inmos

occam 2 toolset user manual – part 2

(occam libraries and appendices)

INMOS Limited

72 TDS 276 02

March 1991

INMOS document number: 72 TDS 276 02

Contents overview

Contents

Preface

Libraries

1	*The* occam *libraries*	Describes the library procedures and functions supplied with the toolset.

Appendices

A	*Names defined by the software*	Lists all names and identifiers used within the toolset.
B	*Transputer instructions*	Lists full and restricted sets of transputer instructions supported by the occam 2 toolset.
C	*Constants*	Lists files of constants supplied with the toolset.
D	*Implementation of* occam *on the transputer*	Describes how the compiler allocates memory and gives details of type mapping, hardware dependencies and language.
E	*Configuration language description*	Defines the syntax of the occam 2 configuration language.
F	*Bootstrap loaders*	Describes bootstrap loaders and lists the standard INMOS scheme.
G	*ITERM*	Describes the format of ITERM terminal support files.
H	*Host file server protocol*	Describes the protocol of the host file server and lists the server functions.
I	*Glossary*	A glossary of terms.
J	*Bibliography*	Literature and documentation for further reading.
	The Index	

Contents

Preface

The '*occam 2 toolset user manual*' is a combined user and reference guide to the occam 2 toolset.

Part 2 '*occam libraries and appendices*' (this book) provides a detailed description of all the libraries supplied with the toolset. Technical reference data is given in the appendices at the end of this book which also includes a glossary of terms and a short bibliography.

A description of the toolset and how it is used to develop and run transputer programs is given in Part 1 '*User guide and tools*' (72 TDS 275 02).

References which span the two parts, take the form of a part number followed by a chapter or section number. Each part contains its own index.

This manual does not contain details of how to install the software, which is to be found in the Delivery Manual that accompanies the shipment.

Host versions

The manual is designed to cover all host versions of the toolset:

IMS D7205	–	IBM and NEC PC running MS-DOS.
IMS D5205	–	Sun 3 systems running SunOS
IMS D4205	–	Sun 4 systems running SunOS
IMS D6205	–	VAX systems running VMS

Conventions used in the manual

Convention	Description
Italics	Used in command line syntax to denote parameters for which values *must* be supplied. Also used for book titles and for emphasis.
Bold	Used for new terms, pin signals, and the text of error messages.
`Teletype`	Used for listings of program examples and to denote user input and terminal output.
[KEY]	Used to denote function keys for the debugger and simulator tools. Keyboard layouts for specific terminals can be found in the Delivery Manual that accompanies the shipment.
▭	Used to indicate the continuation of a function key description.
Braces { }	Used to denote lists of items in command line syntax.
Brackets [ ]	Used to denote optional items in command line syntax.
Option prefix	Examples of command line input are duplicated to show both option prefix characters. Use the line containing the '/' character if you have an MS-DOS or VMS based system and the line containing the '–' character if you are using any other host including UNIX.

Libraries

1 The occam libraries

1.1 Introduction

A comprehensive set of occam libraries is provided for use with the toolset. They include the compiler libraries which are automatically referenced by the compiler, and a number of user libraries to assist with common programming tasks.

The user libraries provide standard mathematical functions, host i/o and file management procedures, file stream i/o support, and many other functions. A full list of all the toolset libraries with brief descriptions of their contents can be found in table 1.1.

Library	Description
Compiler Libraries	Multiple length integer arithmetic Floating point functions 32 bit IEEE arithmetic functions 64 bit IEEE arithmetic functions 2D block move library Bit manipulation and CRC library Arithmetic instruction library
snglmath.lib	Single length mathematical functions
dblmath.lib	Double length mathematical functions
tbmaths.lib	T4 series optimised maths functions
hostio.lib	Host file server library
streamio.lib	Stream i/o library
string.lib	String library
convert.lib	Type conversion library
crc.lib	Block CRC library
xlink.lib	Extraordinary link handling library
debug.lib	Debugging support functions
callc.lib	Mixed languages support library
msdos.lib	DOS specific hostio library

Table 1.1 Toolset libraries

1.1.1 Using the occam libraries

If a library routine is used in a program then the library must be declared with the #USE directive and the declaration must be in scope where the routine is used. The scope of a library, as with all occam declarations, is determined by its level of indentation in the occam text.

An example showing how to declare a library is given below.

```
#USE "hostio.lib"
```

Linking libraries

All libraries used by a program or program module must also be linked with the main progam. This includes the compiler libraries even though they are automatically referenced when the program is compiled.

1.1.2 Listing library contents

You can use the `ilist` tool to examine the contents of a library and determine which routines are available. The tool displays procedural interfaces for routines in each library module and the code size and workspace requirements for individual modules. It can also be used to determine the transputer types and error modes for which the code was compiled. (See chapter 20 for details of `ilist`).

1.1.3 Toolset constants

Constants and protocols used by the toolset libraries are defined in six include files which are supplied with the toolset. Two of the six files provide constants and definitions for the hostio and streamio libraries, a third provides mathematical and trigonometric constants, the fourth contains the absolute transputer link addresses, the fifth contains the rates at which the two transputer clocks increment on the transputer and the sixth provides constants to support the DOS specific library routines. All files containing constant definitions must be declared in the program before the library that references them.

Files of constants provided with the toolset are summarised in table 1.2. Full listings of the files can be found in appendix C.

File	Description
hostio.inc	Constants for the host file server interface
streamio.inc	Constants for the stream i/o interfaces
mathvals.inc	Maths constants
linkaddr.inc	Addresses of transputer links
ticks.inc	Rates of the two transputer clocks
msdos.inc	DOS specific constants

Table 1.2 Files of constants

1.2 Compiler libraries

Compiler libraries are used internally by the compiler to implement multiple length and floating point arithmetic, IEEE functions, and special transputer functions such as bit manipulation and 2D block data moves. They are found automatically by the compiler on the path specified by the **ISEARCH** host environment variable and do not need to be referenced by the **#USE** directive.

The compiler library **virtual.lib**, is disabled (i.e. automatic searching of the library by the compiler can be suppressed) by using the compiler '**Y**' option. The other compiler libraries are disabled by using the compiler '**E**' option.

Separate libraries are supplied for the following processor types:

- T2 family
- T8 family
- 32-bit processors

All error modes are supported by each library.

A full list of the compiler libraries is given below:

File	Processors
occam2.lib	T212/T222/T225/M212
occama.lib	T400/T414/T425/TA/TB
occam8.lib	T800/T801/T805
occamutl.lib	All
virtual.lib	All

The compiler library **occamutl.lib** contains routines which are called from within some of the other compiler libraries and **virtual.lib** is used to support interactive debugging. These two libraries support all processor types and error modes.

File names of the compiler libraries must not be changed. The compiler assumes these filenames, and generates an error if they are not found on the path specified by the host environment variable `ISEARCH`.

Descriptions of some of the compiler library functions and procedures can be found in the '*occam 2 Reference Manual*'.

1.2.1 User functions and procedures

The following routines from the compiler libraries may be of interest to the applications programmer. Calls to these routines can be made directly and do not have to reference the library in a `#USE` statement, provided the compiler '`E`' option is not used.

The functions are grouped as follows: maths functions, including some IEEE and extended arithmetic routines; 2-D block moves; bit manipulation; functions for cyclic redundancy checking (CRC) and supplementary arithmetic support functions.

The procedures listed in this section are grouped as follows: dynamic code loading support; rescheduling process priority queue and procedures to set the transputer error flag.

It is worth noting the difference between the default occam behaviour of arithmetic operations and the behaviour of the equivalent IEEE arithmetic functions. The difference in the implementations concerns the treatment of `NaN`s ('Not a Number') and `Inf`s ('± infinity'). The default occam behaviour of arithmetic operations is to cause an error if such quantities occur, whereas the IEEE functions implement the ANSI/IEEE 754-1985 standard. The IEEE functions use of infinities and `NaN`s to handle errors and overflows may be prefered in some instances, in which case these functions must be explicitly called. For example if

A, B and C are REAL32s:

```
A := B + C  -- default occam behaviour.

A := REAL32OP(B, 0, C) -- IEEE function, round to
                       -- nearest only. The 0
                       -- indicates a '+'
                       -- operation.

A := IEEE32OP(B, 1, 0, C) -- IEEE function with
                          -- rounding option. The
                          -- 1 indicates round to
                          -- nearest. The 0
                          -- indicates a '+'
                          -- operation.
```

The IEEE floating point arithmetic functions are described in more detail in the '*occam 2 Reference Manual*'.

Maths functions

Result(s)	Function	Parameter specifiers
REAL32	ABS	VAL REAL32 x
REAL32	SQRT	VAL REAL32 x
REAL32	LOGB	VAL REAL32 x
INT, REAL32	FLOATING.UNPACK	VAL REAL32 x
REAL32	MINUSX	VAL REAL32 x
REAL32	MULBY2	VAL REAL32 x
REAL32	DIVBY2	VAL REAL32 x
REAL32	FPINT	VAL REAL32 x
BOOL	ISNAN	VAL REAL32 x
BOOL	NOTFINITE	VAL REAL32 x
REAL32	SCALEB	VAL REAL32 x, VAL INT n
REAL32	COPYSIGN	VAL REAL32 x, y
REAL32	NEXTAFTER	VAL REAL32 x, y
BOOL	ORDERED	VAL REAL32 x, y
BOOL, INT32, REAL32	ARGUMENT.REDUCE	VAL REAL32 x, y, y.err
REAL32	REAL32OP	VAL REAL32 x, VAL INT op, VAL REAL32 y
REAL32	REAL32REM	VAL REAL32 x, y
BOOL, REAL32	IEEE32OP	VAL REAL32 x, VAL INT rm, op, VAL REAL32 y
BOOL, REAL32	IEEE32REM	VAL REAL32 x, y
BOOL	REAL32EQ	VAL REAL32 x, y
BOOL	REAL32GT	VAL REAL32 x, y
INT	IEEECOMPARE	VAL REAL32 x, y

Result(s)	Function	Parameter specifiers
REAL64	DABS	VAL REAL64 x
REAL64	DSQRT	VAL REAL64 x
REAL64	DLOGB	VAL REAL64 x
INT, REAL64	DFLOATING.UNPACK	VAL REAL64 x
REAL64	DMINUSX	VAL REAL64 x
REAL64	DMULBY2	VAL REAL64 x
REAL64	DDIVBY2	VAL REAL64 x
REAL64	DFPINT	VAL REAL64 x
BOOL	DISNAN	VAL REAL64 x
BOOL	DNOTFINITE	VAL REAL64 x
REAL64	DSCALEB	VAL REAL64 x, VAL INT n
REAL64	DCOPYSIGN	VAL REAL64 x, y
REAL64	DNEXTAFTER	VAL REAL64 x, y
BOOL	DORDERED	VAL REAL64 x, y
BOOL, INT32, REAL64	DARGUMENT.REDUCE	VAL REAL64 x, y, y.err
REAL64	REAL64OP	VAL REAL64 x, VAL INT op, VAL REAL64 y
REAL64	REAL64REM	VAL REAL64 x, y
BOOL, REAL64	IEEE64OP	VAL REAL64 x, VAL INT rm, op, VAL REAL64 y
BOOL, REAL64	IEEE64REM	VAL REAL64 x, y
BOOL	REAL64EQ	VAL REAL64 x, y
BOOL	REAL64GT	VAL REAL64 x, y
INT	DIEEECOMPARE	VAL REAL64 x, y

Result(s)	Function	Parameter specifiers
INT	LONGADD	VAL INT left, right, carry.in
INT	LONGSUM	VAL INT left, right, carry.in
INT	LONGSUB	VAL INT left, right, borrow.in
INT, INT	LONGDIFF	VAL INT left, right, borrow.in
INT, INT	LONGPROD	VAL INT left, right, carry.in
INT, INT	LONGDIV	VAL INT dividend.hi, dividend.lo, divisor
INT, INT	SHIFTRIGHT	VAL INT hi.in, lo.in, places
INT, INT	SHIFTLEFT	VAL INT hi.in, lo.in, places
INT, INT, INT	NORMALISE	VAL INT hi.in, lo.in
INT	ASHIFTRIGHT	VAL INT argument, places
INT	ASHIFTLEFT	VAL INT argument, places
INT	ROTATELEFT	VAL INT argument, places
INT	ROTATERIGHT	VAL INT argument, places

`SHIFTRIGHT` and `SHIFTLEFT` return zeroes when the number of places to shift is negative, or is greater than twice the transputer's word length. In this case they may take a long time to execute.

