# 4 Distributed-Memory Programming: OCCAM

## 4.1 Introduction

### 4.1.1 Motivation

Parallel processing is not a new concept, and the first ideas can be traced back to 1842 with the work of Charles Babbage who said:

> When a long series of identical computations is to be performed, such as those required for the formation of numerical tables, the machine can be brought in to play so as to give several results at the same time, which will greatly abridge the whole amount of the process".

The idea also appeared in the very first issues of the Communications of ACM [Gorn58]:

> We know that the so-called parallel computers are somewhat faster that the serial ones, and that no matter how faster we make access and arithmetic serially, we can do correspondingly better in parallel. However access and arithmetic speeds seem to be approaching a definite limit ... Does this mean that digital speeds are reaching a limit, that digital computation of multi-variate differential systems must accept it, and that predicting tomorrow's weather must require more than one day? Not if we have truly-parallel machines, with say, a thousand instruction register.

The development in computer technology has concentrated however on the production of newer and faster serial computers as this approach has proved to be successful by resulting in fast, efficient, and cheap machines. There was no ground for developing alternative technology. The trend in hardware technology focuses on increased miniaturisation (from relays to VLSI), speed and reliability. Complex components became available for very low costs wide spreading the applications of computer systems. The construction of more complex applications has then required faster computer, and the limitations on further development of serial computers have become more and more evident.

Factors such as speed of the propagation of signals (limited to the speed of light), the physical limit of miniaturisation and the number of circuit modules which can be packed into a certain area of a chip have became clear. Another important factor has been the decrease in the cost per element which has stimulated the utilisation of more elements connected in parallel. The trend towards even higher degrees of parallelism as a means of improving performance has therefore been inevitable for both technological and economic reasons.

### 4.1.2 Models of computation

The development of new hardware systems provides us opportunities for their exploitation but also presents us with the problem of developing appropriate tools for the treatment of these systems. This problem has also be recognised by [Gorn58]:

> But visualise what it would be like a program for such a machine! If a thousand subroutines were in progress simultaneously, one must program the recognition that they have finished, if one is to use their results together. The programmer must not only synchronise his subroutines, but schedule all his machine units, unless he is willing to have most of them sitting idle most of the time. Not only would programming techniques be vastly different from the serial ones in the serial languages we now use, but they would be human impossible without machine intervention.

Several programming models for parallel computation have been developed. Each one models a different approach to the development of parallel programs. In general, parallelism on multiprocessor systems varies from shared-memory systems to distributed systems. Shared-memory systems are characterised by a group of processing elements, each with a processor and some local memory, communicating via a shared memory. This represents a natural extension of the traditional von Neumann machine. Distributed systems are similar to shared-memory systems except that the processing elements do not share memory but communicate by sending messages along a communication network. Between these two models, there are also hybrid ones which combine features of both of them.

The natural tendency in the design of parallel and distributed languages has been to include ad hoc extensions for parallelism in sequential languages [BalSteTann89]. In these languages, the programming support for distributed applications - that is, the use of multiple processors, the cooperation amongst them and the potential for partial failure is provided by library routines which invoke operating system primitives.

The major disadvantage is that control structures and data types of sequential languages are not usually adequate for parallel programming. This has then led not only to programs which are inefficient and difficult to understand but it has also been an obstacle in the realisation of the full potential of parallel machines by preventing us “thinking parallel” at every stage of the program construction.

Nevertheless, this scenario has changed and several parallel programming models have been developed. These languages (models) are more abstract than those supported by the operating systems, and include mechanisms for the exploitation of many properties of parallel systems. The diversity of parallel languages available allows us to model a parallel application in several ways. These include, for example, communicating sequential systems (CSP) where groups of (sequential) processes run in parallel and communicate through messages; functional models, where the parallelism is implicit and depends upon the order of evaluation of the functions [Hudak86]; and AND/OR or logical models where two clauses occur in parallel until one of them succeeds or both fail [ClarkGreg86].

In this course, we will introduce you to world of parallel programming. As we said above, there are now many models and languages for parallel programming, and would be impossible to cover all of them in this course. Instead, we will try to present you the general aspects of the field and, at the same time, we will give the details of a particular parallel programming language --- occam (the reasons for choosing occam will become clear in the next sections, and during this course). At the end of this course, we expect you to identify the main aspects of a parallel programming language, and be fluent in at least one of them, occam.

### 4.1.3 Transputers

Transputers are high performance microprocessors that support parallel processing through on-chip hardware. They

- be connected together by their serial links in application-specific ways and
- can be used as the building blocks for complex parallel processing systems.

The transputer is a complete microcomputer on a single chip. It has a very fast (single cycle) on-chip memory, on-chip inter-processor links, and a programmable memory interface that allows external memory to be added with the minimum of supporting logic.

The main parts of the transputer are:

- processor;
- on-chip RAM;
- system services;
- link interface;
- application specific interface.

Multi-transputer system can be built very simply. The four high speed links allow transputers to be connected to each other in arrays, trees and many other configurations. All circuitries to drive these links are on the transputer chip, and only two wires are needed to connect two transputers together.

In addition, to providing a communication link between programs running on processors, transputer links

- allow memory to be examined without loading a program, and
- permit programs to be loaded and executed.

It allows whole networks of transputers to be loaded down a single transputer link.

Each single transputer supports parallel processing through a system of internal channels implemented as words in memory. Each transputer has

- a highly efficient built-in run-time scheduler ;
- processes waiting for input or output or waiting on a timer consume on CPU resources.

The programming language occam 2 has been designed to reflect the architecture of the transputer and, for maximum coding efficiency, the whole system can be programmed in occam 2. The inherent security and code efficienty of occam and the ability to use the special features of the transputer make occam 2 a powerful tool for programming concurrent systems.

Transputers can also be programmed in C, Fortran Pascal and Ada, and their optimised design ensures efficient code. Where the programs need to exploit concurrency but still need to use languages other than occam 2, special occam code can be used to link modules together.

## 4.2 The conception of Communicating Sequential Processes (CSP)

From the study of process synchronisation problems grew out the theoretical model of Communicating Sequential Processes (CSP). Occam is in fact an able realisation of CSP conception, we can image it like one of its subset.

The original version of CSP appeared in [Hoare85] as a model for parallel computation where the distributed architecture is directly represented in the model as a set of processes connected through a communication network. CSP models also allow the programmer to explicitly specify the processes, their communication and synchronisation. It is the simplicity and structuring of CSP which make it a powerful and versatile model to specify parallel systems. The simplicity is the result of few simple but very expressive primitive constructs being used. The structuring results from the composition of these primitive constructs in parallel or sequence. This structuring principle is the same as that we learned to apply to sequential programs using while, if-then-else and sequential composition [Wirth71].

CSP is a mathematically-based notation for specifying the behaviour concurrent processes. Within the framework of CSP, a program is a collection of sequential processes, each of which may be executed concurrently with the others. The process may only interact or communicate via inter-process input/output operations - these input/output operations are the only interactions allowed between processes. The communicating processes are fully synchronised, in the sense that when a process reaches an input (output) operation, it waits for the corresponding process to reach the matching output (input) operation. At this point the input/output operation is performed - the processes are in synchronisation - and then both processes resume their execution at their own speeds. There is no queuing or buffering of messages.

A complex system may be completely (mathematically rigorously) specified in CSP. As indicated above, CSP embodies a notation for expressing the execution and interaction of a system comprising a collection of concurrent processes. Table 4-1 shown overleaf contains a few examples of statements in CSP notation.

| | |
|---|---|
| P ; Q | process P is followed sequentially by process Q |
| P ǀ Q | process P executes concurrently with process Q |
| P ≤ b ≥ Q | process P is executed if Boolean b is true, else process Q is executed |
| b * P | process P is executed while Boolean b is true |
| x := e | expression e is assigned to variable x |
| c ! e | the value of expression e is output |
| C ? x | a value is input and assigned to variable x |

*Table 4-1 Examples of statements in CSP notation*

## 4.3 Introduction to Occam

Occam is a programming language which was designed from the outset to support concurrent applications i. e. those systems whose parts may operate independently and sometimes have a need to interact. The name occam is derived from the name of 14th century Oxford philosopher, William of

Occam. He was particularly famous for a quotation known as Occam's Razor: "Entia non sunt multiplicanda prater necessitatem". Translated from Latin, this says: "Entities should not be multiplied beyond necessity" or paraphrasing, "Keep things simple". This is the whole philosophy behind the occam programming language - programming is kept simple. Excess language constructs have not been specified in the occam definition. This simplicity helps understanding. Occam programs should be more reliable, more efficient and easier to reason about.

The language occam was born in 1983 as a result of the experience and work of Prof. Tony Hoare over the previous 30 years, and a team of software and hardware designers from the chip manufacturer INMOS [ChenRem85]. occam was designed as the programming model for the transputer, permitting an efficient map onto the distributed architecture and giving the programmer full control over the hardware (processors and network). At the same time, the location transparency of the model guarantees the design of systems which can be implemented on a distributed network of processors or on a single processor.

