# OCCAM 2.0

## 1 Introduction

A process starts, performs a number of actions, and then either stops or terminates. Each action may be an assignment, an input or an output. An assignment changes the value of a variable, an input receives a value from a channel, and an output sends a value to a channel.

At any time between its start and termination, a process may be ready to communicate on one or more of its channels. Each channel provides a one way connection between two concurrent processes; one of the processes may only output to the channel, and the other may only input from it.

Communication is synchronised. If a channel is used for input in one process, and output in another, communication takes place when both processes are ready. The inputting and outputting processes then proceed, and the value to be output is copied from the outputting process to the inputting process.

A process may be ready to communicate on any one of a number of channels. Communication takes place when another process is ready to communicate on one of the channels.

## 2 Notation

The following examples illustrate the notation used in the description of occam.

The meaning of

<assignment> ::= <variable> := <expression>

is "An <assignment> is a <variable> followed by := followed by an <expression>".

The meaning of

<action> ::= <assignment> | <input> | <output>

is "An <action> is an <assignment> or an <input> or an <output>". This may also be written:

<action> ::= <assignment>
<action> ::= <input>
<action> ::= <output>

The notation { <process> } means "a list of zero or more processes on separate lines" and { ,<expression> } means "a list of zero or more expressions separated from each other by ,".

3 Process

<process> ::= **SKIP** | **STOP** |
<action> |
<construction> |
<block> |
<instance>

<action> ::= <assignment> | <input> | <output>

<construction> ::= <sequence> | <conditional> | <loop> |
<parallel> | <alternative>

**STOP** starts but never proceeds, and never terminates.

**SKIP** starts, performs no action, and terminates.

<assignment> ::= <variable> := <expression>

An assignment evaluates the expression and assigns the result to the variable, provided that the type of the variable is that of the expression. Otherwise the assignment is invalid. All other variables are unchanged in value.

<input> ::= <channel> ? <variable>

An input inputs a value from the channel, assigns it to the variable and then terminates. All other variables are unchanged in value.

<output> ::= <channel> ! <expression>

An output evaluates the expression, outputs the result to the channel and then terminates.

```
<sequence>          ::=   SEQ
                            { <process> }
```

A sequence starts with the start of the first process. Each subsequent process starts if and when its predecessor terminates. The sequence terminates on termination of the last process. A sequence with no component processes behaves like **SKIP**.

```
<conditional>       ::=   IF
                            { <choice> }

<choice>            ::=   <guarded choice> | <conditional>

<guarded choice>    ::=   <boolean>
                            <process>

<boolean>           ::=   <expression>
```

The value of a boolean expression is either true or false. A guarded choice behaves like **STOP** if its boolean is initially false. Otherwise it behaves like **SKIP** and the process, in sequence.

The choices are tested in sequence. The conditional behaves like the first of the choices which can proceed, or like **STOP** if none of them can proceed. A conditional with no component choices behaves like **STOP**.

```
<loop>              ::=   WHILE <boolean>
                            <process>
```

A loop is defined by

```
WHILE b         =   IF
  P                   b
                      SEQ
                        P
                        WHILE e
                          P
                    TRUE
                      SKIP
```

<parallel> ::= **PAR**
{ <process> }

All processes of a parallel start simultaneously, and proceed together. The parallel terminates when all of the processes have terminated. A parallel process is ready to communicate on a channel if any of its components is ready. A parallel with no component processes behaves like **SKIP**.

If a channel is used for input in one process, and output in another, communication takes place when both processes are ready. The inputting and outputting processes then proceed, and the value of the expression specified in the output is assigned to the variable specified in the input.

No variable changed by assignment in any of the component processes of a parallel may be used in any other component, no channel may be used for input in more than one component process, and no channel may be used for output in more than one component process. A parallel is invalid unless these non-interference conditions are satisfied.

<alternative> ::= **ALT**
{ <option> }

<option> ::= <wait option> | <alternative>

<wait option> ::= <wait>
<process>

<wait> ::= <input> |
<boolean> **&** <input> |
<boolean> **& SKIP**

A wait behaves like **STOP** if its boolean is initially false, and like the input or **SKIP** otherwise. A wait option behaves like the wait and the process, in sequence.