`ASHIFTRIGHT`, `ASHIFTLEFT`, `ROTATERIGHT` and `ROTATELEFT` are all invalid when the number of places to shift is negative or exceeds the transputer's word length.

2D block moves

Procedure	Parameter Specifiers
MOVE2D	VAL [][]BYTE Source, VAL INT sx, sy, [][]BYTE Dest, VAL INT dx, dy, width, length
DRAW2D	VAL [][]BYTE Source, VAL INT sx, sy, [][]BYTE Dest, VAL INT dx, dy, width, length
CLIP2D	VAL [][]BYTE Source, VAL INT sx, sy, [][]BYTE Dest, VAL INT dx, dy, width, length

Procedure definitions

MOVE2D

```
PROC MOVE2D (VAL [][]BYTE Source,
             VAL INT sx, sy, [][]BYTE Dest,
             VAL INT dx, dy, width, length)
```

Moves a data block of size `width` by `length` starting at byte `Source[sy][sx]` to the block starting at `Dest[dy][dx]`.

DRAW2D

```
PROC DRAW2D (VAL [][]BYTE Source,
             VAL INT sx, sy, [][]BYTE Dest,
             VAL INT dx, dy, width, length)
```

As `MOVE2D` but only non-zero bytes are transferred.

CLIP2D

```
PROC CLIP2D (VAL [][]BYTE Source,
             VAL INT sx, sy, [][]BYTE Dest,
             VAL INT dx, dy, width, length)
```

As `MOVE2D` but only zero bytes are transferred.

Bit manipulation functions

Result	Function	Parameter Specifiers
`INT`	`BITCOUNT`	`VAL INT Word, CountIn`
`INT`	`BITREVNBITS`	`VAL INT x, n`
`INT`	`BITREVWORD`	`VAL INT x`

Function definitions

`BITCOUNT`

```
INT FUNCTION BITCOUNT (VAL INT Word, CountIn)
```

Counts the number of bits set to 1 in `Word`, adds it to `CountIn`, and returns the total.

`BITREVNBITS`

```
INT FUNCTION BITREVNBITS (VAL INT x, n)
```

Returns an `INT` containing the `n` least significant bits of `x` in reverse order. The upper bits are set to zero. The operation is invalid if `n` is negative or greater than the number of bits in a word.

`BITREVWORD`

```
INT FUNCTION BITREVWORD (VAL INT x)
```

Returns an `INT` containing the bit reversal of `x`.

CRC functions

Result	Function	Parameter Specifiers
`INT`	`CRCWORD`	`VAL INT data, CRCIn, generator`
`INT`	`CRCBYTE`	`VAL INT data, CRCIn, generator`

Function descriptions

CRCWORD

```
INT FUNCTION CRCWORD  (VAL INT data, CRCIn,
                                generator)
```

Performs a cyclic redundancy check over the single word **data** using the CRC value obtained from the previous call. **generator** is the CRC polynomial generator. Can be used iteratively on a sequence of words to obtain the CRC.

CRCBYTE

```
INT FUNCTION CRCBYTE (VAL INT data, CRCIn,
                               generator)
```

As CRCWORD but performs the check over a single byte. The byte processed is contained in the *most* significant byte of the word **data**.

For further information about CRC functions see '*INMOS Technical note 26: Notes on graphics support and performance improvements on the IMS T800*'.

Supplementary arithmetic support functions

Result(s)	Function	Parameter specifiers
INT	FRACMUL	VAL INT x, y
INT, INT, INT	UNPACKSN	VAL INT x
INT	ROUNDSN	VAL INT Yexp, Yfrac, Yguard

Function descriptions

FRACMUL

```
INT FUNCTION FRACMUL  (VAL INT x, y)
```

Performs a fixed point multiplication of **x** and **y**, treating each as a binary fraction in the range [-1, 1), and returning their product rounded to the nearest available representation. The value of the fractions represented by the arguments and result can be obtained by multiplying their INT value by 2^{-31} (on a 32-bit processor) or 2^{-15} (on a 16-bit processor).

The result can overflow if both **x** and **y** are -1.

This routine is compiled inline into a sequence of transputer instructions on 32-bit processors, or as a call to a standard library routine for 16-bit processors.

UNPACKSN

```
INT, INT, INT FUNCTION UNPACKSN (VAL INT x)
```

This returns three parameters; from left to right they are **Xfrac**, **Xexp**, and **Type**. **X** is regarded as an IEEE single length real number (i.e. a **RETYPED REAL32**). The function unpacks **X** into **Xexp**, the (biased) exponent, and **Xfrac** the fractional part, with implicit bit restored. It also returns an integer defining the **Type** of **X**, ignoring the sign bit:

Type	Reason
0	X is zero
1	X is a normalised or denormalised number
2	X is Inf
3	X is NaN

This routine is compiled inline into a sequence of transputer instructions on 32-bit processors such as the **IMS T425**, which do not have a floating support unit, but do have special instructions for floating point operations. For other 32-bit processors the function is compiled as a call to a standard library routine. It is invalid on 16-bit processors, since **Xfrac** cannot fit into an **INT**.

ROUNDSN

```
INT FUNCTION ROUNDSN  (VAL INT Yexp, Yfrac,
                               Yguard)
```

This takes a possibly unnormalised fraction, guard word and exponent, and returns the IEEE floating point value it represents. It takes care of all the normalisation, post-normalisation, rounding and packing of the result. The rounding mode used is round to nearest. The exponent should already be biased.

The function normalises and post-normalises the number represented by **Yexp**, **Yfrac** and **Yguard** into local variables **Xexp**, **Xfrac**, and **Xgaurd**. It then packs the (biased) exponent **Xexp** and fraction **Xfrac** into the result, rounding using the extra bits in **Xguard**. The sign bit is set to 0. If overfow occurs, **Inf** is returned.

This routine is compiled inline into a sequence of transputer instructions on 32-bit processors such as the IMS T425, which do not have a floating support unit, but do have special instructions for floating point operations. For other 32-bit processors the function is compiled as a call to a standard library routine. It is invalid on 16-bit processors, since Xfrac cannot fit into an INT.

Dynamic code loading support procedures

Procedure	Parameter Specifiers
KERNEL.RUN	VAL []BYTE code, VAL INT entry.offset, []INT workspace, VAL INT no.of.parameters
LOAD.INPUT.CHANNEL	INT here, CHAN OF ANY in
LOAD.INPUT.CHANNEL.VECTOR	INT here, []CHAN OF ANY in
LOAD.OUTPUT.CHANNEL	INT here, CHAN OF ANY out
LOAD.OUTPUT.CHANNEL.VECTOR	INT here, []CHAN OF ANY out
LOAD.BYTE.VECTOR	INT here, VAL []BYTE bytes

Procedure definitions

KERNEL.RUN

```
PROC KERNEL.RUN (VAL []BYTE code,
                 VAL INT entry.offset,
                 []INT workspace,
                 VAL INT no.of.parameters)
```

The effect of this procedure is to call the procedure loaded in the code buffer, starting execution at the location code[entry.offset].

The code to be called must begin at a word-aligned address. To ensure proper alignment either start the array at zero or realign the code on a word boundary before passing it into the procedure.

The **workspace** buffer is used to hold the local data of the called procedure. The required size of this buffer and the code buffer must be derived from information in the code file.

The parameters passed to the called procedure should be placed at the top of the **workspace** buffer by the calling procedure before the call of **KERNEL.RUN**. The call to **KERNEL.RUN** returns when the called procedure terminates. If the called procedure requires a separate vector space, then another buffer of the required size must be declared, and its address placed as the last parameter at the top of **workspace**. As calls of **KERNEL.RUN** are handled specially by the compiler it is necessary for **no.of.parameters** to be a constant known at compile time and to have a value ≥ 3.

The workspace passed to **KERNEL.RUN** must be at least:

```
[ws.requirement + no.of.parameters + 2]INT
```

where **ws.requirement** is the size of workspace required, determined when the called procedure was compiled, and stored in the code file and **no.of.parameters** includes the vector space pointer if it is required.

The parameters must be loaded before the call of **KERNEL.RUN**. The parameter corresponding to the first formal parameter of the procedure should be in the word adjacent to the saved **Iptr** word, and the vector space pointer or the last parameter should be adjacent to the top of workspace where the **Wptr** word will be saved.

Note: code developed with the current toolset will not be able to call code compiled by previous toolsets, if channel arrays are used.

LOAD.INPUT.CHANNEL

```
LOAD.INPUT.CHANNEL (INT here, CHAN OF ANY in)
```

The variable **here** is assigned the address of the input channel **in**.

LOAD.INPUT.CHANNEL.VECTOR

```
LOAD.INPUT.CHANNEL.VECTOR (INT here,
                           []CHAN OF ANY in)
```

The variable **here** is assigned the address of the base element of the channel array **in** (i.e. the base of the array of pointers). **Note** this is a change from the previous implementation of this procedure which used to return the actual address of the input channel array.

LOAD.OUTPUT.CHANNEL

```
LOAD.OUTPUT.CHANNEL (INT here, CHAN OF ANY out)
```

The variable **here** is assigned the address of the output channel **out**.

LOAD.OUTPUT.CHANNEL.VECTOR

```
LOAD.OUTPUT.CHANNEL.VECTOR (INT here,
                            []CHAN OF ANY out)
```

The variable **here** is assigned the address of the base element of the channel array **out** (i.e. the base of the array of pointers). **Note** this is a change from the previous implementation of this procedure which used to return the actual address of the output channel array.

LOAD.BYTE.VECTOR

```
LOAD.BYTE.VECTOR (INT here, VAL []BYTE bytes)
```

The variable **here** is assigned the address of the byte array **bytes**.

Transputer error flag manipulation

Procedure	Parameter Specifiers
CAUSEERROR	()
ASSERT	VAL BOOL test

Procedure definitions

CAUSEERROR

```
CAUSEERROR()
```

Inserts a **seterr** instruction into the program. If the program is in STOP or UNIVERSAL mode it inserts a **stopp** instruction as well. The error is then treated in exactly the same way as any other error would be treated in the error mode in which the program is compiled. For example, in HALT mode the whole processor will halt.