The structured programming style of occam is a result of its process-oriented structuring --- that is, the structures of the programs are the processes. From simple primitive processes such as SKIP (null process), STOP (broken process), input and output (communication processes), and assignment, hierarchically structured processes can be constructed by composing processes using conditionals (IF), loops (WHILE), sequences (SEQ), non-determinism (ALT) and parallelism (PAR). The details will be present during this course. The merits of occam go beyond its simplicity, structuring and efficiency (as its model is directly implemented by the transputers). occam is based on strong theoretical foundations [Roscoe84][RoscHoare86]. Therefore, it is not necessary to use a complicated mathematical framework to reason about the programs.

applications, and for embedded systems, for domestic, industrial or military consumation. Its real-time features endow occam with the facilities necessary for the development of real-time applications.

## 4.4 Basics of occam

### 4.4.1 Processes and channels

In occam programming we refer to the parts of a program as **processes**. A process

- starts;
- performs a number of actions and then
- finishes.

In conventional programming languages such as BASIC, much of the activities of a program consist of changing the values, such as numbers or strings of characters, stored in variables. Take for example the next BASIC program:

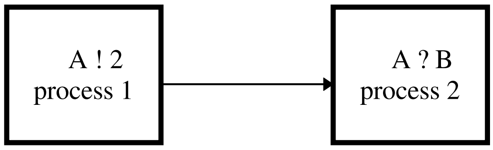
```
10    LET A=2
20    LET B=A
30    PRINT B
40    END
```

The result of running this program is that the value in both variables A and B becomes 2, and the line 30 causes this value to be printed out of the screen. There is communication of a limited sort going on in this program:

- the PRINT command provides one-way communication between the program and the external device, the screen;
- there is also a sense in which the value 2 has been communicated from A to B, Thought we wouldn't normally dignify this act with the name 'communication' because there is only one BASIC program running and it is being executed one line after another. Instead, we tend to regard the value 2 as being stored in both A and B.

Occam permits this sort of communication as a normal feature of programming. More precisely, it makes no difference for occam weather the two programs, communicating this way, are running on different computers or are just two processes running concurrently on the same computer.

A channel is a one-way, point-to-point link from a process to another one. A transfer over a channel is actually an act of copying; if the value is output from a variable, then the variable retains its value and a copy of it is sent through the channel. Occam uses symbol ! for output and ? for input so we could express the above examples by




where A is a channel and B is a variable. It is read (it means) as 'output 2 to A' and 'input from A to B'.

Since processes 1 and 2 are independent, they might be well executed at different times. The act of transferring a value from one end of the channel to the other one can only happen when both processes are ready. In other words, if the output in process 1 is executed before the input in process 2 executes, process 1 will automatically wait for process 2 before sending a value. Vice versa, if the input in process 2 was executed before process had output, process 2 would wait for a value to appear. There is no way for a value to be output into 'thin air' and lost.

The two new features which distinguish channels from the variable are the followings:

- a channel can pass values either between two processes running on the same computer, or between two processes running on different computers. In the first case the channel would in fact be just a location in memory, rather like a variable. In the second case the channel could represent a real hardware link such as a Transputer link or other communication line. Both cases are represented identically in an occam program.
- the key idea is that an occam channel describes communication in the abstract, and does not depend upon its physical implementation. You can trustily write and test a program using channels without having to worry about exactly where the different processes will be executed. The program can be developed on a single processor workstation; when it is finished and proved you may decide to distribute various processes in the program onto different computers by making a few simple declarations at the beginning of the program.
- channels are patient and polite. If an input process finds that no value is ready it will wait until one is supplied without explicit instruction from the programmer. Equally, an output will not be sent until the responsibility for 'timekeeping' of the programmers shoulders.

Communication over self-synchronising channels is a new and powerful part of occam, and it can render the writing of concurrent programs a far less formidable task than it is with conventional languages.

### 4.4.2 Names

In occam the names of objects can be as long as you like, and they must start with a letter of the alphabet. The rest of name, if any, can be made of letters, digit and the dot character. Upper and lower cases are distinguished by occam, so that tom and TOM are different names. These are valid names:

```
x Y fred chan3 Chan3 new.fred old.fred
```

Occam keywords such as SEQ, PAR and CHAN are always in upper case and they are reserved. In other word, they cannot be used as names that you create.

These are not valid names:

```
3chan                         --does not start with the letter
old-fred                      --contains illegal character '-'
fred$                         --contains illegal character '$'
CHAN                          --reserved word CHAN
```

### 4.4.3 Types

Occam, like Pascal and many other languages but unlike BASIC, requires that every object used by a program should have a type which tells occam what sort of object it is dealing with. Furthermore, the type of an object must be specified before it is used in a process. Notice the colon, which is used to end all the different kinds of specification. This colon joins a specification to the process which follows it.

The type can be a primitive type and an array one. The primitive types are the following: channel type, timer type and data type. The data types are: integer type, byte, real and Boolean.

**CHAN OF protocol**

A communication channel is of the type CHAN OF protocol. Each communication channel enables values to be communicated between two concurrent processes according to the specified protocol. It is necessary to specify the data type and structure of the values that they are to carry. This is called the channel protocol. Examples:

```
CHAN OF INT keyboard:
CHAN OF INT screen:
CHAN OF BOOL signal:
```

**Timer**

The timer is of type timer. Each timer provides a clock which can be used by any number of concurrent processes. Example:

```
TIMER clock:
```

**INT**

It is an integer or whole number. INT16, INT32, INT64 are numeric types represented by using 16, 32, 64 bytes, respectively. Examples:

```
INT x, X:
INT a, b, c:
```

**BYTE**

It is a nonnegative integer between 0 and 255, very often used to represent characters. Example:

```
BYTE x:
```

**REAL**

It is a signed real value, represented according to IEEE standard 754-1985. REAL32, REAL64 are numeric types represented by using 16, 32, 64 bytes, respectively. Examples:

**BOOL**

The BOOL type is stored in 8-bits - TRUE having the numeric value 1, False having the numeric value 0. Example:

```
BOOL more:
```

**Array**

Array types are constructed from component types. An array a channel type, timer type or data type, depending on the types of its components. Two arrays have the same type if they have the same number of components and the types of their components equal. In the array type [e] T, the value of e defines the number of the components in an array of the array type, and T defines the type of the components. In occam, array variables are declared in the same way as single variables of any type, but with the number of components in brackets prefixed to the type specified. The size of the array must be fixed when the program is compiled; it cannot be decided or changed while the program is running. Examples:

```
[20] INT fred:              -- an array of 20
                            -- integer called fred
```

```
[100] CHAN OF INT switchboard:   -- an array of 100
                                 -- channels caled
                                 -- switchboard
[12] BOOL jury:                  -- an array of 12
                                 -- truth values called
                                 -- jury
[4][5] INT matrix:               -- an array of
                                 -- 4 elements,
                                 -- every element of
                                 -- each is [5] INT
fred[0]                          -- the first integer
                                 -- in fred
switchboard[20]                  -- the 21th channel in
                                 -- switchboard
jury[11]                         -- the last component
                                 -- of jury
```

**Constants**
A name can be given to a constant value by specifying it with:

|VAL type name IS value:

So we could write:

```
VAL INT year IS 365:
VAL INT leap.year IS 366:
```

The type can be omitted as occam can deduce it from the value:

```
VAL year IS 365:              -- year is an integer
VAL spac IS ' ':              -- space is a BYTE
```

Possible ambiguities over BYTE and INT are resolved by explicitly specifying the type of the value.

**Hexadecimal notation**
Numeric constants can be entered in hexadecimal notation by preceding them with the # sign:
`#FE (equivalent to 254 decimal)`

#### *4.4.3.1 Scope*

The named objects of an occam process, such as constants, variables and channels, must be declared prior to any process in which they are used. The scope or range of definition of the named objects is the range of definition of the process immediately following their declaration. The range of definition of a process is determined by its indentation, and so, the scope of the named objects is defined by the indentation of the following process. The named objects are called “local” to that process and any component processes.

```
INT item:                     -- declaration of item
SEQ                           -- scope of item
  item := 10
  .
  .
  .
  SEQ
    item := item+1
```

In this example, the variable item is available to the inner SEQ constructions since this variable is still in scope.

```
PAR
```

```
INT fred:
SEQ
  chan3 ? fred
  more processes
INT jim:
SEQ
  chan4 ? fred
  more processes
```

An error will be reported because fred exists only inside the first SEQ and jim exists only inside the second SEQ. The second fred will therefore look to occam like an unspecified variable.

The consequence of scoping is that the value of a variable is undefined outside the scope of the declaring process. It means that program objects with different scopes may have the same name but lead to independent existences i. e. they are totally separate objects, and may even have different types.

```
CHAN OF BYTE chan:
PAR
  BYTE char:                      -- first declaration of char
  SEQ                                   -- scope
    char := 'a'
    chan ! char
  BYTE char:
  SEQ
    chan ? char
```

A variable named char is declared for both sequential processes. However, these variables are quite separate, each being local to its own sequential process. Channel chan is declared outside of PAR construction and so the channel is common to both component processes.

