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 $roman\ italic\ font$.

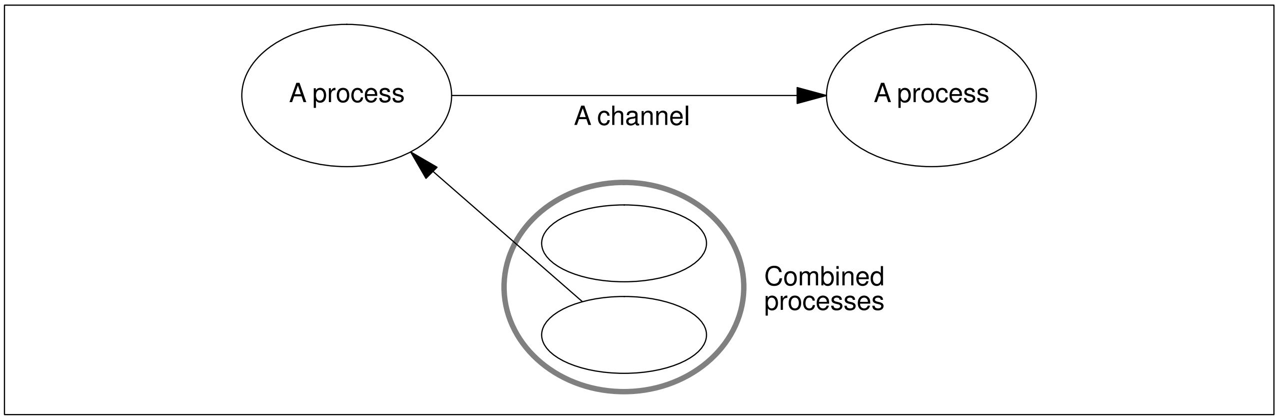

Figure 0.1 Figure conventions

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**.

