1.0 is the value 1.000000119209 (to the nearest 12 digits past the decimal point), any value lying between 1.0 and this value cannot be exactly represented using the representation of type **REAL32**. Thus, values which do lie between 1.0 and 1.00000019209 which are of type **REAL32** must be *rounded* to one of these values.

The rounding of real numbers occurs in arithmetic expression evaluation (page 57), in explicit *type conversions* (page 66), and also when literals are converted to the IEEE representation. An explanation of the IEEE rounding modes is given in the appendix (page 98).

3.2 Named data types

In addition to the primitive data types built into the language it is possible to define new data types derived from these. Each derived type has a name introduced in its *type definition*. These derived types include array types (page 30) and record types (page 31). This section describes how to define new types which have the same values as existing data types.

Consider the definition:

```
DATA TYPE LENGTH IS REAL32 :
DATA TYPE AREA IS REAL32 :
```

This creates new types called **LENGTH** and **REAL** with the same properties as the type **REAL32**.

Named data types might be used in a context in which many different sorts of variable all have the same primitive data type representation. For instance, by defining a **LENGTH** type and an **AREA** type, the type checking system can be used to ensure that a length is not assigned to an area, or that an area is not passed to a procedure where a length is expected.

Named types may also be used for type abstraction. If a fragment of code defining a computation using floating point arithmetic may be used in programs where arithmetic of different precision may be required, then one of the definitions:

```
DATA TYPE REAL IS REAL64 :
DATA TYPE REAL IS REAL32 :
```

may be used, followed by code which uses only the named type **REAL**.

Any legal occam name may be used to name a data type, but it may be desirable to adopt the convention that type names do not use lower case letters. Such conventions are merely to improve readability of code.

The syntax of named types is:

definition	=	**DATA TYPE** *name* **IS** *data.type* **:**
data.type	=	*name*

The data type which is defined may be referred to by its name. Two named data types are only equal when their references are equal. For example, in:

```
DATA TYPE MONEY IS INT64 :
MONEY a, b :
INT64 c :
```

the types of **a** and **b** are the same, but the type of **c** is different. In:

```
DATA TYPE MONEY IS INT64 :
MONEY a :
DATA TYPE MONEY IS INT64 :
MONEY b :
```

the types of **a** and **b** are different because, although the representation of the name of each type is the same, the two type definitions introduce different types.

Any named type derived directly or indirectly by its definition from a primitive type is either a real type, an integer type, a byte type or a boolean type.

By following a chain of one or more type definitions it is possible to determine the *underlying* type of any named type.

3.3 Literals

A literal is a textual representation of a known value, and has a data type. For example, the following are all legal literals:

42	an integer literal in decimal
#2A	an integer literal in hexadecimal
'T'	a byte literal
TRUE	a boolean literal

A number (e.g. **42**) representing a decimal value, or a hexadecimal value introduced by the hash symbol (**#**), is an integer of type **INT**. A character enclosed within a pair of quotation marks (*eg* **'z'**) has a value of type **BYTE**.

Literal values of other types may be expressed by decorating the textual representation of the value with the data type in parentheses, for example:

42(BYTE)	a byte value
'T'(INT)	an integer value
42(INT64)	an integer value with 64 bit representation
42.0(REAL32)	a 32 bit floating point value
386.54(REAL64)	a 64 bit floating point value
1760.0(LENGTH)	a value of a user-defined **REAL** type
587.0E-20(REAL64)	a 64 bit floating point value
+1.0E+123(REAL64)	a 64 bit floating point value
16777217.0(REAL32)	a 32 bit floating point value

The exponent of a **REAL32** literal should have no more than 2 digits, and the exponent of a **REAL64** literal no more than 3 digits. All real number literals must be explicitly decorated with their data type in parentheses after the real number unless the rules in section 3.3.2 allow the decoration to be omitted. A literal of type **REAL32** or **REAL64** or a named type derived from one of these will be rounded (page 26) when the value is converted into the representation of the type. The effect of this rounding can be seen particularly in the last example shown here. The value 16777216.0 is 2^{24} and can be represented precisely in the representation of 32-bit real numbers with a fraction of 23 bits. However, the value 16777217.0 is $(2^{24} + 1)$ and cannot be represented precisely in this representation, and will round to the value 16777216.0. 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**. For example:

54321765439.54(REAL32)	has a nearest representable value of 54321766400.0
54321765400.00(REAL32)	also has a nearest representable value of 54321766400.0

An explanation of the IEEE rounding is given in the appendix (page 98).

The syntax for literals is:

literal	=	*integer*
	|	*byte*
	|	*real*
	|	*integer* **(** *data.type* **)**
	|	*byte* **(** *data.type* **)**
	|	*real* **(** *data.type* **)**
	|	**TRUE**
	|	**FALSE**
integer	=	*digits*
	|	**#** *hex.digits*
byte	=	**'** *character* **'**
real	=	*digits* **.** *digits*
	|	*digits* **.** *digits* **E** *exponent*
exponent	=	**+** *digits*
	|	**-** *digits*

The data type which may appear in parentheses after a literal constant is called a *decoration*. A decoration of a real literal must be a real data type. A decoration of an integer or byte must be an integer or byte type of size appropriate to the value of the literal. A compiler performs any arithmetic to the precision required by the type and may reject literals which produce values which overflow. Only literal constants may be decorated and the decoration serves to inform the compiler of the type of constant value to be created. Type conversion, which may be applied between arbitrary values of most data types, uses type names as operators and is described elsewhere (page 66).

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**. Literals of the types *integer* and *real* may not contain any embedded spaces.

All characters are coded according to their ASCII code. The character **A**, for example, has a value 65, and so on. A table of the ASCII character set is given in appendix G (page 103). A character enclosed in a pair of quotes (e.g. **'T'**) is a byte value, unless explicitly stated otherwise by decoration with an integer data type in parentheses to the right of the enclosing quotes.

The literals **TRUE** and **FALSE** represent the boolean values true and false respectively.

3.3.1 Literals of named types

Literals of a named type derived from a primitive data type are denoted in the same way as literals of the underlying type except that they are decorated with the name of the new type. For example:

```
54321765439.54(LENGTH)
```

The rules for rounding literals of the new type are the same as those for the underlying type so that if the underlying type is **REAL32** then the nearest representable value of this literal is 54321766400.0.

3.3.2 Omitting type decorations from literals

In many expressions the explicit data type decoration is superfluous as a compiler can easily discover the intended type of the literal from the context.

The rules defining when a decoration may be omitted are governed by the principle that where only one possible type decoration would create a legal occam expression then the decoration may be omitted.

The rules are shown below. Some of these rules refer to language features not yet mentioned in this manual. However it is appropriate to include this subsection in the section describing literals. In any case where there is doubt as to whether a decoration may be omitted it is always safe to leave the decoration in.

A type decoration may be omitted when there is only one decoration which would satisfy all the type checks performed by the compiler. In the following circumstances contextual information is used to deduce type:

- Inside a single expression.

  In this context, no information about expression types is passed either 'into' or 'out of' a value process (page 75).

- Expressions in process constructs where only one data type is permitted are assumed to have that type:

  - Array size (page 31) and subscript (page 37) expressions must be of type **INT**.
  - Base and count expressions of replicators (page 10) and segments (page 37) must be of type **INT**.
  - Guards of conditional processes (page 11), expressions following **WHILE** (page 14), and boolean guards of alternatives (page 21)), must be of type **BOOL**.
  - Shift counts (page 62) must be of type **INT**.

- In an assignment (page 5) or output (page 6) the types of the expressions are inferred from the types of the variables or the protocol of the channel.

- In value abbreviations (page 44), the type of the expression is inferred from the specifier of the abbreviation. This rule also applies to the actual parameters of functions and procedures.

- In a **CASE** selection (page 13) the types of the constant case expressions are inferred from type of the selector.

- Inside a record literal (page 32) the type of each expression is inferred from the context of the literal, together with any optional decoration.
- Inside a table construct (page 59) the type of each expression is inferred from the context of the table and the types of any other components of the table, together with any optional decoration.
- In the **RESULT** list of a function (page 78) The types of expressions are inferred from the return type(s) of the function.

The following code fragments are legal for the reasons shown:

```
1(INT32) + 10          -- operands of + must be the same

CHAN OF INT32 c :
c ! 4                  -- 4 must conform to the protocol of c

REAL32 x :
x := 2.0               -- value assigned to x must be REAL32

VAL BYTE ESC IS 13 :   -- 13 must match specifier BYTE

INT reply:
CASE reply
  'Y'                  -- 'Y' must match type of reply
    ...

REAL32 FUNCTION pi() IS 3.14159 :  -- result must match function type
```

Examples of undecorated literals in record literals and tables are given on pages 32 and 58.

3.4 Array data types

An array has a number of consecutively numbered components of the same type which are stored contiguously in memory. Arrays of channels and timers are discussed in chapters 5 and 9. Array data types are non-primitive data types. Such array types may be named in type definitions (page 26). An example of an array type is:

```
[5]INT
```

Arrays of this type have components each of type **INT**. The components are numbered 0, 1, 2, 3, 4. Arrays may have further dimensions specified by adding the size of the dimension, enclosed in square brackets, to the type. The following is an array type with two dimensions:

```
[4][5]INT
```

An array of this type has four components each of type **[5]INT**. Equally, an array of type **[3][4][5]INT** is an array with three components of type **[4][5]INT**, and so on. In this way, arrays with any number of dimensions may be constructed. The type named after the dimension(s) of an array is called the *base type* of the array. The base type of a data type array may be any primitive or named data type.

In theory there is no limit to the number of dimensions an array type may have. In practice however, arrays of data type require memory, both in the compiler and at run time, and an implementation may impose limits. Here are some more array types:

[n]BYTE	a byte array with **n** components
[3][3][3]LENGTH	a three dimensional array of values of type **LENGTH**
[50]BOOL	an array with boolean components.

These examples show the definition of named types which are arrays:

```
DATA TYPE DOSFNAME IS [12]BYTE :        -- a filename for MSDOS
DATA TYPE MATRIX3 IS [3][3]REAL32 :     -- a 3 * 3 matrix
```

The size of each dimension in an array declaration must be specified by a value of type **INT**, and be a value greater than zero. Two arrays are considered to have the same type if they have the same number and type of components. The value of an array is the ordered set of values of its components. An array may receive a new value by input or assignment. An input or assignment to an array is legal only if the value to be assigned is of the same type as the array.

The syntax for array data types is:

data.type = **[** *expression* **]** *data.type*

The syntax shows that any type can be preceded by a value given as an expression in square brackets to denote an array of that base type. The value specifies the number of components in the array and must must be a constant of type **INT**. The syntax is defined recursively showing how multidimensional arrays are defined as arrays of arrays, as illustrated in the examples above.

The declaration of variables of named and unnamed array data types is discussed on page 35. The construction of values of array types is discussed on page 58.

3.5 Record data types

The previous sections have shown how to use primitive and array data types. This section shows how record data types may be defined and used.

A record has a number of fields, each of which is of a data type. Records are used to gather together components of data which make a logical unit. For instance, the real and imaginary components of a complex number form a logical unit:

```
DATA TYPE COMPLEX32
  RECORD
    REAL32 real :
    REAL32 imag :
:
```

This definition creates a new type named **COMPLEX32**. It is a record type with two fields of type **REAL32** named **real** and **imag**.

The declaration of variables of record types and the accessing of fields of such records by subscripting record names with field names is discussed in the next chapter (page 35).

The syntax of record types is:

definition	=	**DATA TYPE** *name* *structured.type* **:**
structured.type	=	**RECORD** { *data.type* $\{_1$ **,** *field.name* } **:** }
field.name	=	*name*

A structured data type definition consists of the keywords **DATA TYPE** followed by a name and, indented by two spaces on the following line, a structured type. A structured type consists of the keyword **RECORD** followed by one or more *field declarations* on succeeding lines indented two spaces. Although a record with no fields is permitted by the syntax, it is not useful as there are no values of records of such a type. Each field declaration consists of a primitive or named data type followed by a comma separated list of field names. Fields of named array or record data types may be defined.

The same field name may be used in the definition of more than one record type. A field name may also be the same as a name defined in any other specification in the scope. Consider:

```
DATA TYPE VOLUME IS REAL32 :
DATA TYPE RECTANGLE
  RECORD
    INT height, width :
:
DATA TYPE KETTLE
  RECORD
    REAL32 power :
    VOLUME capacity :
:
DATA TYPE HEATER
  RECORD
    RECTANGLE size :
    REAL32 power :
:
INT height :
```

In this example, both kettles and heaters have a power field. The field **power** refers to the second field of variables or values of type **HEATER** and to the first field of variables or values of type **KETTLE**. The name **height** is used both for a field and for a variable.

The types of the fields of a record may be any primitive or named types or arrays of such types. Consider:

```
DATA TYPE DIRENTRY
  RECORD
    DOSFNAME fname :
    [4]BYTE access :
    INT32 loc :
:
```

The fields of this record are respectively a named record type, an array of bytes and a primitive integer type.

3.5.1 Record literals

A literal representation of a record gives values for each field and the name of the type. Consider:

```
[0.0(REAL32),1.0(REAL32)](COMPLEX32)
```

This is a literal with data type **COMPLEX32** (defined above). The value of the **real** field is **0.0(REAL32)** and the value of the **imag** field is **1.0(REAL32)**. The association of field values with fields is strictly by position with reference to the record data type definition. An implementation may reorder the fields of a record in the computer's memory for efficiency of storage or access. The type decoration in parentheses may be omitted in contexts where it may be inferred by the rules on page 29.

Syntactically a record literal is a table and this syntax is based on that of tables of array data type (page 58):

table	=	**[** $\{_1$ **,** *expression* $\}$ **] (***name***)**
	|	**[** $\{_1$ **,** *expression* $\}$ **]**

The name used to decorate a record literal must be the name of a record data type. The expressions in a record literal must have the type of the corresponding field in the type definition. For instance, a record literal of type **HEATER** must have two expressions, the first of type **RECTANGLE** and the second of type **REAL64**. The **RECTANGLE** will itself be a record literal constructed from two expressions of type **INT**. For example:

```
[[550, 1000](RECTANGLE),750.0](HEATER)
[[200,500],250.5]   -- in a context where the type may be inferred
```

3.5.2 Packed records

It may be desirable to use record structures to map onto data formats defined by hardware or software outside the programmer's control. Redundant copying or unpacking of such data structures may often be avoided by treating them as arrays of bytes or integers for communication and then accessing the data by *retyping* as defined on page 85. Retyping into a record type enables efficient access to fields within the data.

In these circumstances it is possible for the programmer to inhibit the compiler's freedom to choose how fields of a record are set out in memory. This is achieved by putting the keyword **PACKED** in front of the word **RECORD** in the data type definition.

An implementation may impose restrictions on the alignment of fields of particular types on boundaries defined by the target machine architecture (see page 92). In the absence of **PACKED** the compiler is free to reorder the fields of a record and insert implicit padding fields when mapping the type on to the target machine's memory. In a packed record the fields are mapped onto memory strictly in the order of declaration with no implicit padding. Explicit padding fields may need to be inserted to avoid alignment problems.

Using the keyword **PACKED** guarantees that fields will be assigned positions in order of declaration. A compiler may reject a packed record type definition which cannot be implemented without insertion of padding bytes in order to meet alignment requirements. Programs using **PACKED** are therefore potentially non-portable.

The syntax of packed record types is:

structured.type = **PACKED RECORD**
{ *data.type* $\{_1$ **,** *field.name* $\}$ **:** }

It is possible to use information derived by the compiler concerning the size and position of records and their fields. This uses the keywords **BYTESIN** and **OFFSETOF**, see page 65. Values created in this way may be implementation dependent.

4 Variables and values

occam programs act upon *variables* and communicate using *channels* and *timers*. A variable has a value, and may receive a new value in an *assignment* or *input*. Channels communicate values. Timers produce a value which represents the time.

This chapter describes variables, the declaration of names for variables and values, and their scope.

Channels (page 45) and timers (page 81) are discussed elsewhere in the manual.

4.1 Declaring a variable

The declaration of a variable associates its name with its data type. Consider the following example:

```
INT n :
```

This declaration introduces an integer variable with the name **n** and of the data type **INT**. The variable is not initialised, and therefore the value of the variable is unspecified until it receives a value by an input or assignment. An assignment or input to a variable is valid only if the value to be assigned is of the same data type as the variable. Here is a sequence of variable declarations:

```
BOOL   flag :
BYTE   char :
INT64  big :
REAL32 x :
[5]INT fingers :
```

Variables of named types are declared in the same way as variables of any of the primitive or array data types. For instance, using data types introduced in the previous chapter:

```
LENGTH width :
COMPLEX32 z :
DOSFNAME autoexec :
RECTANGLE rect1 :
DIRENTRY ent1 :
```

The syntax for the declaration and use of variables of any data type is :

declaration	=	*data.type* $\{_1$ **,** *name* $\}$ **:**
variable	=	*name*

A variable declaration consists of the data type, and a name to identify the variable. The declaration appears on a single line, and is terminated by a colon. The name introduced by a declaration may then be used to represent the variable in its scope (page 39). The compiler will allocate memory for a variable and translate uses of the name into uses of the memory allocated.

Where a number of variables of the same type need to be declared, occam permits a single declaration for several names in a comma separated list, as shown in the following example:

```
REAL64 a, b, c :
```

The type of the declaration is determined, and then the declarations are performed. The compiler is not constrained to allocate consecutive memory locations to such variables. This declaration is equivalent to the following sequence of declarations:

```
REAL64 a :
REAL64 b :
REAL64 c :
```

The variable names specified in a multiple declaration are separated by commas. A line break is permitted after a comma. Here are a few more multiple declarations:

```
BOOL flag, switch :
INT16 i, j, k :
REAL64 x, y :
INT64 chains,
      more.chains :
COMPLEX32 u, v, w :
KETTLE k1, k2 :
```

The declaration of an array follows the same form as other declarations, for example:

```
[5]INT x :
```

The declaration of **x** introduces an integer array with five components.

The declaration of an array with multiple dimensions is similar to other declarations, as shown in the following example:

```
[4][5]INT bigx :
MATRIX3 rot :
```

The declaration of **bigx** introduces a 4 by 5 array of integers. The declaration of **rot** introduces a variable of type **MATRIX3** which was defined above (page 31) to be a 3 by 3 array of real numbers. Note that it is optional to name an array data type, but that a record can only be declared with a named type defined in a previous data type definition in whose scope the record is declared.

Here are a few more examples of array declarations:

```
[4]BOOL flag :
[xsize][ysize]REAL64 matrix :
[3][3][3]INT16 cube :
[100]LENGTH histogram :
[256]DOSFNAME directory :
[20]HEATER heating.installation :
```

Several arrays of the same type can be declared together, for example:

```
[users]INT id, privilege :
```

In all declarations the type of the declarations is determined, and then the declarations are performed. This is especially important in the declaration of arrays as it is important to know that the size of the array is computed once only and cannot use variables declared within the declaration. Consider the following declaration:

```
[forms]INT forms, teachers :
```

This declaration introduces two new array variables, **forms** and **teachers**. The size of the arrays is determined by the value **forms**, which is evaluated before the declarations are performed and therefore refers to a variable already in scope when the declaration is performed. See the discussion of scope below (page 39).

