THE PROGRAMMING LANGUAGE

OCCAM

OVERVIEW OF LECTURES ABOUT THE PROGRAMMING LANGUAGE OCCAM

I. Basics

- Processes and channels
- Names
- Types
  - Scope
  - Type conversions
- Operators
  - Arithmetic in occam
  - Relational operators
  - Boolean operators
  - Bitwise operators

II. Primitive processes

- Assignment process
- Input process
- Output process
- SKIP
- STOP

III. Non-primitive processes

- SEQ construction
- PAR construction
- Conditional processes
- Selection processes
- Repetitive processes
- Alternative processes
- Communicating processes
- Replicators

IV. Subroutines

- Abbreviations
  - Expression abbreviations
  - Element abbreviations
- Abbreviations for efficiency
- Retyping
- Procedures
- Parameters
- Functions
- Function definitions

V. Configuration

- Processors
- Allocations
- Hard channels
- Hardware protocols
- Ports

I. BASICS OF OCCAM

• Processes and channels

A process **starts; performs a number of actions and then finishes.**
A channel **is a one-way, point-to-point link from one process to one other process.**

• 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 there is one, can be made up of letters, digit and the dot character. Upper and lower case are distinguished by occam.

• Types

Occam requires that every object that is 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 can be used in a process. Noticed 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 named objects of an occam process, such as constants, variables and channels must be declared in advance of any process in which they are used. The scope or range **of definition of named objects is the range of definition of the process which immediately follows they declaration. The range of definition of a process is determined by its indentation, and so the scope of named objects is defined by the indentation of the following process. The named objects are said to be local to that process and any component processes.**
Sometimes it's convenient to convert one type **to another in a program; it may for instance save having to declare several extra variables for a value that is only required once. Type conversion should be prevent values being used in inappropriate situations.**

• Operators

There are arithmetic, relational, Boolean and bitwise operators.

PROCESSES AND CHANNELS

• Processes

– Conventional programming languages

Much of the activity of a program consists 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
```

– Occam

It permits this sort of communication as a normal feature of programming. More importantly, occam doesn't mind weather the two programs which so communicate are running on different computers or are just two processes running concurrently on the same computer.

• Channels

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

A ! 2 ______ A ? B
process 1   process 2

where A is a channel and B is a variable. This reads as 'output 2 to A' and 'input from A to B'.

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 there is one, can be made up of letters, digit and the dot character. Upper and lower case 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
```

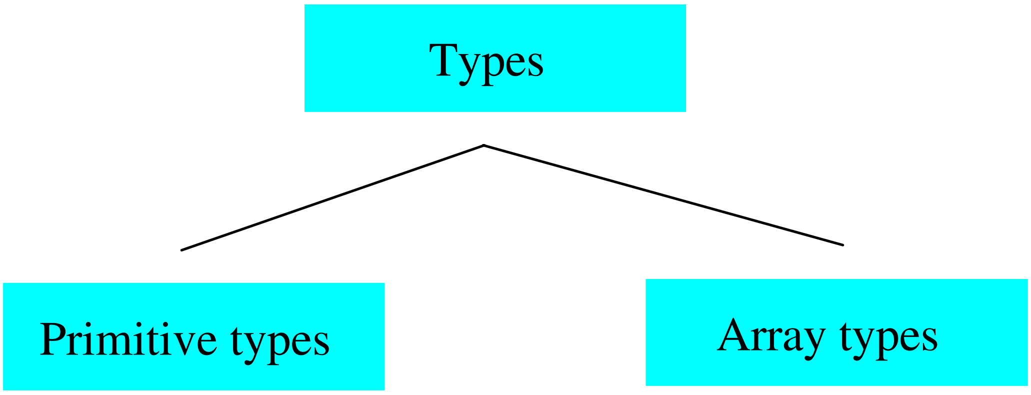
Occam keywords such as SEQ, PAR and CHAN are always in upper case and they are reserved. In other words they cannot be used as names that you create.

These are not valid names:

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

TYPES I.

Occam, like Pascal and many other languages, but unlike BASIC, requires that every object that is 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 can be used in a process. Noticed the colon, which is used to end all the different kinds of specification. This colon joins a specification to the process which follows it.



channel type	**channel type**
timer type	**timer type**
data type	**data type**
integer type	**integer type**
byte type	**byte type**
real type	**real type**
Boolean type	**Boolean type**

TYPES II.

- **CHAN OF protocol**

  A communication channel is of type CHAN OF protocol. Each communication channel enables values to be communicated between two concurrent processes according to the specifies 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**

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

```
TIMER clock:
```

TYPES III.

- **Data types**

INT

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

BOOL

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

```
BOOL more:
```

TYPES IV.

