M.9 Arccosine

```
REAL32 FUNCTION ACOS (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DACOS (VAL REAL64 X)
  ...
:
```

These compute : result = cosine$^{-1}(X)$ (in radians).

Domain : [−1.0, 1.0]
Range : [0, Pi]

All arguments outside the domain generate an **undefined.NaN**.

M.10 Arctangent

```
REAL32 FUNCTION ATAN (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DATAN (VAL REAL64 X)
  ...
:
```

These compute : result = tan$^{-1}(X)$ (in radians).

Domain : [−Inf, Inf]
Range : [−Pi/2, Pi/2]

M.11 Polar Angle

```
REAL32 FUNCTION ATAN2 (VAL REAL32 X, Y)
  ...
:
REAL64 FUNCTION DATAN2 (VAL REAL64 X, Y)
  ...
:
```

These compute the angular co-ordinate tan$^{-1}(Y/X)$ (in radians) of a point whose X and Y co-ordinates are given.

Domain : [−Inf, Inf] x [−Inf, Inf]
Range : (−Pi, Pi]

(0, 0) and (±**Inf**,±**Inf**) give **undefined.NaN**.

M.12 Hyperbolic sine

```
REAL32 FUNCTION SINH (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DSINH (VAL REAL64 X)
  ...
:
```

These compute : result = sinh(X).

Domain : [−Hmax, Hmax] = $[-89.4, 89.4]S$, = $[-710.5, 710.5]D$
Range : (−Inf, Inf)

$X < -$Hmax gives −**Inf**, and $X >$ Hmax gives **Inf**.

M.13 Hyperbolic cosine

```
REAL32 FUNCTION COSH (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DCOSH (VAL REAL64 X)
  ...
:
```

These compute: result = cosh(X).

Domain : [−Hmax, Hmax] = $[-89.4, 89.4]S$, = $[-710.5, 710.5]D$
Range : [1.0, Inf)

$|X| >$ Hmax gives **Inf**.

M.14 Hyperbolic tangent

```
REAL32 FUNCTION TANH (VAL REAL32 X)
  ...
:
REAL64 FUNCTION DTANH (VAL REAL64 X)
  ...
:
```

These compute : result = tanh(X).

Domain : [−Inf, Inf]
Range : [−1.0, 1.0]

M.15 Pseudo-random numbers

```
REAL32, INT32 FUNCTION RAN (VAL INT32 N)
  ...
:
REAL64, INT64 FUNCTION DRAN (VAL INT64 N)
  ...
:
```

This function returns two results, the first is a real between 0.0 and 1.0, and the second is an integer. The integer, which must be used as the parameter in the next call to the function, carries a pseudo-random linear congruential sequence N_k, and must be kept in scope for as long as the function is used. It should be initialised before the first call to the function but not modified thereafter except by the function itself. Consider the following sequence:

```
SEQ
  x, seed := RAN (8)     -- initialise seed
  y, seed := RAN (seed)
  z, seed := RAN (seed)
```

In this example **x**, **y**, and **z** are each assigned a pseudo-random value.

Domain : Integers
Range : [0.0, 1.0) x Integers

N Value, string conversion routines

This appendix describes the standard library of string to value, value to string routines. The library provides primitive procedures to convert a value to and from decimal or hexadecimal representations. High level input/output routines can be easily built using these simple procedures, and a number will typically be provided in an implementation.