4.2 Array components and segments

Individual components of an array may be selected by subscripting. A segment of an array is a sub-array comprising a sequence of consecutively subscripted components.

A subscripted name selects a component of an array. Suppose **clock**, **sf** and **data** are declared as follows:

```
[9]INT clock :
DOSFNAME sf :
[8][9][10]REAL32 data :
```

Consider these examples:

clock[1]	the second component of the array **clock**, of type **INT**.
data[0]	the first component of a dimension of **data**, of type **[9][10]REAL32**.
data[i][0]	the first component of another dimension of **data**, of type **[10]REAL32**.
sf[dotpos]	the **dotpos+1**th component of **sf**, of type **BYTE**.

A subscript appears in square brackets after the name of an array. The component selected has one dimension less than its type for each subscript. A subscript must be an expression of integer type **INT**. A subscript is valid only if the value of the expression is within the bounds of the array. A negative value subscript is always invalid and the value of a subscript must be in the range 0 to $(n-1)$, where n is the number of components in the array.

The syntax of array components is:

variable = *variable* **[** *expression* **]**

The simplest subscripted variable is a name followed by a single subscript expression (which must be a value of type **INT**) in square brackets to the right of the name. This is itself a variable and may also be followed by another subscript in square brackets, and so on, limited only by the number of dimensions in the array. Note that this syntax is also used for the selection of a field from a record (page 39) in which case the expression in square brackets can only be a name which is a field name.

A segment of an array is itself an array. The segment has zero or more components, as shown in the following examples:

[clock FROM 0 FOR 2]	the first two components of **clock**, of type **[2]INT**
[data FOR n]	the first **n** components of **data**,
	of type **[n][9][10]REAL32**.
[data FROM n FOR 6]	six components of the array **data** from **n**,
	of type **[6][9][10]REAL32**.
[data FROM m FOR 0]	an “empty” segment,
	of type **[0][9][10]REAL32**.
[sf FROM dotpos FOR 4]	a sub-array of **sf**, of type **[4]BYTE**.
[sf FROM dotpos]	same as above if **dotpos = 8** and **sf** is an array of 12 bytes.

A segment of an array has the same number of dimensions as the array, and always defines a set of contiguously stored components. A segment with zero components can only be used in an assignment, in a counted array communication (page 48), or as the right hand side of an abbreviation (page 42).

Short forms of segment may be used if the segment starts at the first component of the array or finishes with the last component. The segment **[data FOR n]** denotes the first **n** components of the array **data**. It is equivalent to **[data FROM 0 FOR n]**. The segment **[data FROM 4]** denotes the components of **data** starting with **data[4]** and continuing to the end of the array. In the scope of the above declarations it is equivalent to **[data FROM 4 FOR 4]**.

The syntax of segments is:

variable = **[** *variable* **FROM** *base* **FOR** *count* **]**
| **[** *variable* **FROM** *base* **]**
| **[** *variable* **FOR** *count* **]**

The syntax is defined recursively, and shows how more complex variables can be built. A segment begins with a square bracket, followed on the right by a variable. This may be followed by the keyword **FROM** and a base, which is a value of type **INT**, indicating the first component of the segment. This in turn may be followed by the keyword **FOR** and a count, which is a value of type **INT** which specifies the number of components in the segment. **FROM** *base* or **FOR** *count*, but not both may be omitted.

Line breaks are permitted immediately after the keyword **FROM** and the keyword **FOR**. The segment is valid only if the value of the count is not negative, and does not violate the bounds of the array. That is $(base+count)$ must not exceed the size of the array. Here is another example to consider:

```
[[c FROM j FOR i] FOR 5]
```

This complex looking segment selects the first five components of a variable which is itself a segment, it is in fact equivalent to **[c FROM j FOR 5]** provided **i** $\geq$ 5. Segments may also be subscripted, for example:

```
[x FROM n][3]
```

The subscript in this example selects component number 3 from the segment which starts at **x[n]** and continues to the last component of **x**; so it is equivalent to **x [n + 3]**.

An assignment to a variable selected by a subscript is an assignment to that component of the array, and has no effect on any other component in the array. Consider the following example:

```
x[3] := 42
```

The effect of an assignment to an array or a segment of an array, is to assign to each component the value of the corresponding component of the expression. Assignment to a segment of a variable which is an array, is not valid if a component of the expression is also a component of the array to which it is to be assigned. Thus, the following assignment is not valid:

```
[x FROM 6 FOR 6] := [x FROM 8 FOR 6] -- INVALID!
```

Both these segments share the component **x[8]**, but in different positions, so that the meaning could depend on the order in which an implementation causes the component assignments to be performed. However an assignment which assigns a segment of an array to itself is not invalid as it must always be implemented to have no effect whatsoever.

The combined effect of an input and output, in parallel processes on the same channel, of an array or a segment of an array is equivalent to an assignment from the outputting process to the inputting process. Consider the following example:

```
[x FOR 10] := [y FOR 10]
```

This is a valid assignment, and has the same effect as the following:

```
PAR
  c ! [y FOR 10]
  c ? [x FOR 10]
```

Also consider the following assignment of **v1** to **v2**, where both are arrays of type **[12]INT**:

```
v1 := v2
```

This assignment assigns each component of the array **v2** to each respective component of the array **v1**, and has the same effect as the following communication:

```
PAR
  c ! v1
  c ? v2
```

Assignment is discussed earlier on page 5, input and output are also described earlier on page 6. See the appendix (page 101) to discover how invalid processes behave.

4.3 Record fields

As has been seen in example declarations above, variables of a record type such as **COMPLEX32** are declared in the same way as variables of any other named type. Consider:

```
COMPLEX32 z :
```

This declaration introduces a variable, **z**, of type **COMPLEX32**. This variable may be assigned and communicated in the same way as a variable of a primitive or array data type.

Fields of a record are selected by subscripting the name of the record variable or value of a record literal with a field name in square brackets. The fields **z[real]** and **z[imag]** may be used like ordinary variables in expressions, assignments and communications, in fact in any context where a variable or value of that type may be used. A field that is itself an array or another record may in turn be further subscripted appropriately.

Here are some examples of record fields:

k1[power]	**REAL32** field of record of type **KETTLE**
rect1[height]	**INT** field of record of type **RECTANGLE**
[COS(pi/6.0), SIN(pi/6.0)](COMPLEX32)[imag]	field of record literal
heating.installation[i][size][width]	field of field of array component
ent1[access][2]	component of array that is a field
[ent1[fname] FROM 8]	segment of array that is a field

Note in particular the way that field selection has the form of array subscripting, but is distinguishable by the compiler which notes the use of a record name being subscripted and a field name as a subscript.

As the selection of a field from a record is syntactically indistinguishable from the subscripting of an array, there is no need for a separate definition of this syntax. The syntax of array subscripting was given in the preceding section (page 37). When the variable being subscripted is a record, then the subscript must be one of the field names declared in the definition of the type of the record.

4.4 Scope of names

Earlier sections have explained the declaration of names for variables. This section explains the *scope* of a name, which is the region of the program in which the name may legally be used.

Later chapters of the manual show how to declare other sorts of name, for instance:

```
CHAN OF BYTE c :

PROC add.to (INT x, y)
  x := x + y
:
```

A name in occam denotes one of the following:

Data Type	page 26	*Value abbreviation*	page 43
Field	page 31	*Channel abbreviation*	page 54
Variable	page 35	*Procedure*	page 69
Replication index	page 10	*Function*	page 78
Channel	page 45	*Timer*	page 81
Protocol	page 48	*Timer abbreviation*	page 83
Tag	page 49	*Retypes*	page 85
Variable abbreviation	page 42	*Reshapes*	page 88

We have considered the declaration of variables and the definition of data types. Declarations and definitions are examples of *specifications*. A specification specifies the meaning of one or more names and the scope of those names is the scope of the specification.

The syntax of specifications is:

specification = *declaration*
| *abbreviation*
| *definition*

A specification is a declaration, an abbreviation (e.g. a variable abbreviation, page 42) or a definition (e.g. a protocol definition, page 47).

All specifications are terminated by a colon. The scope of a specification begins at the start of the line following the colon. A specification may appear immediately before a process, choice (in an **IF**), option (in a **CASE**), alternative (in an **ALT**), variant (in a case input or an **ALT**), or value process (See *functions* page 75). Any name specified is then in scope for that process, choice, option, alternative, variant or value process.

The scope of a name can be seen by the level of program indentation. The scope of a name starts on the line following the colon which terminates its specification. The scope includes any other specifications which may immediately follow at the same level of indentation and any further following lines at greater levels of indentation. The scope of a specification concludes when the level of indentation returns to the same level as, or a lesser level than, the original specification.

The following example shows the scope of two declarations of variables **min** and **max**:

```
SEQ
  INT max :             -- specify max
  INT min :                -- scope of max  -- specify min
  SEQ                      --                  -- scope of min
    c ? max                --                  --
    c ? min                --                  --
    IF                     --                  --
      p < max              --                  --
        p := p + 1         --                  --
      p = max              --                  --
        p := min           --                  --
  SEQ
    ...
```

This example increments **p** if it is less than the value specified by **max**. The scope associated with the variable **p** in this example begins at the declaration of **p** earlier in the program.

The association of a name with any particular scope is either *local* or *free* A local name is specified at the start of the scope under consideration, or the name is *free* of local association as are **max** and **min** in the above example. A free name is specified at an outer level of scope (as for **p** in the above example) which includes the scope under consideration, as is **p** in the above example.

Generally all names within a scope in occam are distinct. That is, a name may only have one meaning within any scope. This rule is modified for names of record fields (page 32) or protocol tags (page 53) which may be reused in the same scope as they can always be associated with a particular data type or protocol definition.

The following syntax shows where a specification may occur:

process	=	*specification* *process*
choice	=	*specification* *choice*
option	=	*specification* *option*
alternative	=	*specification* *alternative*
variant	=	*specification* *variant*
value.process	=	*specification* *value.process*

A specification is in scope within any immediately following specifications, as shown by the recursion in the productions above.

In most cases, if a specification is performed which uses an existing name then the new meaning supersedes the old meaning for the duration of the scope of the new specification. This effect is called *hiding*.

The only exceptions to this rule are for names of record fields (page 31) and protocol tags (page 49). These names may be reused in the same scope without hiding as they can always be associated with a particular data type or protocol definition. Examples of such use are given on pages 32 and 53.

The hiding of names is illustrated by the following example:

```
INT x :                                 -- integer variable x
SEQ                                        -- scope of INT x
  dm ? x                                   --
  ALT                                      --
    REAL32 x :      -- REAL32 x hides INT x
    rs ? x              -- scope of REAL32 x
      ...               --
    dm ? y                                 --
      ...                                  --
  ...                                      --
```

The declaration of the **REAL32 x** in the above example has the effect of hiding the earlier use of the name **x** for the duration of its scope.

Because of this hiding rule, all names (except record fields and protocol tags) within a scope are distinct.

Consider the following declaration, which illustrates the use of the same name with more than one meaning in a single line:

```
[n+1]INT n:
```

The two names **n** refer to completely different objects. The scope of the new name **n** being declared as an array does not begin until after the colon, and so the name **n** which is used in the array size expression must be an **INT** constant with the same name which was already in scope before the declaration.

Keywords may not be redefined in specifications. Such names are said to be *reserved*, and an implementation may extend the list of reserved words beyond those defined in this manual (page 102). By convention no names containing lower case letters are ever reserved. An implementation may also introduce further names of *library* routines, which may be respecified by the programmer.

4.5 Abbreviation of variables

A variable abbreviation specifies a new name for a variable. Consider

```
INT n IS m :
```

This abbreviation specifies the name **n** as the new name for **m**. Also, consider the following example:

```
INT user IS lines[8] :
```

This abbreviation specifies the name **user** for a component of the array **lines**. All subscript expressions used in an abbreviation must be valid. The type of the abbreviated variable must be the same as the data type specified, so in this example, **lines** has to be an array of **INT**. Other components of the array **lines** may be used only in abbreviations within the scope (page 39) of **user**, but they must not include the component **lines[8]**. Here are some more examples of abbreviations:

x IS y :	specifies a new name **x** for **y**
INT c IS a[i] :	specifies a name for a component of the array **a**
[]REAL32 s IS [a FROM 8 FOR n] :	specifies a name for a segment of **a**
[]BYTE fname IS [fname FOR len] :	specifies the used part of an array

An abbreviation simply provides a name to identify an existing variable. The name **c** in the above example identifies the existing variable **a[i]**. In the scope of the abbreviation, **c := e** is an assignment to the original variable **a[i]**. A variable used in a subscript to select a component or components of an array may not be assigned to within the *scope* of the abbreviation. For example, no assignment or input can be made to **i** within the scope of **c**. As a result the abbreviation always refers to the same variable throughout its scope. This allows various optimisations to be performed, such as evaluating any expression within the abbreviated variable only once. The original variable **a[i]** may not be used within the scope of the abbreviation **c**.

The abbreviation of **fname** as a leading sub-array of a previously declared array of the same name is a common device for allocating an array whose size is the upper bound of possible sizes and then only using part of it in a particular scope.

The specifier can usually be omitted from an abbreviation, as the type can be inferred from the type of the variable. A specifier **[]** *type* with one or more empty dimensions defines the abbreviation as being any array with components of the specified type. The type of an array on the right hand side of an abbreviation whose specifier has an empty dimension is said to be *compatible* with the specifier.

Where an abbreviation is of a component of an array no other reference may be made to any other part of that array, except in a further abbreviation. Consider the following example:

```
[60][72]INT page :
...
  first.line IS page[0] :
  last.line  IS page[59] :
  SEQ
    first.line := last.line
    last.line  := page[58]           -- This assignment is INVALID!
    ...
    next.to.last.line IS page[58] : -- This abbreviation is valid
    last.line  := next.to.last.line -- and so too, this assignment
    ...
```

Also consider the following example:

```
WHILE i < limit
  this.line IS page[i] :
  next.line IS page[i+1] :
  SEQ
    this.line := next.line
    ...
    i := i + 1                -- this assignment is INVALID!
```

The assignment in the above example is invalid as **i** is used to select components of the array **page** in an abbreviation within the scope of the assignment. This is how the above should be written:

```
WHILE i < limit
  SEQ
    this.line IS page[i] :
    next.line IS page[i+1] :
    SEQ
      this.line := next.line
      ...
    i := i + 1
```

It is important to ensure that all the components of an array remain identified by a single name within any given *scope*. Identification of any component of an array by more than one name constitutes an invalid usage of the component, and it is especially important to be aware of this of when abbreviating components of an array. Once any component of an array is abbreviated then reference to other components of the array must be made by further abbreviation. Checks are made to ensure that two abbreviations which identify segments from the same array do not overlap. When necessary a compiler may generate code to perform this check at run time. Further discussion on abbreviation is given in the chapter on procedures (page 69).

The syntax for abbreviations of variables is:

abbreviation	=	*specifier name* **IS** *variable* **:**
	|	*name* **IS** *variable* **:**
specifier	=	*data.type*
	|	**[]** *specifier*
	|	**[** *expression* **]** *specifier*

The abbreviation of a variable begins with an optional specifier, specifying a data type. The name specified appears to the right of the optional specifier followed by the keyword **IS**, the abbreviated variable appears to the right of the keyword **IS**. The line on which the abbreviation occurs may be broken after the keyword **IS** or at some legal point in the variable. The type of the variable must be the same as the data type specified.

4.6 Abbreviation of values

The last section described variable abbreviations. This section describes abbreviations of values. Consider the example:

```
VAL INT days.in.week IS 7 :
```

This abbreviation specifies the name **days.in.week** for the value 7. This construct is always used in occam for named constants. Here are some more abbreviations for values:

VAL REAL32 y IS (m * x) + c :	specifies a name for the current value of an expression
VAL INT n IS m :	specifies a name for the current value of the variable **m**
VAL []BYTE vowels IS ['a', 'e', 'i', 'o', 'u'] :	specifies a name for a table of values

The abbreviated value must be a valid expression, so it must not overflow, and all subscripts must be in range. Variables used in an abbreviated expression may not receive new values by input or assignment within the *scope* (page 39) of the abbreviation. This ensures that the value of the expression remains constant for the scope of the abbreviation. For example, in the following abbreviation

```
VAL REAL32 y IS (m * x) + c :
```

no assignment or input may be made to **m**, **x**, or **c** within the scope of this abbreviation defining **y**. The effect of the abbreviation may be obtained by replacing each occurrence of **y** in its scope by the abbreviated value **((m * x) + c)**. Similarly for the following abbreviation of the value **[screen FROM line FOR length]**

```
VAL []INT scan IS [screen FROM line FOR length] :
```

no assignment or input may be made to **screen**, **line** or **length** within the scope of **scan**. The effect of the abbreviation is the same as each instance of **scan** being replaced by the abbreviated value, thus

```
VAL []INT scan IS [screen FROM line FOR length] :
SEQ
  row := scan
  ...
```

is equivalent to

```
SEQ
  row := [screen FROM line FOR length]
  VAL []INT scan IS [screen FROM line FOR length] :
  ...
```

The syntax for abbreviations of values is:

abbreviation	=	**VAL** *specifier name* **IS** *expression* **:**
	|	**VAL** *name* **IS** *expression* **:**

The abbreviation of a value begins with the keyword **VAL**. An optional specifier (which specifies the data type of the abbreviation) appears to the right of **VAL**, followed by the name, and the keyword **IS**. The abbreviated value appears to the right of the keyword **IS**. Line breaks are permitted after the keyword **IS**. The type of the value must be compatible with the specifier. The specifier can usually be omitted from the abbreviation, as the type can be inferred from the type of the value. An important exception to this is if the value is an undecorated literal constant. The following will be treated as equivalent by the compiler:

```
VAL REAL32 pi IS 3.14159(REAL32) :
VAL pi IS 3.14159(REAL32) :
VAL REAL32 pi IS 3.14159 :
```

The choice between these is a matter for individual preference.

4.7 Disjoint arrays in parallels

Abbreviations may be used to decompose an array into a number of disjoint parts, so that each part may have a unique name in all or several processes in parallel. Components of each disjoint part may then be selected by a variable subscript (a subscript whose value is dependent on a procedure parameter, a variable, or a replicator index whose base or count is not a constant value), for example:

```
frame1 IS [page FROM 0 FOR 512] :
frame2 IS [page FROM 512 FOR 512] :
PAR
  INT i :
  SEQ
    ...
    c1 ? frame1[i]
    ...
  INT j :
  SEQ
    ...
    c2 ? frame2[j]
    ...
```

This example divides the array **page** into two parts, and provides a name for those parts in each of the two parallel processes. These parts may then be selected by using variable subscripts.

5 Channels and their protocols

occam programs act upon *variables*, and communicate using *channels* and *timers*. A variable has a value, and may receive a new value in an *assignment* or *input*. Channels communicate values. Timers produce a value which represents time.

This chapter describes communication channels, their declaration, the specification of the format and data type of communications, and the construction of arrays of channels.

Variables (page 25) and timers (page 81) are discussed elsewhere in the manual.

A Communication channel provides unbuffered, unidirectional point-to-point communication of values between two concurrent processes, which are components of a parallel or of constructions which are themselves components of a parallel. The format and type of values passed on a channel is specified by the channel *protocol*. The name and protocol of a channel are specified in a channel declaration.