ASSERT

```
PROC ASSERT (VAL BOOL test)
```

At compile time the compiler will check the value of `test` and if it is `FALSE` the compiler will give a compile time error; if it is `TRUE`, the compiler does nothing. If `test` cannot be checked at compile-time then the compiler will insert a run-time check to detect its status.

`ASSERT` is a useful routine for debugging purposes. Once a program is working correctly the compiler option '`NA`' can be used to prevent code being generated to check for `ASSERT`s at run-time. If possible `ASSERT`s will still be checked at compile time.

Rescheduling priority process queue

Procedure	Parameter Specifiers
RESCHEDULE	()

Procedure definition

RESCHEDULE

```
RESCHEDULE()
```

Inserts enough instructions into the program to cause the current process to be moved to the end of the current priority scheduling queue, even if the current process is a 'high priority' process.

1.3 Maths libraries

Elementary maths and trigonometric functions are provided in three libraries, as follows:

Library	Description
`snglmath.lib`	Single length library
`dblmath.lib`	Double length library
`tbmaths.lib`	`TB` optimised library

The single and double length libraries contain the same set of functions in single and double length forms. By convention the double length forms begin with the letter '`D`'. Function names are in upper case.

The `TB` optimised library is a combined single and double length library containing all the single and double length functions optimised for the `T400`, `T414` and `T425` processors. The standard maths libraries can also be used on the `T400`, `T414` and `T425`, but optimum performance on these processors can be achieved by using the optimised functions.

The accuracy of the `T400`/`T414`/`T425` optimised functions is similar to that of the standard single length functions but results returned may not be identical because different algorithms are used.

Functions in the optimised library have the same names as the equivalent functions in the single and double length libraries. This means that the optimised library cannot be used together with either the single or double length library on the same processor. If the optimised library is used in code compiled for any processor except a `T400`, `T414` or `T425`, the compiler reports an error.

A set of constants for the maths libraries are provided in the include file `mathvals.inc`, which is listed in appendix C.

1.3.1 Introduction

This, and the following subsections, contain some notes on the presentation of the elementary function libraries described in section 1.3.2, and the TB version described in section 1.3.3.

These function subroutines have been written to be compatible with the ANSI standard for binary floating-point arithmetic (ANSI-IEEE std 754-1985), as implemented in occam. They are based on the algorithms in:
Cody, W. J., and Waite, W. M. [1980]. *Software Manual for the Elementary Functions*. Prentice-Hall, New Jersey.

The only exceptions are the pseudo-random number generators, which are based on algorithms in:
Knuth, D. E. [1981]. *The Art of Computer Programming, 2nd. edition, Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Mass.

Inputs

All the functions in the library (except RAN and DRAN) are called with one or two parameters which are binary floating-point numbers in one of the IEEE standard formats, either 'single-length' (32 bits) or 'double-length' (64 bits). The parameter(s) and the function result are of the same type.

NaNs and Infs

The functions will accept any value, as specified by the standard, including special values representing **NaNs** ('Not a Number') and **Infs** ('Infinity'). **NaNs** are copied to the result, whilst **Infs** may or may not be in the domain. The domain is the set of arguments for which the result is a normal (or denormalised) floating-point number.

Outputs

Exceptions

Arguments outside the domain (apart from **NaNs** which are simply copied through) give rise to *exceptional results*, which may be **NaN**, **+Inf**, or **−Inf**. **Infs** mean that the result is mathematically well-defined but too large to be represented in the floating-point format.

Error conditions are reported by means of three distinct **NaNs**:

undefined.NaN

This means that the function is mathematically undefined for this argument, for example the logarithm of a negative number.

unstable.NaN

This means that a small change in the argument would cause a large change in the value of the function, so *any* error in the input will render the output meaningless.

inexact.NaN

This means that although the mathematical function is well-defined, its value is in range, and it is stable with respect to input errors at this argument, the limitations of word-length (and reasonable cost of the algorithm) make it impossible to compute the correct value.

The implementations will return the following values for these Not-a-Numbers:

Error	Single length value	Double length value
`undefined.NaN`	#7F800010	#7FF00002 00000000
`unstable.NaN`	#7F800008	#7FF00001 00000000
`inexact.NaN`	#7F800004	#7FF00000 80000000

Accuracy

Range Reduction

Since it is impractical to use rational approximations (i.e. quotients of polynomials) which are accurate over large domains, nearly all the subroutines use mathematical identities to relate the function value to one computed from a smaller argument, taken from the 'primary domain', which is small enough for such an approximation to be used. This process is called 'range reduction' and is performed for all arguments except those which already lie in the primary domain.

For most of the functions the quoted error is for arguments in the primary domain, which represents the basic accuracy of the approximation. For some functions the process of range reduction results in a higher accuracy for arguments outside the primary domain, and for others it does the reverse. Refer to the notes on each function for more details.

Generated Error

If the true value of the function is large the difference between it and the computed value (the 'absolute error') is likely to be large also because of the limited accuracy of floating-point numbers. Conversely if the true value is small, even a small absolute error represents a large proportional change. For this reason the error relative to the true value is usually a better measure of the accuracy of a floating-point function, except when the ouput range is strictly bounded.

If f is the mathematical function and F the subroutine approximation, then the relative error at the floating-point number X (provided $f(X)$ is not zero) is:

$$RE(X) = \frac{(F(X) - f(X))}{f(X)}$$

Obviously the relative error may become very large near a zero of $f(X)$. If the zero is at an irrational argument (which cannot be represented as a floating-point value), the absolute error is a better measure of the accuracy of the function near the zero.

As it is impractical to find the relative error for every possible argument, statistical measures of the overall error must be used. If the relative error is sampled at a number of points X_n (n = 1 to N), then useful statistics are the *maximum relative error* and the *root-mean-square relative error*:

$$MRE = \max_{1 \leq n \leq N} |RE(X_n)|$$

$$RMSRE = \sqrt{\sum_{n=1}^{N} (RE(X_n))^2}$$

Corresponding statistics can be formed for the absolute error also, and are called MAE and $RMSAE$ respectively.

The MRE generally occurs near a zero of the function, especially if the true zero is irrational, or near a singularity where the result is large, since the 'granularity' of the floating-point numbers then becomes significant.

A useful unit of relative error is the relative magnitude of the least significant bit in the floating-point fraction, which is called one 'unit in the last place' (ulp), (i.e. the smallest ϵ such that $1 + \epsilon \neq 1$). Its magnitude depends on the floating-point format: for single-length it is $2^{-23} = 1.19 * 10^{-7}$, and for double-length it is $2^{-52} = 2.22 * 10^{-16}$.

Propagated Error

Because of the limited accuracy of floating-point numbers the result of any calculation usually differs from the exact value. In effect, a small error has been added to the exact result, and any subsequent calculations will inevitably involve this error term. Thus it is important to determine how each function responds to errors in its argument. Provided the error is not too large, it is sufficient just to consider the first derivative of the function (written f').

If the relative error in the argument X is d (typically a few ulp), then the absolute error (E) and relative error (e) in $f(X)$ are:

$$E = |Xf'(X)d| \equiv Ad$$

$$e = \left| \frac{Xf'(X)d}{f(X)} \right| \equiv Rd$$

This defines the absolute and relative error magnification factors A and R. When both are large the function is unstable, i.e. even a small error in the argument, such as would be produced by evaluating a floating-point expression, will cause a large error in the value of the function. The functions return an **unstable.NaN** in such cases which are simple to detect.

The functional forms of both A and R are given in the specification of each function.

Test Procedures

For each function, the generated error was checked at a large number of arguments (typically 100 000) drawn at random from the appropriate domain. First the double-length functions were tested against a 'quadruple-length' implementation (constructed for accuracy rather than speed), and then the single-length functions were tested against the double-length versions.

In both cases the higher-precision implementation was used to approximate the mathematical function (called f above) in the computation of the error, which was evaluated in the higher precision to avoid rounding errors. Error statistics were produced according to the formulae above.

Symmetry

The subroutines were designed to reflect the mathematical properties of the functions as much as possible. For all the functions which are even, the sign is removed from the input at the beginning of the computation so that the sign-symmetry of the function is always preserved. For odd functions, either the sign is removed at the start and then the appropriate sign set at the end of the computation, or else the sign is simply propagated through an odd degree polynomial. In many cases other symmetries are used in the range-reduction, with the result that they will be satisfied automatically.

The Function Specifications

Names and Parameters

All single length functions except **RAN** return a single result of type **REAL32**, and all except **RAN**, **POWER** and **ATAN2** have one parameter, a **VAL REAL32** for the argument of the function.

POWER and **ATAN2** have two parameters which are **VAL REAL32**s for the two arguments of each function.

RAN returns two results of type **REAL32, INT32**, and has one parameter which is a **VAL INT32**.

In each case the double-length version of **name** is called **Dname**, returns a **REAL64** (except **DRAN**, which returns **REAL64, INT64**), and has parameters of type **VAL REAL64** (**VAL INT64** for **DRAN**).

Terms used in the Specifications

A and R Multiplying factors relating the absolute and relative errors in the output to the relative error in the argument.

Exceptions Outputs for invalid inputs (i.e. those outside the *domain*), other than **NaN** (**NaNs** are copied directly to the output and are not listed as exceptions). These are all **Infs** or **NaNs**.

Generated Error The difference between the true and computed values of the function, when the argument is error-free. This is measured statistically and displayed for one or two ranges of arguments, the first of which is usually the *primary domain* (see below). The second range, if present, is chosen to illustrate the typical behaviour of the function.

Domain The range of valid inputs, i.e. those for which the output is a normal or denormal floating-point number.

MAE and RMSAE The Maximum Absolute Error and Root-Mean-Square absolute error taken over a number of arguments drawn at random from the indicated range.

MRE and RMSRE The Maximum Relative Error and Root-Mean-Square relative error taken over a number of arguments drawn at random from the indicated range.

Range The range of outputs produced by all arguments in the *Domain*. The given endpoints are not exceeded.

Primary Domain The range of arguments for which the result is computed using only a single rational approximation to the function. There is no argument reduction in this range.

Propagated Error The absolute and relative error in the function value, given a small relative error in the argument.

ulp The unit of relative error is the 'unit in the last place' (ulp). This is the relative magnitude of the least significant bit of the floating-point fraction (i.e. the smallest ϵ such that $1 + \epsilon \neq 1$).
N.B. this depends on the floating-point format.
For the standard single-length format it is $2^{-23} = 1.19 * 10^{-7}$.
For the double-length format it is $2^{-52} = 2.22 * 10^{-16}$.
This is also used as a measure of absolute error, since such errors can

be considered 'relative' to unity.

Specification of Ranges

Ranges are given as intervals, using the convention that a square bracket '[' or ']' means that the adjacent endpoint is included in the range, whilst a round bracket '(' or ')' means that it is excluded. Endpoints are given to a few significant figures only.

Where the range depends on the floating-point format, single-length is indicated with an S and double-length with a D.

For functions with two arguments the complete range of both arguments is given. This means that for each number in one range, there is at least one (though sometimes only one) number in the other range such that the pair of arguments is valid. Both ranges are shown, linked by an 'x'.

Abbreviations

In the specifications, $XMAX$ is the largest representable floating-point number: in single-length it is approximately $3.4 * 10^{38}$, and in double-length it is approximately $1.8 * 10^{308}$.

Pi means the closest floating-point representation of the transcendental number π, ln(2) the closest representation of $\log_e(2)$, and so on.