N.1 Integer, string conversions

The procedures described here provide conversion between integer values and their decimal or hexadecimal representations held as a string of characters, for example:

```
PROC INTTOSTRING (INT len, []BYTE string, VAL INT n)
  ...
:
```

The procedure **INTTOSTRING** returns the decimal representation of **n** in **string** and the number of characters in the representation in **len**.

```
PROC STRINGTOINT (BOOL error, INT n, VAL []BYTE string)
  ...
:
```

The procedure **STRINGTOINT** returns in **n** the value represented by **string**. **error** is set to **TRUE** if a non numeric character is found in **string**. **+** or a **–** are allowed in the first character position. **n** will be the value of the the portion of **string** up to any illegal character with the convention that the value of an empty string is 0. **error** is also set if the value of **string** overflows the range of **INT**, in this case **n** will contain the low order bits of the binary representation of **string**. **error** is set to **FALSE** in all other cases.

```
PROC HEXTOSTRING (INT len, []BYTE string, VAL INT n)
  ...
:
```

The procedure **HEXTOSTRING** returns the hexadecimal representation of **n** in **string** and the number of characters in the representation in **len**. All the nibbles (a nibble is a word 4 bits wide) of **n** are output so that leading zeros are included. The number of characters will be the number of bits in an **INT** divided by 4.

```
PROC STRINGTOHEX (BOOL error, INT n, VAL []BYTE string)
  ...
:
```

The procedure **STRINGTOHEX** returns in **n** the value represented by the hexadecimal **string**. **error** is set to **TRUE** if a non hexadecimal character is found in **string**. Here **n** will be the value of the the portion of **string** up to the illegal character with the convention that the value of an empty string is 0. **error** is also set to **TRUE** if the value represented by **string** overflows the range of **INT**. In this case **n** will contain the low order bits of the binary representation of **string**. In all other cases **error** is set to **FALSE**.

Similar procedures are provided for the types **INT16**, **INT32** and **INT64**. These procedures use equivalent parameters of the appropriate type. The procedures are:

INTTOSTRING	INT16TOSTRING	INT32TOSTRING	INT64TOSTRING
STRINGTOINT	STRINGTOINT16	STRINGTOINT32	STRINGTOINT64
HEXTOSTRING	HEX16TOSTRING	HEX32TOSTRING	HEX64TOSTRING
STRINGTOHEX	STRINGTOHEX16	STRINGTOHEX32	STRINGTOHEX64

N.2 Boolean, string conversion

The procedures described here provide conversion between boolean values and their textual representation "**TRUE**" and "**FALSE**".

```
PROC BOOLTOSTRING (INT len, []BYTE string, VAL BOOL b)
  ...
:
```

The procedure **BOOLTOSTRING** returns "**TRUE**" in **string** if **b** is **TRUE** and "**FALSE**" otherwise. **len** contains the number of characters in the string returned.

```
PROC STRINGTOBOOL (BOOL error, b, VAL []BYTE string)
  ...
:
```

The procedure **STRINGTOBOOL** returns **TRUE** in b if first 4 characters of **string** are "**TRUE**", **FALSE** if first 5 characters are "**FALSE**" and **b** is undefined in other cases. **TRUE** is returned in **error** if **string** is not exactly "**TRUE**" or **"FALSE"**.

N.3 Real, string conversion

The procedures described here provide conversion between real values and their representation as strings, for example:

```
PROC STRINGTOREAL32 (BOOL error, REAL32 r, VAL []BYTE string)
  ...
:
PROC STRINGTOREAL64 (BOOL error, REAL64 r, VAL []BYTE string)
  ...
:
```

These two procedures each take a string containing a decimal representation of a real number and convert it into the corresponding real value. If the value represented by **string** overflows the range of the type then an appropriately signed infinity is returned. Errors in the syntax of **string** are signalled by a Not-a-Number being returned and **error** being set to **TRUE**. The string is scanned from the left as far as possible while the syntax is still legal. If there any characters after the end of the longest correct string then error is set to **TRUE**, otherwise it is **FALSE**. For example if **string** was **"12.34E+2+1.0"** then the value returned would be 12.34×10^2 with **error** set to **TRUE**. Strings which represent real values are those specified by the syntax for *real* literals, for example:

```
12.34
587.0E-20
+1.0E+123
-3.05
```