5.1 Channel type

The type of a channel is:

channel.type	=	**CHAN OF** *protocol*

The keyword **CHAN** is always followed by the keyword **OF** which is followed by a protocol according to syntax elaborated below (page 47). Channel types cannot be named, but protocols can.

5.2 Declaring a channel

A channel is declared in the same way as variables are declared. Consider the following example:

```
CHAN OF BYTE screen :
```

This declaration introduces a channel named **screen** with a protocol of type **BYTE**. The protocol in this example specifies that each communication on this channel must be a value of type **BYTE**. An output on this channel could be:

```
screen ! 'H'
```

Several channels with the same protocol can be declared together, for example:

```
CHAN OF BYTE screen, keyboard :
CHAN OF INT from.a, from.b, from.c :
```

The type of the declarations is determined, and then the declarations are made.

The syntax of channel declarations is:

declaration	=	*channel.type* $\{_1$ `,` *name* $\}$ **:**
channel	=	*name*

A channel declaration consists of the channel type, and a comma separated list of names to identify the channels. The declaration appears on a single line, which may be split after any comma, and is terminated by a colon.

5.3 Arrays of channels

Arrays of channels can be declared in the same way as arrays of variables (see page 36). The following, for example, declares an array of channels:

```
[4]CHAN OF BYTE screens :
```

This declaration introduces an array **screens** of four channels.

Multidimensional arrays of channels are built in the same way as multidimensional arrays of variables, for example:

```
[5][5]CHAN OF PACKETS node :
```

There is a subtle semantic distinction to be made between an array of data type and an array of channels. An array of variables is itself a variable, as it may receive a new value by assignment or input. However, an array of channels is not itself a channel, as only single components of the array may be used in input/output, but a means of referencing a number of distinct channels identified by consecutive subscripts. This distinction is not made in the description of the syntax of channels.

Several arrays of the same type can be declared together. Consider the following example:

```
[users]CHAN OF BYTE screen, keyboard :
```

The type of the declarations is determined, and then the declarations are made.

The syntax of channel types is extended with:

channel.type = **[***expression***]** *channel.type*

A channel type may be preceded by an expression in square brackets. The value of the expression must be a constant and is the number of components in arrays of the channel type.

5.3.1 Channel array components and segments

Components and segments of channel arrays are denoted in the same way as components and segments of variable arrays.

Subscripted names select a component of an array. Suppose **user.in** is declared as follows:

```
[12]CHAN OF MESSAGES user.in :
```

Consider the example:

user.in[9]	the tenth component of the array **user.in**, of type **CHAN OF MESSAGES**.

A segment of an array is itself an array. The segment has zero or more components, as shown in the following examples:

[user.in FROM 9 FOR 1]	the tenth component of the array **user.in**, of type **[1]CHAN OF MESSAGES**.
[user.in FROM 9]	the tenth, eleventh and twelfth components of the array **user.in**, of type **[3]CHAN OF MESSAGES**.

A segment of an array has the same number of dimensions as the array.

The syntax is:

channel = *channel***[***expression***]**
| **[***channel* **FROM** *base* **FOR** *count***]**
| **[***channel* **FROM** *base* **]**
| **[***channel* **FOR** *count***]**

This syntax matches exactly the corresponding syntax for variables (page 37).

5.4 Channel protocol

A channel communicates values between two concurrent processes. The format and data type of these values is specified by the channel protocol. The channel protocol is specified when the channel is declared. Each input and output must be compatible with the protocol of the channel used. Channel protocols enable the compiler to check the usage of channels, and to ensure the same effect whether the sending or the receiving process is ready to communicate first.

5.4.1 Simple protocols

The simplest protocols consist of a data type. Examples of channels with byte and integer protocols have already been given. A protocol with an array or record type can be declared in the same way, for example, using types declared in chapter 3:

```
CHAN OF [36]BYTE message :          -- explicit array type
CHAN OF DOSFNAME fdir :             -- named array type
CHAN OF COMPLEX32 imp :             -- named record type
```

The first declaration introduces a channel with a byte array protocol which is identified by the name **message**. The protocol of this channel specifies that each communication on the channel consists of a byte array with 36 components. The second declaration declares **fdir** to be a channel on which are communicated file names of type **DOSFNAME**, defined on page 31 to be arrays of 12 bytes. The third declaration declares a channel for communicating complex numbers, each constructed as a record of type **COMPLEX32** (page 31).

The following outputs use these channels:

```
message ! "The vorpal blade went snicker-snack."
fdir ! "AUTOEXEC.BAT"(DOSFNAME)
fdir ! sourcename
imp ! [cosx, sinx](COMPLEX32)
```

The variable **sourcename** in the third example must be an array of 12 bytes of type **DOSFNAME**. Note that in the fourth example it would be possible to omit the type decoration **(COMPLEX32)** because the type of the record can be inferred from the type of the protocol.

Counted array protocols

It is often desirable to have a channel that will pass arrays of values, where the number of components in the array is not known until the output occurs. A special protocol, called a *counted array* protocol, enables this kind of array communication by passing a length and that number of components from the array. A declaration for such a channel looks like this:

```
CHAN OF INT::[]BYTE message :
```

This declaration introduces a channel which passes an integer value and that number of components from the array. An output on this channel will look like this:

```
message ! 16::"The vorpal blade went snicker-snack."
```

This has the effect of outputting the integer **16** and then the string **"The vorpal blade"**, the first **16** bytes of the array. The associated input could look like this:

```
message ? len::[buffer FROM start]
```

This input receives an integer value (**16** in this example), which is assigned to the variable **len**, and that number of components, which are assigned to components of the array starting at **buffer[start]**. The assignments to **len** and **buffer** happen in parallel and therefore the same rules apply as for parallel assignment. That is, no name appearing to the left of **::** may be used in the array variable on its right and *vice versa*[1]. The input is invalid if the number of components in the destination array is less than the count value input to **len**.

[1] In occam 2, the count was input first and the parallel assignment rules did not apply. Some occam 2 programs are invalidated by the new rule and implementors may provide a compiler option to accept programs written assuming the old definition.

All the above protocols are called *simple protocols*, the syntax of these and of inputs and outputs using them is:

simple.protocol	=	*data.type*
	|	*data.type* **::[]** *data.type*
input	=	*channel* **?** *input.item*
input.item	=	*variable*
	|	*variable* **::** *variable*
output	=	*channel* **!** *output.item*
output.item	=	*expression*
	|	*expression* **::** *expression*
protocol	=	*simple.protocol*

This syntax has extended the earlier syntax for *input* and *output* (page 6). A simple protocol is either a data type or a counted array as described above. A counted array is specified by the data type of the count (which may be either an integer or byte type), followed by a double colon, square brackets (**::[]**), and the specifier indicating the type of the components.

An input is a channel followed by the symbol **?** and an input item. An input item may be either a variable, of any data type, or a variable of integer or byte type followed by a double colon (**::**) and a variable of an array data type.

An output is a channel followed by the symbol **!** and an output item. An output item may be either an expression, of any data type, or an expression of integer or byte type followed by a double colon (**::**) and an expression of an array data type.

5.4.2 Naming a protocol

A protocol can be given a name in a *protocol definition*, as shown in the following example:

```
PROTOCOL CHAR IS BYTE :
```

A channel can now be declared with the protocol **CHAR**, for example:

```
CHAN OF CHAR screen :
```

While the naming of simple protocols is optional, a protocol definition must be used if more complex protocols, like the *sequential protocol* described in the following section are required.

The syntax for protocol definition is:

definition	=	**PROTOCOL** *name* **IS** *simple.protocol* **:**
	|	**PROTOCOL** *name* **IS** *sequential.protocol* **:**
protocol	=	*name*

A protocol definition defines a name for the simple protocol or sequential protocol (described in the following section) which appears to the right of the keyword **IS**. A simple or sequential protocol definition appears on a single line, and is terminated by a colon. The line may be broken after the keyword **IS** or after a semi-colon in a sequential protocol.

5.4.3 Sequential protocol

Simple protocols have been discussed earlier. Sequential protocols specify a protocol for communication which consists of a sequence of simple protocols. Consider the following example:

```
PROTOCOL COMPLEX IS REAL32; REAL32 :
```

Channels declared with this protocol (**CHAN OF COMPLEX**) pass pairs of values. An input or output on a channel with sequential protocol is a sequence of distinct inputs or outputs. An input on a channel with the above protocol **COMPLEX** is shown below:

```
items ? real.part; imaginary.part
```

Each value is input in sequence and assigned to each variable in turn.

It is important to note the difference between this protocol and the simple protocol **COMPLEX32** derived from the record data type declared on page 31. Communications using a simple protocol derived from a single record type pass a complete record value of several fields in a single communication, but communications using a sequential protocol use a sequence of separate communications and so do not require either the outputting or the inputting processes to have declared variables of record type from which and to which the data will be communicated. There will be situations in which each of these alternatives is the better choice of coding style.

Consider also the example:

```
COMPLEX32 z1, z2 : -- COMPLEX32 already declared as a record data type
PROTOCOL COMPLEX32 IS COMPLEX32 :   -- unnecessary protocol definition
CHAN OF COMPLEX32 cz :
PAR
  cz ! z1
  cz ? z2
```

This is a legal occam program, but the effect of the named protocol definition is to introduce a new meaning of **COMPLEX32** as a protocol and so in the scope of that declaration it would not be possible to declare new variables of the record type **COMPLEX32**. However variables of the record type which were declared before the protocol definition are still in scope after it and may be used in processes within its scope. In the absence of the protocol definition the channel declaration would still be legal as it would reference the simple protocol derived from the type definition of **COMPLEX32**. In this case new variables of this type may be declared in the scope of the channel declaration.

Here are some more examples of sequential protocol definitions:

```
PROTOCOL DIR.ENTRY IS INT16; [14]BYTE :
PROTOCOL INODE IS INT16;INT16;INT32;INT32;INT16;[7]INT16;INT16;INT16 :
PROTOCOL LINE IS INT16::[]BYTE :
```

Declarations of channels with these protocols would look like this:

```
CHAN OF DIR.ENTRY directory :
CHAN OF INODE  sys :
CHAN OF LINE blocks :
```

The syntax of sequential protocols, which can only appear in protocol definitions, is:

sequential.protocol	=	$\{_1$ **;** *simple.protocol* $\}$
input	=	*channel* **?** $\{_1$ **;** *input.item* $\}$
output	=	*channel* **!** $\{_1$ **;** *output.item* $\}$

A sequential protocol is one or more simple protocols separated by semicolons. A communication on a channel with a sequential protocol is legal provided the type of each item input or output is compatible with the corresponding component of the protocol.

5.4.4 CASE protocol

It is often convenient to use a single channel to communicate messages with different formats. A **CASE** protocol specifies a number of possible formats for communication on a single channel. Consider the following example:

```
PROTOCOL FILES
  CASE
    request;  BYTE
    filename; DOSFNAME
    word;     INT16
    record;   INT32; INT16::[]BYTE
    error;    INT16; BYTE::[]BYTE
    halt
:
```

This example defines a case protocol named **FILES**. **CASE** combines a number of *tags*, each of which may identify a tagged sequential protocol. The case protocol defined here has six variants.

A channel declared with this protocol would look like this:

```
CHAN OF FILES to.dfs :
```

A communication on this channel first sends a tag to inform the receiving process of the format for the rest of the communication. So, for example

```
to.dfs ! request; get.record
```

first sends the tag **request** followed by a **BYTE** value (**get.record**). Consider the output:

```
to.dfs ! halt
```

This output sends only the tag **halt** and according to the above **CASE** protocol definition requires no further output.

The syntax for a case protocol and the associated output is:

definition	=	**PROTOCOL** *name*
		  CASE
		    { *tagged.protocol* }
		:

tagged.protocol	=	*tag*
	|	*tag* **;** *sequential.protocol*
tag	=	*name*
output	=	*channel* **!** *tag*
	|	*channel* **!** *tag* **;** $\{_1$ **;** *output.item* $\}$

In a definition of a case protocol the name which identifies the protocol appears to the right of the keyword **PROTOCOL**, this is followed at an indentation of two spaces by the keyword **CASE**, which in turn is followed on succeeding lines at a further indentation of two spaces by a number of tagged protocols. The definition of a case protocol is terminated by a colon, which appears on a line by itself, at the same level of indentation as the **P** of the keyword **PROTOCOL**. A tagged protocol is either a tag by itself or a tag followed by a semi-colon, and a sequential protocol.

A name may be used as a tag within multiple protocols in the same scope. A tag name may also be the same as a name defined in any other specification in the scope.

An output on a channel of case protocol is a tag by itself or a tag followed by a number of output items separated by semi-colons. The output is valid only if the tag and the associated output items are compatible with one of the tagged protocols specified in the definition of the protocol.

An implementation may restrict the number of tags in a protocol definition so that a tag may be represented by a value in a limited range, but this range must be at least 0 to 255.

Input on a channel with case protocol

So far only output on a channel with case protocol has been shown. A special form of input is required (called *case input*) to provide for input on channels with a case protocol. The previous example is suggestive of a *conversation* with a *disc filing system*, and is a reminder that channels are unidirectional. So, for a user process to “listen to” the other side of this conversation, another channel must be declared, as shown below:

```
CHAN OF FILES from.dfs :
```

This example declares another channel with the protocol **FILES**. The process which outputs **request; get.record**, might reasonably expect to receive a reply on a channel with this protocol. Consider a more complete example of this conversation:

```
SEQ
  to.dfs ! request; get.record
  from.dfs ? CASE
    record; rnumber; rlen::buffer
      ...  do whatever
    error;  enumber; elen::buffer
      ...  handle error
```

Illustrated in the above example is a case input on the channel **from.dfs**. This accepts a variant input with either the tag **record** or the tag **error**; any other tag would be invalid and the input would behave like the primitive process **STOP**.

A special form of case input simply receives a tag from the channel named on the left of the case input symbol (**? CASE**), and then compares the tag for equality with the tag of the tagged list which appears to the right of the symbol. A tag is input, then if the tags match the process next inputs the remainder of the tagged list. If the tags do not match, the process next behaves like the primitive process **STOP**. For example:

```
from.dfs ? CASE filename; name.buffer
```

This process inputs a tag. If the tag is **filename** the input is completed, and a value assigned to the variable **name.buffer**. Otherwise, no further input is performed, and the input behaves like the primitive process **STOP** (page 7). A case input is legal only if the tagged lists are compatible with tagged protocols specified in the definition of the case protocol.

Consider the following:

```
PROTOCOL COMMS
  CASE
    packet;INT::[]BYTE
    sync
:
CHAN OF COMMS route :
PAR
  SEQ
    route ! packet; 11::"Hello world"
    R ()
  SEQ
    route ? CASE sync
    S ()
```

In this example the input **route ? CASE sync** will behave like the primitive process **STOP** as the tags do not match. The associated output will also behave like **STOP**, for although the output of the tag **packet** succeeds, the output **11::"Hello world"** does not. In this example the procedures **R()** and **S()** will not be performed. Also consider the following:

```
PAR
  SEQ
    route ! sync
    P ()
  SEQ
    route ? CASE packet; length::message
    Q ()
```

Each communication of a sequential protocol, or of a tagged sequential protocol is in fact a sequence of separate communications. So, in the above example, the input **route ? CASE packet; length::message** will behave like the primitive process **STOP** because the tags do not match. However, the associated output **route ! sync** will succeed as the output of the tag has completed, and the variant requires no further output. Thus, the *procedure* (page 69) **P()** will be performed, and the procedure **Q()** will not be performed.

The syntax for case input is:

case.input	=	*channel* **? CASE**
		{ *variant* }
variant	=	*tagged.list*
		process
	|	*specification*
		variant
tagged.list	=	*tag*
	|	*tag* **;** $\{_1$ **;** *input.item* }
process	=	*case.input*
input	=	*channel* **? CASE** *tagged.list*

A process which is a case input receives a tag from the channel named on the left of the case input symbol (**? CASE**), and then the tag is used to select one of the variants. These appear on the following lines, indented by two spaces and optionally preceded by local specifications. A tag is input, then if a variant with that tag is present the process next inputs the remainder of the tagged list, and an associated process, indented a further two spaces, is performed. If no variant with that tag is found the process next behaves like the primitive process **STOP**.

An input may contain the case input symbol followed by a single tagged list only, as shown in the earlier examples.

Variants in alternatives

A case input may also be used as an input in an alternation (chapter 2, page 19). Consider the following example:

```
ALT
  from.dfs ? CASE
    request; query
      ...  do query
    error; enumber; elen::buffer
      ...  handle dfs error
    record; rnumber; rlen::buffer
      ...  accept record

  from.network ? CASE
    request; query
      ...  do query
    error; enumber; elen::buffer
      ...  handle network error
    record; rnumber; rlen::buffer
      ...  accept record
```

This alternation accepts input from either of the two channels (**from.dfs** and **from.network**). These inputs are explained in the previous section. This alternation could have included a mix of case inputs, and the alternatives described on page 19. The syntax for case inputs in an alternative is:

alternative	=	*channel* **? CASE**
		{ *variant* }
	|	*boolean* **&** *channel* **? CASE**
		{ *variant* }

A case input as an alternative is either a case input with variants as described in the earlier syntax, or such a case input preceded by a boolean guard and an ampersand (`&`) to the left of the channel name. The case input is not considered by the alternation if the boolean guard is false.

Scope of tag names

A name may be used as a tag within more than one case protocol in the same scope. For example:

```
PROTOCOL LINES
  CASE
    aline; INT::[]BYTE
    eof
:
PROTOCOL HEATING
  CASE
    kettle; KETTLE
    heater; HEATER
    eof
:
CHAN OF LINES document:
CHAN OF HEATING installation:
PAR
  ...
  SEQ
    ...
    document ! eof                  -- eof of LINES
    ...
    installation ! eof              -- eof of HEATING
    ...
```

It is possible for a tag name of a tagged protocol to be used again in a specification or as a field name in a record during the scope of the protocol. For example:

```
PROTOCOL COMMS
  CASE
    packet;INT::[]BYTE
    sync
:
[4]BYTE packet :
```

However, it is still possible to use the name **`packet`** to denote a tag of the protocol **`COMMS`** in outputs and **`CASE`** inputs on channels with protocol **`COMMS`**. Consider the following simple program:

```
PROTOCOL COMMS
  CASE
    packet;INT::[]BYTE
    sync
:
[4]BYTE packet :
CHAN OF COMMS c :
INT len :
PAR
  c ! packet; 3::"xyz"
  c ? CASE packet; len::packet
```

The name **`packet`** is used to denote the variant tag in both the input and the output. The name is also used to specify the destination of the input. Confusing usages such as this are usually better avoided.

5.4.5 Anarchic protocol

In some situations it may be necessary to specify a channel protocol where the format of the protocol for some reason cannot be defined. Such situations are rare, and are likely to occur only when communicating with an external device such as a printer, terminal or other device controller. Such a device can be considered an *alien process* where the protocol for communication with that process is dictated by the nature of the device. A special protocol exists which allows the input and output of any format without checking. The protocol is specified by the keyword **ANY**, as illustrated in the following examples:

```
CHAN OF ANY mouse:
PROTOCOL PRN IS ANY :
CHAN OF PRN printer :
```

The keyword **ANY** may be used directly in a channel declaration such as that of **mouse** above, or it may be used as a simple protocol as in the definition of **PRN**.