An alternative behaves like any one of the options which can proceed, and can proceed if any of the options can proceed. An alternative with no component options behaves like **STOP**.

4 Replicator

```
<sequence>      ::=  SEQ <replicator>
                       <process>

<parallel>      ::=  PAR <replicator>
                       <process>

<conditional>   ::=  IF <replicator>
                       <choice>

<alternative>   ::=  ALT <replicator>
                       <option>

<replicator>    ::=  <name> = <base> FOR <count>

<base>          ::=  <expression>

<count>         ::=  <expression>
```

Let n be a name, and B and C be expressions of type **INT** with values b and c. Let **X** be one of **SEQ**, **PAR**, **ALT** or **IF** and let Y(n) be a corresponding process, choice or option (in which n may be used as a value of type **INT**) according to the syntax above. Then

```
SEQ  n = B FOR 0          =   SKIP
  Y(n)

IF   n = B FOR 0          =   STOP
  Y(n)

PAR  n = B FOR 0          =   SKIP
  Y(n)

ALT  n = B FOR 0          =   STOP
  Y(n)
```

If c > 0

```
X    n = B FOR C          =   X
  Y(n)                          Y(b)
                                Y(b+1)
                                ...
                                Y(b+c-1)
```

If c < 0

```
X    n = B FOR C              is invalid
  Y(n)
```

## 5 Types

<type> ::= <primitive type> |
<array type> |
<procedure type>

<primitive type> ::= **CHAN** |
**TIMER** |
**BOOL** |
**BYTE** |
**INT** |
<extra type>

A communication channel is of type **CHAN**. Each communication channel provides communication between two concurrent processes. A timer is of type **TIMER**. Each timer provides a clock which can be used by any number of concurrent processes. All other primitive types are data types.

Every variable, expression and value has a data type, which defines the length and interpretation of values of the type.

The values of type **BOOL** are true and false.

The values of type **BYTE** are nonnegative numbers less than 256.

A value of any integer type is interpreted as a signed integer n in the range

$$-(N/2) <= n < (N/2)$$

where N is the number of different values which may be represented by variables of the integer type.

**INT** is the type of signed integer values most efficiently provided by the implementation.

An implementation of occam must provide the primitive types types **CHAN**, **TIMER**, **INT**, **BYTE** and **BOOL**. It may also provide extra types as extensions. However, an implementation which provides extensions must also provide the user with the ability to disable them.

<extra type> ::= **INT16** |
**INT32** |
**INT64** |
**REAL32** |
**REAL64**

A signed real value is of type **REAL32** or **REAL64**, and is represented according to IEEE standard P754 draft 10.0. A value v of type **REAL32** is represented using a sign bit s, an 8 bit exponent e and a 23 bit fraction f. The value v is positive if s=0, negative if s=1; its magnitude is:

```
(2 ** (e-127)) * 1.f        if 0<e and e<255
(2 ** -126) * 0.f           if e=0 and f<>0
0                           if e=0 and f=0
```

Similarly, a value v of type **REAL64** is represented using a sign bit s, an 11 bit exponent e and a 52 bit fraction f. The value v is positive if s=0, negative if s=1; its magnitude is:

```
(2 ** (e-1023)) * 1.f       if 0<e and e<2047
(2 ** -1022) * 0.f          if e=0 and f<>0
0                           if e=0 and f=0
```

<array type> ::= [ <expression> ] <type>

Array types are constructed from component types. An array type is a channel type, timer type or data type, depending on the type 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 components in an array of the array type, and T defines the type of the components. A component of an array can be selected by a nonnegative value less than the size of the array.

Procedure types are described below together with procedure definitions.

## 6 Scope

<block> ::= <specification>:
<scope>

<specification> ::= <declaration> |
<abbreviation>
<definition>

<scope> ::= <process>

A block behaves like its scope; the specification specifies a name, which may be used with this specification only within its scope.