In describing the algorithms, 'X' is used generically to designate the argument, and 'result' (or RESULT, in the style of occam functions) to designate the output.

1.3.2 Single length and double length elementary function libraries

The versions of the libraries described by this section have been written using only floating-point arithmetic and pre-defined functions supported in occam. Thus they can be compiled for any processor with a full implementation of occam, and give identical results.

These two libraries will be efficient on processors with fast floating-point arithmetic and good support for the floating-point predefined functions such as **MULBY2** and **ARGUMENT.REDUCE**. For 32-bit processors without special hardware for floating-point calculations the alternative optimised library described in section 1.3.3 using fixed-point arithmetic will be faster, but will not give identical results.

A special version has been produced for 16-bit transputers, which avoids the use of any double-precision arithmetic in the single precision functions. This is distinguished in the notes by the annotation 'T2 special'; notes relating to the version for T8 and TB are denoted by 'standard'.

Single and double length maths functions are listed below. Descriptions of the functions can be found in succeeding sections.

To use the single length library a program header must include the line

```
#USE "snglmath.lib"
```

To use the double length library a program header must include the line

```
#USE "dblmath.lib"
```

Result(s)	Function	Parameter specifiers
REAL32	ALOG	VAL REAL32 X
REAL32	ALOG10	VAL REAL32 X
REAL32	EXP	VAL REAL32 X
REAL32	POWER	VAL REAL32 X, VAL REAL32 Y
REAL32	SIN	VAL REAL32 X
REAL32	COS	VAL REAL32 X
REAL32	TAN	VAL REAL32 X
REAL32	ASIN	VAL REAL32 X
REAL32	ACOS	VAL REAL32 X
REAL32	ATAN	VAL REAL32 X
REAL32	ATAN2	VAL REAL32 X, VAL REAL32 Y
REAL32	SINH	VAL REAL32 X
REAL32	COSH	VAL REAL32 X
REAL32	TANH	VAL REAL32 X
REAL32,INT32	RAN	VAL INT32 X

Result(s)	Function	Parameter specifiers
REAL64	DALOG	VAL REAL64 X
REAL64	DALOG10	VAL REAL64 X
REAL64	DEXP	VAL REAL64 X
REAL64	DPOWER	VAL REAL64 X, VAL REAL64 Y
REAL64	DSIN	VAL REAL64 X
REAL64	DCOS	VAL REAL64 X
REAL64	DTAN	VAL REAL64 X
REAL64	DASIN	VAL REAL64 X
REAL64	DACOS	VAL REAL64 X
REAL64	DATAN	VAL REAL64 X
REAL64	DATAN2	VAL REAL64 X, VAL REAL64 Y
REAL64	DSINH	VAL REAL64 X
REAL64	DCOSH	VAL REAL64 X
REAL64	DTANH	VAL REAL64 X
REAL64,INT64	DRAN	VAL INT64 X

Function definitions

ALOG
DALOG

```
REAL32 FUNCTION ALOG (VAL REAL32 X)
REAL64 FUNCTION DALOG (VAL REAL64 X)
```

Compute $\log_e(X)$.

Domain:	$(0, XMAX]$
Range:	$[MinLog, MaxLog]$ (See note 2)
Primary Domain:	$[\sqrt{2}/2, \sqrt{2}) = [0.7071, 1.4142)$

Exceptions

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

Propagated Error

$A \equiv 1, \qquad R = 1/\log_e(X)$

Generated Error

Primary Domain Error:	MRE	RMSRE
Single Length(Standard):	1.7 ulp	0.43 ulp
Single Length(T2 special):	1.6 ulp	0.42 ulp
Double Length:	1.4 ulp	0.38 ulp

The Algorithm

1 Split X into its exponent N and fraction F.

2 Find LnF, the natural log of F, with a floating-point rational approximation.

3 Compute $\ln(2) * N$ with extended precision and add it to LnF to get the result.

Notes

1) The term $\ln(2) * N$ is much easier to compute (and more accurate) than LnF, and it is larger provided N is not 0 (i.e. for arguments outside the primary domain). Thus the accuracy of the result improves as the modulus of $\log(X)$ increases.

2) The minimum value that can be produced, MinLog, is the logarithm of the smallest denormalised floating-point number. For single length $Minlog$ is -103.28, and for double length it is -745.2. The maximum value $MaxLog$ is the logarithm of $XMAX$. For single-length it is 88.72, and for double-length it is 709.78.

3) Since **Inf** is used to represent *all* values greater than $XMAX$ its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

`ALOG10`
`DALOG10`

```
REAL32 FUNCTION ALOG10 (VAL REAL32 X)
REAL64 FUNCTION DALOG10 (VAL REAL64 X)
```

Compute $\log_{10}(X)$.

Domain:	$(0, XMAX]$
Range:	$[MinL10, MaxL10]$ (See note 2)
Primary Domain:	$[\sqrt{2}/2, \sqrt{2})$ = [0.7071, 1.4142)

Exceptions

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

Propagated Error

$A \equiv \log_{10}(e)$, $R = \log_{10}(e)/\log_e(X)$

Generated Error

Primary Domain Error:	**MRE**	**RMSRE**
Single Length(Standard):	1.70 ulp	0.45 ulp
Single Length(T2 special):	1.71 ulp	0.46 ulp
Double Length:	1.84 ulp	0.45 ulp

The Algorithm

1 Set $temp$:= `ALOG(X)`.

2 If $temp$ is a **NaN**, copy it to the output, otherwise set result = log(e) $*$ $temp$

Notes

1) See note 1 for `ALOG`.

2) The minimum value that can be produced, $MinL10$, is the base-10 logarithm of the smallest denormalised floating-point number. For single length $MinL10$ is −44.85, and for double length it is −323.6. The maximum value $MaxL10$ is the base-10 logarithm of $XMAX$. For single length $MaxL10$ is 38.53, and for double-length it is 308.26.

3) Since **Inf** is used to represent *all* values greater than $XMAX$ its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

`EXP`
`DEXP`

```
REAL32 FUNCTION EXP (VAL REAL32 X)
REAL64 FUNCTION DEXP (VAL REAL64 X)
```

Compute e^X.

Domain: [−**Inf**, $MaxLog$) = [−**Inf**, 88.72)S, [−**Inf**, 709.78)D

Range:	[0, **Inf**)		(See note 4)
Primary Domain:	$[-Ln2/2, Ln2/2)$	=	[−0.3466, 0.3466)

Exceptions

All arguments outside the domain generate an **Inf**.

Propagated error

$A = Xe^X, \quad R = X$

Generated error

Primary Domain Error:	**MRE**	**RMSRE**
Single Length(Standard):	0.99 ulp	0.25 ulp
Single Length(T2 special):	1.0 ulp	0.25 ulp
Double Length:	1.0 ulp	0.25 ulp

The Algorithm

1 Set N = integer part of $X/\ln(2)$.

2 Compute the remainder of X by ln(2), using extended precision arithmetic.

3 Compute the exponential of the remainder with a floating-point rational approximation.

4 Increase the exponent of the result by N. If N is sufficiently negative the result must be denormalised.

Notes

1) $MaxLog$ is $\log_e(XMAX)$.

2) For sufficiently negative arguments (below −87.34 for single-length and below −708.4 for double-length) the output is denormalised, and so the floating-point number contains progressively fewer significant digits, which degrades the accuracy. In such cases the error can theoretically be a factor of two.

3) Although the true exponential function is never zero, for large negative arguments the true result becomes too small to be represented as a floating-point number, and **EXP** underflows to zero. This occurs for arguments below −103.9 for single-length, and below −745.2 for double-length.

4) The propagated error is considerably magnified for large positive arguments, but diminished for large negative arguments.

POWER
DPOWER

```
REAL32 FUNCTION POWER (VAL REAL32 X, Y)
REAL64 FUNCTION DPOWER (VAL REAL64 X, Y)
```

Compute X^Y.

Domain:	[0, **Inf**] x [−**Inf**, **Inf**]
Range:	(−**Inf**, **Inf**)
Primary Domain:	See note 3.

Exceptions

If the first argument is outside its domain, **undefined.NaN** is returned. If the true value of X^Y exceeds $XMAX$, **Inf** is returned. In certain other cases other **NaNs** are produced: See note 2.

Propagated Error

$A = YX^Y(1 \pm \log_e(X))$, $R = Y(1 \pm \log_e(X))$ (See note 4)

Generated error

Example Range Error:	MRE	RMSRE	(See note 3)
Single Length(Standard):	1.0 ulp	0.25 ulp	
Single Length(T2 special):	63.1 ulp	13.9 ulp	
Double Length:	21.1 ulp	2.4 ulp	

The Algorithm

Deal with special cases: either argument = 1, 0, +**Inf** or −**Inf** (see note 2). Otherwise:

(a) For the standard single precision:

1 Compute $L = \log_e(X)$ in double precision, where X is the first argument.

2 Compute $W = Y \times L$ in double precision, where Y is the second argument.

3 Compute $RESULT = e^W$ in single precision.

(b) For double precision, and the single precision special version:

1 Compute $L = \log_2(X)$ in extended precision, where X is the first argument.

2 Compute $W = Y \times L$ in extended precision, where Y is the second argument.

3 Compute $RESULT = 2^W$ in extended precision.

Notes

1) This subroutine implements the mathematical function x^y to a much greater accuracy than can be attained using the **ALOG** and **EXP** functions, by performing each step in higher precision. The single-precision version is more efficient than using **DALOG** and **EXP** because redundant tests are omitted.

2) Results for special cases are as follows:

First Input (X)	Second Input (Y)	Result
< 0	ANY	**undefined.NaN**
0	≤ 0	**undefined.NaN**
0	$0 < Y \leq XMAX$	0
0	**Inf**	**unstable.NaN**
$0 < X < 1$	**Inf**	0
$0 < X < 1$	**-Inf**	**Inf**
1	$-XMAX \leq Y \leq XMAX$	1
1	$\pm$ **Inf**	**unstable.NaN**
$1 < X \leq XMAX$	**Inf**	**Inf**
$1 < X \leq XMAX$	**-Inf**	0
Inf	$1 \leq Y \leq$ **Inf**	**Inf**
Inf	**-Inf**$\leq Y \leq -1$	0
Inf	$-1 < Y < 1$	**undefined.NaN**
otherwise	0	1
otherwise	1	X

3) Performing all the calculations in extended precision makes the double-precision algorithm very complex in detail, and having two arguments makes a primary domain difficult to specify. As an indication of accuracy, the functions were evaluated at 100 000 points logarithmically distributed over (0.1, 10.0), with the exponent linearly distributed over (−35.0, 35.0) (single-length), and (−300.0, 300.0) (double-length), producing the errors given above. The errors are much smaller if the exponent range is reduced.

4) The error amplification factors are calculated on the assumption that the relative error in Y is $\pm$ that in X, otherwise there would be separate factors for both X and Y. It can be seen that the propagated error will be greatly amplified whenever $\log_e(X)$ or Y is large.

```
SIN
DSIN
```

```
REAL32 FUNCTION SIN (VAL REAL32 X)
REAL64 FUNCTION DSIN (VAL REAL64 X)
```

Compute sine(X)   (where X is in radians).

Domain: $[-Smax, Smax]$ = [−205887.4, 205887.4]S (Standard),
[$-4.2 * 10^6, 4.2 * 10^6$]S (T2 special)
= [$-3.4 * 10^9, 3.4 * 10^9$]D

Range: [−1.0, 1.0]
Primary Domain: $[-Pi/2, Pi/2]$ = [−1.57, 1.57]

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except **±Inf**, which generates an **undefined.NaN**.