A channel with a protocol defined to be **ANY** in either of these ways can only input or output data values. It cannot handle case protocols. The effect of an output on a channel with an **ANY** protocol is that the value is mapped down into its constituent bytes, and output as an array of bytes. An input on a channel with any **ANY** protocol inputs the array of bytes and converts (by retyping conversion, see page 85) the value to the type of the receiving variable.

A protocol defined to be **ANY** may be redefined to have any other protocol by a channel retyping definition (page 86). Such retypings are necessary when passing a channel with anarchic protocol as a actual parameter to a procedure whose corresponding formal has any other protocol.

The syntax extends that of simple protocol:

simple.protocol	=	**ANY**

5.5 Abbreviation of channels

Channel abbreviations are similar to variable abbreviations (see page 42). A channel abbreviation specifies a new name for a channel or a channel array. Consider the following examples:

```
[]CHAN OF INT clients IS [from.a, from.b, from.c, from.d, from.e] :
CHAN OF INT client2 IS clients[2] :
clients01 IS [clients FOR 2] :
otherclients IS [clients FROM 3] :
```

This introduces the name **clients** for the constructed array shown. It then introduces **client2** as the new name for **clients[2]** and **clients01** and **otherclients** for non-overlapping segments of the array **clients**.

The syntax of channel abbreviation is:

abbreviation	=	*specifier name* **IS** *channel* **:**
	|	*name* **IS** *channel* **:**
	|	*specifier name* **IS** **[**$\{_1$ **,** *channel* $\}$**]** **:**
	|	*name* **IS** **[**$\{_1$ **,** *channel* $\}$**]** **:**
specifier	=	*channel.type*
	|	**[]** *specifier*
	|	**[** *expression* **]** *specifier*

The specifier must specify a channel type but may be omitted whenever the type of the abbreviation can be inferred from the type of the channel. Channel abbreviations are subject to the same usage restrictions as variable abbreviations. These are summarised in appendix E.

5.5.1 Channel array constructors

In the syntax above it can be seen that a channel array in an abbreviation may be constructed as a table of individual channels. Consider the example:

```
sources IS [from.a, from.b, from.c] :
sources[i] ? x
```

The three channels originally declared separately have been constructed into an array which can then be named in an abbreviation so that it may be subscripted or segmented. This can be particularly useful when **i** is the replication index of a replicated alternation, and the construction is being used to receive input on one of a group of channels of the same protocol.

This example shows how the constructor might be used to combine an arbitrary collection of channels of the same protocol for use by a process which listens to all of them:

```
CHAN OF BYTE from.a, from.b, from.c:
PAR
  SEQ
    ...  process a
    from.a ! 'A'
    ...
  SEQ
    ...  process b
    from.b ! 'B'
    ...
  SEQ
    ...  process c
    from.c ! 'C'
    ...
  sources IS [from.a, from.b, from.c]:
  [3]BYTE bb:
  SEQ j = 0 FOR 3
    ALT i = 0 FOR 3
      sources[i] ? bb[i]
        SEQ
          ...
```

This example has been so simplified that the need for a constructor is not immediately obvious. The need becomes greater if the individual channels are allocated to non-contiguous hardware addresses (page 94).

The channels combined in a channel array constructor must all be used in the same direction. Channel array constructors can only be used within channel abbreviations (and as actual parameters to procedure instances, see page 70).

6 Expressions

This chapter is about *expressions*, and describes the range of *operators* provided by occam. The chapter also describes *tables* and *data type conversions*.

An expression is evaluated and produces a result. The result of an expression has a value and a data type. The simplest expressions are literals and variables. More complex expressions are constructed from *operands*, *operators* and *parentheses*. An operand is a *variable* (page 35), a literal, a table, or an expression in parentheses. An operator performs an operation, for example an addition, upon its operand(s). The following are all valid expressions:

5(INT64)	a literal value
x	a variable
6 * 4	multiplication of two literal operands
x * y	multiplication of two variable operands
y << 2	shift of a variable operand by a literal
x = y	comparison of two variable operands
NOT TRUE	a boolean expression

An expression in parentheses may itself be an operand in an expression. In this way larger expressions are built, as shown in the following examples:

(1 + 2) - 1	subtract **1** from the result of **(1 + 2)**
(x * y) * (w * z)	multiply the results of the expressions **(x * y)** and **(w * z)**

There is no operator precedence and so the hierarchical structure of a large expression must be defined by parentheses. With the exception of shift operations, where the number of bits to be shifted must be indicated by a value of type **INT**, the data type of the two operands in a dyadic expression must be of the same type. In an assignment the value of the expression must be of the same data type as the variable to which it is to be assigned. Consider in detail the following example:

```
y := (m * x) + c
```

Each of the variables in this example (**y**, **m**, **x** and **c**) must be of the same data type. The value of an expression is of the same type as its operand(s). The expression in this example - **(m * x) + c** - has two operators. The parentheses indicate that the expression **(m * x)** is an operand of the operator **+**, and thus must be evaluated before the **+** operation can be performed.

The right hand second operand of a shift operator must always be, or be converted to be, of the explicit type **INT**. Apart from this, where an operator is defined in this chapter to operate on operands of a particular primitive data type, it may be assumed also to operate on operands of a named type derived from that underlying type. The two operands of a dyadic operator (excluding shifts) must both be of the same type, and so must be explicitly converted (see page 66) to the same type if they are declared as, or computed to be, of different types.

The syntax for expressions includes:

expression	=	*monadic.operator operand*
	|	*operand dyadic.operator operand*
	|	*operand*
	|	*conversion*
operand	=	*variable*
	|	*literal*
	|	*table*
	|	**(***expression***)**
variable	=	*name*
monadic.operator	=	**-** | **MINUS** | **~** | **BITNOT** | **NOT** | **SIZE**
dyadic.operator	=	**+** | **-** | ***** | **/** | **** | **REM** | **PLUS** | **MINUS** | **TIMES**
	|	**/** | **\\/** | **><** | **BITAND** | **BITOR** | **AND** | **OR**
	|	**=** | **<>** | **<** | **>** | **>=** | **<=** | **AFTER**

A monadic operator precedes its one operand. A dyadic operator appears between its two operands. Tables, operators and conversions are detailed in the following sections. Variables (page 35) and literals (page 27) have been explained earlier. An operand which is a table can only be used where an array may be used. Conversions are decribed on page 66.

6.1 Tables and strings

A table constructs an array of values from a number of expressions which must yield values of the same data type. A string is a compact representation of a table of bytes. The value of each component of the array is the value of the corresponding expression. Consider the following example:

```
[1, 2, 3]
```

This example constructs an array with three components, each of type **INT**. Here are some more examples:

['a', 'b', 'c']	a table of three bytes (equivalent to **"abc"**)
[x, y, z]	a table of three values
[x * y, x + 4]	a table with two component values
[(m * x) + c]	a table with a single component
[6(INT64), 8888]	a table of two **INT64** integers
"occam"	a string literal
"RUNHELLO.BAT"(DOSFNAME)	a string literal of a named type

A table that is not a string is one or more expressions of the same data type, separated by commas, and enclosed in square brackets, optionally followed by the type name of an array or record type in parentheses. Line breaks are permitted after a comma.

A table is a value of an array type. If the variables **m**, **x** and **c** are of type **INT**, then the table **[(m * x) + c]** is an expression whose type is **[1]INT**. **['a', 'b', 'c']** is an expression whose type is **[3]BYTE**, and so on.

A table may be subscripted or segmented like an array variable (page 35).

If a table appears in a context where its type is known, the type of its components are then known and may be untyped literals (page 29) whose type is implicitly determined. However if a table appears in a context where its type is not known the type of its components will be inferred from the first component found to have a known type. Consider:

```
VAL []REAL32 tab1 IS [0.0, 1.1] :
VAL tab2 IS [2.0(REAL32), 2.1, 2.2] :
VAL tab2a IS [1.9, 2.0(REAL32), 2.2] :
VAL tab3 IS [3.0, 3,1] :     -- ILLEGAL
DATA TYPE VEC2 IS [2]REAL32 :
VAL tab4 IS [4.0, 4.1](VEC2) :
```

The type of **tab1** is determined from the specifier **[]REAL32**. The type of **tab2** is determined from its first component which is an explicitly decorated literal. The type of **tab2a** is determined from its second component which is an explicitly decorated literal. The type of **tab3** cannot be determined and so the abbreviation is illegal. The type of **tab4** is determined by the explicit decoration of the table, which must be done by a type name rather than a type specifier including subscript brackets.

A string is a sequence of characters enclosed in double quotes. A string of a named byte array type may be decorated with a type name in parentheses. Note that though a string is lexically a literal constant, it is treated syntactically as a table of bytes.

The type of a string is an array of type **[]BYTE**. The string **"zen"** is an array of type **[3]BYTE**. Each component of the array is the ASCII value of the corresponding character in the string.

Special character sequences allow certain control values such as Tabulation and Carriage Return to be included in strings. Full details of the occam character set and special characters are given in the appendix (page 103).

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. The indentation of the continuation should be no less than the current indentation, as illustrated in the following example:

```
occam := "Beware the jabberwock my son, the jaws that bite, the*
         * claws that catch, beware the jubjub bird, and shun the*
         * frumious bandersnatch."
```

The syntax for tables and strings is:

table	=	*string*
	|	*string* **(** *name* **)**
	|	**[** $\{_1$ **,** *expression* $\}$ **]**
	|	**[** $\{_1$ **,** *expression* $\}$ **]** **(** *name* **)**
	|	*table* **[** *expression* **]**
	|	**[** *table* **FROM** *base* **FOR** *count* **]**
	|	**[** *table* **FROM** *base* **]**
	|	**[** *table* **FOR** *count* **]**

A table is either a string or is a sequence of expressions separated by commas inside a pair of square brackets. It may optionally be decorated by a an array data type name in parentheses. A table may be subscripted or segmented in the same way as an array variable (see page 37). A segmented or subscripted table may not be decorated.

Tables are syntactically very similar to record literals, (page 32), the difference being that all components of a table must be of the same type, and so the application of the rules of literal type inference are different within a table.

6.2 Operations on values

An operation evaluates its operand(s) and produces a result. The result of an operation has a value and a data type. These are the operators which operate on values, which are defined in related groups in subsections below:

+	addition	**>>**	shift right
-	subtraction	**<<**	shift left
*****	multiplication	**AND**	boolean and
/	division	**OR**	boolean or
\ REM	remainder	**NOT**	boolean not
PLUS	modulo addition	**=**	equal
MINUS	modulo subtraction	**<>**	not equal
TIMES	modulo multiplication	**<**	less than
/\ BITAND	bitwise and	**>**	greater than
\/ BITOR	bitwise or	**<=**	less than or equal
><	bitwise exclusive or	**>=**	greater than or equal
~ BITNOT	bitwise not	**AFTER**	later than
SIZE	array size	**BYTESIN**	element size

Some operators are symbols, others keywords. Those symbols which use characters which may be absent in some national variants of ASCII (i.e. **** or **~**) have alternative representations as keywords.

6.2.1 Arithmetic operators

The arithmetic operators are:

+	addition
–	subtraction
*****	multiplication
/	division
\ REM	remainder

Arithmetic operators perform an arithmetic operation upon operands of the same real, integer or byte type (not on booleans), for example:

39 + 3	produces a value of 42
'T' + 32	produces a value equal to **'t'**
45 – 3	produces a value of 42
6 * 7	produces a value of 42
2.0(REAL32) * 3.14159	produces an approximation to 2π
126 / 3	produces a value of 42
128 REM 3	produces a value of 2

The final example in this list may also be written: **128 \ 3**. The symbols **REM** and **** both signify the remainder operation. A remainder operation produces a value which is the remainder of the division of the two operands. The sign of an integer remainder operation is the sign of the left hand operand (except where the result is zero) regardless of the sign of the right hand operand. The result of an integer division is rounded toward zero (i.e. truncated), for example:

3 / 2	produces a value of 1
(–3) / 2	produces a value of -1
(–9) / 4	produces a value of -2
(–9) REM 4	produces a value of -1

The operator **–** is also a monadic negation operator, which has the effect of negating the value of its operand which must be of an integer or real type, for example:

– x	has the value (0 – **x**)
– 5	minus 5
– 1.0E+06(REAL32)	minus one million

An arithmetic operation produces a result of the same data type as the operands. An arithmetic operation is not valid if the resulting value cannot be represented by the same data type as the operands, for example where the result of a multiplication of two large integers produces a value which exceeds the range of the type (arithmetic overflow). Division by zero is also treated as invalid.

Remainder operations, on both integers and reals, obey the following law:

$$((x/y) * y) + (x\,\mathbf{REM}\,y) = x$$

Here are some examples of real expressions, in which **x** is a value of **39.0(REAL32)**, and **y** is a value of **3.0(REAL32)**:

x + y	produces a value of 42.0 of type **REAL32**
x – y	produces a value of 36.0 of type **REAL32**
x * y	produces a value of 117.0 of type **REAL32**
x / y	produces a value of 13.0 of type **REAL32**
x REM y	produces a value of 0.0 of type **REAL32**

Rounding the results of real operations

The result of a real arithmetic expression (which is considered to be infinitely precise) is rounded to the nearest value which can be represented by the type. That is, the value will be adjusted, if necessary, to fit into the representation of its type. The precision of an operation is that of the type of the operands.

It is possible for the result of a real remainder operation to be negative. Consider the following example:

```
1.5(REAL32) REM 2.0(REAL32)
```

The result of this expression is (-0.5). If x and y are real values, the result of x **REM** y is $(x-(y*n))$, where n is the result of dividing x and y rounded toward zero. Applying this to the above example, n is 0.75 rounded to the nearest integer (1), leaving : $(1.5-(2.0*1)) = (-0.5)$.

Full details of IEEE rounding modes are given in the appendix (page 98).

6.2.2 Modulo arithmetic operators

The modulo arithmetic operators are:

PLUS	modulo addition
MINUS	modulo subtraction
TIMES	modulo multiplication

These modulo arithmetic operators perform an operation upon operands of the same integer or byte data type (not on reals or booleans). Whilst the effect of these operations is similar to the corresponding arithmetic operations, no overflow checking takes place, and thus the values are cyclic. For example, adding one to the most positive integer will produce a value equal to the most negative integer (i.e. ($MOSTPOS$ **PLUS** 1) = $MOSTNEG$), and subtracting one from the most negative integer will produce a value equal to the most positive integer (i.e. ($MOSTNEG$ **MINUS** 1) = $MOSTPOS$). Consider these examples:

`32767(INT16) + 1(INT16)`	causes an arithmetic overflow. **INVALID!**
`32767(INT16) PLUS 1(INT16)`	produces the value -32768.
`(-32768(INT16)) - 1(INT16)`	causes an arithmetic overflow. **INVALID!**
`(-32768(INT16)) MINUS 1(INT16)`	produces the value 32767.
`20000(INT16) * 10(INT16)`	causes an arithmetic overflow. **INVALID!**
`20000(INT16) TIMES 10(INT16)`	produces the value 3392
`255(BYTE) + 1(BYTE)`	causes an arithmetic overflow. **INVALID!**
`255(BYTE) PLUS 1`	produces the byte value 0
`128(BYTE) TIMES 3`	produces the byte value 128

MINUS is also a valid monadic operator. **MINUS z** is always equivalent to **0 MINUS z** and so if **z** is of a byte type then it is equal to **256 - z**.

6.2.3 Bit operations

Bitwise operators perform operations on the bit pattern of a value of an integer or byte type. The bitwise operators are:

/\ BITAND	bitwise and
\/ BITOR	bitwise or
><	bitwise exclusive or
~ BITNOT'	bitwise not

Here are some example expressions using the bitwise operators. The results shown are correct if the value of **pixel** is **#1010**, and the value of **pattern** is **#FFFF**, and their type is **INT16**:

pixel /\ pattern	produces a result **#1010(INT16)**
~ pixel	produces a result **#EFEF(INT16)**
pixel \/ pattern	produces a result **#FFFF(INT16)**
pixel >< pattern	produces a result **#EFEF(INT16)**

The operands of **/**, **\/** and **><** must both be of the same integer or byte type. The following table illustrates how each bit of the result is produced from the corresponding bits in the operand.

1 >< 0 = 1	1 /\ 0 = 0	1 \/ 0 = 1
0 >< 0 = 0	0 /\ 0 = 0	0 \/ 0 = 0
1 >< 1 = 0	1 /\ 1 = 1	1 \/ 1 = 1
0 >< 1 = 1	0 /\ 1 = 0	0 \/ 1 = 1

The bitwise not operator (**~**) has a single operand which must be of an integer or byte type. Each bit of the result is the inverse of the corresponding bit in the operand, as shown in the following table:

˜1 = 0
˜0 = 1

The result of a bitwise operation is of the same integer or byte type as the operand(s). The keywords **BITAND**, **BITOR** and **BITNOT** are equivalent to **/**, **\/**, **~** respectively, and are included especially for implementations which have a restricted character set.

6.2.4 Shift operations

The shift operators perform a logical shift on a value of an integer or byte type. The shift operators are:

>>	shift right
<<	shift left

The shift operators shift the bit pattern of a value of any integer or byte type by a number of places determined by a count value which must be of type **INT**. For example, if the value of **n** is **#FFFF**, and of type **INT16**:

n << 4	produces a result **#FFF0(INT16)**
n >> 4	produces a result **#0FFF(INT16)**

The result is of the same integer or byte type as **n**. The bits vacated by the shift become zero, the bits shifted out of the pattern are lost. The left shift operator shifts toward the most significant end of the pattern, the right shift operator shifts toward the least significant end of the pattern.

Consider these further examples, where **n** is a value of type **INT32**:

n << 0	produces the value **n**
n >> 0	produces the value **n**
n >> 32	produces the value 0
n << 32	produces the value 0

A shift by a negative value, or by a value which exceeds the number of bits in the representation, is invalid.

6.2.5 Boolean operations

The boolean operators operate on operands of boolean type, and produce a boolean result. The boolean operators are:

AND	boolean and
OR	boolean or
NOT	boolean not

The following table shows the results for each operation:

$false$	**AND**	$true$	$= false$	$false$	**OR**	$true$	$= true$	**NOT**	$false$	$= true$
$false$	**AND**	$false$	$= false$	$false$	**OR**	$false$	$= false$	**NOT**	$true$	$= false$
$true$	**AND**	$false$	$= false$	$true$	**OR**	$false$	$= true$			
$true$	**AND**	$true$	$= true$	$true$	**OR**	$true$	$= true$			

The operand to the left of a boolean operator is evaluated, and if the result of the operation can be determined evaluation ceases. This differs from the behaviour of other expressions. Consider the following example:

```
IF
  ((ch >= 'a') AND (ch <= 'z')) OR ((ch >= 'A') AND (ch <= 'Z'))
    ...
  (ch = cr) OR (ch = down) OR (ch = up)
    ...
  ((ch = escape) AND shift)) OR ((ch = escape) AND control))
    ...
```

Note that parentheses may be omitted between expressions containing adjacent **AND** or **OR** operators. The evaluation of the boolean expression **((ch >= 'a') AND (ch <= 'z'))** ceases if the expression **(ch >= 'a')** is false, in which case the evaluation of the expression **(ch <= 'z')** does not take place. If the result is true, the expression **((ch >= 'A') AND (ch <= 'Z'))** to the right of **OR** is not evaluated. The rule is that evaluation of a boolean expression will cease if the operand to the left of **AND** is false, or if the operand to the left of **OR** is true.