Let x and y be names, and let P(x) and P(y) be processes which are similar except that P(x) contains x wherever P(y) contains y, and vice versa. Let S(x) and S(y) be specifications which are similar except that S(x) is a specification of x and S(y) is a specification of y. Then

S(x): = S(y):
P(x) P(y)

Using this rule it is possible to express a process in a canonical form in which no name is specified more than once.

<declaration> ::= <type> <name>

A declaration T x declares x as a new channel, variable, timer or array of type T.

<abbreviation> ::= <specifier> <name> **IS** <element> |
<specifier> <name> **IS** <operation> |
**VAL** <specifier> <name> **IS** <expression>

An abbreviation S n **IS** <element> specifies n as an abbreviation for a variable, channel, timer, procedure, array or value. Any subscript within the element takes the value it has at the start of the block.

An abbreviation S n **IS** <operation> specifies n as an abbreviation for the value taken by the operation at the start of the block.

An abbreviation **VAL** S n **IS** <expression> specifies n as an abbreviation for the value taken by the expression at the start of the block. The abbreviation is valid only if the expression is of primitive type.

<specifier> ::= <type> |
[] <specifier>

The type of the element, operation or expression in an abbreviation must be compatible with the specifier. A type T is compatible with a specifier T. A type [n]T is compatible with a specifier []T.

Let x be an element, operation or expression and S a compatible specifier. Then **VAL** n **IS** x is a syntactic abbreviation for **VAL** S n **IS** x, and n **IS** x is a syntactic abbreviation for S n **IS** x.

## 7 Procedure

```
<definition>    ::=  PROC <name> ( {, <formal> } )
                       <body>

<formal>        ::=  <specifier> <name> |
                     VAL <specifier> <name>

<body>          ::=  <process>
```

The definition

```
PROC n ( { ,<formal> } )
  B
```

defines n as the name of a procedure.

```
<instance>      ::=  <name> ( { ,<actual> } )

<actual>        ::=  <element> |
                     <operation> |
                     <expression>
```

Let X be a program expressed in the canonical form in which no name is specified more than once. If X contains a procedure definition P ( F0, F1, ... , Fn ) with body B, then within the scope of P

```
P ( A0, A1, ... , An )    =  F0 IS A0 :
                             F1 IS A1 :
                             ...
                             Fn IS An :
                             B
```

provided that each definition Fi **IS** Ai is valid.

```
<procedure type>  ::=  PROC ( { , <specifier> } )
```

A procedure type specifies the formal parameters of a procedure of the procedure type. Procedure types are used only in abbreviations and definitions, and enable procedures to be used as procedure parameters.

A procedure can always be compiled either by substitution of its body as described above or as a closed subroutine.

## 8 Element

An element has a type, which may be a channel type, timer type or data type. An element of data type also has a value. Elements enable channels, timers, variables, values or arrays to be selected from arrays.

```
<element>       ::=  <element> [ <subscript> ] |
                     [ <element> FROM <subscript> FOR <subscript> ] |
                     <name> |
                     <literal> |
                     <constructor>

<subscript>     ::=  <expression>
```

Let v be of type [n]T, and e an expression of type **INT** and value s. Then v[e] is valid only if 0<=s and s<n; it is the component of v selected by s.

Let v be of type [n]T. Then [v **FROM** b **FOR** c] is valid only if c>0, b>=0 and (b+c)<n; it is an array of type [c]T with components v[b], v[b+1], ... v[(b+c)-1].

The type of an element consisting of a name is that of the name, and the type and value of an element consisting of a literal is that of the literal.

```
<constructor>   ::=  [ { ,<expression> } ]
```

The value of a constructor is an array in which the value and type of each component is the value and type of the corresponding expression in the constructor.

Literals and constructors may not be changed by assignment or input.

## 9 Variable, Channel, Timer and Expression

<variable> ::= <element>

Every variable has a value which may be changed by assignment or input. The value of a variable is the value most recently assigned to it, or is arbitrary if no value has been assigned to it.

Let v be a variable of type [n]T, and e an expression of type T. If 0<=s and s<n, then v[s] := e assigns to v a new value in which the value of the component selected by s is replaced by the value of e, and all other components are unchanged. Otherwise the assignment is invalid.