Propagated Error

$A = X\cos(X)$, $R = X\cot(X)$

Generated error (See note 1)

	Primary Domain		$[0, 2Pi]$	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length (Standard):	0.94 ulp	0.23 ulp	0.96 ulp	0.19 ulp
Single Length (T2 special):	0.92 ulp	0.23 ulp	0.94 ulp	0.19 ulp
Double Length:	0.9 ulp	0.22 ulp	0.91 ulp	0.18 ulp

The Algorithm

1 Set N = integer part of $|X|/Pi$.

2 Compute the remainder of $|X|$ by Pi, using extended precision arithmetic (double precision in the standard version).

3 Compute the sine of the remainder using a floating-point polynomial.

4 Adjust the sign of the result according to the sign of the argument and the evenness of N.

Notes

1) For arguments outside the primary domain the accuracy of the result depends crucially on step 2. The extra precision of step 2 is lost if N becomes too large, and the cut-off $Smax$ is chosen to prevent this. In

any case for large arguments the 'granularity' of floating-point numbers becomes a significant factor. For arguments larger than $Smax$ a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

2) The propagated error has a complex behaviour. The propagated relative error becomes large near each zero of the function (outside the primary range), but the propagated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

3) Since only the remainder of X by Pi is used in step 3, the symmetry $\sin(x + n\pi) = \pm\sin(x)$ is preserved, although there is a complication due to differing precision representations of π.

4) The output range is not exceeded. Thus the output of `SIN` is always a valid argument for `ASIN`.

COS
DCOS

```
REAL32 FUNCTION COS (VAL REAL32 X)
REAL64 FUNCTION DCOS (VAL REAL64 X)
```

Compute cosine(X) (where X is in radians).

Domain: $[-Cmax, Cmax]$ = $[-205887.4, 205887.4]$S (Standard),
$[-12868.0, 12868.0]$S (T2 special)
= $[-3.4 * 10^9, 3.4 * 10^9]$D

Range: $[-1.0, 1.0]$
Primary Domain: See note 1.

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except **±Inf**, which generates an **undefined.NaN**.

Propagated Error

$A = -X\sin(X)$, $R = -X\tan(X)$ (See note 4)

Generated error

Range:	[0, $Pi/4$)		[0, $2Pi$]	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length (Standard):	0.93 ulp	0.25 ulp	0.88 ulp	0.18 ulp
Single Length (T2 special):	1.1 ulp	0.3 ulp	0.94 ulp	0.19 ulp
Double Length:	1.0 ulp	0.28 ulp	0.9 ulp	0.19 ulp

The Algorithm

1 Set N = integer part of $(|X|+Pi/2)/Pi$ and compute the remainder of $(|X|+Pi/2)$ by Pi, using extended precision arithmetic (double precision in the standard version).

2 Compute the sine of the remainder using a floating-point polynomial.

3 Adjust the sign of the result according to the evenness of N.

Notes

1) Inspection of the algorithm shows that argument reduction always occurs, thus there is no 'primary domain' for COS. So for all arguments the accuracy of the result depends crucially on step 2. The standard single-precision version performs the argument reduction in double-precision, so there is effectively no loss of accuracy at this step. For the T2 special version and the double-precision version there are effectively K extra bits in the representation of π(K = 8 for the former and 12 for the latter). If the argument agrees with an odd integer multiple of $\pi/2$ to more than k bits there is a loss of significant bits from the computed remainder equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The difference between COS evaluated at successive floating-point numbers is given approximately by the absolute error amplification factor, A. For arguments larger than $Cmax$ this difference may be more than half the significant bits of the result, and so the result is considered to be essentially indeterminate and an **inexact.NaN** is returned. The extra precision of step 2 in the double-precision and T2 special versions is lost if N becomes too large, and the cut-off at $Cmax$ prevents this also.

3) For small arguments the errors are not evenly distributed. As the argument becomes smaller there is an increasing bias towards negative

errors (which is to be expected from the form of the Taylor series). For the single-length version and X in [−0.1, 0.1], 62% of the errors are negative, whilst for X in [−0.01, 0.01], 70% of them are.

4) The propagated error has a complex behaviour. The propagated relative error becomes large near each zero of the function, but the propagated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

5) Since only the remainder of $(|X| + Pi/2)$ by Pi is used in step 3, the symmetry $\cos(x+n\pi) = \pm\cos(x)$ is preserved. Moreover, since the same rational approximation is used as in **SIN**, the relation $\cos(x) = \sin(x + \pi/2)$ is also preserved. However, in each case there is a complication due to the different precision representations of π.

6) The output range is not exceeded. Thus the output of **COS** is always a valid argument for **ACOS**.

TAN
DTAN

```
REAL32 FUNCTION TAN (VAL REAL32 X)
REAL64 FUNCTION DTAN (VAL REAL64 X)
```

Compute $\tan(X)$ (where X is in radians).

Domain: $[-Tmax, Tmax]$ = $[-102943.7, 102943.7]$S(Standard),
$[-2.1 * 10^6, 2.1 * 10^6]$S(T2 special),
= $[-1.7 * 10^9, 1.7 * 10^9]$D

Range: (−**Inf**, **Inf**)
Primary Domain: $[-Pi/4, Pi/4]$ = $[-0.785, 0.785]$

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except ±**Inf**, which generate an **undefined.NaN**. Odd integer multiples of $\pi/2$ may produce **unstable.NaN**.

Propagated Error

$A = X(1 + \tan^2(X))$, $R = X(1 + \tan^2(X))/\tan(X)$ (See note 3)

Generated error

Primary Domain Error:	**MRE**	**RMSRE**
Single Length(Standard):	1.44 ulp	0.39 ulp
Single Length(T2 special):	1.37 ulp	0.39 ulp
Double Length:	1.27 ulp	0.35 ulp

The Algorithm

1 Set N = integer part of $X/(Pi/2)$, and compute the remainder of X by $Pi/2$, using extended precision arithmetic.

2 Compute two floating-point rational functions of the remainder, $XNum$ and $XDen$.

3 If N is odd, set $RESULT = -XDen/XNum$, otherwise set $RESULT = XNum/XDen$.

Notes

1) R is large whenever X is near to an integer multiple of $\pi/2$, and so tan is very sensitive to small errors near its zeros and singularities. Thus for arguments outside the primary domain the accuracy of the result depends crucially on step 2, so this is performed with very high precision, using double precision $Pi/2$ for the standard single-precision function and two double-precision floating-point numbers for the representation of $\pi/2$ for the double-precision function. The T2 special version uses two single-precision floating-point numbers. The extra precision is lost if N becomes too large, and the cut-off $Tmax$ is chosen to prevent this.

2) The difference between **TAN** evaluated at successive floating-point numbers is given approximately by the absolute error amplification factor, A. For arguments larger than $Smax$ this difference could be more than half the significant bits of the result, and so the result is considered to be essentially indeterminate and an **inexact.NaN** is returned.

3) Tan is quite badly behaved with respect to errors in the argument. Near its zeros outside the primary domain the relative error is greatly magnified, though the absolute error is only proportional to the size of the argument. In effect, the error is seriously amplified in an interval about each irrational zero, whose width increases roughly in proportion to the size of the argument. Near its singularities both absolute and relative errors become large, so any large output from this function is liable to be seriously contaminated with error, and the larger the argument, the smaller the maximum output which can be trusted. If step 3 of the algorithm requires division by zero, an **unstable.NaN** is produced

instead.

4) Since only the remainder of X by $Pi/2$ is used in step 3, the symmetry $\tan(x + n\pi) = \tan(x)$ is preserved, although there is a complication due to the differing precision representations of π. Moreover, by step 3 the symmetry $\tan(x) = 1/\tan(\pi/2 - x)$ is also preserved.

ASIN
DASIN

```
REAL32 FUNCTION ASIN (VAL REAL32 X)
REAL64 FUNCTION DASIN (VAL REAL64 X)
```

Compute $\text{sine}^{-1}(X)$ (in radians).

Domain:	$[-1.0, 1.0]$
Range:	$[-Pi/2, Pi/2]$
Primary Domain:	$[-0.5, 0.5]$

Exceptions

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

Propagated Error

$A = X/\sqrt{1 - X^2}, \quad R = X/(\sin^{-1}(X)\sqrt{1 - X^2})$

Generated Error

	Primary Domain		$[-1.0, 1.0]$	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length:	0.58 ulp	0.21 ulp	1.35 ulp	0.33 ulp
Double Length:	0.59 ulp	0.21 ulp	1.26 ulp	0.27 ulp

The Algorithm

1 If $|X| > 0.5$, set $Xwork$:= **SQRT** $((1 - |X|)/2)$. Compute $Rwork$ = arcsine($-2 * Xwork$) with a floating-point rational approximation, and set the result = $Rwork + Pi/2$.

2 Otherwise compute the result directly using the rational approximation.

3 In either case set the sign of the result according to the sign of the argument.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error: however since both domain and range are bounded the *absolute* error in the result cannot be large.

2) By step 1, the identity $\sin^{-1}(x) = \pi/2 - 2\sin^{-1}(\sqrt{(1-x)/2})$ is preserved.

ACOS
DACOS

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

Compute $\text{cosine}^{-1}(X)$ (in radians).

Domain:	$[-1.0, 1.0]$
Range:	$[0, Pi]$
Primary Domain:	$[-0.5, 0.5]$

Exceptions

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

Propagated Error

$A = -X/\sqrt{1-X^2}$, $R = -X/(\sin^{-1}(X)\sqrt{1-X^2})$

Generated Error

	Primary Domain		[−1.0, 1.0]	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length:	1.06 ulp	0.38 ulp	2.37 ulp	0.61 ulp
Double Length:	0.96 ulp	0.32 ulp	2.25 ulp	0.53 ulp

The Algorithm

1 If $|X| > 0.5$, set $Xwork$:= SQRT ((1 − $|X|$)/2). Compute $Rwork$ = arcsine($2 * Xwork$) with a floating-point rational approximation. If the argument was positive, this is the result, otherwise set the result = $Pi - Rwork$.

2 Otherwise compute $Rwork$ directly using the rational approximation. If the argument was positive, set result = $Pi/2 - Rwork$, otherwise result = $Pi/2 + Rwork$.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error, although this interval is larger near 1 than near −1, since the function goes to zero with an infinite derivative there. However since both the domain and range are bounded the *absolute* error in the result cannot be large.

2) Since the rational approximation is the same as that in **ASIN**, the relation $\cos^{-1}(x) = \pi/2 - \sin^{-1}(x)$ is preserved.

ATAN
DATAN

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

Compute $\tan^{-1}(X)$ (in radians).

Domain:	[−**Inf**, **Inf**]
Range:	$[-Pi/2, Pi/2]$
Primary Domain:	$[-z, z]$, $z = 2 - \sqrt{3} = 0.2679$

Exceptions

None.