6.2.6 Relational operations

The relational operators perform a comparison of their operands, and produce a boolean result. The relational operators are:

=	equal
<>	not equal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal

Here are examples of relational expressions using **=** and **<>**. In these examples the operands, **x** and **y**, can be any primitive data type:

x = y	is true if the value of **x** is equal to the value of **y** the result is false otherwise
x <> y	is true if the value of **x** is not equal to the value of **y** the result is false otherwise

The following are examples using the other relational operators. In these examples the operands, **x** and **y**, can be an integer, byte or real type, but may not be a boolean:

x < y	is true if the value of **x** is less than the value of **y** the result is false otherwise
x > y	is true if the value of **x** is greater than the value of **y** the result is false otherwise
x <= y	is true if the value of **x** is less than or equal to the value of **y** the result is false otherwise
x >= y	is true if the value of **x** is greater than or equal the value of **y** the result is false otherwise

AFTER (later than)

The special modulo operator **AFTER** performs a comparison operation, and returns a boolean result, for example:

```
(a AFTER b)
```

This expression is true if **a** is later in a cyclic sequence than **b**, just as one o'clock *pm* can be considered later than eleven o'clock *am*. The first operand is considered the starting point on a "clock face" of integer values. If the shortest route to the value of the second operator is clockwise, then the value is later than the first operand and the result of the expression is true. If the shortest route to the value of the second operand is anticlockwise, or the routes are equal in length, then the value of the second operand is earlier, and the result of the expression is false.

If the type of the operands is a (signed) integer type (a **AFTER** b) produces the same value as (a **MINUS** b) > 0.

If the type of the operands is a byte type (which is unsigned) (a **AFTER** b) produces the same value as ((a **MINUS** b)**MINUS**1) < 127.

6.2.7 **SIZE** (number of components in an array)

The special operator **SIZE** may have a single operand of any array type, and produces an integer value of type **INT**, equal to the number of components in the array. For example, if **a** is an array of type **[8]INT**, then:

SIZE a	produces the value 8

If **b** is of type **[8][4]DOSFNAME** (see page 31), then:

SIZE b	produces the value 8
SIZE b[1]	produces the value 4
SIZE b[0][0]	produces the value 12

This operator is most often used to determine the size of an array passed as a formal parameter in a procedure or function (pages 69 and 76). The operator **SIZE** may also be applied to named array data types (page 65).

6.2.8 **BYTESIN** (bytes occupied by implementation of array element)

The operation of **BYTESIN** on operands is discussed below together with its operation on types (page 65).

6.3 Operations on types

An operation on a type produces a result based on a property of the implementation of all values of the type. The result of an operation on a type has a value and a data type.

6.3.1 MOSTPOS and MOSTNEG (integer range)

MOSTNEG	most negative
MOSTPOS	most positive

The operator **MOSTPOS** produces the most positive value of an integer or byte type. The operator **MOSTNEG** produces the most negative value of a (signed) integer type, and the value 0 for an (unsigned) byte type. The type of the result is the same as the operand. Consider the following examples:

MOSTPOS BYTE	has the value 255
MOSTPOS INT16	has the value 32767
MOSTPOS INT32	has the value 2147483647
MOSTPOS INT64	has the value 9223772036854775807
MOSTNEG BYTE	has the value 0
MOSTNEG INT16	has the value −32768

The syntax for these operators extends that of *expression*, and is:

expression = **MOSTPOS** *data.type*
| **MOSTNEG** *data.type*

The keyword (**MOSTPOS** or **MOSTNEG**) appears to the left of the name of an integer or byte data type.

6.3.2 SIZE (number of components in an array type)

SIZE	array size

The operator **SIZE** may have a single operand which is an array data type, and produces an integer value of type **INT**, equal to the number of components in arrays of that type. For example, if **DOSFNAME** is a name defined as an array of type **[12]BYTE**, then:

SIZE DOSFNAME	produces the value 12

The syntax for **SIZE** applied to types extends that of *expression*, and is:

expression = **SIZE** *data.type*

where the data type is any array data type. The operator **SIZE** may also be applied to arrays, see page 64.

6.3.3 BYTESIN and OFFSETOF (field positions in records)

BYTESIN	bytes in a record
OFFSETOF	offset of record field

It is sometimes necessary to discover how the compiler has mapped the fields of a record data type onto the computer's memory. The mapping may be determined by applying the **BYTESIN** operator to discover the total number of bytes occupied by a variable of the record type, and the **OFFSETOF** operator to a pair of names to discover the offset of a field within variables of a particular record type. The values returned by these operators are of type **INT**.

The **BYTESIN** operator may be applied to any data type or to any expression yielding a value of that type and returns the number of bytes that would be occupied by a component of that type in an array. This takes account of any need to add padding at the end of a record to meet implementation dependent alignment restrictions. This implies that:

BYTESIN([n**]** *type***)** $= n *$ **(BYTESIN(***type***))**

It is illegal to apply **BYTESIN** to an object whose size cannot be represented as an integral number of bytes.

The **OFFSETOF** operator is applied to two operands in parentheses. The first operand must be the name of a record data type and the second operand the name of a field declared in the definition of that type. The value returned is the offset in bytes from the start of the record in memory and the start of the field indicated.

OFFSETOF is invalid if its first operand is not the name of a record or packed record type, or if its second operand is not a field of that record type. It is invalid to apply **OFFSETOF** to a field whose offset cannot be represented as an integral number of bytes.

Let us assume that the following record type meets any restrictions on alignment imposed by the implementation and that the compiler has not needed to force the insertion of padding bytes. If so the operations yield the results indicated:

```
DATA TYPE MIXED
  PACKED RECORD
    BYTE b1, b2:
    INT16 i1:
    REAL32 r1:
    REAL64 r2:
    DOSFNAME n1:
:
MIXED m1 :
INT a, b, c, d, e, ob1, ob2, or2, on1 :
SEQ
  a := BYTESIN (MIXED)          -- 28
  b := BYTESIN (m1)             -- 28
  c := BYTESIN (m1[r1])         -- 4
  d := BYTESIN (m1[n1])         -- 12
  e := BYTESIN (m1[n1][0])      -- 1
  ob1 := OFFSETOF (MIXED, b1)   -- 0
  ob2 := OFFSETOF (MIXED, b2)   -- 1
  or2 := OFFSETOF (MIXED, r2)   -- 8
  on1 := OFFSETOF (MIXED, n1)   -- 16
```

The syntax for operands using these operators is:

operand	=	**BYTESIN (** *operand* **)**
	|	**BYTESIN (** *data.type* **)**
	|	**OFFSETOF (** *name* **,** *field.name* **)**

The operand of **BYTESIN** may be syntactically either an operand or any data type. The operands of **OFFSETOF** must be names, the first of a structured data type and the second of a field of that type. Note that **BYTESIN** and its operand in parentheses and **OFFSETOF** and its operands in parentheses are treated syntactically as operands, as if they were function instances, see page 78.

6.4 Data type conversion

With the exception of logical shifts (where the number of bits to shift must be of type **INT**), the types of the operands in an expression must be of the same type. Operands may explicitly have their data type converted.

A data type conversion permits a value of any non-array data type to be converted to a numerically similar value of another non-array data type. A data type conversion produces the value of its operand as a value of the specified data type, for example:

```
j := (k * 4.5(REAL64)) * (REAL64 n)
```

The value of **n** in this example is converted to a value of type **REAL64**. Note that **4.5(REAL64)** is a literal value of type **REAL64**, whereas **(REAL64 n)** is a data type conversion of the value of **n**.

The syntax for data type conversions is:

conversion	=	*data.type operand*
	|	*data.type* **ROUND** *operand*
	|	*data.type* **TRUNC** *operand*

The data type must be a primitive data type, or a named type derived from a primitive data type and appears to the left of the operand. A data type conversion which includes the keyword **ROUND** as described by the syntax, produces a value rounded to the nearest value of the specified type. Where two values are equally near, the value is rounded toward the nearest even number. A data type conversion which includes the keyword **TRUNC** as described by the syntax, produces a value truncated (rounded toward zero) to a value of the specified type. A real value of appropriate magnitude can be rounded or truncated to a value of type **BYTE**.

A conversion between any of the integer types, and conversions between those types and type **BYTE**, is valid only if the value produced is within the range of the receiving type. Byte and integer values may be converted to boolean values if their value is one or zero. The boolean value is true if the value is one, and false if the value is zero. That is:

BOOL 1	evaluates to **TRUE**
BOOL 0	evaluates to **FALSE**
INT TRUE	evaluates to 1
INT FALSE	evaluates to 0

Conversions from integer or byte values to real values, and vice versa, must specify whether the result is to be rounded or truncated. A value of type **REAL32** can be extended to an exact value of type **REAL64**. Values of type **REAL64** can be converted to values of type **REAL32**, providing the value is in the range of the **REAL32** type. The conversion must specify if the value is to be rounded or truncated. Consider these examples, where **n**, and **m** are integers of type **INT64**, and **n** has a value 255 and **m** has a value 3:

BYTE n	produces a byte value 255
REAL32 ROUND n	produces a **REAL32** value 255.0
REAL64 TRUNC n	produces a **REAL64** value 255.0
REAL64 ROUND(n * m)	produces a **REAL64** value 765.0
(REAL64 ROUND n) * (REAL64 ROUND m)	produces a **REAL64** value 765.0

Conversions may be applied to operands of the same type, but will have no effect. The truncation and rounding of integer types to real types occurs where the integer cannot be exactly represented as a value of the real type. Consider the following example:

```
SEQ
  i := 33554435 (INT32)  -- hex #2000003
  a := REAL32 ROUND i
  b := REAL32 TRUNC i
```

The value in this example has been chosen specifically to illustrate the behaviour of explicitly rounding an integer value which cannot be directly represented in the floating point representation of **REAL32**. The value of **a** after this sequence is 33554436.0, and the value of **b** is 33554432.0. For **b**, the two least significant bits of the integer representation have been lost (they had held the value 3). For **a** the value of those bits has been rounded to the next nearest representable value. Further detail of rounding is given in the appendix on page 98.

Conversion of real values to integers has the effect illustrated by the following examples:

INT32 ROUND 0.75(REAL32)	produces a value of 1
INT32 ROUND 0.25(REAL32)	produces a value of 0
INT32 TRUNC 0.75(REAL32)	produces a value of 0
INT32 TRUNC 0.25(REAL32)	produces a value of 0

Consider these examples, where **x**, and **y** are type **REAL32**, **x** has a value 3.5, **y** has a value 2.5.:

INT16 TRUNC y	produces the value 2, **y** truncated
INT16 ROUND y	produces the value 2, **y** rounded to nearest even
INT32 ROUND x	produces the value 4, **x** rounded to nearest even
INT16 TRUNC (x / y)	produces the value 1
(INT ROUND x) * 10	produces the value 40
REAL64 x	produces the value 3.5

A full explanation of the IEEE rounding modes is given in the appendix (page 98).

Type conversions may be needed if named types have been introduced to improve type abstraction in a program. For example if the types **AREA** and **LENGTH** (page 26) have been defined with real underlying types then it may be necessary to convert expressions of this type if they are to be used as actual parameters of calls to mathematical functions:

```
AREA a:
LENGTH b:
b := LENGTH ROUND (DSQRT (REAL64 a))
```

This example shows a named type being used as a conversion operator. As the conversion from **REAL64** to **LENGTH** may not be exact then a rounding operator must also be used.

Similarly if a named integer type has been defined then conversions will be needed in contexts which explicitly require a value of type **INT**. Consider:

```
DATA TYPE INDEX IS INT :
INDEX i :
VAL INDEX max IS 8192 :
[INT max]INT heap :   -- explicit conversion for array size
SEQ
  i := max - 3        -- 3 is an implicitly typed INDEX
  heap [INT i] := 4   -- explicitly converted INDEX
  ...
```

7 Procedures

This chapter describes *procedures* in occam. A procedure definition in occam defines a name for a process. Consider the following example:

```
PROC increment (INT x)
  x := x + 1
:
```

This example defines **increment** as the name for the process, **x := x + 1**. *Formal parameters* of a procedure are specified in parentheses after the procedure name. In this example, **x** is a formal parameter, and its specifier is the type **INT**. The procedure **increment** may be used as shown in the following example:

```
INT y :
SEQ
  ...
  increment (y)
  ...
```

A formal parameter is an *abbreviation* of the *actual parameter* used in an *instance* of a procedure. The type of the actual parameter matches the type of the corresponding formal parameter.

An *instance* of a procedure has the same effect as the substitution of the process named in the procedure's definition. This instance of **increment** can be expanded to show its effect:

```
INT y :
SEQ
  ...
  x IS y :                -- variable abbreviation
  x := x + 1
  ...
```

which is equivalent to

```
INT y :
SEQ
  ...
  y := y + 1
  ...
```

Here is a further example:

```
PROC writestr (CHAN OF BYTE stream, VAL []BYTE string)
  SEQ i = 0 FOR SIZE string
    stream ! string[i]
:
```

This procedure takes a channel (**stream**) and an array value (**string**) as parameters, and outputs the components of the array to the channel. Note the use of the operator **SIZE** (page 64) on the formal to obtain the number of components in the actual parameter when the specifier of a formal has an empty dimension.

An instance of the procedure **writestr** looks like this:

```
SEQ
  ...
  writestr (screen, "Hello world!")
  ...
```

The type of the second actual parameter is compatible (page 42) with the specifier of the formal parameter.

Again, this instance can be expanded to show the effect:

```
SEQ
  ...
  CHAN OF BYTE stream IS screen :         -- channel abbreviation
  VAL []BYTE string IS "Hello World!" :   -- value abbreviation
  SEQ i = 0 FOR SIZE string
    stream ! string[i]
  ...
```

To show the use of a user defined named type this example may be rewritten to use the type **DOSFNAME** defined on page 31, together with the use of the operator **SIZE** on such a type (page 65):

```
PROC writefname (CHAN OF BYTE stream, VAL DOSFNAME string)
  SEQ i = 0 FOR SIZE DOSFNAME
    stream ! string[i]
:
```

This procedure has the same effect as **writestr** but is restricted in its application to strings of type **DOSFNAME**.

An instance of the procedure **writefname** looks like this:

```
SEQ
  ...
  writefname (screen, "autoexec.bat"(DOSFNAME))
  -- the decoration (DOSFNAME) could have been omitted here!
  ...
```

Again, this instance can be expanded to show the effect:

```
SEQ
  ...
  CHAN OF BYTE stream IS screen :          -- channel abbreviation
  VAL DOSFNAME string IS "autoexec.bat" : -- value abbreviation
  SEQ i = 0 FOR SIZE DOSFNAME
    stream ! string[i]
  ...
```

The correspondence between formal parameters and actual parameters is defined in terms of abbreviations. There are three kinds of abbreviation: value abbreviations (page 43), variable abbreviations (page 42) and channel abbreviations (page 54) and procedure parameters may also be of these three kinds (they may also be timers, page 83).

If the formal is qualified by **VAL**, then it is a value parameter and the corresponding actual must be an expression as the abbreviation is a value abbreviation (page 43). If the formal is a data type not qualified by **VAL**, then the actual must be a variable which can receive a value by input or assignment as the abbreviation is a variable abbreviation (page 42).

A name which is *free* in the body of the procedure (page 40) is statically bound to the specification of the name in scope at the position of the procedure definition, for example:

```
INT step :
SEQ
  step := 39
  PROC next.item (INT next, VAL INT this)
    next := this + step
  :
  INT g, step :
  SEQ
    step := 7
    next.item (g, 3)
    ...        -- at this point the value of g is 42
```

The free variable **step**, in scope when the procedure **next.item** was defined, is *bound* to the occurrence of the name in the procedure **next.item**. The rules of occam ensure that distinct names identify distinct objects. The second declaration of a variable with the name **step** introduces a distinct new name. This means that in the example, the scope and binding of the variables can be seen more clearly by making systematic changes of name. Once this is done, the example is equivalent to:

```
INT step :
SEQ
  step := 39
  INT g, curb :  -- name step changed to curb
  SEQ
    curb := 7
    INT next IS g :  -- expand instance of next.item
    VAL INT this IS 3 :
    next := this + step
    ...          -- at this point the value of g is 42
```

In this transformation of the earlier example, it can be seen that the variable used in the instance of **`next.item`** is the variable named **`step`** declared before the procedure definition of **`next.item`**, and not the second variable declared with the same name.

Further rules for procedure parameters follow from those for abbreviations (see appendix E). The principal purpose of these rules is to ensure that at any one point in a program there is only one name that refers to any one variable. *Alias checking* is the name given to the checks performed by a compiler to ensure the absence of illegal *aliases*.

The rules for abbreviations (page 99) lead to restrictions on the actual parameters which may be used in procedure instances. Consider the procedure:

```
INT x, y, step :
PROC next.item (INT next, VAL INT this)
  next := this + step
:
```

And now consider the following equivalences of instances that may appear in the scope of the procedure:

`next.item (x, y)`	is equivalent to:	**`INT next IS x :`** **`VAL INT this IS y :`** **`next := this + step`**	
`next.item (x, step)`	is equivalent to:	**`INT next IS x :`** **`VAL INT this IS step :`** **`next := this + step`**	
`next.item (step, x)`	is equivalent to:	**`INT next IS step :`** **`VAL INT this IS x :`** **`next := this + step`**	which is **ILLEGAL!**
`next.item (x, x)`	is equivalent to:	**`INT next IS x :`** **`VAL INT this IS x :`** **`next := this + step`**	which is **ILLEGAL!**

Here it can be seen how the meaning of each procedure parameter is defined in terms of an abbreviation, the ordering of parameters corresponds to a sequence of abbreviations. **`next.item (step, x)`** is illegal because the variable **`step`** is used in the expression **`next := this + step`**, after it has been abbreviated, and the example **`next.item (x, x)`** is illegal as **`x`** has already been used in the previous abbreviation of the variable **`x`** (and the rules state that a variable used in such an abbreviation may not be used within the associated scope). Notice also the effect with the order of parameters used in **`next.item`** changed:

```
INT x, y, step :
PROC next.item (VAL INT this, INT next)
  next := this + step
:
```

With this re-ordering, **`next.item (x, x)`** is still illegal, although now for a different reason, as follows:

`next.item (x, x)` is equivalent to:

```
VAL INT this IS x :
INT next IS x :
next := this + step
```

which is **ILLEGAL!**

`next.item (x, x)` is illegal here as there is an assignment to **`x`** (via **`next`**) within the scope of the first abbreviation. Now consider the following example:

```
PROC nonsense (INT x, VAL INT y)
  SEQ
    x := x + y
    x := x - y
:
```

This procedure should leave the value of the variable used as the actual parameter for **`x`**, unchanged, as the following expansion shows:

`nonsense (n, 3)` is equivalent to:

```
INT x IS n :
VAL INT y IS 3 :
SEQ
  x := x + y
  x := x - y
```

and by substitution:

```
SEQ
  n := n + 3
  n := n - 3
```

The value of **`n`** after this instance is **`n`**, as might be expected. However, the following instance is illegal, which is just as well, as the effect is non-intuitive:

`nonsense (n, n)` is equivalent to:

```
INT x IS n :
VAL INT y IS n :
SEQ
  x := x + y
  x := x - y
```

which is **ILLEGAL!**

and by substitution:

```
SEQ
  n := n + n
  n := n - n
```

a non-intuitive effect!