Let v be a variable of type [nT. **Let s be** [v **FROM** b **FOR** c] and e an expression of the same type as s, [c]T. The assignment s := e assigns to each component of s the corresponding component of e, provided that b>=0 and (b+c)<n. Otherwise s := e is invalid.

Let x be a channel, s be [v **FROM** b **FOR** c], e an expression. The combined effect of x?s and x!e is s := e.

<channel> ::= <element>

<timer> ::= <element>

A channel element used for input or output is invalid unless it is of type **CHAN**. A timer element used for timer input or delayed input is invalid unless it is of type **TIMER**.

<expression> ::= <element> |
<operation>

An expression has a data type and a value, which are those of the element or operation.

## 10 Literal

A literal has a data type and a value.

```
<literal>       ::=  <integer> |
                     <byte> |
                     <integer>_<type> |
                     <real>_<type> |
                     <string> |
                     TRUE | FALSE

<integer>       ::=  <digits> |
                     - <digits> |
                     # <digits>

<byte>          ::=  '<character>'

<real>          ::=  <digits>.<digits> |
                     <digits>.<digits>E<exponent>

<exponent>      ::=  +<digits> | -<digits>
```

An integer literal is a signed decimal number, or # followed by a hexadecimal number. A byte literal is an ASCII character enclosed in single quotation marks: '.

An integer literal is of type **INT**, and a byte literal is of type **BYTE**. Let x be an integer or byte literal, and T be **BYTE** or an integer type. A literal x_T is a value of type T and value x, provided that x can be exactly represented as a value of type T. Otherwise x_T is invalid.

Let T be a real type. A literal f_T is of type T and value f. A real fEe_T is of value f * (10 ** e).

A string is represented as a sequence of ASCII characters, enclosed by double quotation marks: ". Let s be a string of n characters. The value of s is an array of type [n]**BYTE**; the value of each component of the array is the value of the corresponding character of the string.

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

## 11 Operation

An operation has a data type and a value. Expressions are constructed from elements of data type, operators and parentheses.

<operation> ::= <monadic operator> <operand> |
<operand> <dyadic operator> <operand> |
<conversion>

<operand> ::= <element> | (<operation>)

The type and value of an operation enclosed in parentheses are those of the operation.

The arithmetic operators +, -, *, /, **REM** yield the arithmetic sum, difference, product, quotient and remainder, respectively. Both operands of an arithmetic operator must be of the same integer or real type, and the result is of the same type as the operands. The arithmetic operators treat integer operands as signed integer values, and produce signed integer results. The value of - x is T(0 - x), where T is the type of x.

Let m and n be integers. The result of m / n is rounded towards zero, being positive if both m and n are of the same sign, negative otherwise. The result of m **REM** n is the remainder of m / n, and its sign is the same as the sign of m. Regardless of the signs of m and n and provided that n is nonzero

m = ((n* (m/n)) + (m **REM** n))

The result of a real arithmetic expression e of type t is the value of e, rounded to the nearest value of type t, provided that the value of e differs from a value of type t by at most one half in the least significant bit position. If two values of type t are equally near, the one in which the least significant bit is zero is chosen. If x and y are real, the result r of x **REM** y is x - (y * n), where n is the integer result of x / y rounded to its nearest value; **REM** can therefore yield a negative value.

The operators +, -, *, /, **REM** are invalid if the result cannot be represented as a signed value of the same type as the operands.

Both operands of the modulo operators **PLUS**, **MINUS** and **TIMES** must be of the same integer type, and the result is of the same type as the operands. Let r be the number of different values which may be represented using the type of the expression. The modulo operators obey the following rules:

(m **PLUS** n) = (m + n) + (k * r)

where k is the unique integer for which

(m + n) + (k * r) >= -(r/2) and
(m + n) + (k * r) < (r/2)

Similarly:

(m **MINUS** n) = (m - n) + k * r
(m **TIMES** n) = (m * n) + k * r

The operator **AFTER** is defined by:

(m **AFTER** n) = (m **MINUS** n) > 0

The bitwise operators **BITAND**, **BITOR**, >< yield the bitwise and, or and exclusive or of their operands. Both operands must be of the same integer type, and the result is of the same type as the operands. Each bit of the result is produced from the corresponding bits of the operands according to the following rules:

b >< 0 = b
b **BITAND** 0 = 0
b **BITOR** 0 = b

0 >< 1 = 1
b **BITAND** 1 = b
b **BITOR** 1 = 1

1 >< 1 = 0

where b is 0 or 1.