If an object is declared within the scope of an existing object with the same name, and then all references to that name refer to the more recent declaration i. e. the more recent declaration takes precedence in a clash of scope. The older declaration is suspended for the duration of the scope of the more recent declaration but resumes its role afterwards.

```
BYTE item:                     -- declaration as BYTE
SEQ                            -- scope as BYTE
  item := 50 (BYTE)
  INT item:                    -- declaration as INT
  SEQ
    item := 500
    chan2 ! item               -- gives 500
  chan1 ! item                 -- gives 50 (BYTE declaration
                               -- restored)
```

where chan1 is specified as type BYTE, and chan2 is specified as type INT. In this example, the second declaration of variable item masks the first declaration for the duration of the second's scope. Any reference to item during this period will refer to the second declaration. Thus, the output via channel chan2 produces the value 500 as the second declaration in force here. However, the output via channel chan1 produces the value 50 after the first declaration is back in force - the second one is no longer in the scope.

Specifying a variable in occam does not initialise its value to zero. The value of a variable is undefined garbage until it has been assigned to or has input a value. The value of a variable has meaning only during the execution of the process for which it is declared. It is important to realise that occam does not guarantee one the value of a local variable to be kept from one execution of procedure to the next one. If a value of a variable needs to be retained between executions, then the variable must be declared in a higher level process. Such a variable is called free variable.

```
INT x:
  WHILE x >= 0                 -- unwise: x is garbage here
    SEQ
```

```
        input ? x
        output ! x

    INT x:
    SEQ
      x := 0
      WHILE x >= 0                -- correct
        SEQ
        input ? x
        output ! x
```

#### *4.4.3.2 Type conversions*

Sometimes it is convenient to convert one type to another in a program; it may, for instance, save to declare several extra variables for a value that is only required once. Type conversion should be prevent values being used in inappropriate situations.

If **`number`** has been declared as INT and **`digit`** as byte, we could still add them together like this:

```
number := (number * 10) + (INT digit)
```

The reverse conversion of INT to BYTE is legal only if the value is within the byte range of 0 to 255. For example, to output a number between 0 and 9 as a character we could write:

```
output ! BYTE (number + (INT '0') )
```

Values of type BOOL can be converted to type INT or BYTE and vice versa, using the following definitions:

```
INT TRUE or BYTE TRUE          is 1
INT FALSE or BYTE FALSE        is 0
BOOL 1                         is TRUE
BOOL 0                         is FALSE
```

so if value of **`active`** is FALSE, INT **`active`** is 0.

### 4.4.4 Operators

#### *4.4.4.1 Arithmetic in occam*

The basic arithmetic operation are the followings:

| | |
|---|---|
| x + y | -- add y to x |
| x - y | -- subtract y from x |
| x * y | -- multiply x by y |
| x / y | -- quotient when x is divided by y |
| x REM y | -- remainder when x is divided by y |

These operations can be performed on INT or REAL numbers type.

All operators have the same priority in occam so parentheses must be used in complex expressions to enclose component operations and allow them to be treated as single operands. This also established the order of evaluation. For example:

```
fred = (2 + jim)
(fred + jane) < jim
```

#### *4.4.4.2 Relational operators*

They allow operands to be compared, the result of the comparison is one of the logical truth values TRUE or FALSE. The occam provides the following tests:

| | |
|---|---|
| = | -- equal to |
| <> | -- not equal to |
| > | -- greater than |
| < | -- less than |
| >= | -- greater than or equal to |
| <= | -- less than or equal to |

These tests may only be applied to two values of the same type, and they always yield a value of type BOOL. For example, the test 2 <> 3 yields the value TRUE since 2 does not equal 3. The truth values TRUE and FALSE are occam constants which can be used in any situation where a test could be used; you can regard them as tests whose outcome is decided in advance.

Operands, which may be constants, variables or expressions, used with the equality (=) and inequality (<>) operators may be any primitive data type but the operands used with the remaining relational operators must be of byte, integer or real types only. Characters and strings may be compared with relational operators, according to their ASCII ordering. For example, if comp is declared to be type BOOL, then

```
comp := (3 = 25)              -- evaluates to FALSE
comp := (2 < 25)              -- evaluates to TRUE
comp := (7.15(REAL(32) >= 3.25(REAL32)-- evaluates to TRUE
comp := ('A' < 'B')           -- evaluates to TRUE
comp := (fred < mary)         -- evaluates to TRUE
```

#### *4.4.4.3 Boolean operators*

Boolean operators comprise the usual logical connectives, AND, OR and NOT. They allow the logical combination of Boolean operands - either simple Boolean variables or Boolean expressions. The result is a Boolean value, TRUE or FALSE. For example:

```
(fred < mary) AND (alfa > beta)
```

is the logical AND combination of the Boolean expressions fred < mary and alfa > beta. The value of the result depends on the current value of the numeric variables being compared.

#### *4.4.4.4 Bitwise operators*

To allow low level operations on the individual bits in a value, occam provides bitwise operators.

**Setting, masking**

These operators allow various operations to be performed on the individual bits or pattern of bits comprising the value of a constant, variable or expression of integer type. The first operand represents the bit pattern which operates on this value, and may be itself a constant, variable or expression of integer type.

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

For example:

```
int \/ #3F
```

shows the bitwise oring of a variable int with the constant #3F. The bit pattern of #3F is 0011 1111, and so this operation will set to one the 6 low-order bits of int.

**Shift operators**

The bit comprising the value of an integer constant, variable or expression may be shifted left or right by a specified number of bit positions.

| |
|---|
| << left shift |
| >> right shift |

Such shifting is equivalent to multiplying or dividing by multiples of two. The number of the bit positions to be shifted is given by the second operand. This operand, which may be itself a constant, variable or an expression, must be of INT type. The number of bits to be shifted must not be greater than that of the bits allowed for the integer type of the first operand. The bit positions of the first operand, vacated by the shift operation, are filled with 0 bits. For example:

```
int << 3
```

shifts the value of the variable int left by three bit positions; this is equivalent to multiplying by 8.

```
int >>7
```

shifts the value of the variable into right by seven bit positions; this is equivalent to dividing by 128.

## 4.5 Primitive processes of Occam

### 4.5.1 Assignment process

An assignment process changes the value of the variable, just as in most conventional languages. The symbol for assignment in occam is :=. So the assignment process:

```
fred := 2
```

makes the value in variable **fred** two. The value assigned to a variable could be an expression such as:

```
fred := 2 + 5
```

and this expression could contain other variables:

```
fred := 5 - jim
```

Multiple assignment, assignment to more than one variable at the same time, is also possible in occam:

```
fred, john := 2, 3
```

This multiple assignment process makes the value in the variable **fred** two, and the value in the variable **john** three. This is really useful for swapping the value of variables:

```
fred, john := john, fred
```

### 4.5.2 Input process

An input process inputs the value from the channel into a variable. The symbol for input in occam is ?. The input process:

```
chan3 ? fred
```

takes a value from a channel called **chan3** and puts it into variable **fred**.

Input processes can input values only to variables. It is quite meaningless to input to a constant or to an other expression.

An input process cannot proceed, until a corresponding output process on the same channel is ready.

### 4.5.3 Output process

An output process outputs a value to a channel. The symbol for output in occam is !. The process:

```
chan3 ! 2
```

outputs the value 2 to a channel **chan3**.

The value output to a channel can be anything that you could assign to a variable, so it may be a variable or an expression, and the expression may contain variables.

An output process cannot proceed until a corresponding input process on the same channel is ready.

Thus, communication over a channel can occur only when both input and output processes are ready. If during the execution of a program, an input process is reached before its corresponding output process is reached, the input will wait until the output becomes ready. Should the output be reached first, it will wait for its input.

A value, communicated over a channel, is copied to the input variable and the value of the output variable remains unchanged.

### 4.5.4 SKIP

The special process SKIP may be regarded as it represents a process which does nothing. It might be used in a partly complemented program in place of process which will be written later but for the moment is allowed to do nothing.

For example, a process which is to drive an electric motor could be replaced by SKIP when testing the program without motor. There are also occasions when you do not want anything to happen but the syntax of the occam requires a process to be present.

### 4.5.5 STOP

The special process STOP may be regarded a process which does not work, or is 'broken'. It might be used, like SKIP, to stand in for a process which has yet to be written.

For example, a process for handling errors could be replaced by STOP in the early stages of testing a program.

The effect of 'broken' process tends to spread because any process which communicates with a broken process will itself never finish, and hence it becomes broken, too.

So far, we have loosely used the term 'finish' when referring to a process. Concurrent programming in occam requires us to be more precise this. A process which completes all its actions is said to be terminate. Normally a process starts, proceeds and then terminates. A process which cannot proceed is said to be stopped by waiting for an event which will never happen, due to a programming error, in which case it is said to be deadlocked.

Correct termination of concurrent programs is not a trivial matter since, they may have many parallel processes which communicate with one another.

## 4.6 Non-primitive processes

Several primitive processes can be combined into a larger process by specifying them to be performed one after the other, or all at the same time. This larger process is called a construction and it begins with an occam keyword which stated how the component processes are to be combined.

### 4.6.1 SEQ construction

The simplest construction to understand is the SEQ, short for sequence, which merely says ‘do the following processes one after another’. Here is an example:

```
SEQ
  chan3 ? fred
  jim := fred + 1
  chan4 ! jim
```

It means, do in sequence, input from **chan3** to **fred**, assign **fred+1** to **jim** and output **jim** to **chan4.** In sequence means, to be more precise, that the next process does not start until the previous has terminated. A SEQ process therefore works just like a program in any conventional programming language; it finishes when its last component process finishes.