The value of **`n`** after this instance, if it were legal, would be 0, which is counterintuitive. The following example highlights the problem further.

`nonsense (i, v[i])` is equivalent to:

```
INT x IS i :
VAL INT y IS v[i] :
SEQ
  x := x + y
  x := x - y
```

which is **ILLEGAL!**

and by substitution:

```
SEQ
  i := i + v[i]
  i := i - v[i]
```

a non-intuitive effect!

If this instance were legal, the value of **`i`** after the instance of **`nonsense`** would be difficult to predict, as in each of the assignments **`v[i]`** will probably reference a different component of **`v`**, as the value of the subscript **`i`** may be changed by the first assignment.

The syntax for a procedure definition is:

definition	=	**PROC** *name* **(** $\{_0$ **,** *formal* $\}$ **)**
		process
		:
formal	=	*specifier* $\{_1$ **,** *name* $\}$
	|	**VAL** *specifier* $\{_1$ **,** *name* $\}$

The keyword **PROC**, the name of the procedure, and a formal parameter list enclosed in parentheses is followed by a process, indented two spaces, which is the body of the procedure. Any data type may be used as a specifier following **VAL**, any type as a specifier otherwise. The procedure definition is terminated by a colon which appears on a new line at the same indentation level as the start of the definition.

The syntax for a procedure instance is:

proc.instance	=	*name* **(** $\{_0$ **,** *actual* $\}$ **)**
actual	=	*variable*
	|	*channel*
	|	*timer*
	|	*expression*
process	=	*proc.instance*

An instance of a procedure is the procedure name followed by a list of zero or more actual parameters in parentheses. An actual parameter is a variable, channel, timer or expression. The list of actual parameters must correspond directly to the list of formal parameters used in the definition of the procedure. The actual parameter list must have the same number of entries, each of which must be compatible with the kind (**VAL** or non-**VAL**) and type of the corresponding formal parameter. In a program in which all names are distinct, an instance of a procedure behaves like the substitution of the procedure body. Notice that all programs can be expressed in a form in which all names are made distinct by systematic changes of name. A channel parameter or free channel may only be used for input or output (not both) in the procedure.

Procedures in occam cannot be recursive as the procedure name is not defined until the end of its body and so cannot itself be used inside it. It is, however, possible to produce the effect of recursion, to a fixed maximum depth, by means of multiple definitions of the same procedure, in each of which a reference to the procedure name identifies the immediately preceding definition.

An instance of a procedure defined with zero parameters must be followed by empty parentheses. Where a number of parameters of the same type appear in the formal parameter list, a single specifier may specify several names. For example:

```
PROC snark (VAL INT butcher, beaver, LENGTH boojum, jubjub)
  ...
:
```

This example, is equivalent to:

```
PROC snark (VAL INT butcher, VAL INT beaver,
            LENGTH boojum,   LENGTH jubjub)
  ...
:
```

The optimisation of procedure instances by generation of inline code is discussed in appendix A.

8 Functions

The previous chapter discusses named processes (called *procedures*). This chapter describes *functions* in occam. A function defines a name for a special kind of process, called a *value process*. A value process produces a result of any data type, and may appear in expressions. Value processes may also produce more than one result, which may be assigned in a multiple assignment. occam functions are free from all side effects, as they are forbidden to communicate or to assign to free variables. This helps to ensure that programs are clear and easy to maintain.

8.1 Value processes

A value process performs an enclosed process and produces a result. Consider the following example:

```
total := subtotal + (INT sum :
                      VALOF
                        SEQ
                          sum := 0
                          SEQ i = 0 FOR SIZE v
                            sum := sum + v[i]
                        RESULT sum
                     )
```

In the example shown here, the value process produces the sum of the array **v**, and is equivalent to

$$\sum_{i=0}^{(SIZEv)-1} v[i]$$

The syntax of value processes is:

value.process	=	**VALOF**
		  process
		  RESULT *expression.list*

	|	*specification* *value.process*
operand	=	**(** *value.process* **)**
expression.list	=	**(** *value.process* **)**

A value process consists of zero or more specifications which precede the keyword **VALOF**, followed by a process at an indentation of two spaces, and the keyword **RESULT** at the same indentation. The keyword **RESULT** is followed by an expression list on the same line. The expressions in this list may be of any fixed size data type. The line may be broken after a comma, or at a legal point in an expression. An operand of an expression may consist of a left parenthesis, a value process, and a right parenthesis. The structured parentheses appear at the same indentation as each other, and are equivalent to the left hand and right hand parentheses of a bracketed expression respectively. So, where the value process produces a single result, the upper bracket may be preceded by an operator, or the lower bracket may be followed by an operator.

8.2 Functions

More commonly the value process is the body of a function definition, as illustrated in the following example:

```
INT FUNCTION sum (VAL []INT values)
  INT accumulator :
  VALOF
    SEQ
      accumulator := 0
      SEQ i = 0 FOR SIZE values
        accumulator := accumulator + values[i]
    RESULT accumulator
:
```

This function definition defines the name **sum** for the associated value process. The type of the result of this function is **INT**. The result type or types, which may be any explicit or named fixed size data types appear in a comma separated list before the keyword **FUNCTION**.

Just as the behaviour of procedure instances is defined by the substitution of the procedure body, instances of functions behave like a substitution of the function body. In fact the value process is introduced into the language for the purpose of providing a clean definition of function instances in terms of substitution, rather than as a construct that will normally be used in its own right. It follows that the example which starts this chapter is an expansion of the following:

```
total := subtotal + sum (v)
```

A function definition may also define a name for an expression list which does not need a value process for its evaluation, so that simple, single line functions can be defined in the following fashion:

```
BYTE FUNCTION noparity (VAL BYTE ch) IS (ch /\ #7F) :
BOOL FUNCTION lowercase (VAL BYTE ch) IS (ch >= 'a') AND (ch <= 'z') :
BOOL FUNCTION uppercase (VAL BYTE ch) IS (ch >= 'A') AND (ch <= 'Z') :
BOOL FUNCTION isletter (VAL BYTE ch) IS uppercase (ch) OR lowercase (ch) :
```

Each of these functions returns a single byte or boolean result. The definition of the function **isletter** is equivalent to the following:

```
BOOL FUNCTION isletter (VAL BYTE ch)
  VALOF
    SKIP
    RESULT uppercase (ch) OR lowercase (ch)
:
```

A number of rules apply to functions to ensure they are free from side effects. As for procedures, the correspondence between the formal and actual parameters of a function is defined in terms of *abbreviations*, and follows the associated scope rules. However, an argument of a function may only be a value parameter. Only variables declared within the body of a value process or function may be assigned to. There may be no parallels within a value process. There can be no inputs, outputs or alternations within a value process. The evaluation of a function can therefore never be explicitly delayed by the action of another process running concurrently.

Any procedure used within a function must also be free from side effects on variables outside the function. A variable which is free within the result list or value process (Scope, page 39) can be used only in expressions within the value process or function body; it may not receive new values by assignment. Consider the following:

```
[]THING stack:
INT stack.pointer:
SEQ
  ...
  THING FUNCTION read.top.of.stack () IS stack[stack.pointer] :
  BOOL FUNCTION empty () IS (stack.pointer = 0) :
```

stack and **stack.pointer** are free in the definition of **read.top.of.stack** and **stack.pointer** in the definition of **empty**, but can be used in expressions in those definitions.

A value process may produce more than one result, which may then be assigned using a multiple assignment. Consider the following example:

```
point, found := (VAL BYTE char IS 'g' :
                 VAL []BYTE string IS message :
                 BOOL ok :
                 INT ptr :
                 VALOF
                   IF
                     IF i = 0 FOR SIZE string
                       string[i] = char
                         SEQ
                           ok := TRUE
                           ptr := i
                     TRUE
                       SEQ
                         ok := FALSE
                         ptr := -1
                   RESULT ptr, ok
                )
```

This value process searches the byte array **string** for the character **'g'**. The result is produced from the expression list which follows **RESULT**, and is then assigned to **point**, and **found**. This value process can be given a name in a function definition, as follows:

```
INT, BOOL FUNCTION instr (VAL BYTE char, VAL []BYTE string)
  BOOL ok :
  INT ptr :
  VALOF
    IF
      IF i = 0 FOR SIZE string
        string[i] = char
          SEQ
            ok := TRUE
            ptr := i
      TRUE
        SEQ
          ok := FALSE
          ptr := -1
    RESULT ptr, ok
:
VAL message IS "Twas brillig and the slithy toves" :
INT point :
BOOL found :
SEQ
  point, found := instr ('g', message)
  ...
```

This example finds the position of **'g'** in the string **message**. After the multiple assignment in this example, the value of **point** will be **11**, and the value of **found** will be **TRUE**.

Single line functions with multiple results may also be defined:

```
INT, INT FUNCTION div.rem (VAL INT x, y) IS x / y, x REM y :
```

This function produces the quotient and remainder when **x** is divided by **y**. If an error occurs within a function or value process, it will behave like the primitive process **STOP**. This behaviour is equivalent to the behaviour of a numerical overflow in an arithmetic expression (see page 101 for details of the behaviour of invalid processes). Consider the behaviour of an instance of the following function:

```
INT FUNCTION factorial (VAL INT n)
  INT product :
  VALOF
    SEQ
      product := 1
      SEQ i = 1 FOR n
        product := product * i
    RESULT product
:
```

This function will behave like the primitive process **STOP** if **n** is less than zero, or if an overflow occurs in the evaluation of the factorial. In either case the behaviour is equivalent to the behaviour of any other in valid expression (page 101).

The syntax for functions is:

function.header	=	**FUNCTION** *name* **(** $\{_0$ **,** *formal* $\}$**)**
definition	=	$\{_1$ **,** *data.type* $\}$ *function.header* *value.process* **:**
	|	$\{_1$ **,** *data.type* $\}$ *function.header* **IS** *expression.list* **:**
operand	=	*name* **(** $\{_0$ **,** *expression* $\}$ **)**
	|	*operand* **[** *subscript* **]**
expression.list	=	*name* **(** $\{_0$ **,** *expression* $\}$ **)**

A function header consists of the keyword **FUNCTION**, followed by the name of the function and a formal parameter list enclosed in parentheses. A function definition consists of a comma separated list of the types of the result(s) produced by the function, followed by a function header. This is followed by a value process, indented two spaces, which forms the body of the function. The function definition is terminated by a colon which appears on a new line at the same indentation level as the start of the definition. Alternatively, a function definition may consist of the result type list and function header followed by the keyword **IS**, an expression list, and a colon, on the same line. The line may be broken after the keyword **IS**, a comma, or at a legal point in an expression.

An instance of a function is an operand and consists of a function name followed by a list of actual parameter expressions in parentheses. An operand that is a function instance or value process returning a single result of an array or record type may be subscripted by an expression or a field name respectively. An instance of a function defined to have zero parameters must be followed by empty parentheses. Where a number of parameters of the same type appear in the formal parameter list, a single specifier may specify several names. For example, using the type **COMPLEX32** declared on page 31:

```
INT FUNCTION alice (VAL COMPLEX32 tweedle.dum, tweedle.dee,
                    VAL INT cheshire.cat)
  ...
:
```

This example is equivalent to:

```
INT FUNCTION alice (VAL COMPLEX32 tweedle.dum,
                    VAL COMPLEX32 tweedle.dee,
                    VAL INT cheshire.cat)
  ...
:
```

A function may return results of a record type such as **COMPLEX32**. For example the multiplication of complex numbers may be expressed by this function:

```
COMPLEX32 FUNCTION cmul (VAL COMPLEX32 z1, z2)
  COMPLEX32 z :
  VALOF
    SEQ
      z[real] := (z1[real] * z2[real]) - (z1[imag] * z2[imag])
      z[imag] := (z1[real] * z2[imag]) + (z1[imag] * z2[real])
    RESULT z
:
COMPLEX32 w1, w2, w3 :
SEQ
  ...
  w1 := cmul (w2, w3)
  ...
```

The final example of a function shows both the use of a named array type as a result type and the use of two results to return two values. An explicit array type could have been used here instead of a named one. Given an array of program names separated by commas this function will select one of these names, append **".exe"** to it and return it with its length:

```
INT, DOSFNAME FUNCTION exename (VAL[]BYTE names, VAL INT j)
  -- extract (j-1)'th name from comma separated names and append ".exe"
  DOSFNAME fname :
  INT n :
  VALOF
    INT m, s :
    SEQ
      m, s := 0, 0
      SEQ i = 0 FOR j
        SEQ
          WHILE names [m] <> ','
            m := m + 1
          s := m + 1                -- names [s] starts next name
          m := s
      WHILE isletter (names [m])    -- instance of FUNCTION isletter
        m := m + 1
      n := m - s                    -- n is length of j'th name
      [fname FOR n] := [names FROM s FOR n]
      [fname FROM n FOR 4] := ".exe"
    RESULT n + 4, fname
:
```

Use of this function to select a name from a list and then use of that name as a parameter to a procedure is shown in this example:

```
VAL prognames IS "oc,occonf,ilink,icollect,imakef,":
DOSFNAME tool :
INT len :
SEQ
  len, tool := exename (prognames, 2)
  callprog ([tool FOR len])
```

Note that the expression list returned as a function result cannot be used directly as an actual parameter list, but must be assigned to a variable list (or used after **IS** or **RESULT** in another function definition).

The optimisation of function instances by generation of inline code is discussed in appendix A.

9 Timers

Timers produce a value which represents the time, and allow processes to be delayed until the time has reached or passed a particular value. The use of timers is essential in most real time control systems.

This chapter describes timers, the declaration of timers, and access to them.

Syntactically timers have many similarities to channels (discussed on page 45).

A timer provides a clock which can be accessed by any number of concurrent processes. The relationship between the time returned by an occam timer and real time is not defined by the language, i.e. implementations may differ in the granularity of timers and consequently in their cycle period.

9.1 Timer type

The type of a timer is:

timer.type	=	**`TIMER`**

Timer arrays have type similar to channel arrays, for example:

```
[10]TIMER
```

The syntax of timer array types is:

timer.type	=	`[`*expression*`]` *timer.type*

9.2 Declaring a timer

A timer is declared in a manner similar to channels and variables. Consider the following example:

```
TIMER clock :
```

This declaration introduces a timer which is identified by the name **`clock`**. Several timers may be declared together, for example:

```
TIMER clockA, clockB :
```

The type of the declarations is determined, and then the declarations are made. Timer arrays are declared in the same way as other arrays, for example:

```
[10]TIMER clocks :
```

Components and segments of timer arrays are denoted in the same way as components and segments of variable arrays (page 36) and channel arrays (page 46).

The syntax of timer declarations is:

declaration	=	*timer.type* $\{_1$ `,` *name*$\}$ `:`
timer	=	*name*
	|	*timer*`[`*expression*`]`
	|	`[` *timer* **`FROM`** *base* **`FOR`** *count* `]`
	|	`[` *timer* **`FROM`** *base* `]`
	|	`[` *timer* **`FOR`** *count* `]`

A value input from a timer provides an integer value of type **`INT`** representing the time. The value is derived from a clock, which changes by an increment at regular intervals. The value of the clock is cyclic (*ie* when the value reaches the most positive integer value, an increment results in the most negative integer value). The special operator **AFTER** can be used to compare times even though the value may have crossed from most positive to most negative, just as one o'clock *pm* may be considered later than eleven o'clock *am*. If **t1** and **t2** are successive inputs from the same timer, then the expression **t1 AFTER t2** is true if **t1** is later than **t2**. This behaviour is only sensible if the second value (**t2**) is input within half a cycle of the timer. **AFTER** is also explained in the chapter on expressions (page 57).

The rate at which a timer is incremented is implementation dependent, and may depend on the priority at which a process is run (see page 94).

9.3 Timer input

Timers are accessed by special forms of *input* called *timer inputs*, which are syntactically similar to channel inputs, for example:

```
clock ? t
```

This example inputs a value from the timer **clock** and assigns the value to the variable **t**. Unlike channels, inputs from the same timer may appear in any number of components of a parallel.

Another special timer input (called a *delayed input*) specifies a time, after which the input terminates, for example:

```
clock ? AFTER t
```

This input waits until the value of the timer **clock** is later than the value of **t**. In other words, if **c** is the value of the timer **clock**, then the input will wait until **(c AFTER t)** is true. The value of **t** is unchanged.

A delay can be caused by this sequence:

```
SEQ
  clock ? now
  clock ? AFTER now PLUS delay
```

The first input inputs a value representing the current time and assigns it to the variable **now**. The second (delayed) input waits until the value input from **clock** is later than the value of **now PLUS delay**. **PLUS** (page 61) is a *modulo operator*. Note that because of the cyclic nature of timers it is important to use **PLUS** and **MINUS** when adding and subtracting integers derived from a timer.

The syntax for timer inputs is:

input	=	*timer.input*
	|	*delayed.input*
timer.input	=	*timer* **?** *variable*
delayed.input	=	*timer* **? AFTER** *expression*

A timer input receives a value from the timer named on the left of the input symbol (**?**), and assigns that value to the variable named on the right of the symbol. A delayed input waits until the value of the timer named on the left of the input symbol (**?**) is later than the value of the expression on the right of the keyword **AFTER**.

9.4 Timers in alternations

Timer inputs and delayed inputs may be used as guards in alternations. This gives a simple way in which to program timeouts wherein a process waits for one or more inputs on communication channels for up to some maximum time and performs some other action if no input was received. Consider the process:

```
SEQ
  to.server ! request
  time ? request.time
  ...
  ALT
    from.server ? reply
      ...  the server has replied in time
    time ? AFTER request.time PLUS time.out
      ...  the server has missed the deadline
```

In this example, the process sends a request to a server and notes the time at which the request was sent. When the process is ready to receive the reply, it waits alternatively for the server to become ready with the reply or for the timeout period to pass. If the server has not become ready to reply before the end of the timeout period, then the process will execute the branch of the alternation associated with the delayed input. Notice that the timeout period starts from the time of the timer input following the request, not from the beginning of the alternation.

9.5 Timer abbreviation

Timers may be abbreviated in the same way as variables (page 42) and channels (page 54). The same rules, summarised in appendix E, apply to abbreviated timer names as apply to abbreviated variable or channel names. Timers may be used as actual parameters in procedure instances.

The syntax of timer abbreviation is

abbreviation	=	*specifier name* **IS** *timer* **:**
	|	*name* **IS** *timer* **:**
specifier	=	*timer.type*
	|	**[]** *specifier*
	|	**[** *expression* **]** *specifier*
actual	=	*timer*

This syntax is closely related to the corresponding syntax for channels.