Further examples are given in the section on literals on page 27.

```
PROC REAL32TOSTRING (INT len, []BYTE string,
                     VAL REAL32 r, VAL INT m,n)
  ...
:
PROC REAL64TOSTRING (INT len, []BYTE string,
                     VAL REAL64 r, VAL INT m,n)
  ...
:
```

These two procedures return a string representing the value **r** in the first **len** **BYTE**s of **string**. The format of the representation is determined by **m** and **n**. Free format is selected by passing **0** in **m** and **n** into the procedure. Where possible a fixed point representation is used when this does not indicate more accuracy than is available and does not have more than 3 "`0`"s after the decimal point before significant digits. Otherwise exponential form is used. The number of characters returned in **`string`** here depends on the input but will be no more than 15 in **`REALTOSTRING32`** and 24 in **`REALTOSTRING64`**. **`string`** is left justified in free format.

If **`m`** is non-zero then if possible the procedure returns a fixed point representation of **`r`** with **`m`** digits before the decimal point and **`n`** places after with padding spaces being added when needed. If this is not possible then an exponential representation is returned with the same field width as the fixed point representation would have had. If **`m`** and **`n`** are both very small then an exponential representation may not fit in the field width so two special values "**`Un`**" and "**`Ov`**" with a sign are returned to indicate a value under or over the representable fixed point values. In all these cases **`string`** is padded with spaces so that it contains ($\mathbf{m} + \mathbf{n} + 2$) characters - **`m`** before the decimal point, **`n`** after, as well as the sign and decimal point characters.

If **`m`** is zero but **`n`** is not then an exponential representation is returned where the number of digits of fraction returned is **`n`**. The form of the fraction is *digit.digits* except when **`n`** is 1. In this case the output is not a proper representation as the fraction will be of the form *' ' digit* where the padding space is added due to the absence of a decimal point. For this reason the case $\mathbf{m} = 0$, $\mathbf{n} = 1$ should not be used in general. When **`m`** is 0 **`string`** will contain ($\mathbf{n} + 6$) characters for **`REALTOSTRING32`** and ($\mathbf{n} + 7$) for **`REALTOSTRING64`**.

Each procedure returns a string "**`Inf`**" preceded by a sign character for infinities and a string "**`NaN`**" for Not-a-Numbers. In free format a leading space on either string is dropped. Both these will be padded on the right with spaces to fill the field width when free format output is not being used.

O Programming support routines

This appendix describes some additional routines that have been added to occam implementations to provide support for the programmer. For full specifications of these see documentation provided with the compiler.

O.1 Rescheduling the processor

In real time processing it is sometimes necessary to force the compiler to ensure that a process in a parallel suspends itself to give another process a chance to be executed. A call of **RESCHEDULE()** will have this effect.

```
PROC RESCHEDULE ()
  ...
:
```

O.2 Assertion checking

This procedure is provided to enable the programmer to make assertions involving the variables of the program, to ensure that if such an assertion is found to be false at compile time, then the compiler will report an error, or if it is found to be false at run time then the program will stop at that point.

```
PROC ASSERT (VAL BOOL assertion)
  ...
:
```

An instance of **ASSERT** may usefully be included as a “stub” at all places in a program under development that will eventually be replaced by suitable error checking code. The careful use of assertions is at the heart of defensive programming.

An implementation may provide compiler options to control the action which occurs when an assertion is found to be false.

P Changes from occam 2

This appendix enumerates the changes which have been made to occam 2 to derive the extended language occam 2.1, and the more prominent changes in the style and structure of this manual to facilitate the description of the extended language.