### 4.6.2 PAR construction

The PAR construction means to do the following processes all at the same time', I. e. parallel. All the component processes of a PAR start execution simultaneously. For example:

```
PAR
  SEQ
    chan3 ? fred
    fred := fred + 1
  SEQ
    chan4 ? jim
    jim := jim + 1
```

says: at the same time, input from **chan3** to **fred** and then add one to the result, while receiving input from **chan4** to **jim** and then adding one of the result.

The important thing to keep in mind is that in a PAR, the written order of the component processes is irrelevant as they are all performed in the same time. PAR is not quite so easy to understand as SEQ because the idea of things happening simultaneously in computer programs is new to many programmers.

All component processes in a PAR start at the same time, and the PAR construction itself terminates when all its component processes have terminated, and that is all we need to know.

### 4.6.3 Conditional processes

In addition to repetition, all programming languages have to provide a way for programs to choose to do different things according to condition i. e. the result of a test. In occam, one form of conditional choice is provided by the construction called **IF**.

**IF** can take any number of processes, each of which has a test placed before it, and make them into a single process. Only one of the component processes will actually be executed, and that will be the first one (in the order in which they are written) whose test is true:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
```

In this fragment of program (we assume **x, y, chan1** are **chan2** and declared elsewhere), the value of **y** will either be output on **chan1** or **chan2** depending upon whether the value of **x** is 1 or 2.

The tests **x = 1** and **x = 2** are Boolean expressions which are used to choose which component of the **IF** is to be executed. The component parts of the **IF**, each composed of a Boolean expression and a process, are called choices.

What if the value x were 3? Then the **IF** process would cause the program to stop just as if **STOP** had been executed. The program can only proceed if one of the choices is executed.

(An **IF** with no choices in it just acts like a **STOP** A **PAR** or **SEQ** with no component processes on the other hand acts like **SKIP** i. e. the program continues as if it were not there at all.)

In many cases it will not be acceptable to stop the program if x is neither 1 nor 2. In that case we must add another choice which will be executed no matter what the value of x. This is accomplished by using **TRUE**.

```
IF
  x = 1
```

```
    chan1 ! y
  x = 2
    chan2 ! y
TRUE
  chan3 ! y
```

Now, y will be output on **chan3** if x has any other value but 1 or 2, because of test on the last choice is always executed by default if no previous choice has been executed. If we want nothing to happen at all when x was not 1 or 2 we can say:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
TRUE
  SKIP
```

This provides one example of the utility of **SKIP**; occam requires some sort of process after the guard and will not allow just blank space.

A better way of writing the example above is to explicitly state each case as in the following example:

```
IF
  x = 1
    chan1 ! y
  x = 2
    chan2 ! y
(x <> 1) AND (x <> 2)
  SKIP
```

It is a much better way to write conditional processes since it is totally unambiguous. For convenience and simplicity, in this tutorial we will in some places continue to use **TRUE** as a condition. In real programs you should avoid doing so.

To make more complex choices, **IF**s can be nested by using an **IF** process in a choice of another **IF** construct.

```
IF
  x = 1
    chan1 ! y
  x = 2
    IF
      chan2 ! y
    TRUE
    chan3 ! y
  TRUE
    SKIP
```

In this process **chan3** is used for the output if x is 2 and y has any other value than 1.

### 4.6.4 Selection processes

Like many other programming languages, occam provides a further means of making a choice depending upon the value of variables. Such a construction is called a selection in occam and provides an efficient means of selecting one of the numbers of options in a **CASE**.

**CASE** can take any number of processes, each of which has a list of one or more expressions placed before it, and can combine them into single process. Only one of the component processes will actually be executed, and that will be the first one (again, in the order in which they are written) with an expression, which has the same values as the selecting variable:

```
CASE x
```

```
    1
      chan1 ! y
    2
      chan2 ! y
```

In this fragment of program (which is similar to the example, used to describe IF), the value of y will either be output on **chan1** or **chan2** depending upon whether the value of x is 1 or 2. Typically, the constants used in **CASE** will be named, and also there can be more than one **CASE** expressions:

```
CASE x
  i, j
    chan1 ! y
  k
    chan2 ! y
```

In this program fragment each constant expression has been given a name: i, j and k. The value of y will be output on **chan1** if x has the same value as i, and will be output on **chan2** if x has the same value as k.

What if the value of x was none of these values? Then the process would cause the program to stop just as if **STOP** had been executed. The program can only proceed if one of the option is executed, just as an **IF** may only proceed if a choice is executed.

Once again (as we saw with **IF**), in many cases it will be not acceptable to stop the program if x does not have the same as one of the case expressions. We must then add a further option which will execute no matter what the value of x is. This is accomplished by using **ELSE**:

```
CASE x
  i, j
    chan1 ! y
  k
    chan2 ! y
  ELSE
    chan2 ! y
```

Now y will be output on **chan3** if x has any value other than i, j or k.

The component parts of the **CASE**, each composed of an expression and a process, are called options.

### 4.6.5 Repetitive processes

All programming languages provide for some means of looping, i. e. performing an action repeatedly. In general, it is a convention to distinguish two kinds of repetition: repeat for a specified number of times, or repeat while a given condition holds. occam has both types of repetition. The second conditional loop is performed by a construction called **WHILE**, which includes a test such as x < 0 or fred = 100. The resulting process is executed while the test results is true, or looked at another way, until it becomes false.

For example:

```
INT x:
SEQ
  x := 0
  WHILE x >= 0
    SEQ
      input ? x
      output ! x
```

will continue to read values from channel **input** and send them to **output** so long as the value is not less than zero. Every time the inner **SEQ** process terminates, the **WHILE** process will be performed again and the test will be repeated. It continues so long, as the test result is **TRUE** i. e. so long as **x** is greater or equal to zero. When a negative value is received the **WHILE** process terminates.

The net effect of this process is to buffer (i. e. store) a single value on its way from input to output. Occam programs are often designed by making the major processes communicate on a channel to buffer, filter, or transform the transmitted values, almost as if they were electrical components rather than programs.

The logical values **TRUE** and **FALSE** can be used as constant in an occam program, anywhere that a test could be used so:

```
WHILE TRUE
  INT x:
  SEQ
    input ? x
    output ! x
```

it will continue to read values for ever (or until you pull the plug), whereas:

```
WHILE FALSE
  INT x:
  SEQ
    input ? x
    output ! x
```

is a pointless sort of process which terminates immediately and will read no values at all.

### 4.6.6 Alternative processes

A process may use several channels for input, or for output, or for both, in order to communicate with another process. So we can make choices according to the state of channels. It is made possible by the ALT construction, whose name is short for alternation.

ALT joins together any number of components into a single construction but the component part of an ALT called alternatives is some more complicated.

The simplest kind of ALT has as each alternative an input process followed by a process to be executed. The ALT watches all the input processes and executes the process, associated with the first,input to become ready. Thus, ALT is basically a first-past-the-post race between a group of channels, with only the winner's process being executed:

```
CHAN OF INT chan1, chan2, chan3 :
INTx:
ALT
  chan1 ? x
     .... first process
  chan2 ? x
     .... second process
  chan3 ? x
     .... third process
```

If chan2 was the first to produce an input, then only the second process would be executed.

Here choice is being decided in the time dimension, the inputs causing the program to wait until one of them is ready.

### 4.6.7 Communicating processes

Communication between two processes is the essence of occam programming. At its simplest way version it requires two processes executing in parallel and a channel joining them:

```
INT x:
CHAN OF INT comm:
PAR
  comm ! 2
  comm ! x
```

This trivial program merely outputs the value 2 from one process and inputs it into the variable x in the second. Its overall effect is exactly as if we had a single process which assigned 2 to x.

Communication between the component processes of a **PAR** must only be done by using channels, occam does not allows us to pass values between parallel processes using shared variable.

In fact, if a component of a **PAR** contains an assignment or input to a variable, then the variable must not be used at all in any other component:

```
INT x, y :
PAR
  SEQ
    x := 2
    ... more processes
  SEQ
    y := x                    -- illegal!
    ... more processes
```

Keeping variables local to component processes and using channels to communicate values is the right way to do it.

It can be seen like a severe restriction to programmers who have experience with conventional languages. It will certainly be the biggest source of errors when first programming in occam.

Like a prohibitions it will be more easily borne if the reason for it is understood. The reason is both simple and necessary.

Parallel processes run at the same time and, in general, they run asynchronously i. e. They come at they own place, only coming into synchronisation with each other briefly when forced to by communication over a channel.

If occam allowed one parallel process to read from a variable which has its value altered in another parallel process, what value will be read? It depends upon whether or not the other process has altered it yet, and this cannot be known since the processes are asynchronous. And what if the altering process chooses to alter the variable’s value at the moment when the second process is reading it? What would the value be then?

Such a scheme is obviously unworkable hence the prohibition. But couldn't we organise it so that a variable warns the other process that it has had its value charged? We could indeed; the resulting object already exists in occam and it is called a channel Q. E. D.