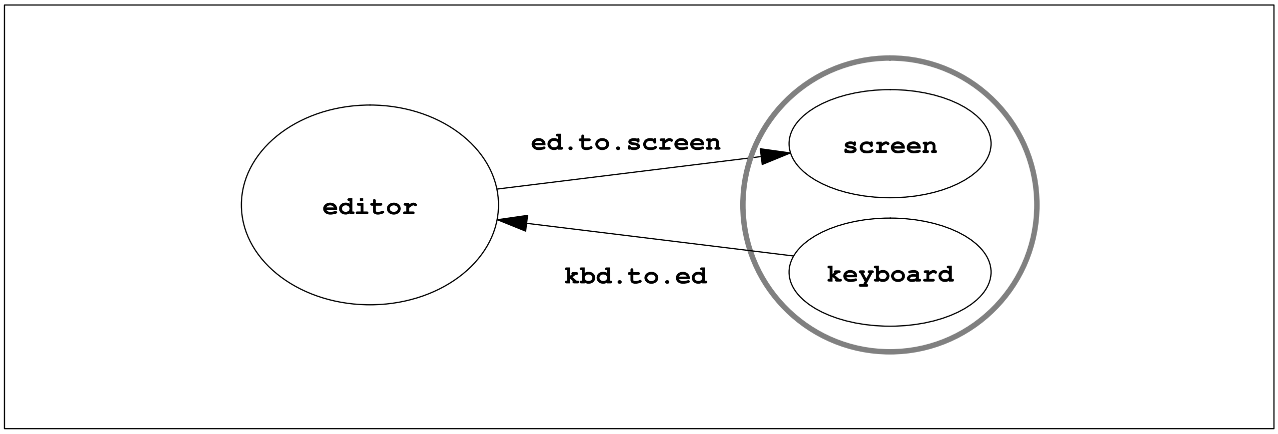

Figure 2.1 Communicating concurrent processes

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)
```

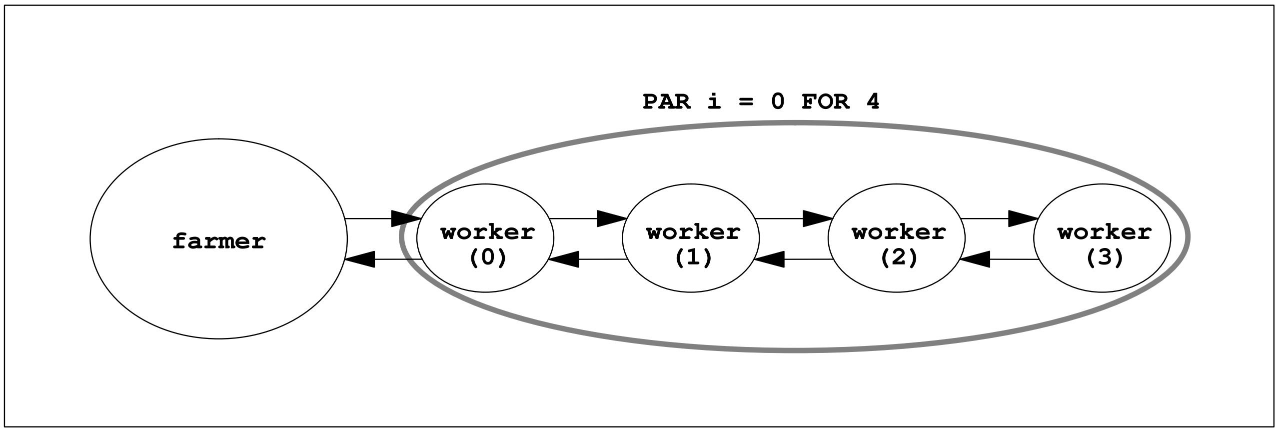

Figure 2.2 A farm of parallel processes

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.

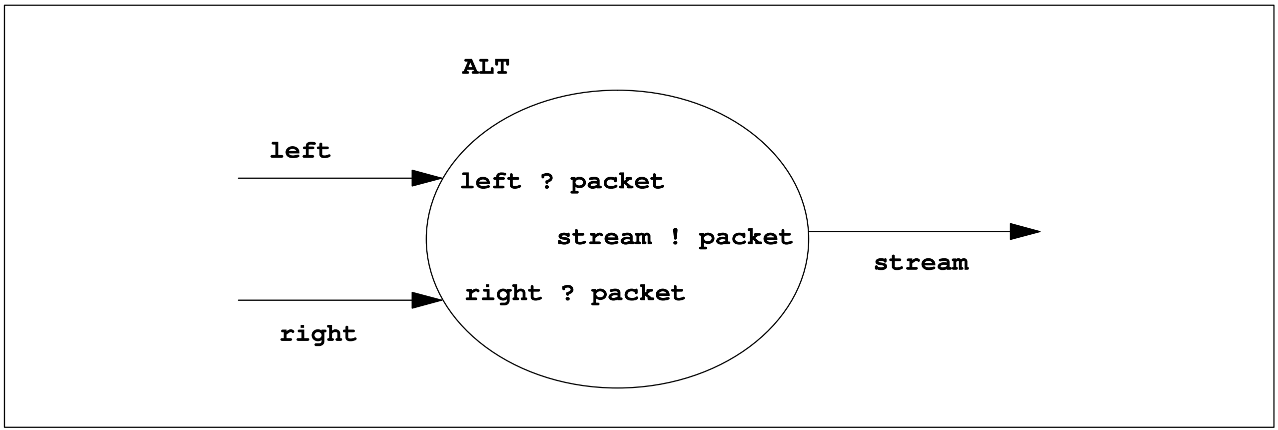

Figure 2.3 Merging the flow of data

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
```