Propagated Error

$A = X/(1 + X^2)$, $R = X/(\tan^{-1}(X)(1 + X^2))$

Generated Error

Primary Domain Error:	**MRE**	**RMSRE**
Single Length:	0.56 ulp	0.21 ulp
Double Length:	0.52 ulp	0.21 ulp

The Algorithm

1 If $|X| > 1.0$, set $Xwork = 1/|X|$, otherwise $Xwork = |X|$.

2 If $Xwork > 2 - \sqrt{3}$, set $F = (Xwork * \sqrt{3} - 1)/(Xwork + \sqrt{3})$, otherwise $F = Xwork$.

3 Compute $Rwork = \arctan(F)$ with a floating-point rational approximation.

4 If $Xwork$ was reduced in (2), set $R = Pi/6 + Rwork$, otherwise $R = Rwork$.

5 If X was reduced in (1), set $RESULT = Pi/2 - R$, otherwise $RESULT = R$.

6 Set the sign of the $RESULT$ according to the sign of the argument.

Notes

1) For $|X| > ATmax$, $|\tan^{-1}(X)|$ is indistinguishable from $\pi/2$ in the floating-point format. For single-length, $ATmax = 1.68 * 10^7$, and for double-length $ATmax = 9 * 10^{15}$, approximately.

2) This function is numerically very stable, despite the complicated argument reduction. The worst errors occur just above $2 - \sqrt{3}$, but are no more than 3.2 ulp.

3) It is also very well behaved with respect to errors in the argument, i.e. the error amplification factors are always small.

4) The argument reduction scheme ensures that the identities $\tan^{-1}(X) = \pi/2 - \tan^{-1}(1/X)$, and
$\tan^{-1}(X) = \pi/6 + \tan^{-1}((\sqrt{3} * X - 1)/(\sqrt{3} + X))$ are preserved.

ATAN2
DATAN2

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

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: $(-Pi, Pi]$
Primary Domain: See note 2.

Exceptions

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

Propagated Error

$A = X(1 \pm Y)/(X^2 + Y^2)$, $R = X(1 \pm Y)/(\tan^{-1}(Y/X)(X^2 + Y^2))$ (See note 3)

Generated Error (See note 2)

The Algorithm

1 If X, the first argument, is zero, set the result to $\pm\pi/2$, according to the sign of Y, the second argument.

2 Otherwise set $Rwork$:= **ATAN** (Y/X). Then if $Y < 0$ set $RESULT = Rwork - Pi$, otherwise set $RESULT = Pi - Rwork$.

Notes

1) This two-argument function is designed to perform rectangular-to-polar co-ordinate conversion.

2) See the notes for **ATAN** for the primary domain and estimates of the generated error.

3) The error amplification factors were derived on the assumption that the relative error in Y is ± that in X, otherwise there would be separate factors for X and Y. They are small except near the origin, where the polar co-ordinate system is singular.

SINH
DSINH

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

Compute sinh(X).

Domain: $[-Hmax, Hmax]$ = [−89.4, 89.4]S, [−710.5, 710.5]D

Range:	(−**Inf**, **Inf**)
Primary Domain:	$(-1.0, 1.0)$

Exceptions

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

Propagated Error

$A = X\cosh(X)$, $R = X\coth(X)$ (See note 3)

Generated Error

	Primary Domain		$[1.0, XBig]$ (See note 2)	
	MRE	**RMSRE**	**MRE**	**RMSRE**
Single Length:	0.91 ulp	0.26 ulp	1.41 ulp	0.34 ulp
Double Length:	0.67 ulp	0.22 ulp	1.31 ulp	0.33 ulp

The Algorithm

1 If $|X| > XBig$, set $Rwork$:= **EXP** $(|X| - \ln(2))$.

2 If $XBig \geq |X| \geq 1.0$, set temp:= **EXP** $(|X|)$, and set $Rwork = (temp - 1/temp)/2$.

3 Otherwise compute $\sinh(|X|)$ with a floating-point rational approximation.

4 In all cases, set $RESULT = \pm Rwork$ according to the sign of X.

Notes

1) $Hmax$ is the point at which $\sinh(X)$ becomes too large to be represented in the floating-point format.

2) $XBig$ is the point at which $e^{-|X|}$ becomes insignificant compared with $e^{|X|}$, (in floating-point). For single-length it is 8.32, and for double-length it is 18.37.

3) This function is quite stable with respect to errors in the argument. Relative error is magnified near zero, but the absolute error is a better measure near the zero of the function and it is diminished there. For large arguments absolute errors are magnified, but since the function is itself large, relative error is a better criterion, and relative errors are not magnified unduly for any argument in the domain, although the output does become less reliable near the ends of the range.

COSH
DCOSH

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

Compute cosh(X).

Domain: $[-Hmax, Hmax]$ = [−89.4, 89.4]S, [−710.5, 710.5]D

Range: [1.0, **Inf**)
Primary Domain: $[-XBig, XBig]$ = [−8.32, 8.32]S
[−18.37, 18.37]D

Exceptions

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

Propagated Error

$A = X\sinh(X)$, $R = X\tanh(X)$ (See note 3)

Generated Error

Primary Domain Error:	**MRE**	**RMS**
Single Length:	1.24 ulp	0.32 ulp
Double Length:	1.24 ulp	0.33 ulp

The Algorithm

1 If $|X| > XBig$, set $result$:= **EXP** ($|X| - \ln(2)$).

2 Otherwise, set $temp$:= **EXP** ($|X|$), and set $result = (temp + 1/temp)/2$.

Notes

1) $Hmax$ is the point at which cosh(X) becomes too large to be represented in the floating-point format.

2) $XBig$ is the point at which $e^{-|X|}$ becomes insignificant compared with $e^{|X|}$ (in floating-point).

3) Errors in the argument are not seriously magnified by this function, although the output does become less reliable near the ends of the range.

TANH
DTANH

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

Compute tanh(X).

Domain:	[−**Inf**, **Inf**]
Range:	$[-1.0, 1.0]$
Primary Domain:	[−Log(3)/2, Log(3)/2] = $[-0.549, 0.549]$

Exceptions

None.

Propagated Error

$A = X/\cosh^2(X)$, $R = X/\sinh(X)\cosh(X)$

Generated Error

Primary Domain Error:	**MRE**	**RMS**
Single Length:	0.53 ulp	0.2 ulp
Double Length:	0.53 ulp	0.2 ulp

The Algorithm

1 If $|X| > \ln(3)/2$, set $temp$:= **EXP** ($|X|/2$). Then set $Rwork = 1 - 2/(1 + temp)$.

2 Otherwise compute $Rwork = \tanh(|X|)$ with a floating-point rational approximation.

3 In both cases, set $RESULT = \pm Rwork$ according to the sign of X.

Notes

1) As a floating-point number, tanh(X) becomes indistinguishable from its asymptotic values of ± 1.0 for $|X| > HTmax$, where $HTmax$ is 8.4 for single-length, and 19.06 for double-length. Thus the output of **TANH** is equal to ± 1.0 for such X.

2) This function is very stable and well-behaved, and errors in the argument are always diminished by it.

RAN
DRAN

```
REAL32,INT32 FUNCTION RAN (VAL INT32 X)
REAL64,INT64 FUNCTION DRAN (VAL INT64 X)
```

These produce a pseudo-random sequence of integers, or a corresponding sequence of floating-point numbers between zero and one. **X** is the seed integer that initiates the sequence.

Domain: Integers (see note 1)
Range: [0.0, 1.0] x Integers

Exceptions

None.

The Algorithm

1 Produce the next integer in the sequence: $N_{k+1} = (aN_k + 1)_{mod\,M}$

2 Treat N_{k+1} as a fixed-point fraction in [0,1), and convert it to floating point.

3 Output the floating point result and the new integer.

Notes

1) This function has two results, the first a real, and the second an integer (both 32 bits for single-length, and 64 bits for double-length). The integer is used as the argument for the next call to **RAN**, i.e. it 'carries' the pseudo-random linear congruential sequence N_k, and it should be kept in scope for as long as **RAN** is used. It should be initialised before the first call to **RAN** but not modified thereafter except by the function itself.

2) If the integer parameter is initialised to the same value, the same sequence (both floating-point and integer) will be produced. If a different sequence is required for each run of a program it should be initialised to some 'random' value, such as the output of a timer.

3) The integer parameter can be copied to another variable or used in expressions requiring random integers. The topmost bits are the most random. A random integer in the range $[0, L]$ can conveniently be produced by taking the remainder by $(L+1)$ of the integer parameter shifted right by one bit. If the shift is not done an integer in the range $[-L, L]$ will be produced.

4) The modulus M is 2^{32} for single-length and 2^{64} for double-length, and

the multipliers, a, have been chosen so that all M integers will be produced before the sequence repeats. However several different integers can produce the same floating-point value and so a floating-point output may be repeated, although the *sequence* of such will not be repeated until M calls have been made.

5) The floating-point result is uniformly distributed over the output range, and the sequence passes various tests of randomness, such as the 'run test', the 'maximum of 5 test' and the 'spectral test'.

6) The double-length version is slower to execute, but 'more random' than the single-length version. If a highly-random sequence of single-length numbers is required, this could be produced by converting the output of `DRAN` to single-length. Conversely if only a relatively crude sequence of double-length numbers is required, `RAN` could be used for higher speed and its output converted to double-length.

1.3.3 IMS T400, T414 and T425 elementary function library

The version of the library described by this section has been written for 32-bit processors without hardware for floating-point arithmetic. Functions from it will give results very close, but not identical to, those produced by the corresponding functions from the single and double length libraries.

This is the version specifically intended to derive maximum performance from the IMS T400, T414 and T425 processors. The single-precision functions make use of the `FMUL` instruction available on 32-bit processors without floating-point hardware. The library is compiled for transputer class `TB`.

The tables and notes at the beginning of section 1.3 apply equally here. However all the functions are contained in one library. To use this library a program header must include the line:

```
#USE "tbmaths.lib"
```

Function definitions

ALOG

```
REAL32 FUNCTION ALOG (VAL REAL32 X)
REAL64 FUNCTION DALOG (VAL REAL64 X)
```

These compute: $\log_e(X)$

Domain:	$(0, XMAX]$
Range:	$[MinLog, MaxLog]$ (See note 2)
Primary Domain:	$[\sqrt{2}/2, \sqrt{2})$ = [0.7071, 1.4142)

Exceptions

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

Propagated Error

$A \equiv 1, \quad R = 1/\log_e(X)$

Generated Error

Primary Domain Error:	MRE	RMSRE
Single Length:	1.19 ulp	0.36 ulp
Double Length:	2.4 ulp	1.0 ulp

The Algorithm

1 Split X into its exponent N and fraction F.

2 Find the natural log of F with a fixed-point rational approximation, and convert it into a floating-point number LnF.

3 Compute $\ln(2) * N$ with extended precision and add it to LnF to get the result.

Notes

1) The term $\ln(2) * N$ is much easier to compute (and more accurate) than LnF, and it is larger provided N is not 0 (i.e. for arguments outside the primary domain). Thus the accuracy of the result improves as the modulus of $\log(X)$ increases.

2) The minimum value that can be produced, $MinLog$, is the logarithm of the smallest denormalised floating-point number. For single length

Minlog is −103.28, and for double length it is −745.2. The maximum value *MaxLog* is the logarithm of $XMAX$. For single-length it is 88.72, and for double-length it is 709.78.