In occam variables are used for storing values, while channels are used for communicating values.

Lets now return to the main track with a more complicated example of **PAR** which performs some arithmetic on a value before passing it on:

```
CHAN OF INT comm:
PAR
  INT x:
  SEQ
    input ? x
    comm ! 2 * x
  INT y:
  SEQ
  comm ? y
    output ! y + 1
```

Here we have two channels called **input** and **output** which lead to other processes or perhaps to the outside world. We assumed that they have been declared elsewhere in a larger program. This piece of program uses two processes working in parallel one of which multiplies an input value by two , the other adds one to the result and sends it on its way to the output. The times-two process and the add-one process communicate on channel **comm**.

In this case it worries you this is not a particularly useful thing to do; it is purely for illustration. It would be much simpler to do times-two and add-one in a single SEQ process, or indeed in a single expression. But later on, when we have more of occam at our disposal, we shall see how this sort of thing can be very useful, indeed. At this early stage, all examples of communicating PARs will tend, unfortunately, to appear trivial.

It was told several times already that an occam channel is a one-way link between a pair of processes, but it is useful now to examine exactly what it implies. In a communicating PAR construct it means that:

1. Only two component processes of the PAR may use any particular channel, one as the sender and the other as receiver.

```
CHAN OF INT comm:
PAR
  SEQ
    comm ! 2
  INT y:
  SEQ
    comm ? y
  INT x:  -- ILLEGAL! two processes
  SEQ     -- inputting from the same channel
    comm ? y
```

2. The sender process can only contain outputs to the channel and the receiver must only contain inputs from the channel

```
CHAN OF INT comm:
PAR
  SEQ
    comm ! 2
  INT y:
  SEQ
    comm ? y
    comm ! y+1
     -- ILLEGAL! input and output
     -- from the same channel in the same process
```

For two-way communication between two processes we need two channels:

```
CHAN OF INT comm1, comm2:
PAR
  INT x:
  SEQ
    comm1 ! 2
    comm2 ? x
  INT y:
  SEQ
    comm1 ? y
    comm2 ! 3
```

The effect is that each process sends a value to the other; x ends up with the value 3 and y with the value 2. The order of the inputs and outputs in each SEQ matters very much here and it is important to understand why.

If we were to write:

```
CHAN OF INT comm1, comm2:
PAR
  INT x:
  SEQ
    comm2 ? x
    comm1 ! 2
  INT y:
```

```
SEQ
  comm1 ? y
  comm2 ! 3
```

then the program would never terminate; we have the dreaded deadlock.

Why deadlock? Because both SEQs wait patiently for an input to become ready. But since each is waiting for the other to output, neither can proceed to make the necessary output! It is rather like those comical scenes when two men passing in a narrow doorway repeatedly step to the same side to make way, so repeatedly blocking each other. Swapping the input and output in either process resolves the deadlock.

Take care: sequence your programs to ensure that two parallel processes are never waiting for a sequentially later output from the other. This is the only circumstance in which occam requires you to worry about such matters but watch out for it. Like certain stalemates in the game chess, it may be disguised in complex processes.

### 4.6.8 Replicators

One of the most powerful features of occam is that it allows the construction of arrays of processes in addition to data and channel arrays.

The device replicator is used together with one of the occam constructs SEQ, PAR, ALT and IF to create an array of similar processes of the corresponding kind. Individual processes in a repliated construct can be referred to using the replicator index in just the same way that components of an array are selected using a subscript. The general form of a replicator is:

```
REP index = base FOR count
  process
```

where Rep is one SEQ, PAR, ALT or IF. Take care: if the count of replicator is zero, the process behaves like a single construct with no components i. e. SEQ will act like SKIP (do nothing) but IF and ALT act like STOP (stop the process).

**Replicated SEQ**

The most straightforward replicated construct is the replicated SEQ. If input is specified as a channel then

```
INT x:
SEQ i = 0 FOR 5
  input ? x
```

says “create five replicas of the input process and execute them in sequence”. The effect is as we had written:

```
INT x:
SEQ
  input ? x
  input ? x
  input ? x
  input ? x
  input ? x
```

which in its turn is as if we had specified a loop with five iterations:

```
INT x, i:
SEQ
  i := 0
  while i < 5
    SEQ
      input ? x
      i := i + 1
```

In other words a replicated **SEQ** is equivalent to a counted loop. Using a replicated **SEQ** is more concise than using **WHILE** because there is no need to specify, increment and index variable and test its value each time round.

The replicator index variable, which can be given any name, increases by one from the base value for count times, and can be used to reference particular pass through the loop. Note that the index does not need to be separately specified, and always of type INT. An array of any size could be filled with input values like this:

```
SEQ i = 0 FOR SIZE big.array
  input ? big.array[i]
```

which has the same effect as:

```
SEQ
  input ? big.array[0]
  input ? big.array[1]

  input ? big.array[(size big.array - 1)]
```

occam allows either or both bases and count values in the replicated SEQ to be variables rather than constant.

Take care: it is forbidden to input or assign to the replicator index. There is no way to cause partial execution of a replicated SEQ (or any other replicated construct); remember that they are not loops but arrays or processes which only terminate when all their processes are terminated.

**Replicated PAR**

A replicated **PAR** builds an array of structurally similar parallel processes. Any process can be referred to by means of the replicator index.

The replicated **PAR** is of paramount importance in occam programming. Used in conjuction with an array of channels, it permits an economical and elegant expression of some of the stock data structures used by programmers, such as buffers and queues, but futhermore, it allows the exploitation of multiple concurrent processors using pipelining and other techniques.

Technical note: current implementation of occam does not allow the count in a replicated **PAR** to be a variable value. This is in order that the compiler can know all the reseources needed by a process at compile time.

As an example let`s look first a simple queue in Figure 4-1. A queue is precisely what you would expect from the everyday meaning of the word, a structure through which a number of values pass, with the first value to arrive being also the first value to leave:

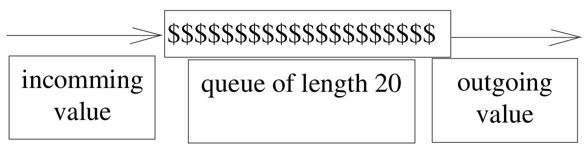



*Figure 4-1 A simple queue*

Queues are used extensively in programming, often as buffers for processes which consume data more slowly then they are being supplied. In order that no data should be lost, they are buffered or queued to wait their tum. Such buffers are often called FIFO buffers, short for First In/First Out.

A typical application would be in a computer terminal`s keyboard. Some programs may accept characters slower than a fast typist can enter them. If a queue is used to accept the characters, then users can type as fast as they wish, the queue is growing as they type ahead of the program.

In occam a queue can be simulated by an array of parallel processes, passing data from one to the other like a bucket chain:

```
[21]CHAN OF INT slot :
```

```
PAR i = 0 FOR 20
  WHILE TRUE
    INT x :
     SEQ
       slot [i] ? x
       slot [i+1] ! x
```

The replicated **PAR** sets up 20 parallel processes each of which continually transfers values between two slots in the queue, which is represented by an array of 21 channels. The net effect is that each value has to pass through the whole queue before leaving from **slot [20]**.

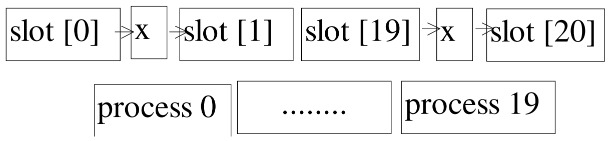



*Figure 4-2 Queue implementation with replicated PAR*

As it is seen it is not a complete program because the first process in the queue needs to input the values from somewhere, and the last process needs to outputs to somewhere. Otherwise each will be waiting and the whole queue will be deadlocked. We assume that the queue is a part of the larger program which feeds data into **slot [0]** and bleeds data from **slot [20]**, like this:

```
PAR
...feed
...queue
...bleed
```

If the supply of data were temporally interrupted, data being already in the queue would continue to be passed along, the earlier processes would wait automatically by the very nature of the occam input process.

When the supply starts again, the early processes can proceed but later processes anywhere in the line are to be waiting even if the flow of data goes on - only if the queue is allowed to become empty of full will all the processes be waiting.

Replicated **PAR** exemplies the occam approach to programming and its difference from conventional languages.

## 4.7 Subroutines in Occam

### 4.7.1 Abbreviations

An abbreviation is an occam language feature for producing a succinct alias, or new name, for an occam expression or element. Within an occam process the abbreviation is then used instead of the original expression or element. This feature allows simplification of complicated occam statements. Essentially, the abbreviation behaves as a macro for the expression or element. In execution the effect of an abbreviation is equivalent to the substitution of the abbreviation name by the original expression or element. The usual scoping rules apply to an abbreviation. When a new name is in force due to an abbreviation, the old name of the expression or element may not be used within the scope of the abbreviation. We can remark that the constant declaration is a simple form of abbreviation. For example:

```
VAL maximum IS 1000:
```

Abbreviations are categorised as being either expression abbreviations or element abbreviations.