10 Retyping and reshaping

This chapter describes retyping conversions. A retyping conversion is syntactically a specification (a definition) in whose scope the data type of a bit pattern is changed from one data type to another. There are three kinds of retyping conversions: conversions which convert a variable, conversions which convert the value of an expression, and conversions which change the protocol of a channel. There is also the related reshaping conversion.

The length (i.e. the number of bits) of the new type specified must be the same as the length of the bit pattern being retyped. A retyping conversion has no effect upon the bit pattern, and differs from *type conversion* (page 66) where the value of one type is converted into an equivalent value of another type.

10.1 Retyping variables and values

As retyping conversions are dependent on the exact mapping of types in the computer, the word size, the significance of the order of bytes in a word, etc., their use can lead to programs which are not portable between different target computer types. See also the discussion of word size on page 92 in appendix A.

The retyping conversion of a value may be used to specify a name for a particular bit pattern described by a hexadecimal constant. Consider the following example:

```
VAL REAL32 root.NaN RETYPES #7F840000(INT32) :
```

The advantage of the above conversion is that it has been possible to specify the exact representation of a value otherwise difficult to represent. Use of this conversion may be dependent on the target machine's use of IEEE floating point arithmetic. Consider also the following example:

```
VAL INT64 pattern RETYPES 42.0(REAL64) :
```

The bit pattern for the real representation of the value 42.0 is mapped to a name **pattern** of type **INT64**. As for the *abbreviation* (page 43) of expressions, no variable used in the expression may receive a new value by input or assignment within the scope of the conversion.

The retyping conversion may also specify a name of a new type for an existing variable of the same length. For example:

```
INT64 condition :
...
SEQ
  [8]BYTE state RETYPES condition :
  ...
```

In this example, **condition**, a variable of type **INT64**, is made accessible as an array of 8 bytes. Each byte is accessible via subscript, any change to the bit pattern as a result of an assignment or input will directly affect the value of the original variable.

The same rules apply to names specified by retyping conversions as apply to abbreviations (page 99). That is, no variable used in a subscript, base or count expression which selects a component or segment of an array may receive a new value by input or assignment within the *scope* (page 39, the region of a program where use of the name is legal) of the conversion. The variable converted may not be used within the scope of the conversion.

The syntax for retyping conversion is:

definition	=	*specifier name* **RETYPES** *variable* **:**
	|	**VAL** *specifier name* **RETYPES** *expression* **:**

A retyping conversion is syntactically a definition. The retyping conversion of a variable begins with a specifier, followed by the name specified, and the keyword **RETYPES**, the variable appears to the right of the keyword **RETYPES**. The retyping conversion of a value begins with the keyword **VAL**, a data type specifier appears to the right of **VAL**, followed by the name specified, and the keyword **RETYPES**; the expression appears to the right of the keyword **RETYPES**. The number of bytes in the representation of values of the type of the specifier must be the same as in the variable or value retyped. The line on which a retyping conversion occurs may be broken after the keyword **RETYPES**, or at any legal point in the expression.

A retyping conversion where the destination type is an array type may use a specifier with one empty dimension. The size of this dimension is determined from the size of the variable or value being retyped. For example:

```
[10]REAL64 w :
SEQ
  []BYTE bw RETYPES w :
  -- here we can send the binary array to a file
  ...
  [10][]INT iw RETYPES w :
  -- here we can look at each REAL as a bit pattern
  ...
```

If a variable being retyped is implemented with different alignment requirements to the destination type then the retyping conversion is invalid unless it is determined that alignment mismatches do not occur. This can be important when using retyping to pack binary values into a buffer for sending to a file. Consider the following example:

```
[4096]BYTE diskbuff :
INT di :
DATA TYPE MYINT IS INT :
MYINT x :
SEQ
  ...
  MYINT db0 RETYPES [diskbuff FOR BYTESIN (MYINT)] :
  --  ok because start of array always aligned
  di := x
  []BYTE bx RETYPES x :
  [diskbuff FROM di FOR SIZE bx] := bx
  ...
  MYINT dbi RETYPES [diskbuff FROM di FOR BYTESIN (MYINT)] : -- INVALID!
  -- fails because value of di might cause misaligment
  ...
```

An implementation may reject an invalid definition as above at compile time or may cause the generation of code to reject it at run time.

10.2 Retyping channels

The protocol of a channel is used by a compiler to check the format and content of messages that are communicated by inputs and outputs. Sometimes it is necessary to allow different protocols to be used for communications using the same channel at different places within a program.

This may be achieved by using a retyping conversion applied to channels. Consider this example:

```
PROC real.buffer (CHAN OF REAL32 rin, rout)
  REAL32 r :
  WHILE TRUE
    SEQ
      rin ? r
      rout ! r
:
PROTOCOL COMPLEX IS REAL32; REAL32 :
PROC generate.complex (CHAN OF COMPLEX c)
  ...  process outputting complex values on c
:
PROC consume.complex (CHAN OF COMPLEX c)
  ...  process inputting complex values on c
:
CHAN OF COMPLEX cin, cout :
PAR
  generate.complex (cin)
  CHAN OF REAL32 crin RETYPES cin :
  CHAN OF REAL32 crout RETYPES cout :
  real.buffer (crin, crout)
  consume.complex (cout)
```

This is a trivial example of a situation that sometimes arises in occam programming. The procedure **real.buffer** represents an existing general purpose procedure that it is not convenient to rewrite. This happens to buffer single values. Similarly the procedure **generate.complex** has been written to output complex numbers as pairs of values using the protocol **COMPLEX**. As this is a sequential protocol each value will be sent as a single communication and so will match one input in the buffer procedure. The use of channel retyping allows these separately written procedures to be called without changing either.

If channel retyping is used in only one of the processes that use a channel as here, then it is important to ensure that individual inputs and outputs remain compatible in terms of the numbers of bytes in each communication. Channel retyping may also be used to change the protocol both in the process that outputs on a channel and the process which inputs on the channel. In this case there is no need to take special care about the details of implementation of communication.

A common use of channel retyping is to apply protocol checking to the use of one or both ends of a channel that was declared with an anarchic protocol (see page 53).

The syntax of channel retyping is similar to variable retyping:

definition = *specifier name* **RETYPES** *channel* **:**

The specifier must be of the form **CHAN OF** protocol. Channel arrays may be retyped or reshaped (see below).

10.3 Reshaping

A special kind of retyping conversion may be used to change the interpretation of an array in terms of its subdivision into arrays of fewer dimensions. A reshaping conversion may increase or decrease the number of dimensions or may merely change the sizes of the dimensions. A reshaping conversion is independent of details of representation and therefore programs using them are portable to different target computer architectures. Consider this example:

```
VAL [2][6]REAL32 nums IS [[0.0, 1.0, 2.1, 3.1, 4.2, 5.2]
                          [6.3, 7.3, 8.4, 9.4, 10.5, 11.5]] :
SEQ
  --  here we want to use two arrays of six reals each
  ...
  VAL [6][2]REAL32 nums.6.2 RESHAPES nums :
  SEQ
    --  here we want to use six arrays of two reals each
    ...
  VAL [12]REAL32 nums.12 RESHAPES nums :
  SEQ
    --  here we want to use one array of twelve reals
    ...
  VAL [2][2][]REAL32 nums.2.2. RESHAPES nums :
  SEQ
    --  here we want a 3-dimensional subdivision
    ...
```

In all the reshapings shown the number of components of the array base type on the two sides of the keyword **RESHAPES** must be the same. One pair of brackets in the specifier on the left may be empty, and such a reshaping is valid if and only if there is an integer such that the product of the sizes of all dimensions on the left is equal to the number of base type components in the array on the right. The empty brackets may be in any position. In the example shown the type of **nums.2.2.** will be found by the compiler to be **[2][2][3]REAL32**.

The syntax for reshaping conversion is:

definition	=	*specifier name* **RESHAPES** *variable* **:**
	|	**VAL** *specifier name* **RESHAPES** *expression* **:**
	|	*specifier name* **RESHAPES** *channel* **:**

A reshaping conversion of a variable begins with a specifier for an array type, followed by the name specified, and the keyword **RESHAPES**. The variable which must be an array built from components of the same base type as the specifier appears to the right of the keyword **RESHAPES**.

A reshaping conversion of a value begins with the keyword **VAL**, a data type specifier appears to the right of **VAL**, followed by the name specified, and the keyword **RESHAPES**. The expression which must be of an array type built from components of the same base type as the specifier appears to the right of the keyword **RESHAPES**.

A reshaping conversion of a channel begins with a specifier for a channel array type, followed by the name specified, and the keyword **RESHAPES**. The channel which must be an array built from components of the same base type as the specifier appears to the right of the keyword **RESHAPES**.

The line on which a conversion occurs may be broken after the keyword **RESHAPES**, or at any valid point in the variable or expression. Specifiers appearing in reshaping conversions may contain one empty pair of brackets (open dimension).

Appendices

A Implementation dependent features

As far as possible an implementation of occam should place no unreasonable quantitative restrictions on the components of an occam program. If any restrictions that impact the class of acceptable occam programs are imposed by a compiler then they must be described in accompanying documentation. Examples of such limits and restrictions would be those concerning the length of names, length of formal parameter lists, number of tags in a protocol or requirements that certain constants be known at compile time.

An implementation of occam may need to extend the language in various ways to facilitate the solution of common software engineering problems such as source code management, optimisations and interfacing to other software and special purpose hardware. Such extensions should always obey the spirit of occam in keeping things as simple as possible and should as far as possible be limited to structures introduced by explicit keywords easily recognised by a compiler or human reader.

Directives to a compiler which can occupy a line on their own should always do so and be introduced by a keyword whose first character is the special character **#**. This should be the first lexical item on a line and should be allowed at any indentation consistent with the surrounding occam source code. Allowable indentations match those for comments (page 4). Any such compiler directive line must be allowed to end with an occam comment, and apart from its interpretation at compile time is treated as a comment.

These keywords and others which need to be embedded within occam source should all obey the convention that no keyword contains a lower case letter.

It is not the purpose of this appendix to describe any such language extensions in detail, but to mention some of those that have been found useful in existing implementations, and so reduce the likelihood of new extensions being devised for similar purposes when existing ones could sensibly be reused. Readers should consult detailed documentation accompanying a particular compiler for details of the syntax and semantics of these extensions.

A.1 Compiler directives

From the strict syntactic point of view compiler directives behave as comments and may appear wherever a comment can appear. They are interpreted by the compiler. Particular directives may be allowed by a compiler only in a more restricted set of positions, and their indentation may be significant if they introduce additional specifications, whose scope they will determine.

#INCLUDE introduces (by name in some filing system or operating system name space) a source text file to be inserted at the current source line position and indentation.

#USE introduces an object file or library file from which interface information from a pre-compiled body of occam code is to be inserted at the current source line position and indentation.

#OPTION introduces a list of compilation options.

#PRAGMA introduces further information that may influence what a compiler does to the source text during compilation. Versions of this directive have also been used to define interfacing to code compiled from languages other than occam, selective control of usage checking, etc.

#COMMENT introduces a text string that is to be explicitly copied to an object file. This may be needed to ensure that copyright etc is identified.

A.2 Special keywords introducing language extensions

INLINE

This keyword may be inserted before **PROC** or **FUNCTION** to suggest to the compiler that calls to the routine should be compiled to inline code rather than to a closed subroutine.

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**	(15)
		{ *process* }	
	|	**PAR** *replicator*	(17)
		process	
	|	**PRI PAR**	(94)
		{ *process* }	
	|	**PRI PAR** *replicator*	(94)
		process	
	|	*placedpar*	(93)
placedpar	=	**PLACED PAR**	(93)
		{ *placedpar* }	
	|	**PLACED PAR** *replicator*	(93)
		placedpar	
	|	**PROCESSOR** *expression*	(93)
		process	
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*	(41)
		process	
	|	*allocation*	(95)
		process	

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*	(13)
		{ *option* }	
selector	=	*expression*	(13)
sequence	=	**SEQ**	(9)
		{ *process* }	
	|	**SEQ** *replicator*	(10)
		process	
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**	(31)
		{ *data.type* {$_1$ **,** *field.name* } **:** }	
	|	**PACKED RECORD**	(31)
		{ *data.type* {$_1$ **,** *field.name* } **:** }	

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)
  ...
:
```

SQRT(**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)
  ...
:
```

ISNAN(**X**) = **TRUE** $\Leftrightarrow$ **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)
  ...
:
```

NOTFINITE(**X**) = **TRUE** $\Leftrightarrow$ **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)
  ...
:
```

SCALEB(**X**, **n**) = **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)
  ...
:
```

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

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

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	± **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**.

M.9 Arccosine

```
REAL32 FUNCTION ACOS (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DACOS (VAL REAL64 X)
  ...
:
```

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

Domain : $[-1.0, 1.0]$
Range : [0, Pi]

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

M.10 Arctangent

```
REAL32 FUNCTION ATAN (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DATAN (VAL REAL64 X)
  ...
:
```

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

Domain : [$-$Inf, Inf]
Range : [$-$Pi/2, Pi/2]

M.11 Polar Angle

```
REAL32 FUNCTION ATAN2 (VAL REAL32 X, Y)
  ...
:
REAL64 FUNCTION DATAN2 (VAL REAL64 X, Y)
  ...
:
```

These compute the angular co-ordinate tan$^{-1}(Y/X)$ (in radians) of a point whose X and Y co-ordinates are given.

Domain : [$-$Inf, Inf] x [$-$Inf, Inf]
Range : ($-$Pi, Pi]

(0, 0) and ($\pm$**Inf**,$\pm$**Inf**) give **undefined.NaN**.

M.12 Hyperbolic sine

```
REAL32 FUNCTION SINH (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DSINH (VAL REAL64 X)
  ...
:
```

These compute : result = sinh(X).

Domain :	[−Hmax, Hmax] = $[-89.4, 89.4]S$, = $[-710.5, 710.5]D$
Range :	(−Inf, Inf)

$X < -$Hmax gives −**Inf**, and $X >$ Hmax gives **Inf**.

M.13 Hyperbolic cosine

```
REAL32 FUNCTION COSH (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DCOSH (VAL REAL64 X)
  ...
:
```

These compute: result = cosh(X).

Domain :	[−Hmax, Hmax] = $[-89.4, 89.4]S$, = $[-710.5, 710.5]D$
Range :	[1.0, Inf)

$|X| >$ Hmax gives **Inf**.

M.14 Hyperbolic tangent

```
REAL32 FUNCTION TANH (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DTANH (VAL REAL64 X)
  ...
:
```

These compute : result = tanh(X).

Domain :	[−Inf, Inf]
Range :	[−1.0, 1.0]

M.15 Pseudo-random numbers

```
REAL32, INT32 FUNCTION RAN (VAL INT32 N)
  ...
:
REAL64, INT64 FUNCTION DRAN (VAL INT64 N)
  ...
:
```

This function returns two results, the first is a real between 0.0 and 1.0, and the second is an integer. The integer, which must be used as the parameter in the next call to the function, carries a pseudo-random linear congruential sequence N_k, and must be kept in scope for as long as the function is used. It should be initialised before the first call to the function but not modified thereafter except by the function itself. Consider the following sequence:

```
SEQ
  x, seed := RAN (8)     -- initialise seed
  y, seed := RAN (seed)
  z, seed := RAN (seed)
```

In this example **x**, **y**, and **z** are each assigned a pseudo-random value.

Domain : Integers
Range : [0.0, 1.0) x Integers

N Value, string conversion routines

This appendix describes the standard library of string to value, value to string routines. The library provides primitive procedures to convert a value to and from decimal or hexadecimal representations. High level input/output routines can be easily built using these simple procedures, and a number will typically be provided in an implementation.

N.1 Integer, string conversions

The procedures described here provide conversion between integer values and their decimal or hexadecimal representations held as a string of characters, for example:

```
PROC INTTOSTRING (INT len, []BYTE string, VAL INT n)
  ...
:
```

The procedure **INTTOSTRING** returns the decimal representation of **n** in **string** and the number of characters in the representation in **len**.

```
PROC STRINGTOINT (BOOL error, INT n, VAL []BYTE string)
  ...
:
```

The procedure **STRINGTOINT** returns in **n** the value represented by **string**. **error** is set to **TRUE** if a non numeric character is found in **string**. **+** or a **–** are allowed in the first character position. **n** will be the value of the the portion of **string** up to any illegal character with the convention that the value of an empty string is 0. **error** is also set if the value of **string** overflows the range of **INT**, in this case **n** will contain the low order bits of the binary representation of **string**. **error** is set to **FALSE** in all other cases.

```
PROC HEXTOSTRING (INT len, []BYTE string, VAL INT n)
  ...
:
```

The procedure **HEXTOSTRING** returns the hexadecimal representation of **n** in **string** and the number of characters in the representation in **len**. All the nibbles (a nibble is a word 4 bits wide) of **n** are output so that leading zeros are included. The number of characters will be the number of bits in an **INT** divided by 4.

```
PROC STRINGTOHEX (BOOL error, INT n, VAL []BYTE string)
  ...
:
```

The procedure **STRINGTOHEX** returns in **n** the value represented by the hexadecimal **string**. **error** is set to **TRUE** if a non hexadecimal character is found in **string**. Here **n** will be the value of the the portion of **string** up to the illegal character with the convention that the value of an empty string is 0. **error** is also set to **TRUE** if the value represented by **string** overflows the range of **INT**. In this case **n** will contain the low order bits of the binary representation of **string**. In all other cases **error** is set to **FALSE**.

Similar procedures are provided for the types **INT16**, **INT32** and **INT64**. These procedures use equivalent parameters of the appropriate type. The procedures are:

INTTOSTRING	INT16TOSTRING	INT32TOSTRING	INT64TOSTRING
STRINGTOINT	STRINGTOINT16	STRINGTOINT32	STRINGTOINT64
HEXTOSTRING	HEX16TOSTRING	HEX32TOSTRING	HEX64TOSTRING
STRINGTOHEX	STRINGTOHEX16	STRINGTOHEX32	STRINGTOHEX64

N.2 Boolean, string conversion

The procedures described here provide conversion between boolean values and their textual representation "**TRUE**" and "**FALSE**".

```
PROC BOOLTOSTRING (INT len, []BYTE string, VAL BOOL b)
  ...
:
```

The procedure **BOOLTOSTRING** returns "**TRUE**" in **string** if **b** is **TRUE** and "**FALSE**" otherwise. **len** contains the number of characters in the string returned.

```
PROC STRINGTOBOOL (BOOL error, b, VAL []BYTE string)
  ...
:
```

The procedure **STRINGTOBOOL** returns **TRUE** in b if first 4 characters of **string** are "**TRUE**", **FALSE** if first 5 characters are "**FALSE**" and **b** is undefined in other cases. **TRUE** is returned in **error** if **string** is not exactly "**TRUE**" or **"FALSE"**.

N.3 Real, string conversion

The procedures described here provide conversion between real values and their representation as strings, for example:

```
PROC STRINGTOREAL32 (BOOL error, REAL32 r, VAL []BYTE string)
  ...
:
PROC STRINGTOREAL64 (BOOL error, REAL64 r, VAL []BYTE string)
  ...
:
```

These two procedures each take a string containing a decimal representation of a real number and convert it into the corresponding real value. If the value represented by **string** overflows the range of the type then an appropriately signed infinity is returned. Errors in the syntax of **string** are signalled by a Not-a-Number being returned and **error** being set to **TRUE**. The string is scanned from the left as far as possible while the syntax is still legal. If there any characters after the end of the longest correct string then error is set to **TRUE**, otherwise it is **FALSE**. For example if **string** was **"12.34E+2+1.0"** then the value returned would be 12.34×10^2 with **error** set to **TRUE**. Strings which represent real values are those specified by the syntax for *real* literals, for example:

```
12.34
587.0E-20
+1.0E+123
-3.05
```

Further examples are given in the section on literals on page 27.