P.1 Language changes

The following groups of changes have been made to occam 2:

1. Addition of named types. (page 26)
2. Addition of record and packed record types. (page 31)
3. New operators **BYTESIN** and **OFFSETOF**. (page 65)
4. Arithmetic and other operations on values of **BYTE** type. (page 57)
5. Implicit typing of literal constants. (page 29)
6. Functions returning results of any fixed length type. (page 78)
7. Segments with optional **FROM** and **FOR**. (page 37)
8. Reshaping of arrays. (page 88)
9. Scope of **PROTOCOL** tags. (page 53)
10. Revised treatment of anarchic protocols. (page 53)
11. Channel retyping. (page 86)
12. Channel array constructors. (page 55)
13. Revised semantics of counted array communication. (page 47)
14. Restrictions on the positioning of allocations. (page 95)

P.2 Manual changes

The opportunity has been taken in this revised manual to make some structural changes, and many detailed changes in style of presentation.

The following structural changes have been made:

- A new chapter on variables and values collects together material that was previously spread over the chapters on arrays and elements, scope and abbreviation.
- The chapter on timers has been delayed until after the more familiar procedures and functions have been introduced.
- Description of priority alternation has been moved from the appendix to the chapter on constructed processes.
- The description of retyping has been moved from an appendix to a new chapter together with the new feature of reshaping.
- The complete syntax in the appendix is fully annotated with cross references to the places in the text where the semantic restrictions applicable to each production are defined.

- A new appendix on implementation dependent features has been added.

In addition to material defining and illustrating the new extensions several revisions have been made to the detailed presentation of language syntax in the manual. In particular:

The syntactic category *element* has been dropped, to simplify the separate handling of *variable*, *channel* and *timer*, which have many syntactic similarities but also some significant differences in how they are used.

The treatment of *string* has been revised to emphasise its semantic closeness to *table*.

The syntactic categories *primitive.type* and *array.type* have been replaced by the categories *data.type*, *channel.type*, *timer.type* and *port.type*, each of which may be used for scalars and arrays.

The syntactic categories *action*, *function.body*, *procedure.body*, and *valof* which served no particularaly useful syntactic purpose and had only descriptive value have been eliminated.

The syntactic categories *monadic.operator* and *dyadic.operator* have been explicitly defined in the syntax, and the lexical level categories *name*, *digits*, *hex.digits*, *character* and *string* in the lexical appendix.

Q Glossary of terms

Abbreviation An abbreviation specifies a *name* as an *alias* for an existing *variable, channel, port* or *timer* or for the value of an *expression*. The meaning of the alias is defined by substitution of the abbreviated expression, variable, etc.

Action An action is an assignment, an input, or an output.

Actual parameter A parameter used in an *instance* of a procedure or function.

Alias check Ensure all variables and channels are identified by a single name within a given *scope*.

Allocation Place a *variable*, *channel*, *timer*, *array* or *port* at an absolute location in memory.

Alternation Combines a number of processes guarded by inputs, and performs the process associated with an input which is ready.

Alternative A component of an *alternation*.

Anarchic protocol A protocol compatible with any communication using a simple protocol.

Argument A parameter used in an instance of a function.

Array A sequence of components of the same type stored in consecutive memory locations.

Assignment Evaluates an expression or list of expressions, and assigns each resulting value to a corresponding variable.

Base An expression defining the starting value of a *replication index* or the subscript of the first component of a *segment*.

Bitwise operation Operation on the individual bits in the representation of a value.

Boolean A truth value used in a guard or loop.

Boolean operation Logical operation on truth values.

Case input Uses tag to select the protocol of an input on a single channel with a case protocol.

Case protocol A protocol comprising a collection of tagged protocols.

Channel Unbuffered, uni-directional point-to-point connection for communication between two processes executing in parallel.

Channel array constructor An array of channels expressed as a sequence of component channels in square brackets.

Character A literal value of byte type expressed alone between single quotes, or as part of a string in double quotes.

Choice A component of a conditional.

Combination The collection of processes in a *construction*.

Communication The communication of values between concurrent processes. Each communication involves a rendezvous between an input and an output.

Component An array is a sequence of components of the same type selected by an integer subscript.

Concurrency Two or more processes are concurrent if they may be executed at the same time.

Conditional A construction (**`IF`**) which combines a number of processes each of which is guarded by a boolean. Behaves as the first to evaluate to true.