#### *4.7.1.1 Expression abbreviations*

This form is used to abbreviate the values of expressions. The value of an abbreviated expression must remain constant whilst the abbreviation is in scope. The simplest format is:

```
VAL type name IS expression:
```

where

- *type*, the type of the abbreviation, is one of the primitive data types. The inclusion of type is optional and may be omitted if the type may be determined from the data type of the expression.
- *name* is an occam identifier of the abbreviation, and
- *expression* is a valid occam expression, such as a constant, a variable, an array component or some combination of these. The type of the expression must be the same as the type, if it is specified. Any variables used in expression must not be changed by assignment or input within the scope of the abbreviation. Any array component used in the expression must have a valid subscript.

The use of the reserved word VAL underlines the constant value of this type of abbreviation.

Example 1

```
VAL Return is '*c':

VAL AbsolutZero IS −273;

VAL TwoPi IS 2.0 (REAL32) * 3.14159 (REAL32):

VAL BufferFull IS COUNT > Limit:

VAL INT16 Maximum IS 32767 (INT16):

VAL REAL32 Volume IS (Length*Breadth)*Height:

VAL REAL64 Error IS Theory – Experiment:
```

So for example **`TwoPi`** may be used to represent the value of the expression **`2.0 (REAL32) * 3.14159 (REAL32)`** and **`Volume`** may be used to represent the value of the expression **`(Length*Breadth)*Height`**, calculated with the current values of the variables **`Length,`** **`Breadth`** and **`Height`** at the time **`Volume`** is referenced.

Hence the assignments

```
Circumference := 2.0 (REAL32) * 3.14159 (REAL32) * Radius
```

and

```
ThisVolume := (Length*Breadth)*Height
```

may be written more succinctly as

```
Circumference := TwoPi * Radius
```

and

```
ThisVolume := Volume
```

As noted above, because of the restriction to constant values, it is important that the value of any variable used in expression is not be changed by either assignment or input within the scope of abbreviation i. e. while the abbreviation is active. So, for example, **`Length, Breadth`** and **`Height`** must remain constant within the scope of **`Volume`**.

Example 2

```
VAL TerminalMask IS #FF << 8:
```

or

```
VAL Shift IS 8:
VAL Mask IS #FF:
VAL TerminalMask IS Mask << Shift:
```

shows the use of abbreviations to define a further abbreviation.

Other versions of the expression abbreviation allow the abbreviation of a table of expressions to array of values. Such a mechanism is convenient for data initialisation. The format is

```
VAL [size] type name IS expression:
```

or

```
VAL [ ] type name IS expression:
```

where the second alternative demonstrates that the actual specification of the array size is optional. Again, the specification of type may be omitted if it may be determined by the data type of the expression.

Example 3

```
VAL [10] INT Primes IS [2,3,5,7,11,13,17,19,23,29]:
SEQ Index = 0 FOR 10
  Chan ! Primes [Index}
```

depicts the use of a table to initialise an array. The type may be omitted as it may be determined from the table components, so the above abbreviation may be written as

```
VAL Primes IS [2,3,5,7,11,13,17,19,23,29]:
```

The first version is useful when it is required to emphasise the size of the array.

Example 4

```
VAL [5] [2] BYTE Vowels IS [['a','A'], ['e','E'], ['i','I'],
['o','O'], ['u','U']:
```

and

```
VAL [5] [2]  Vowels IS [['a','A'], ['e','E'], ['i','I'],
['o','O'], ['u','U']:
```

are equivalent initialisations of the two-dimensional array **`Vowels`**.

Example 5

```
VAL WhiteQueen IS ChessBoard [3] [0]:
```

```
VAL Punctuation IS ['.', ',', ':', ';', '?', '!']:

VAL SummerMonths IS {"Jun", "Jul", "Aug"]:

VAL [] INT ValidRange IS [Spectrum FROM Start FOR Length}:
```

are further examples of abbreviations using arrays - the last demonstrating the abbreviation of a segment. So it is possible to have statements of the form

```
NextPieceToMove := WhiteQueen
```

instead of

```
NextPieceToMove := ChessBoard [3] [0]
```

and

```
Acceptable := ValidRange
```

instead of

```
Acceptable := [Spectrum FROM Start FOR Length]
```

(where **Acceptable** is an array of the same type as **ValidRange**.

Example 6

```
VAL Vowels IS ['a', 'e', 'i', 'o', 'u', ]:
BYTE Char:
INT VowelCount:
SEQ
  VowelCount:= 0
  WHILE TRUE
    SEQ
      InChan ? Char
      IF
        IF Index = 0 FOR 5
          Char = Vowels[Index]
            VowelCount + 1
        TRUE
           SKIP
```

represents an occam fragment to count the number of vowels appearing in a piece of text, where the channel, **InChan** is declared to be type BYTE.

#### *4.7.1.2 Element abbreviations*

This form of abbreviations is used to give a new name to an element. (occam gives the generic title *element* to variables of the primitive data types, channel and timer types, and also to arrays-components, segments or whole arrays. ) The format is:

*type element* IS *element*:

where *type* is the type of the abbreviation, and which may be omitted if the type may be determined by the type of *element*. If *type* is present, the type of the element must match the *type*. This type of abbreviation is not limited to constant values. Any change in the value of the abbreviation is reflected by a change in the value of the element being abbreviated.

For example,

```
INT Rook IS Castle:

REAL32 Average IS Results [99]:

StatusLine IS Screen {20}:

INT Element IS Array [Subscript]:
```

For example, **`Results [99]`** would now be referenced by **`Average`**.

When the abbreviation is an alias for any array component, then certain rules must be observed within the scope of the abbreviation.

- any variable selecting i. e. indexing the array component must remain constant.

For example:

```
INT Middle:
[50] REAL32 List:

   -- initialise array

Middle := 25
Pivot IS List[Middle]: --specify Pivot as an abbreviation


  -- scope of Pivot
```

The value of **`Middle`** may not be changed by input or assignment within the scope of definition of **`Pivot`**.

- no reference must be made to *any* component of the array, except via abbreviations.

For example:

```
INT Middle:
[50] REAL32 List:

   -- initialise array

Middle := 25
Pivot IS List[Middle]:
Initial IS List[0]:

  -- process array with abbreviations Pivot and Initial
```

The array component **`List[Middle]`** must not be referenced as it was within the scope of **`Pivot`** - any reference to **`List[Middle]`** is only valid via a reference to **`Pivot`**. Moreover, references to other components of **`List`**, for example, **`List[0]`**, are illegal within the scope of **`Pivot`**. Other components of **`List`** may only be referenced via other abbreviations for these components; for example, declaring the abbreviation **`Initial`** for **`List[0]`** and referencing **`List[0]`** via this abbreviation.

Like the previous category of abbreviation, an array version exists for this category.

The format is:

*[size] type name* IS *element*:

or

*[ ] type name* IS *element*:

Example 7

```
[30] INT WholeRange:
MiddleRange IS [WholeRange FROM 10 FOR 10]:
```

declares that **`MiddleRange`** is an alias for a segment of the array, **`WholeRange`**, and may be referenced as an array. So, for example, **`MiddleRange[4]`** refers to **`WholeRange[4]`**.

#### *4.7.1.3 Abbreviations for efficiency*

One important use of abbreviations is the production of efficient code - both in terms of memory space and execution time - for handling large arrays.

For example

```
VAL Start IS 1000:
[5000] INT Vector:
SEQ
  Vector[Start+Offset1} := Value
  Vector[Start+Offset2} := Value
```

(where **`Offset2`**, **`Offset2`** etc. have been declared as type INT constants), it is better to write

```
VAL Lenght IS 3000:
VAL Start IS 1000:
[5000] INT Vector:
Array IS [Vector FROM Start FOR Length]
SEQ
  Vector[Start+Offset1} := Value
  Vector[Start+Offset2} := Value
```

where the large array **`Vector`** has been abbreviated by the segment **`Array`**, .

Since the large array has been abbreviated by a segment, and the segment is indexed by a constant, the occam compiler has no need to generate run-time range-checking code. All the checking may be performed at compile-time. This leads to efficiency in saving memory space and execution time.

### 4.7.2 Retyping

Retyping allows a given constant or variable of one data type to be expressed as a different data type, essentially mapping the given bit pattern to a named constant or variable of the different type. Retyping differs from the previously discussed data conversion in that retyping (as the name suggests) only changes the *type* of the given constant or variable and does not alter the bit pattern to produce an equivalent value of a different type. For example, the data conversion from an integer value to a real value involves the change from two's complement form to the IEEE floating point form. Retyping, moreover, is a specification and not an operation, as the data conversion is.

The format of retyping declaration is :

VAL *type name* RETYPE *expression*:

where *name* is the occam identifier of the constant (*expression*) or variable (*element*) being retyped.

The retyped constant or variable is governed by the usual scoping rules. Within the scope of the retyping, the name of the constant or variable being retyped may not be used.

Example 8

```
INT32 PackedNumber:
[4] BYTE SmallNumber RETYPES PackedNumber:
```

retypes the integer **`PackedNumber`** as a byte array **`SmallNumber`**. Individual bytes of the integer may then be referenced via the array. The size of the array must be correspond to the word-size of the integer being retyped.