The bitwise operator **BITNOT** yields the bitwise not of its operand, which must be of integer type. Each bit of the result is produced from the operand as follows:

**BITNOT** 1 = 0

**BITNOT** 0 = 1

In a shift expression n << c or n >> c, n and the result are of the same integer type, and c is of type **INT**. The shift operators yield results according to the following rules:

n << 1 = n **PLUS** n
n >> 1 = m, where m>=0 and ((m << 1) + (n **BITAND** 1)) = n

Let **O** be << or >>, and let b be the number of bits needed to represent a value of type n. Then

(n **O** 0) = n
if c < 0 or c > b
if c > 0

n **O** c is invalid
n **O** c = (n **O** 1) **O** (c-1)

The boolean operators **NOT**, **AND**, **OR** yield boolean results according to the following rules:

**NOT** false = true
false **AND** b = false
false **OR** b = b

**NOT** true = false
true **AND** b = b
true **OR** b = true

where b is a boolean value.

The relational operators =, <>, <, <=, >, >= yield a result of type **BOOL**. Both operators of a relational operator must be of the same primitive type. The operands of = and <> may be of any primitive type. The operands of <, <=, >, >= must be of the same integer or real type. The result of x = y is true if the value of x is equal to that of y. The result of x < y is true if the signed integer value of x is strictly less than that of y. The other operators obey the following rules:

(x <> y) = **NOT** (x = y)  (x >= y) = **NOT** (x < y)
(x > y) = (y < x)  (x <= y) = (y >= x)

where x and y are any values.

The operand of the monadic operator **SIZE** must be an array type. Let x be of type [n]T. The expression **SIZE** x is of type **INT**; its value is n.

<conversion> ::= <type> <operand> |
<type> **ROUND** <operand> |
<type> **TRUNC** <operand>

The type of a conversion T e, T **ROUND** e or T **TRUNC** e is T; its value is the value of e converted to a value of type T. Both T and the type of e must be primitive types.

Let T be any integer type, or **BYTE**. Then:

T **TRUE** = 1  **BOOL** 1 = **TRUE**
T **FALSE** = 0  **BOOL** 0 = **FALSE**

Let e be an expression of value v, and of integer or **BYTE** type. Let I be an integer type and N the number of values representable in type I. The value of I(e) is v, provided that -(N/2) <= v < (N/2). The value of **BYTE** e is v, provided that 0 <= v < 256. Let R be a real type. The value of R **ROUND** e is the value of type R nearest to v, and the the value of R **TRUNC** e is the value of type T nearest to and not larger in magnitude than v. R e is invalid.

Let e be an expression of value v, and of real type. Let R be a real type, and T an integer or real type. The result of the conversion R e is v, provided that v is exactly representable as a value of type R. The value of T **ROUND** e is the value of type T nearest to v, and the value of T **TRUNC** e is the value of type T nearest to and not larger in magnitude than v.

The conversion T **ROUND** e is invalid unless the value of e differs numerically from a value of type T by at most one half in the least significant bit position. If two values of type T are equally near to the value of e, the one in which the least significant bit is zero is chosen.

## 12 Timer input

<input> ::= <timer input> |
<delayed input>

<timer input> ::= <timer> ? <variable>

A timer input sets the variable to a value of type **INT** representing the time. The value is derived from a clock, which changes at regular intervals. The successive values of the clock are produced by:

clock := clock **PLUS** 1

<delayed input> ::= <timer> ? **AFTER** <expression>

A delayed input is unable to proceed until the value of the clock satisfies (clock **AFTER** e), where e is the value of the expression.