Configuration Configuration associates the components of an occam program with a set of physical resources on one or more processors.

Construction A construction combines processes. occam programs are built from processes, by combining primitive processes and other constructions to form constructions of *sequence* (**`SEQ`**), *conditional* (**`IF`**), *selection* (**`CASE`**), *loop* (**`WHILE`**), *parallel* (**`PAR`**) or *alternation* (**`ALT`**).

Conversion Use of a type name as an operator to convert a value to an equivalent value of another type.

Count An expression defining the number of replications in a *replicator* or number of components in a *segment*.

Counted array An array communicated on a channel, whose size is defined by an explicit count communicated previously.

Data type The data type of a variable defines which values can be stored in that variable. The data type of a value defines which operations can be performed on the value.

Deadlock A state in which two or more concurrent processes can no longer proceed due to a communication interdependency.

Declaration Specifies the name, type and scope of a *variable*, *channel*, *port*, *timer* or an array of one of these types.

Decoration The use of a type name in parentheses to define the type of a literal or table.

Delayed input A special *timer input* which will wait until the timer has incremented beyond a specified time before terminating. Useful for adding a simple delay in a process.

Definition A specification introducing a name for a procedure, function, protocol, data type, retyping or reshaping.

Digits Decimal or hexadecimal digits as a component of a literal of a real, integer or byte type.

Expression The representation of a value within a program. The data type of an expression can always be determined by the compiler.

Expression list A list of expressions separated by commas; used in *multiple assignment* and *functions*.

Field A field is a component of a record type. A field is selected from a record by a *field name* in square brackets following the record variable or value.

Formal parameter Parameter specified in the definition of a procedure or function. A formal parameter acts as an *abbreviation* for the *actual parameter* used in an *instance* of the procedure or function.

Free channel A channel whose name is a free name.

Free name A name which occurs within a process, but is not specified within the process.

Free variable A variable whose name is a free name.

Function definition Specifies a name for a value process or expression list.

Guard Determines the execution of an associated process in a choice (*boolean guard*) or alternative (*input guard*).

Indentation An offset from the left hand edge of the page. In occam indentation is critical, and serves to define the structure of processes. Indentation increases by two spaces for each nested construct in a program, and decreases by two spaces at the end of that construct.

Inline When the compiler implements an instance of a procedure or function by an explicit copy of the body, with appropriate parameter substitutions, rather than by a closed subroutine.

Input Receive a value from a channel into a variable.

Input guard An input which guards an alternative in an alternation.

Instance A process which is a call of a procedure or an operand or expression list which is a call of a function.

Invalid process A process whose behaviour has for some reason become undefined, and as a result may cause a system to stop.

Library function See *library procedure*.

Library procedure A procedure whose definition is compiled separately and whose compiled code may be referenced within a program by an implementation dependent language extension.

Literal A literal is a textual representation of a known value, and has a data type.

Livelock A divergent process, one which may remain internally active but not perform further communication.

Loop A (`WHILE`) loop executes the associated process as long as the specified condition is true; if the condition is initially false the associated process is not executed.

Modulo operator A modulo operator performs its operation (**PLUS**, **MINUS**, **TIMES**) with no check for overflow. The value returned as a result is the cyclic value within the range of the operand type.

Name A sequence of letters (and digits or dots) introduced in a specification as the name of a variable, field, tag, channel, timer, procedure, function, data type, protocol, etc.

Network a network consists of a number of processing devices, microcomputers perhaps, with the facility to communicate with each other.

Offset The position in memory of a field within a record.

Operand Yields a value in an expression.

Operator A monadic operator performs an operation on a single operand and a dyadic operator performs an operation on two operands.

Output Send the value of an expression to a channel.

Packed record A record in which the programmer explicitly defines the order and implicitly the offsets of all the fields.

Placement A configuration statement which places a process on a particular processing device.