3) Since **Inf** is used to represent *all* values greater than $XMAX$ its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

ALOG10

```
REAL32 FUNCTION ALOG10 (VAL REAL32 X)
REAL64 FUNCTION DALOG10 (VAL REAL64 X)
```

These compute: $\log_{10}(X)$

Domain:	$(0, XMAX]$
Range:	$[MinL10, MaxL10]$ (See note 2)
Primary Domain:	$[\sqrt{2}/2, \sqrt{2}) = [0.7071, 1.4142)$

Exceptions

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

Propagated Error

$A \equiv \log_{10}(e), \qquad R = \log_{10}(e)/\log_e(X)$

Generated Error

Primary Domain Error:	**MRE**	**RMSRE**
Single Length:	1.43 ulp	0.39 ulp
Double Length:	2.64 ulp	0.96 ulp

The Algorithm

1 Set $temp$:= `ALOG(X)`.

2 If $temp$ is a **NaN**, copy it to the output, otherwise set result = $\log(e) * temp$

Notes

1) See note 1 for `ALOG`.

2) The minimum value that can be produced, $MinL10$, is the base-10 logarithm of the smallest denormalised floating-point number. For single length $MinL10$ is −44.85, and for double length it is −323.6. The maximum value $MaxL10$ is the base-10 logarithm of $XMAX$. For single length $MaxL10$ is 38.53, and for double-length it is 308.26.

3) Since **Inf** is used to represent *all* values greater than $XMAX$ its logarithm cannot be defined.

4) This function is well-behaved and does not seriously magnify errors in the argument.

EXP

```
REAL32 FUNCTION EXP (VAL REAL32 X)
REAL64 FUNCTION DEXP (VAL REAL64 X)
```

These compute: e^X

Domain: [−**Inf**, $MaxLog$) = [−**Inf**, 88.72)S, [−**Inf**, 709.78)D

Range:	[0, **Inf**)	(See note 4)
Primary Domain:	$[-Ln2/2, Ln2/2)$	= [−0.3466, 0.3466)

Exceptions

All arguments outside the domain generate an **Inf**.

Propagated Error

$A = Xe^X, \qquad R = X$

Generated Error

Primary Domain Error:	**MRE**	**RMSRE**
Single Length:	0.51 ulp	0.21 ulp
Double Length:	0.5 ulp	0.21 ulp

The Algorithm

1 Set N = integer part of $X/\ln(2)$.

2 Compute the remainder of X by ln(2), using extended precision arithmetic.

3 Convert the remainder to fixed-point, compute its exponential using a fixed-point rational function, and convert the result back to floating point.

4 Increase the exponent of the result by N. If N is sufficiently negative the result must be denormalised.

Notes

1) $MaxLog$ is $\log_e(XMAX)$.

2) The analytical properties of e^x make the relative error of the result proportional to the absolute error of the argument. Thus the accuracy of step 2, which prepares the argument for the rational approximation, is crucial to the performance of the subroutine. It is completely accurate when $N = 0$, i.e. in the primary domain, and becomes less accurate as the magnitude of N increases. Since N can attain larger negative values than positive ones, **EXP** is least accurate for large, negative arguments.

3) For sufficiently negative arguments (below −87.34 for single-length and below −708.4 for double-length) the output is denormalised, and so the floating-point number contains progressively fewer significant digits, which degrades the accuracy. In such cases the error can theoretically be a factor of two.

4) Although the true exponential function is never zero, for large negative arguments the true result becomes too small to be represented as a floating-point number, and **EXP** underflows to zero. This occurs for arguments below −103.9 for single-length, and below −745.2 for double-length.

5) The propagated error is considerably magnified for large positive arguments, but diminished for large negative arguments.

POWER

```
REAL32 FUNCTION POWER (VAL REAL32 X, Y)
REAL32 FUNCTION DPOWER (VAL REAL64 X, Y)
```

These compute: X^Y

Domain:	[0, **Inf**] x [−**Inf**, **Inf**]
Range:	(−**Inf**, **Inf**)
Primary Domain:	See note 3.

Exceptions

If the first argument is outside its domain, **undefined.NaN** is returned. If the true value of X^Y exceeds $XMAX$, **Inf** is returned. In certain other cases other **NaNs** are produced: See note 2.

Propagated Error

$A = YX^Y(1 \pm \log_e(X))$, $R = Y(1 \pm \log_e(X))$ (See note 4)

Generated Error

Example Range Error:	MRE	RMSRE	(See note 3)
Single Length:	1.0 ulp	0.24 ulp	
Double Length:	13.2 ulp	1.73 ulp	

The Algorithm

Deal with special cases: either argument = 1, 0, +**Inf** or −**Inf** (see note 2). Otherwise:

(a) For single precision:

1 Compute $L = \log_2(X)$ in fixed point, where X is the first argument.

2 Compute $W = Y \times L$ in double precision, where Y is the second argument.

3 Compute 2^W in fixed point and convert to floating-point result.

(b) For double precision:

1 Compute $L = \log_2(X)$ in extended precision, where X is the first argument.

2 Compute $W = Y \times L$ in extended precision, where Y is the second argument.

3 Compute $RESULT = 2^W$ in extended precision.

Notes

1) This subroutine implements the mathematical function x^y to a much greater accuracy than can be attained using the **ALOG** and **EXP** functions, by performing each step in higher precision.

2) Results for special cases are as follows:

First Input (X)	Second Input (Y)	Result
< 0	ANY	**undefined.NaN**
0	≤ 0	**undefined.NaN**
0	$0 < Y \leq XMAX$	0
0	**Inf**	**unstable.NaN**
$0 < X < 1$	**Inf**	0
$0 < X < 1$	**–Inf**	**Inf**
1	$-XMAX \leq Y \leq XMAX$	1
1	± **Inf**	**unstable.NaN**
$1 < X \leq XMAX$	**Inf**	**Inf**
$1 < X \leq XMAX$	**–Inf**	0
Inf	1 $\leq Y \leq$ **Inf**	**Inf**
Inf	**–Inf**$\leq Y \leq -1$	0
Inf	$-1 < Y < 1$	**undefined.NaN**
otherwise	0	1
otherwise	1	X

3) Performing all the calculations in extended precision makes the double-precision algorithm very complex in detail, and having two arguments makes a primary domain difficult to specify. As an indication of accuracy, the functions were evaluated at 100 000 points logarithmically distributed over (0.1, 10.0), with the exponent linearly distributed over (–35.0, 35.0) (single-length), and (–300.0, 300.0) (double-length), producing the errors given above. The errors are much smaller if the exponent range is reduced.

4) The error amplification factors are calculated on the assumption that the relative error in Y is ± that in X, otherwise there would be separate factors for both X and Y. It can be seen that the propagated error will be greatly amplified whenever $\log_e(X)$ or Y is large.

SIN

```
REAL32 FUNCTION SIN (VAL REAL32 X)
REAL64 FUNCTION DSIN (VAL REAL64 X)
```

These compute: sine(X)   (where X is in radians)

Domain:	$[-Smax, Smax]$	= [−12868.0, 12868.0]S,
		$[-2.1 * 10^8, 2.1 * 10^8]$D
Range:	[−1.0, 1.0]	
Primary Domain:	$[-Pi/2, Pi/2]$	= [−1.57, 1.57]

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except ±**Inf**, which generates an **undefined.NaN**.

Propagated Error

$A = X\cos(X)$, $R = X\cot(X)$

Generated Error (See note 3)

Range:	Primary Domain		[0, 2*Pi*]	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length:	0.65 ulp	0.22 ulp	0.74 ulp	0.18 ulp
Double Length:	0.56 ulp	0.21 ulp	0.64 ulp	0.16 ulp

The Algorithm

1 Set N = integer part of $|X|/Pi$.

2 Compute the remainder of $|X|$ by Pi, using extended precision arithmetic.

3 Convert the remainder to fixed-point, compute its sine using a fixed-point rational function, and convert the result back to floating point.

4 Adjust the sign of the result according to the sign of the argument and the evenness of N.

Notes

1) For arguments outside the primary domain the accuracy of the result depends crucially on step 2. The extended precision corresponds to K extra bits in the representation of π ($K = 8$ for single-length and 12 for double-length). If the argument agrees with an integer multiple of π to more than K bits there is a loss of significant bits in the remainder, equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The extra precision of step 2 is lost if N becomes too large, and the

cut-off $Smax$ is chosen to prevent this. In any case for large arguments the 'granularity' of floating-point numbers becomes a significant factor. For arguments larger than $Smax$ a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

3) The propagated error has a complex behaviour. The propagated relative error becomes large near each zero of the function (outside the primary range), but the propagated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

4) Since only the remainder of X by Pi is used in step 3, the symmetry $\sin(x + n\pi) = \pm\sin(x)$ is preserved, although there is a complication due to differing precision representations of π.

5) The output range is not exceeded. Thus the output of `SIN` is always a valid argument for `ASIN`.

COS

```
REAL32 FUNCTION COS (VAL REAL32 X)
REAL64 FUNCTION DCOS (VAL REAL64 X)
```

These compute: cosine (X) (where X is in radians)

Domain:	$[-Smax, Smax]$	$= [-12868.0, 12868.0]$S, $[-2.1 * 10^8, 2.1 * 10^8]$D
Range:	$[-1.0, 1.0]$	
Primary Domain:	See note 1.	

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except **±Inf**, which generates an **undefined.NaN**.

Propagated Error

$A = -X\sin(X)$, $R = -X\tan(X)$ (See note 4)

Generated Error

Range:	[0, $Pi/4$)		[0, $2Pi$]	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length:	1.0 ulp	0.28 ulp	0.81 ulp	0.17 ulp
Double Length:	0.93 ulp	0.26 ulp	0.76 ulp	0.18 ulp

The Algorithm

1 Set N = integer part of $(|X| + Pi/2)/Pi$.

2 Compute the remainder of $(|X| + Pi/2)$ by Pi, using extended precision arithmetic.

3 Compute the remainder to fixed-point, compute its sine using a fixed-point rational function, and convert the result back to floating point.

4 Adjust the sign of the result according to the evenness of N.

Notes

1) Inspection of the algorithm shows that argument reduction always occurs, thus there is no 'primary domain' for COS. So for all arguments the accuracy of the result depends crucially on step 2. The extended precision corresponds to K extra bits in the representation of π ($K = 8$ for single-length and 12 for double length). If the argument agrees with an odd integer multiple of $\pi/2$ to more than K bits there is a loss of significant bits in the remainder, equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The extra precision of step 2 is lost if N becomes too large, and the cut-off $Smax$ is chosen to prevent this. In any case for large arguments the 'granularity' of floating-point numbers becomes a significant factor. For arguments larger than $Smax$ a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

3) For small arguments the errors are not evenly distributed. As the argument becomes smaller there is an increasing bias towards negative errors (which is to be expected from the form of the Taylor series). For the single-length version and X in [−0.1, 0.1], 62% of the errors are negative, whilst for X in [−0.01, 0.01], 70% of them are.

4) The propagated error has a complex behaviour. The propagated relative error becomes large near each zero of the function, but the propa-

gated absolute error only becomes large for large arguments. In effect, the error is seriously amplified only in an interval about each irrational zero, and the width of this interval increases roughly in proportion to the size of the argument.

5) Since only the remainder of $(|X| + Pi/2)$ by Pi is used in step 3, the symmetry $\cos(x+n\pi) = \pm\cos(x)$ is preserved. Moreover, since the same rational approximation is used as in `SIN`, the relation $\cos(x) = \sin(x+\pi/2)$ is also preserved. However, in each case there is a complication due to the different precision representations of π.

6) The output range is not exceeded. Thus the output of `COS` is always a valid argument for `ACOS`.