```
PROC REAL32TOSTRING (INT len, []BYTE string,
                     VAL REAL32 r, VAL INT m,n)
  ...
:
PROC REAL64TOSTRING (INT len, []BYTE string,
                     VAL REAL64 r, VAL INT m,n)
  ...
:
```

These two procedures return a string representing the value **r** in the first **len BYTE**s of **string**. The format of the representation is determined by **m** and **n**. Free format is selected by passing **0** in **m** and **n** into the procedure. Where possible a fixed point representation is used when this does not indicate more accuracy than is available and does not have more than 3 "**0**"s after the decimal point before significant digits. Otherwise exponential form is used. The number of characters returned in **string** here depends on the input but will be no more than 15 in **REALTOSTRING32** and 24 in **REALTOSTRING64**. **string** is left justified in free format.

If **m** is non-zero then if possible the procedure returns a fixed point representation of **r** with **m** digits before the decimal point and **n** places after with padding spaces being added when needed. If this is not possible then an exponential representation is returned with the same field width as the fixed point representation would have had. If **m** and **n** are both very small then an exponential representation may not fit in the field width so two special values "**Un**" and "**Ov**" with a sign are returned to indicate a value under or over the representable fixed point values. In all these cases **string** is padded with spaces so that it contains (**m** + **n** + 2) characters - **m** before the decimal point, **n** after, as well as the sign and decimal point characters.

If **m** is zero but **n** is not then an exponential representation is returned where the number of digits of fraction returned is **n**. The form of the fraction is *digit.digits* except when **n** is 1. In this case the output is not a proper representation as the fraction will be of the form *' ' digit* where the padding space is added due to the absence of a decimal point. For this reason the case **m** = 0, **n** = 1 should not be used in general. When **m** is 0 **string** will contain (**n** + 6) characters for **REALTOSTRING32** and (**n** + 7) for **REALTOSTRING64**.

Each procedure returns a string "**Inf**" preceded by a sign character for infinities and a string "**NaN**" for Not-a-Numbers. In free format a leading space on either string is dropped. Both these will be padded on the right with spaces to fill the field width when free format output is not being used.

O Programming support routines

This appendix describes some additional routines that have been added to occam implementations to provide support for the programmer. For full specifications of these see documentation provided with the compiler.

O.1 Rescheduling the processor

In real time processing it is sometimes necessary to force the compiler to ensure that a process in a parallel suspends itself to give another process a chance to be executed. A call of **RESCHEDULE()** will have this effect.

```
PROC RESCHEDULE ()
  ...
:
```

O.2 Assertion checking

This procedure is provided to enable the programmer to make assertions involving the variables of the program, to ensure that if such an assertion is found to be false at compile time, then the compiler will report an error, or if it is found to be false at run time then the program will stop at that point.

```
PROC ASSERT (VAL BOOL assertion)
  ...
:
```

An instance of **ASSERT** may usefully be included as a “stub” at all places in a program under development that will eventually be replaced by suitable error checking code. The careful use of assertions is at the heart of defensive programming.

An implementation may provide compiler options to control the action which occurs when an assertion is found to be false.

P Changes from occam 2

This appendix enumerates the changes which have been made to occam 2 to derive the extended language occam 2.1, and the more prominent changes in the style and structure of this manual to facilitate the description of the extended language.

P.1 Language changes

The following groups of changes have been made to occam 2:

1. Addition of named types. (page 26)
2. Addition of record and packed record types. (page 31)
3. New operators **BYTESIN** and **OFFSETOF**. (page 65)
4. Arithmetic and other operations on values of **BYTE** type. (page 57)
5. Implicit typing of literal constants. (page 29)
6. Functions returning results of any fixed length type. (page 78)
7. Segments with optional **FROM** and **FOR**. (page 37)
8. Reshaping of arrays. (page 88)
9. Scope of **PROTOCOL** tags. (page 53)
10. Revised treatment of anarchic protocols. (page 53)
11. Channel retyping. (page 86)
12. Channel array constructors. (page 55)
13. Revised semantics of counted array communication. (page 47)
14. Restrictions on the positioning of allocations. (page 95)

P.2 Manual changes

The opportunity has been taken in this revised manual to make some structural changes, and many detailed changes in style of presentation.

The following structural changes have been made:

- A new chapter on variables and values collects together material that was previously spread over the chapters on arrays and elements, scope and abbreviation.
- The chapter on timers has been delayed until after the more familiar procedures and functions have been introduced.
- Description of priority alternation has been moved from the appendix to the chapter on constructed processes.
- The description of retyping has been moved from an appendix to a new chapter together with the new feature of reshaping.
- The complete syntax in the appendix is fully annotated with cross references to the places in the text where the semantic restrictions applicable to each production are defined.

- A new appendix on implementation dependent features has been added.

In addition to material defining and illustrating the new extensions several revisions have been made to the detailed presentation of language syntax in the manual. In particular:

The syntactic category *element* has been dropped, to simplify the separate handling of *variable*, *channel* and *timer*, which have many syntactic similarities but also some significant differences in how they are used.

The treatment of *string* has been revised to emphasise its semantic closeness to *table*.

The syntactic categories *primitive.type* and *array.type* have been replaced by the categories *data.type*, *channel.type*, *timer.type* and *port.type*, each of which may be used for scalars and arrays.

The syntactic categories *action*, *function.body*, *procedure.body*, and *valof* which served no particularaly useful syntactic purpose and had only descriptive value have been eliminated.

The syntactic categories *monadic.operator* and *dyadic.operator* have been explicitly defined in the syntax, and the lexical level categories *name*, *digits*, *hex.digits*, *character* and *string* in the lexical appendix.

Q Glossary of terms

Abbreviation An abbreviation specifies a *name* as an *alias* for an existing *variable, channel, port* or *timer* or for the value of an *expression*. The meaning of the alias is defined by substitution of the abbreviated expression, variable, etc.

Action An action is an assignment, an input, or an output.

Actual parameter A parameter used in an *instance* of a procedure or function.

Alias check Ensure all variables and channels are identified by a single name within a given *scope*.

Allocation Place a *variable*, *channel*, *timer*, *array* or *port* at an absolute location in memory.

Alternation Combines a number of processes guarded by inputs, and performs the process associated with an input which is ready.

Alternative A component of an *alternation*.

Anarchic protocol A protocol compatible with any communication using a simple protocol.

Argument A parameter used in an instance of a function.

Array A sequence of components of the same type stored in consecutive memory locations.

Assignment Evaluates an expression or list of expressions, and assigns each resulting value to a corresponding variable.

Base An expression defining the starting value of a *replication index* or the subscript of the first component of a *segment*.

Bitwise operation Operation on the individual bits in the representation of a value.

Boolean A truth value used in a guard or loop.

Boolean operation Logical operation on truth values.

Case input Uses tag to select the protocol of an input on a single channel with a case protocol.

Case protocol A protocol comprising a collection of tagged protocols.

Channel Unbuffered, uni-directional point-to-point connection for communication between two processes executing in parallel.

Channel array constructor An array of channels expressed as a sequence of component channels in square brackets.

Character A literal value of byte type expressed alone between single quotes, or as part of a string in double quotes.

Choice A component of a conditional.

Combination The collection of processes in a *construction*.

Communication The communication of values between concurrent processes. Each communication involves a rendezvous between an input and an output.

Component An array is a sequence of components of the same type selected by an integer subscript.

Concurrency Two or more processes are concurrent if they may be executed at the same time.

Conditional A construction (`IF`) which combines a number of processes each of which is guarded by a boolean. Behaves as the first to evaluate to true.

Configuration Configuration associates the components of an occam program with a set of physical resources on one or more processors.

Construction A construction combines processes. occam programs are built from processes, by combining primitive processes and other constructions to form constructions of *sequence* (`SEQ`), *conditional* (`IF`), *selection* (`CASE`), *loop* (`WHILE`), *parallel* (`PAR`) or *alternation* (`ALT`).

Conversion Use of a type name as an operator to convert a value to an equivalent value of another type.

Count An expression defining the number of replications in a *replicator* or number of components in a *segment*.

Counted array An array communicated on a channel, whose size is defined by an explicit count communicated previously.

Data type The data type of a variable defines which values can be stored in that variable. The data type of a value defines which operations can be performed on the value.

Deadlock A state in which two or more concurrent processes can no longer proceed due to a communication interdependency.

Declaration Specifies the name, type and scope of a *variable*, *channel*, *port*, *timer* or an array of one of these types.

Decoration The use of a type name in parentheses to define the type of a literal or table.

Delayed input A special *timer input* which will wait until the timer has incremented beyond a specified time before terminating. Useful for adding a simple delay in a process.

Definition A specification introducing a name for a procedure, function, protocol, data type, retyping or reshaping.

Digits Decimal or hexadecimal digits as a component of a literal of a real, integer or byte type.

Expression The representation of a value within a program. The data type of an expression can always be determined by the compiler.

Expression list A list of expressions separated by commas; used in *multiple assignment* and *functions*.

Field A field is a component of a record type. A field is selected from a record by a *field name* in square brackets following the record variable or value.

Formal parameter Parameter specified in the definition of a procedure or function. A formal parameter acts as an *abbreviation* for the *actual parameter* used in an *instance* of the procedure or function.

Free channel A channel whose name is a free name.

Free name A name which occurs within a process, but is not specified within the process.

Free variable A variable whose name is a free name.

Function definition Specifies a name for a value process or expression list.

Guard Determines the execution of an associated process in a choice (*boolean guard*) or alternative (*input guard*).

Indentation An offset from the left hand edge of the page. In occam indentation is critical, and serves to define the structure of processes. Indentation increases by two spaces for each nested construct in a program, and decreases by two spaces at the end of that construct.

Inline When the compiler implements an instance of a procedure or function by an explicit copy of the body, with appropriate parameter substitutions, rather than by a closed subroutine.

Input Receive a value from a channel into a variable.

Input guard An input which guards an alternative in an alternation.

Instance A process which is a call of a procedure or an operand or expression list which is a call of a function.

Invalid process A process whose behaviour has for some reason become undefined, and as a result may cause a system to stop.

Library function See *library procedure*.

Library procedure A procedure whose definition is compiled separately and whose compiled code may be referenced within a program by an implementation dependent language extension.

Literal A literal is a textual representation of a known value, and has a data type.

Livelock A divergent process, one which may remain internally active but not perform further communication.

Loop A (`WHILE`) loop executes the associated process as long as the specified condition is true; if the condition is initially false the associated process is not executed.

Modulo operator A modulo operator performs its operation (**PLUS**, **MINUS**, **TIMES**) with no check for overflow. The value returned as a result is the cyclic value within the range of the operand type.

Name A sequence of letters (and digits or dots) introduced in a specification as the name of a variable, field, tag, channel, timer, procedure, function, data type, protocol, etc.

Network a network consists of a number of processing devices, microcomputers perhaps, with the facility to communicate with each other.

Offset The position in memory of a field within a record.

Operand Yields a value in an expression.

Operator A monadic operator performs an operation on a single operand and a dyadic operator performs an operation on two operands.

Output Send the value of an expression to a channel.

Packed record A record in which the programmer explicitly defines the order and implicitly the offsets of all the fields.

Placement A configuration statement which places a process on a particular processing device.

Port A memory mapped peripheral control register from which input, or to which output, may be directed.

Predefine A predefined procedure or function is one that may be used in a program without explicit specification. An implementation may define a set of such names and specify their formal parameters.

Primitive data type A primitive data type is a integer, boolean, byte or real type. A named type derived from such a type or an array or structured type is not a primitive type.

Priority Priority can be given to the component processes of a parallel executing on a single processing device. Lower priority processes on such a device may only continue when all higher priority processes are unable to. The inputs which guard alternatives in an alternation may be given a selection priority. If two or more inputs are ready, then the input with the highest priority is selected.

Procedure definition A procedure definition specifies a name for a process.

Procedure instance An instance of a procedure is a use of the procedure, and behaves like a substitution of the process named in the procedure definition. The phrase "procedure call" is used in many other languages, to indicate the use of a procedure, and has a similar meaning. Although the behaviour of an occam procedure is clearly defined as the substitution of the procedure body, a procedure may be implemented as either a substitution or as a call to a closed subroutine.

Process A process starts, performs a number of actions, and then either stops without completing or terminates completely. occam programs are built from the primitive processes *assignment* (**:=**), *input* (**?**), *output* (**!**), **SKIP** and **STOP**. These primitives are combined in **SEQ**, **IF**, **CASE**, **WHILE**, **PAR** and **ALT** constructions.

Protocol The format and *type* of values passed (communicated) on a channel, or a name defined to represent such a format. Communication is valid only if the output and input are compatible; i.e. each communication matches the type(s) specified by the channel protocol.

Real A numerical value represented according to the IEEE standard for floating point arithmetic.

Real time The actual time taken for a physical process to occur. The time returned by a *timer* is directly related to real time.

Record A record consists of a number of named *fields* each of which has a specified data type. A value of record type associates a value of appropriate type with each of the fields.

Record layout The concrete representation of a record data type in store.

Relational operation A relational operation compares its operands and yields a boolean result.

Replication A replicator produces a number of similar components of a construction, distinguished by the value of a *replication index*.

Reshaping conversion A reshaping conversion changes the program's view of the internal structure of a (multi-dimensional) array.

Retyping conversion A retyping conversion changes the data type of a bit pattern, from one data type to another. There are three kinds of retyping conversions: conversions which change the protocol of a channel, conversions which convert a variable, and conversions which convert the value of an expression. Data type retyping has no effect upon the bit pattern, and differs from *type conversion* where a value of one type is converted into an equivalent value of another type.

Scope The region of a program in which a particular specification of a name is valid.

Segment A segment is one or more consecutive components of an array, defined by a base and a count. An abbreviated segment may have a computed number of components which may be zero but not negative.

Selection A selection process (**CASE**) executes a process from a list of associated options. The options are selected by matching a selector with a constant case expression associated with the option.

Sequence A sequential process (**SEQ**) is one where one action follows another.

Sequential protocol A sequential protocol specifies a sequence of simple protocols as the format of communication on a channel.

Shift operation Perform logical shift of the bit pattern of a value.

Skip Start, perform no action and terminate immediately.

Specification A specification is either a declaration, an abbreviation or a definition and specifies a name which may be used within the associated scope.

Specifier Identifies the type of an name specified in an abbreviation, formal parameter, or definition.

Stop Start, perform no further action and do not terminate.

String A sequence of characters in double quotes equivalent to a table of bytes.

Subscript An integer expression in square brackets (**[]**) which selects a component of an array.

Table An array of values of the same type, or a record of values of possibly different types, optionally decorated by a type name in parentheses, used in expressions.

Tag Identifier of a protocol variant specified in a *case protocol* definition. Used in an output or **CASE** input.

Tagged list Component of a variant in a *case input*.

Tagged protocol Specifies a possible case protocol variant, distinguished by its tag, for communication on a single channel.

Timer A timer is a clock which can be accessed by any number of concurrent processes.

Timer input A timer input inputs a value from a timer.

Type conversion A type conversion converts the value of an expression of one data type into an equivalent value of another data type.

Usage check A usage check performed by a compiler ensures that variables and channels are not shared illegally between parallel components.

Value process A value process produces one or more results, each of a data type.

Variable A variable is declared or abbreviated and may receive a new value by input or assignment. A variable normally requires the allocation of an addressable storage location in the computer.

Variable list A list of variables used in a *multiple assignment*.

Variable subscript A variable subscript is a subscript whose value depends on a variable, a procedure parameter, or the index of a replicator with a base or count which is not a constant or constant expression.

Variant A variant is a component of a *case input* consisting of a tagged list of variables and an associated process to be executed when the tag input matches the tag in the tagged list.

R Occam Bibliography

R.1 Books

Bowler, K.C, Kenway, R.D, Pawley, G.S. and Roweth, D *Introduction to OCCAM 2 Programming* Chartwell-Bratt 1989 ISBN 0 86238 227 0

Brookes, Graham R *Introduction to Occam 2 on the Transputer* Macmillan 1989 ISBN 0 333 45340 9

Burns, Alan *Programming in Occam 2* Addison-Wesley 1987 ISBN 0 201 17371 9

Carling, Alison *Parallel Processing, Transputer and Occam* Sigma 1988 ISBN 1 85058 07 4

Cok, Ronald S *Parallel Programs for the Transputer* Prentice-Hall 1991 ISBN 0 13 651480 4

Dowsing, R.L. *Introduction to Concurrency Using Occam* Van Nostrand Reinhold 1988 ISBN 0 278 00059 2

East, I *Parallel Processing: A First Course* Pitman 1989

Ellison, David *Understanding Occam and the Transputer : Through Complete Working Programs* Sigma ISBN 1 85058 206 8

Galletly, J *Occam 2* Pitman 1990 ISBN 0 273 03067 1
Japanese edition: ISBN 4-526-02874-6

Hoare, C.A.R *Communicating Sequential Processes* Prentice-Hall 1985 ISBN 0 13 153271 5

Inmos Limited *Occam Programming Manual* Prentice-Hall 1984 ISBN 0 13 629296 8
Japanese edition: ISBN 4-7665-0133-0

Inmos Limited (tr Fontaine A.B) *Occam 2: Manuel de Reference* (in French) Masson 1989

Inmos Limited *Occam 2 Reference Manual* Prentice-Hall 1988 ISBN 0 13 629312 3

Inmos Limited *Communicating Process Architecture* Prentice-Hall 1988 ISBN 0 13 629320 4

Inmos Limited *Transputer Technical Notes* Prentice-Hall 1989 ISBN 0 13 929126 1

Inmos Limited *Transputer Development System, Second Edition* Prentice-Hall 1990 ISBN 0 13 929068 0

Jones, Geraint *Programming in occam* Prentice-Hall 1987 ISBN 0 13 729773 4

Jones, G and Goldsmith, M *Programming in Occam 2* Prentice-Hall 1988 ISBN 0 13 730334 3

Kerridge, J *Occam Programming: A Practical Approach* Blackwell Scientific 1987 ISBN 0 632 01659 0

Pountain, D and May, D *A Tutorial Introduction to Occam Programming* Blackwell Scientific 1987 ISBN 0 632 01847 X

Roscoe, A.W. *Laws of Occam Programming* Oxford University Computing Laboratory, Programming Research Group 1986 ISBN 0 902928 34 1

Wexler, John *Concurrent Programming in occam2* Ellis Horwood 1989 ISBN 745 80394 6

R.2 Conference Proceedings

A large number of these have been published by IOS Press: ISSN 0925-4986

R.3 Journals, etc

Transputer Communications Wiley 1993- ISSN 1070-454X

WoTUG News (formerly *OUG Newsletter*) Published privately by the user group 1984-

Index