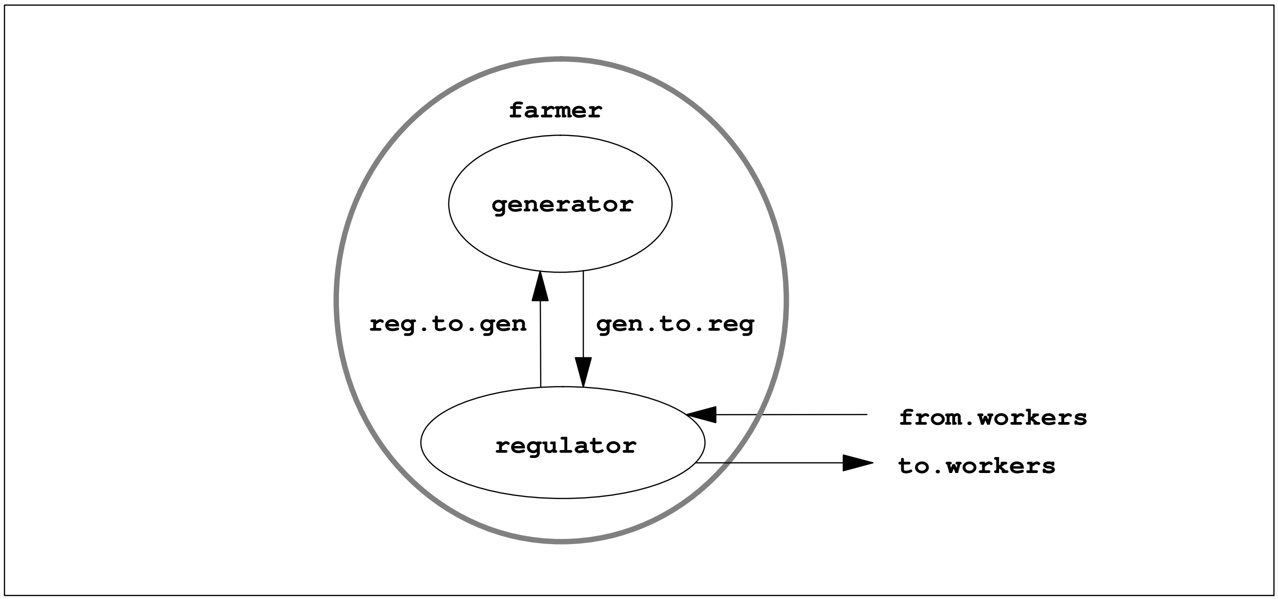

Figure 2.4 Regulating the flow of data

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.

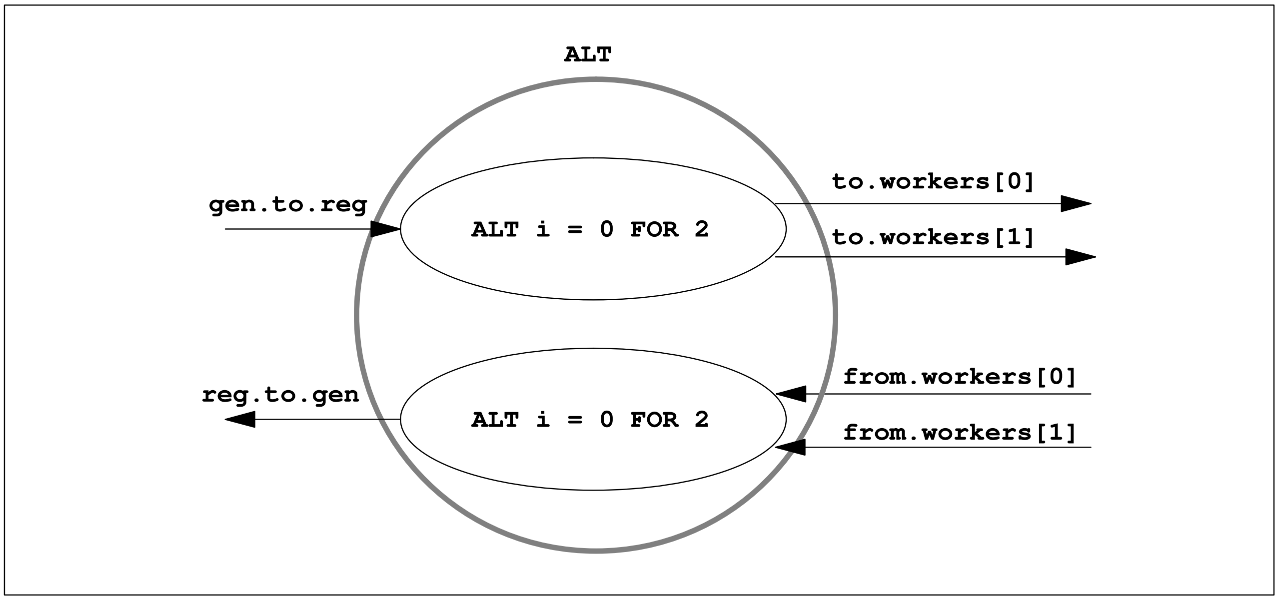

Figure 2.5 A tree structured farm of parallel processes

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** ≥ 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.