### 4.7.3 Procedures

Procedure is a means of giving a name to an occam process, and, as such, leads to more compact, transparent and structured programs. Instead of the statements of the same process being repeated many times within the program, the process may be defined once and then referenced many times by referring to its name.

One possible drawback of an occam procedure is that recursion is not supported. Memory allocation in occam is static - fixed at compile-time, and static allocation does not allow recursion.

An occam procedure is another example of a specification statement. Constant and variable declarations were earlier examples of specification statements.

The procedure has the format:

PROC *name* ()
  *procedure body*
:

i. e. an occam procedure definition consists of

- a *procedure heading* comprising the keyword PROC, *name*, the procedure identifier and a pair of matching brackets,
- followed by the *procedure body*. The procedure body must be indented by two spaces and may consist of a primitive or more complex process. Like any other process, this process may also contain local declarations of constants, variables, etc. required by the procedure.

The procedure, like other specifications is terminated by a colon. However, the terminating colon of a procedure must appear on a line by itself, directly aligned with the P of PROC. The procedure is bound to the following process in the same way as a constant or variable declaration. It is governed by similar scoping rules i. e. it is before any process which references them. Procedures may be nested within other procedures.

Constants and variables, specified before the specification of a procedure but being still in scope when the procedure is in scope, are accessible from the procedure.

Example 9

```
PROC WriteErrorMessage
  SKIP
```

specifies a trivial procedure named **`WriteErrorMessage`**. The body of this procedure solely comprises the primitive SKIP process. Thus, essentially this procedure behaves as a procedure stub - representing not yet written, procedures represented by procedure stubs, to be filled in later as the development processes.

Example 10

```
PROC Delay ()
  VAL Limit IS 5000:
  INT Count:
  SEQ
    Count := 0
```

```
    WHILE Count < Limit
      Count:=Count+1
```

depicts a procedure which produces a delay by (rather wastefully) looping until an incremented variable reaches a certain value. (A more efficient method of producing a delay is to use and occam timer.)

Procedures are only executed when invoked from another process. A procedure is invoked from another process by referring to its name - this is called an instance of the procedure.

The format is

name ()

where *name* is the procedure identifier of the procedure being invoked. (The procedure brackets must be present.)

Example 11

```
PROC Delay ()                 -- |
  VAL Limit IS 5000:          -- |
  INT Count:                  -- |
  SEQ                         -- |procedure specification
    Count := 0                -- |
    WHILE Count < Limit       -- |
      Count:=Count+1          -- |


SEQ
  Delay ()                    -- instance procedure
```

shows an instance that procedure *Delay* is a following process.

Unlike procedures in some other programming languages with a branch and return mechanism, an instance of an occam procedure produces an in-line expansion of the procedure body. It behaves as a macro expansion.

For example, using the *Delay* procedure once more.

```
.
.                             -- procedure specification

SEQ                           -- |
  VAL Limit IS 5000:          -- |
  INT Count:                  -- | instance of Delay
  SEQ                         -- | procedure expanded
    Count := 0                -- | in-line
    WHILE Count < Limit       -- |
      Count:=Count+1          -- |
```

### 4.7.4 Parameters

Procedures may have parameters which allow the effect of the procedure to be applied to different values of variables, if required, each time an instance of a procedure occurs.

PROC *name (parameter 1, .... parameter n)*
*procedure body*
:

where *parameter 1, .... parameter n* are the formal parameters of the procedure, each separated by a comma if there is more than one parameters. The parameters may be of any occam type. This type must be completely specified within the procedure heading. Parameters of the same type may be grouped together with a single specifier of the type.

Parameters may be one or two kinds

- constant, or
- variable

If the value of a parameter remains constant i. e. unchanged within the procedure, then the type specification of the parameter in the procedure heading should be preceded by the reserved word VAL. If the value of a parameter may be changed i. e. it is a variable within the procedure, then VAL should be omitted from the parameter specification.

For example

```
PROC SkipSpaces (VAL INT Number)
```

indicates the value of the parameter. Number, remains constant within the procedure (an instance of this procedure would use an evaluated expression as a parameter) whilst

```
PROC Exchange (INT Item1, Item2)
```

indicates the values of the parameters, **Item1** and **Item2**, may be changed within the procedure (an instance of this procedure would use named variables and parameters), and

```
PROC MaxMin (VAL INT Item1, Item2, INT Max, Min)
```

indicates the values of the first two parameters, **Item1** and **Item2** remains constant, whilst the values of the second two, **Max** and **Min** are variable and may be modified within the procedure.

Example 12

```
PROC SkipSpaces (VAL INT Number)
  VAL Space IS ' ':
  BYTE Char:
  INT Skip:
  SEQ
    Skip := Number
    WHILE Skip := Number
    WHILE Skip > 0
      SEQ
        Chan ? Char
        IF
          Char = Space
            Skip := Skip - 1
          TRUE
             SKIP
```

is a procedure to skip given number of spaces, specified as the parameter value **Number**. The channel **Chan** is assumed to declared with type BYTE. It would be preferable to pass this channel name to the procedure as a channel parameter (see example 18)

Example 13

```
PROC Exchange (INT Item1, Item2)
  SEQ
    Item1, Item2 := Item2, Item1
```

represents the procedure to swap the values of its two parameters.

Example 14

```
PROC Circle (VAL REAL32 Radius REAL32 Area, Circumference)
  VAL Pi IS 3.14159 (REAL32):
  SEQ
```

```
      Area := Pi * (Radius * Radius)
      Circumference := 2.0 (REAL32) * (Pi * Radius)
```

is a procedure to calculate the area and circumference of a circle, given to radius.

Example 15

```
PROC RunningAverage (VAL REAL32 Data, REAL32 Average, INT Count)
SEQ
  Average := ((Average * (ROUND Count)) + Data) / ROUND (Count + 1))
  Count := Count + 1
```

is a procedure to calculate the running average of a list of numbers, given one number at a time.

In an instance of procedure within a process, actual parameters replace the formal parameters. Actual parameters must agree in type and number with the formal ones. The format of a procedure instance with parameters is just an extension of a simple format

*name (actual1 ... actualn)*

For example
Exchange (Int1, Int2)

In execution the formal parameter behaves as an abbreviation for the value of the actual parameter; a VAL type parameter behaving as an expression abbreviation and a non-VAL type behaving as an element abbreviation.

For example,

SkipSpaces (N)

is equivalent to

```
VAL Number IS N:            --|
VAL BYTE Space IS ' ':      --|
  BYTE Char:                --|
  INT Skip:                 --| in-line expansion
  SEQ                       --| of SkipSpaces (N)
  .                         --|
  .                         --|
  .                         --|
```

Any change in the value of a non-VAL formal parameter in the procedure body produces a corresponding change in the value of the actual parameter when used in an instance of the procedure.
For example

```
PROC Circle (VAL REAL32 Radius REAL32 Area, Circumference)
  VAL Pi IS 3.14159 (REAL32):
  SEQ
    Area := Pi * (Radius * Radius)
    Circumference := 2.0 (REAL32) * (Pi * Radius)
    .
    .
SEQ
  Circle (R, A, C)          --| instance of procedure
```

is equivalent to

```
 .
```

```
  -- procedure specification
.
SEQ                              --|
  VAL Radius IS R:               --| in-line
  Area IS A:                     --| expansion
  Circumference IS C:            --| of procedure
  VAL Pi IS 3.14159 (REAL32):--| instance
  SEQ                            --|
```

The actual parameters R, A and C are abbreviated to formal parameters **Radius**, **Area** and **Circumference** within the procedure will cause corresponding changes to the values of A and C.

In occam, when a variable passed as an actual parameter to a procedure, it is the variable replaces the formal parameter throughout the procedure. Anything done to the formal parameter, is done to the variable, which may therefore have its value changed.

Example:

```
PROC decrement (INT number)
  number := number -1
:
```

If we use **decrement (x)** as an instance of the procedure, the value of x will be reduced by one when **decrement (x)** terminates.

Pay attention, the behaviour differs from that found in certain other widely used languages. The commonly used call-by-value convention (available in C and Pascal) has the effect of evaluating the variable (actual) parameter, and using the result as the initial value of the formal parameter, which behaves as a local variable of the procedure body. Consequently, as assignment to the formal parameter has no effect on the actual parameter. The occam convention is more nearly equivalent to Pascal's call-by-reference (or VAR) parameters. This point is emphasised because it may trip up programmers who are experienced in these other languages.

Sometimes it is preferable that a procedure should not alter the value of a variable passed to it as a parameter.

Example:

```
PROC delay (VAL INT interval)
  INT n:
  SEQ
    n:= interval
    WHILE n> 0
      n:= n-1
```

decant the value of the parameter into a local variable (n in this example) and do any manipulations on this local value. Use this method if you need to translate Pascal procedures or similar with value parameters, into occam.

If only the original value of a variable is needed in a procedure, i. e. if the formal parameter is never altered by assignment of input, then a more efficient program may result if we explicitly say that only the value is to be passed , using VAL.

Parameters may be any occam-type. In particular, this includes channels and arrays. In specification of the procedure heading, occam does not require the size of any formal array parameter to be declared. Instead, an empty array dimension may be supplied. Used in conjunction with the SIZE operator, this shows that quite general array handling procedures can be written, without having to be specific about the size of arrays catered for.