Port A memory mapped peripheral control register from which input, or to which output, may be directed.

Predefine A predefined procedure or function is one that may be used in a program without explicit specification. An implementation may define a set of such names and specify their formal parameters.

Primitive data type A primitive data type is a integer, boolean, byte or real type. A named type derived from such a type or an array or structured type is not a primitive type.

Priority Priority can be given to the component processes of a parallel executing on a single processing device. Lower priority processes on such a device may only continue when all higher priority processes are unable to. The inputs which guard alternatives in an alternation may be given a selection priority. If two or more inputs are ready, then the input with the highest priority is selected.

Procedure definition A procedure definition specifies a name for a process.

Procedure instance An instance of a procedure is a use of the procedure, and behaves like a substitution of the process named in the procedure definition. The phrase “procedure call” is used in many other languages, to indicate the use of a procedure, and has a similar meaning. Although the behaviour of an occam procedure is clearly defined as the substitution of the procedure body, a procedure may be implemented as either a substitution or as a call to a closed subroutine.

Process A process starts, performs a number of actions, and then either stops without completing or terminates completely. occam programs are built from the primitive processes *assignment* (**:=**), *input* (**?**), *output* (**!**), **SKIP** and **STOP**. These primitives are combined in **SEQ**, **IF**, **CASE**, **WHILE**, **PAR** and **ALT** constructions.

Protocol The format and *type* of values passed (communicated) on a channel, or a name defined to represent such a format. Communication is valid only if the output and input are compatible; i.e. each communication matches the type(s) specified by the channel protocol.

Real A numerical value represented according to the IEEE standard for floating point arithmetic.

Real time The actual time taken for a physical process to occur. The time returned by a *timer* is directly related to real time.

Record A record consists of a number of named *fields* each of which has a specified data type. A value of record type associates a value of appropriate type with each of the fields.

Record layout The concrete representation of a record data type in store.

Relational operation A relational operation compares its operands and yields a boolean result.

Replication A replicator produces a number of similar components of a construction, distinguished by the value of a *replication index*.

Reshaping conversion A reshaping conversion changes the program's view of the internal structure of a (multi-dimensional) array.

Retyping conversion A retyping conversion changes the data type of a bit pattern, from one data type to another. There are three kinds of retyping conversions: conversions which change the protocol of a channel, conversions which convert a variable, and conversions which convert the value of an expression. Data type retyping has no effect upon the bit pattern, and differs from *type conversion* where a value of one type is converted into an equivalent value of another type.

Scope The region of a program in which a particular specification of a name is valid.

Segment A segment is one or more consecutive components of an array, defined by a base and a count. An abbreviated segment may have a computed number of components which may be zero but not negative.

Selection A selection process (`CASE`) executes a process from a list of associated options. The options are selected by matching a selector with a constant case expression associated with the option.

Sequence A sequential process (`SEQ`) is one where one action follows another.

Sequential protocol A sequential protocol specifies a sequence of simple protocols as the format of communication on a channel.

Shift operation Perform logical shift of the bit pattern of a value.

Skip Start, perform no action and terminate immediately.

Specification A specification is either a declaration, an abbreviation or a definition and specifies a name which may be used within the associated scope.

Specifier Identifies the type of an name specified in an abbreviation, formal parameter, or definition.

Stop Start, perform no further action and do not terminate.

String A sequence of characters in double quotes equivalent to a table of bytes.

Subscript An integer expression in square brackets (`[]`) which selects a component of an array.

Table An array of values of the same type, or a record of values of possibly different types, optionally decorated by a type name in parentheses, used in expressions.

Tag Identifier of a protocol variant specified in a *case protocol* definition. Used in an output or `CASE` input.

Tagged list Component of a variant in a *case input*.

Tagged protocol Specifies a possible case protocol variant, distinguished by its tag, for communication on a single channel.

Timer A timer is a clock which can be accessed by any number of concurrent processes.