## 13 Character set

The occam characters are:

Alphabetic characters

**ABCDEFGHIJKLMNOPQRSTUVWXYZ**
**abcdefghijklmnopqrstuvwxyz**

Digits

**0123456789**

Special characters

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

The space character

Strings and character constants may contain any occam character (except *, ' and "). Certain characters are represented as follows:

| | |
|---|---|
| ***c** | carriage return |
| ***n** | newline |
| ***t** | horizontal tabulate |
| ***s** | space |
| ***'** | quotation mark |
| ***"** | double quotation mark |
| ****** | asterisk |

Any character can be represented by an asterisk followed by a two digit hexadecimal constant.

A name consists of a sequence of alphabetic characters, decimal digits and underscores (_), the first of which must be an alphabetic character.

An implementation may provide other characters for use in strings and character constants.

## 14 Notation

Let T be a type and N1, N2, ..., Nn names. Then

```
T N1:
T N2:
...                          =   T N1, N2, ... , Nn:
T Nn:                            P
P
```

Let S be a specifier and N1, N2, ..., Nn names. Then

```
S N1 IS X1:
S N2 IS X2:
...                          =   S N1 IS X1, N2 IS X2, ... , Nn IS Xn:
S Nn IS Xn:                      P
P

                             =   N1 IS X1, N2 IS X2, ... , Nn IS Xn:
                                 P
```

and

```
VAL S N1 IS X1:
VAL S N2 IS X2:
...                          =   VAL S N1 IS X1, N2 IS X2, ... , Nn IS Xn:
VAL S Nn IS Xn:                  P
P

                             =   VAL N1 IS X1, N2 IS X2, ... , Nn IS Xn:
                                 P
```

Let F be a <formal>. Then

```
F x1, F x2, ... , F xn       =   F x1, x2, ... , xn
```

Let c be a channel, e1, e2, ... en be expressions and v1, v2, ... vn be variables. Then

```
SEQ
  c!e1
  c!e2                       =   c!e1; e2; ... ; en
  ...
  c!en

SEQ
  c?v1
  c?v2                       =   c?v1; v2; ... ; vn
  ...
  c?vn

c?v1
  SEQ
    c?v2                     =   c?v1; v2; ... ; vn
    ...
    c?vn
```

and if **e** is an expression

```
e & c?v1
  SEQ
    c?v2                    =    e & c?v1; v2; ... ; vn
    ...
    c?vn
```

Let **O** be one of the operators **AND**, **OR**.

```
e1 O e2 O ... O en          =    (e1 O (e2 O ( ... O en ) ... ))
```

In an implementation which provides the characters \ and ~:

```
\                           =    REM
/\                          =    BITAND
\/                          =    BITOR
~                           =    BITNOT
```

## 15 Configuration

Configuration does not affect the logical behavior of a program. However, it does enable the program to be arranged to ensure that performance requirements are met.

```
<parallel>      ::=  PLACED PAR
                       { <placement> }   |
                     PLACED PAR <replicator>
                       <placement>

<placement>     ::=  PROCESSOR <expression>
                       <process>
```

Each placement is executed by a separate processor. The value of the expression in a placement is the number of the processor executing the component process. The variables and timers used in a placement must be declared within the placement.

```
<parallel>      ::=  PRI PAR
                       { <process> }   |
                     PRI PAR <replicator>
                       <process>
```

Each process is executed at a separate priority. The first process is the highest priority, the last the lowest. If P and Q are two concurrent processes with priorities p and q such that p < q, then Q is only allowed to proceed when P cannot proceed.

```
<alternative>   ::=  PRI ALT
                       { <option> }   |
                     PRI ALT <replicator>
                       <option>
```

If several options can proceed, the alternative behaves like the first in textual sequence.

```
<process>       ::=  <allocation> :
                     <process>

<allocation>    ::=  PLACE <name> AT <expression> |
                     PLACE <name> IN <element> |
                     WORKSPACE <element>
```

An allocation **PLACE** n **AT** e allocates the variable, channel, timer or array n to the physical address e.

Let s be an array of type [n]**INT**. An allocation **PLACE** p **IN** s, where p is the name of a procedure or an abbreviation of a constant, places the value of the constant or the compiled instructions for the procedure in the array s.