• Array

Array types are constructed from component types. An array type is 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 are the same. 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. But 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
```

TYPES V.

- **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 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)
```

TYPES VI.

- **Scope**

The named objects of an occam process, such as constants, variables and channels must be declared in advance of any process in which they are used. The scope or range of definition of named objects is the range of definition of the process which immediately follows they declaration. The range of definition of a process is determined by its indentation, and so the scope of named objects is defined by the indentation of the following process. The named objects are said to be 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.

TYPES VII.

- **Type conversions**

Sometimes it's convenient to convert one type to another in a program; it may for instance save having 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 only legal 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.

OPERATORS I.

- **Arithmetic in occam**

The basic arithmetic operations are these:

```
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 numbers of type INT or REAL.

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
```

OPERATORS II.

- **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 may like to think of 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
```

OPERATORS III.

- **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.

OPERATORS IV.

- **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 itself may be 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.

OPERATORS V.

- **Shift operators**

The bit comprising the value of an integer constant, variable or expression may be shifted left or right 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 bit positions to be shifted is given by the second operand. This operand, which itself may be a constant, variable or an expression, must be of type INT. The number of bits to be shifted must be not be greater than the number of 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.

II. PRIMITIVE PROCESSES

• Assignment process

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

```
fred := 2
```

makes the value in variable `fred` two.

• 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`.

• 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`.

• SKIP

The special process SKIP may be thought of as representing a process which does nothing. It might be used in a partly complemented program in place of process which will be written later, but which for the moment can be allowed to do nothing.

• STOP

The special process STOP may be thought of as representing a process which doesn't work, or is 'broken'. It might be used, like SKIP, to stand in for a process which has yet to be written.

III. NON-PRIMITIVE PROCESSES

Several primitive processes can be combined into a larger process by specifying that they should 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.

SEQ CONSTRUCTION

he 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
```

This says, 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.

PAR CONSTRUCTION

The PAR construction says: do the following processes all at the same time', I. e. parallel. All the component processes of a PAR start to execute 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.

CONDITIONAL PROCESSES

In occam the conditional choice is provided by the construction called `IF`. It 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.)

SELECTION PROCESSES

The construction called selection in occam provides an efficient means of selecting one of a number 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 combine them into an 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 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.

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.

REPETITIVE PROCESSES

All programming languages provide some means of looping, i. e. performing an action repeatedly. In general it's 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 result 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 repeated. This 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.

ALTERNATIVE PROCESSES

A process may use several channels for input, or several for output, or both, in order to communicate with one other process. So we can make choices according to the state of channels. This 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 are 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 were 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.

COMMUNICATING PROCESSES

Communication between two processes is the essence of occam programming. At its simplest 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 using channels, occam doesn't allows us to pass values between parallel processes using shared variable.

REPLICATORS I.

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).

REPLICATORS II.

- **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 efect 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`s no need to specify and increment and index variable and test its value each time round.

REPLICATORS III.

• 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.

As an example let`s look first a simple queue. 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]`.

IV. SUBROUTINES

- **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.

- **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.

- **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.

IV. SUBROUTINES (CONT.)

- **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.

- **Functions**

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

- **Function definitions**

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

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.**

- **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 an optional and may be omitted if the type my 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 type, if type 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:

```
VAL Return is '*c':
VAL AbsolutZero IS –273;
VAL TwoPi IS 2.0 (REAL32) * 3.14159 (REAL32):
VAL BufferFull IS COUNT > Limit:
```

- **Element abbreviations**

  This form of abbreviations 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 from 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]:
```

  So for example, `Results [99]` would now be referenced by `Average`.

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, rather than write

```
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.

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:

```
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 such so as to correspond to the word-size of the integer being retyped.

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.

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 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.

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 parameter. 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).

FUNCTIONS

In addition to procedures the occam language definition also includes functions which are another form of 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.

FUNCTION DEFINITION

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.

V. CONFIGURATION

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.

- Processor

  The actual processor in the network on which a set of processes are to be execute.

- 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 consist of the number of a processor and a process to be run on it.

- Hard channels

  Each physical link will support two occam channels - one input channel and one output channel.

V. CONFIGURATION (CONT.)

- **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.

- **Ports**

  In addition to hard channels, occam can address I/O ports as used in conventional computer systems.

PROCESSOR

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

```
PROCESSOR number 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**

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 consist of the number of a processor and a process to be run on it.

HARD CHANNELS

The four communication links of a transputer have fixed adresses 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, muts be mapped onto he transputer links. *Soft* channels, on the other hand, are implemented via memory locations since the processes using the 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 correct type, namely input or output. A link on one transputer need not 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.

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.

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 serial1:
```

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 is more in keeping with 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 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 noo 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 inut may depend uppon the time at which the input was executed, and inputting at an invalid time would produuce unusable data.

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