Timer input A timer input inputs a value from a timer.

Type conversion A type conversion converts the value of an expression of one data type into an equivalent value of another data type.

Usage check A usage check performed by a compiler ensures that variables and channels are not shared illegally between parallel components.

Value process A value process produces one or more results, each of a data type.

Variable A variable is declared or abbreviated and may receive a new value by input or assignment. A variable normally requires the allocation of an addressable storage location in the computer.

Variable list A list of variables used in a *multiple assignment*.

Variable subscript A variable subscript is a subscript whose value depends on a variable, a procedure parameter, or the index of a replicator with a base or count which is not a constant or constant expression.

Variant A variant is a component of a *case input* consisting of a tagged list of variables and an associated process to be executed when the tag input matches the tag in the tagged list.

R Occam Bibliography

R.1 Books

Bowler, K.C, Kenway, R.D, Pawley, G.S. and Roweth, D *Introduction to OCCAM 2 Programming* Chartwell-Bratt 1989 ISBN 0 86238 227 0

Brookes, Graham R *Introduction to Occam 2 on the Transputer* Macmillan 1989 ISBN 0 333 45340 9

Burns, Alan *Programming in Occam 2* Addison-Wesley 1987 ISBN 0 201 17371 9

Carling, Alison *Parallel Processing, Transputer and Occam* Sigma 1988 ISBN 1 85058 07 4

Cok, Ronald S *Parallel Programs for the Transputer* Prentice-Hall 1991 ISBN 0 13 651480 4

Dowsing, R.L. *Introduction to Concurrency Using Occam* Van Nostrand Reinhold 1988 ISBN 0 278 00059 2

East, I *Parallel Processing: A First Course* Pitman 1989

Ellison, David *Understanding Occam and the Transputer : Through Complete Working Programs* Sigma ISBN 1 85058 206 8

Galletly, J *Occam 2* Pitman 1990 ISBN 0 273 03067 1
Japanese edition: ISBN 4-526-02874-6

Hoare, C.A.R *Communicating Sequential Processes* Prentice-Hall 1985 ISBN 0 13 153271 5

Inmos Limited *Occam Programming Manual* Prentice-Hall 1984 ISBN 0 13 629296 8
Japanese edition: ISBN 4-7665-0133-0

Inmos Limited (tr Fontaine A.B) *Occam 2: Manuel de Reference* (in French) Masson 1989

Inmos Limited *Occam 2 Reference Manual* Prentice-Hall 1988 ISBN 0 13 629312 3

Inmos Limited *Communicating Process Architecture* Prentice-Hall 1988 ISBN 0 13 629320 4

Inmos Limited *Transputer Technical Notes* Prentice-Hall 1989 ISBN 0 13 929126 1

Inmos Limited *Transputer Development System, Second Edition* Prentice-Hall 1990 ISBN 0 13 929068 0

Jones, Geraint *Programming in occam* Prentice-Hall 1987 ISBN 0 13 729773 4

Jones, G and Goldsmith, M *Programming in Occam 2* Prentice-Hall 1988 ISBN 0 13 730334 3

Kerridge, J *Occam Programming: A Practical Approach* Blackwell Scientific 1987 ISBN 0 632 01659 0

Pountain, D and May, D *A Tutorial Introduction to Occam Programming* Blackwell Scientific 1987 ISBN 0 632 01847 X

Roscoe, A.W. *Laws of Occam Programming* Oxford University Computing Laboratory, Programming Research Group 1986 ISBN 0 902928 34 1

Wexler, John *Concurrent Programming in occam2* Ellis Horwood 1989 ISBN 745 80394 6

R.2 Conference Proceedings

A large number of these have been published by IOS Press: ISSN 0925-4986

R.3 Journals, etc

Transputer Communications Wiley 1993- ISSN 1070-454X

WoTUG News (formerly *OUG Newsletter*) Published privately by the user group 1984-

Index