Let w be an array of type [n]**INT**. Then **WORKSPACE** w : P allocates w as the workspace for the process P.

## 16 Invalid processes

In a checked implementation invalid processes which are not detected by the compiler behave like **STOP**; in an unchecked implementation an invalid process which is not detected by the compiler may do anything.

Similarly, in a checked implementation a process containing an invalid expression operation behaves like **STOP**; in an unchecked implementation an inavlid expression has an arbitrary value.

## 17 External input and output

<type> ::= <type> **PORT**

<port> ::= <element>

A process may communicate with external devices which are connected to the processor's memory system. A port specification is similar to a variable specification, and the type used in a port specification must be a data type.

<input> ::= <port> ? <variable>

<output> ::= <port> ! <expression>

A port input inputs a value from the port, assigns it to the variable and then terminates. A port output evaluates the expression and outputs the result to the port. A program is invalid if any port is used for input or output in more than one component of a parallel.

## 18 Retyping

<definition> ::= <specifier> <name> **RETYPES** <element>

The representation of values and variables in an implementation is defined by the representation function REP(e), where e is an element.

Within the scope of the definition S n **RETYPES** e, either REP(n) = REP(e) or the definition is invalid. Any subscript within the element e takes the value it has at the start of the block.

The definition T n **RETYPES** e specifies n as an element of type T, and []T n **RETYPES** e specifies n as an element of type [x]T.

A procedure may be converted to a data value using a retyping conversion, provided that all identifiers used in the procedure are defined within the procedure.

The use of the retyping conversion will normally result in implementation dependant processes, as the definition of REP will vary from one implementation to another.

## TRANSPUTER IMPLEMENTATION

### 1 Representation and Retyping

The length and alignment of values of each primitive data type for a 16 bit transputer are

| T | length(T) | alignment(T) |
|---|---|---|
| **BOOL** | 1 | 1 |
| **BYTE** | 1 | 1 |
| **INT** | 2 | 2 |
| **INT16** | 2 | 2 |
| **INT32** | 2 | 2 |
| **INT64** | 2 | 2 |
| **REAL32** | 2 | 2 |
| **REAL64** | 2 | 2 |

and those for a 32 bit transputer are

| T | length(T) | alignment(T) |
|---|---|---|
| **BOOL** | 1 | 1 |
| **BYTE** | 1 | 1 |
| **INT** | 4 | 4 |
| **INT16** | 2 | 2 |
| **INT32** | 4 | 4 |
| **INT64** | 4 | 4 |
| **REAL32** | 4 | 4 |
| **REAL64** | 4 | 4 |

The length of an array [n]T is n * length(T) and its alignment is that of T. The alignment of a procedure is that of **INT**.

Let e be an element of type E. The definition T n **RETYPES** E is valid provided that length(T) = length(E), and (alignment(E) REM alignment(T)) = 0.

The definition []T n **RETYPES** E is valid provided that (length(E) REM length(T)) = 0 and (alignment(E) REM alignment(T)) = 0.

The length of any element e is the value of **SIZE** n, where n is specified by []**BYTE** n **RETYPES** e.

## 2 Standard Procedures

The following arithmetic procedures are provided. They all treat their parameters as unsigned integer values.

**LongAdd ( INT Sum, INT A, INT B, INT Carry_In)**

**LongSum ( INT Carry_Out, INT Sum, INT A, INT B, INT Carry_In)**

**LongSub ( INT Diff, INT A, INT B, INT Borrow_In)**

**LongDiff ( INT Borrow_Out, INT Diff, INT A, INT B, INT Borrow_In)**

**LongProd ( INT Carry_out, INT Prod, INT A, INT B, INT Carry_In )**

**LongDiv ( INT Quot, INT Rem, INT Dvd_hi, INT Dvd_lo, INT Dvsr )**

**ShiftRight ( INT S_hi, INT S_lo, INT A_hi, INT A_lo, INT Places )**

**ShiftLeft ( INT S_hi, INT S_lo, INT A_hi, INT A_lo, INT Places )**

**Normalise ( INT Places, INT N_hi, INT N_Lo, INT A_hi, INT A_Lo )**