Example 16

```
PROC Initialise ([] INT Buffer)
  SEQ Index = 0 FOR SIZE Buffer
    Buffer [Index] := 0
```

is a procedure which may be called with any size of array as parameter; the use of the SIZE operator within the procedure body catering for any size of array.

Example 17

```
PROC Transpose (VAL [] [] REAL32 Matrix, {} {} REAL32 TranMatrix)
  SEQ Row = 0 FOR SIZE Matrix
    SEQ Column = 0 FOR SIZE Matrix
      TranMatrix [Row, Column] := Matrix [Column, Row]
```

represents a procedure to calculate the transpose of a two-dimensional *square* matrix.

Example 18

```
PROC Buffer (CHAN OF BYTE InChan, OutChan)
  WHILE TRUE
    BYTE Char
    SEQ
      InChan ? Char
      OutChan ! Char
```

is a simple buffer process written as procedure. The channels used by the buffer are passed as parameters. The ability to pass channel names as parameters is an important feature of occam and is frequently used in programs.

### 4.7.5 Functions

In addition to procedures the occam language definition also includes functions which are another form of a process. In common with functions in other programming languages, the occam function returns a value as a result of some computation within the function. The simplest format is

*type* FUNCTION *name (parameters)*
  *declarations*
  VALOF
    *function body*
    RESULT *expression*
*:*

where

- *type* is one of the primitive types. The function returns a value of this type.
- *name* is the function identifier,
- *parameters* are the optional parameters, separated by commas. The kind of any parameter used in function must be VAL, and
- *function* body is an occam process which effects the computation of the function, and may be a primitive process or a more complex one. The result of the computation is returned via the value of *expression* (which is composed of any combination of parameters, constants, variables, etc. specified in *declarations* and literals). The expression must result in a value which has the same data type as *type*. The function body may contain further local declarations of constants, variables etc. required by the function.

The occam reserved word, VALOF, must be indented by two spaces with respect to the first letter of the specification, and the function body and the reserved word, RESULT, indented a further two spaces. Like the procedure, a function is terminated by a colon. The colon must appear on a line by itself directly underneath the first letter of the type specification.

Example 19

```
INT FUNCTION Maximum (VAL INT X, Y)
  INT Max:
    VALOF
      SEQ
        IF
          X>Y
            Max := X
          TRUE
            Max := Y
      RESULT
:
```

is a function which delivers the maximum of two integers.

Example 20

```
INT FUNCTION Factorial (VAL INT Number)
  INT Fact:
    VALOF
      SEQ
        Fact := 1
        SEQ Index = 1 FOR Number
          Fact := Index * Fact
      RESULT
:
```

depicts a function for calculating the factorial (n! = 1 * 2 * 3 * ...n) of given integer.

Example 21

```
REAL32 FUNCTION Average (VAL [] REAL32 List)
  REAL32 Sum:
    VALOF
      SEQ
        Sum = 0.0 (REAL32)
        SEQ Index = 0 FOR SIZE List
          Sum := Sum + List [Index]
       RESULT Sum / REAL32 ROUND (SIZE List)
```

represents a function which calculates the average for a list of reals.

Instance of a function behaves as an in-line expansion of the function body and any formal parameters behave as expression abbreviations of the actual parameters. Also, like a procedure, a function is referenced by referring to its name. However, unlike a procedure, because the function behaves like an expression, it is referenced via an assignment statement. Thus,

*variable := name (actual parameters)*

For example,

```
[100] REAL32 Data
SEQ
.
.                             -- input data into array
.
Mean := Average (Data)        -- reference function Average
```

A special and interesting feature of the occam function is that it is guaranteed not to produce any side-effects. To enable this feature, certain conditions must be observed in the use of the function

- the function body must not contain any parallel or alternation constructions
- the function body must not contain any inputs or outputs
- only variables declared within the function may be assigned to.

More sophisticated functions may be specified which return more than one value. The simple format of the function is expanded to cater for this situation.

*type1, ... type n* FUNCTION *name (parameters)*
 *declarations*
 VALOF
  *function body*
  RESULT *expression1, ... expression n*
*:*

where the list of types, separated by commas, must much in type and number the list or expression, likewise separated by commas.

For example,

```
REAL32, REAL32 FUNCTION Statistics (VAL REAL32 List)
```

would be the function heading for a function which returned the mean and standard deviation of a list of numbers.
Such a function must be referenced via a multiple assignment statement.
For example,

```
Mean, StdDev := Statistics (Data)
```

represents an instance of this multi-valued function.

### 4.7.6 Function definitions

A function definition provides a convenient notation for the specification of simple functions which are expressible as a single expression. Essentially, function definition have a null function body.

```
REAL32 FUNCTION InchesToCms (VAL REAL32 Inches) IS 2.54 (REAL32) * Inches

REAL32 FUNCTION Area (VAL REAL32 Length, Breadth) IS Length * Breadth

BOOL FUNCTION BufferFull (VAL INT Count) IS Count > 1024:

REAL32 FUNCTION Disc (VAL REAL32 A, B, C) IS (B*B) – (4.0 (REAL32) *
(A*C):

BOOL FUNCTION IsaDigit (VAL BYTE Char) IS (Char >= ‘0’) AND (Char <=
‘9’):
```

are examples of function definitions.
Like a simple function, a function definition may be generalised to deliver multiple values.

## 4.8 Configuration in Occam

The key idea is that the configuration does not affect the logical behavior of a program, it does enable the program to be aranged so that the performance requirements are met.

### 4.8.1 Processor

The actual processor in the network on which a set of processes are to be executed and the type of the processor are identified with the PROCESSOR statement.

PROCESSOR *nuber type*

where

- *number* is an integer value identifying the particular processor. This is just a logical numbering and is used to facilitate debugging, and
- *type* is the type of the processor - T2, T4 or T8 - and used to check that the program has been compiled for the correct processor:

| | | |
|---|---|---|
| T2 | - | 16-bit transputer T212 |
| T4 | - | 32-bit transputer T414 |
| T8 | - | 32-bit transputer T800 |

### 4.8.2 Allocations

Occam programs may be designed, written, tested and debugged on a single processor workstation, and then transferred to a network of parallel computers.

The final stage of such a development cycle is to allocate parallel processes in the program to different processors. This allocation is performed by replacing PAR with PLACED PAR in the appropriate parts of the program. PLACED PAR is followed by a placement, which consists of the number of a processor and a process to be run on it.

### 4.8.3 Hard channels

The four communication links of a transputer have fixed addresses in the transputer's memory space. These addresses are the first eight words of the internal memory. Conventionally, the addresses are mapped into occam identifiers using VAL abbreviation, as follows:

```
VAL link0out IS 0, link0in IS 4 :
VAL link1out IS 1, link1in IS 5 :
VAL link2out IS 2, link2in IS 6 :
VAL link3out IS 3, link3in IS 7 :
```

Each physical link will support two occam channels - one input channel and one output channel. The *hard* channels, which provide iner-processor communication or communication with external devices, must be mapped onto he transputer links. *Soft* channels, on the other hand, are implemented via memory locations since the processes using such channels for interprocess communication reside on the same transputer. Any link may be choosen to act as a channel to a communicating processor, provided it is of the correct type, namely input or output. A link on one transputer does not need to be connected to a correspondingly named link on another transputer. Thus, for example, **`link1out`** on one transputer may be connected to **`link1in`** on another transputer, but may equally well be connected to **`link0in, link2in, link3in`**. The corresponding physical links must be connected together between transputers as per the configuration.

Each collection of occam processes, which will execute on a single transputer in the network, must be grouped together to form an outher procedure with the additon of a procedure heading and terminator. The parameters for this outher procedure are necessary hard channels for communicating with the outer procedures on other transputers. It is an Inmos characteristic, that each outher procedure must be separately compiled.

Each hard channel between two processes on different transputers must be specified in two separate PLACE statements; once as an input channel of one transputer and once as an output channel for the other transputer. The order of declarations must follow the general format:

```
global declarations
```

```
PLACED PAR
  PROCESSOR
    local declarations
    procedure instance
```

### 4.8.4 Hardware protocols

Some hardware devices will require various protocols to be adhered to, that is certain special commands and non-data characters may need to be sent (and received) to control the devices, in addition to the actual data.

These are very hardware dependent and cannot be covered in any detail here.

### 4.8.5 Ports

In addition to hard channels, occam can address I/O ports as used in conventional computer systems. A port declaration has a data type, e. g.

```
PORT OF BYTE seriall:
```

and the allowed processes are input and output only:

```
serial1 ! 'a'
serial2 ? x
```

This allows ports to be used like channels, rather than like variables, which better correspond to the occcam style of using channels for all communication and variables for storage.

Ports behave like occam channels in that only one process may input from a port, and in only one process may output to a port. Thus ports provide a secure method of accessing external memory mapped status registers etc.

Note that there is no synchronisation mechanism assotiated with port input and output. Any timing constraints which result from the use of asynchronous external hardware will have to be programmed explicitly. For example, a value read by a port input may depend the time at which the input was executed, and inputting at an invalid time would unusable data.

During applications development it is recomended to model the peripherial by an occam process connected via channel.