TAN

```
REAL32 FUNCTION TAN (VAL REAL32 X)
REAL64 FUNCTION DTAN (VAL REAL64 X)
```

These compute: $\tan(X)$ (where X is in radians)

Domain:	$[-Tmax, Tmax]$	$= [-6434.0, 6434.0]$S
		$[-1.05 * 10^8, 1.05 * 10^8]$D
Range:	(−**Inf**, **Inf**)	
Primary Domain:	$[-Pi/4, Pi/4]$	$= [-0.785, 0.785]$

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except ±**Inf**, which generate an **undefined.NaN**. Odd integer multiples of $\pi/2$ may produce **unstable.NaN**.

Propagated Error

$A = X(1 + \tan^2(X))$, $R = X(1 + \tan^2(X))/\tan(X)$ (See note 4)

Generated Error

Primary Domain Error:	MRE	RMSRE
Single Length:	3.5 ulp	0.23 ulp
Double Length:	0.69 ulp	0.23 ulp

The Algorithm

1 Set N = integer part of $X/(Pi/2)$.

2 Compute the remainder of X by $Pi/2$, using extended precision arithmetic.

3 Convert the remainder to fixed-point, compute its tangent using a fixed-point rational function, and convert the result back to floating point.

4 If N is odd, take the reciprocal.

5 Set the sign of the result according to the sign of the argument.

Notes

1) R is large whenever X is near to an integer multiple of $\pi/2$, and so tan is very sensitive to small errors near its zeros and singularities. Thus for arguments outside the primary domain the accuracy of the result depends crucially on step 2. The extended precision corresponds to K extra bits in the representation of $\pi/2$ ($K = 8$ for single-length and 12 for double-length). If the argument agrees with an integer multiple of $\pi/2$ to more than K bits there is a loss of significant bits in the remainder, approximately equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

2) The extra precision of step 2 is lost if N becomes too large, and the cut-off $Tmax$ is chosen to prevent this. In any case for large arguments the 'granularity' of floating-point numbers becomes a significant factor. For arguments larger than $Tmax$ a change in the argument of 1 ulp would change more than half of the significant bits of the result, and so the result is considered to be essentially indeterminate.

3) Step 3 of the algorithm has been slightly modified in the double-precision version from that given in Cody & Waite to avoid fixed-point underflow in the polynomial evaluation for small arguments.

4) Tan is quite badly behaved with respect to errors in the argument. Near its zeros outside the primary domain the relative error is greatly magnified, though the absolute error is only proportional to the size of the argument. In effect, the error is seriously amplified in an interval about each irrational zero, whose width increases roughly in proportion to the size of the argument. Near its singularities both absolute and relative errors become large, so any large output from this function is liable to be seriously contaminated with error, and the larger the argument, the smaller the maximum output which can be trusted. If step 4 of the algorithm requires division by zero, an **unstable.NaN** is produced instead.

5) Since only the remainder of X by $Pi/2$ is used in step 3, the symmetry $\tan(x + n\pi) = \tan(x)$ is preserved, although there is a complication due

to the differing precision representations of π. Moreover, by step 4 the symmetry $\tan(x) = 1/\tan(\pi/2 - x)$ is also preserved.

ASIN

```
REAL32 FUNCTION ASIN (VAL REAL32 X)
REAL64 FUNCTION DASIN (VAL REAL64 X)
```

These compute: $\text{sine}^{-1}(X)$ (in radians)

Domain:	$[-1.0, 1.0]$
Range:	$[-Pi/2, Pi/2]$
Primary Domain:	$[-0.5, 0.5]$

Exceptions

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

Propagated Error

$A = X/\sqrt{1 - X^2}$, $R = X/(\sin^{-1}(X)\sqrt{1 - X^2})$

Generated Error

	Primary Domain		$[-1.0, 1.0]$	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length:	0.53 ulp	0.21 ulp	1.35 ulp	0.33 ulp
Double Length:	2.8 ulp	1.4 ulp	2.34 ulp	0.64 ulp

The Algorithm

1 If $|X| > 0.5$, set $Xwork$:= `SQRT ((1 - |X|)/2)`.
Compute $Rwork = \text{arcsine}(-2 * Xwork)$ with a floating-point rational approximation, and set the result $= Rwork + Pi/2$.

2 Otherwise compute the result directly using the rational approximation.

3 In either case set the sign of the result according to the sign of the argument.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error: however since both

domain and range are bounded the *absolute* error in the result cannot be large.

2) By step 1, the identity $\sin^{-1}(x) = \pi/2 - 2\sin^{-1}(\sqrt{(1-x)/2})$ is preserved.

ACOS

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

These compute: $\text{cosine}^{-1}(X)$ (in radians)

Domain:	$[-1.0, 1.0]$
Range:	$[0, Pi]$
Primary Domain:	$[-0.5, 0.5]$

Exceptions

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

Propagated Error

$A = -X/\sqrt{1-X^2}$, $R = -X/(\sin^{-1}(X)\sqrt{1-X^2})$

Generated Error

	Primary Domain		$[-1.0, 1.0]$	
	MRE	**RMSRE**	**MAE**	**RMSAE**
Single Length:	1.1 ulp	0.38 ulp	2.4 ulp	0.61 ulp
Double Length:	1.3 ulp	0.34 ulp	2.9 ulp	0.78 ulp

The Algorithm

1 If $|X| > 0.5$, set $Xwork$:= SQRT ((1 − $|X|$)/2). Compute $Rwork$ = arcsine ($2 * Xwork$) with a floating-point rational approximation. If the argument was positive, this is the result, otherwise set the result = $Pi - Rwork$.

2 Otherwise compute $Rwork$ directly using the rational approximation. If the argument was positive, set result = $Pi/2 - Rwork$, otherwise result = $Pi/2 + Rwork$.

Notes

1) The error amplification factors are large only near the ends of the domain. Thus there is a small interval at each end of the domain in which the result is liable to be contaminated with error, although this interval is larger near 1 than near −1, since the function goes to zero with an infinite derivative there. However since both the domain and range are bounded the *absolute* error in the result cannot be large.

2) Since the rational approximation is the same as that in **ASIN**, the relation $\cos^{-1}(x) = \pi/2 - \sin^{-1}(x)$ is preserved.

ATAN

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

These compute: $\tan^{-1}(X)$ (in radians)

Domain:	[−**Inf**, **Inf**]
Range:	$[-Pi/2, Pi/2]$
Primary Domain:	$[-z, z]$, $z = 2 - \sqrt{3} = 0.2679$

Exceptions

None.

Propagated Error

$A = X/(1 + X^2)$, $R = X/(\tan^{-1}(X)(1 + X^2))$

Generated Error

Primary Domain Error:	MRE	RMSRE
Single Length:	0.53 ulp	0.21 ulp
Double Length:	1.27 ulp	0.52 ulp

The Algorithm

1 If $|X| > 1.0$, set $Xwork = 1/|X|$, otherwise $Xwork = |X|$.

2 If $Xwork > 2 - \sqrt{3}$, set $F = (Xwork * \sqrt{3} - 1)/(Xwork + \sqrt{3})$, otherwise $F = Xwork$.

3 Compute $Rwork = \arctan(F)$ with a floating-point rational approximation.

4 If $Xwork$ was reduced in (2), set $R = Pi/6 + Rwork$, otherwise $R = Rwork$.

5 If X was reduced in (1), set $RESULT = Pi/2 - R$, otherwise $RESULT = R$.

6 Set the sign of the $RESULT$ according to the sign of the argument.

Notes

1) For $|X| > ATmax$, $|\tan^{-1}(X)|$ is indistinguishable from $\pi/2$ in the floating-point format. For single-length, $ATmax = 1.68 * 10^7$, and for double-length $ATmax = 9 * 10^{15}$, approximately.

2) This function is numerically very stable, despite the complicated argument reduction. The worst errors occur just above $2 - \sqrt{3}$, but are no more than 1.8 ulp.

3) It is also very well behaved with respect to errors in the argument, i.e. the error amplification factors are always small.

4) The argument reduction scheme ensures that the identities $\tan^{-1}(X) = \pi/2 - \tan^{-1}(1/X)$, and
$\tan^{-1}(X) = \pi/6 + \tan^{-1}((\sqrt{3} * X - 1)/(\sqrt{3} + X))$ are preserved.

ATAN2

```
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:	$(-Pi, Pi]$
Primary Domain:	See note 2.

Exceptions

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

Propagated Error

$A = X(1 \pm Y)/(X^2 + Y^2)$, $R = X(1 \pm Y)/(\tan^{-1}(Y/X)(X^2 + Y^2))$ (See note 3)

Generated Error

See note 2.

The Algorithm

1 If X, the first argument, is zero, set the result to $\pm\pi/2$, according to the sign of Y, the second argument.

2 Otherwise set $Rwork$:= **ATAN** (Y/X). Then if $Y < 0$ set $RESULT = Rwork - Pi$, otherwise set $RESULT = Pi - Rwork$.

Notes

1) This two-argument function is designed to perform rectangular-to-polar co-ordinate conversion.

2) See the notes for **ATAN** for the primary domain and estimates of the generated error.

3) The error amplification factors were derived on the assumption that the relative error in Y is $\pm$ that in X, otherwise there would be separate factors for X and Y. They are small except near the origin, where the polar co-ordinate system is singular.

SINH

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

These compute: sinh(X)

Domain: $[-Hmax, Hmax]$ = $[-89.4, 89.4]$S, $[-710.5, 710.5]$D

Range: (−**Inf**, **Inf**)

Primary Domain: $(-1.0, 1.0)$

Exceptions

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

Propagated Error

$A = X\cosh(X)$, $R = X\coth(X)$ (See note 3)

Generated Error

	Primary Domain		[1.0, $XBig$] (See note 2)	
	MRE	**RMSRE**	**MRE**	**RMSRE**
Single Length:	0.89 ulp	0.3 ulp	0.98 ulp	0.31 ulp
Double Length:	1.3 ulp	0.51 ulp	1.0 ulp	0.3 ulp

The Algorithm

1 If $|X| > XBig$, set $Rwork$:= **EXP** ($|X| - ln(2)$).

2 If $XBig \geq |X| \geq 1.0$, set $temp$:= **EXP** ($|X|$), and set $Rwork = (temp - 1/temp)/2$.

3 Otherwise compute $Rwork = \sinh(|X|)$ with a fixed-point rational approximation.

4 In all cases, set $RESULT = \pm Rwork$ according to the sign of X.

Notes

1) $Hmax$ is the point at which $\sinh(X)$ becomes too large to be represented in the floating-point format.

2) $XBig$ is the point at which $e^{-|X|}$ becomes insignificant compared with $e^{|X|}$, (in floating-point). For single-length it is 8.32, and for double-length it is 18.37.

3) This function is quite stable with respect to errors in the argument. Relative error is magnified near zero, but the absolute error is a better measure near the zero of the function and it is diminished there. For large arguments absolute errors are magnified, but since the function is itself large, relative error is a better criterion, and relative errors are not magnified unduly for any argument in the domain, although the output does become less reliable near the ends of the range.

COSH

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

These compute: $\cosh(X)$

Domain: $[-Hmax, Hmax]$ = $[-89.4, 89.4]$S, $[-710.5, 710.5]$D

Range: [1.0, **Inf**)
Primary Domain: $[-XBig, XBig]$ = $[-8.32, 8.32]$S
$[-18.37, 18.37]$D

Exceptions

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

Propagated Error

$A = X\sinh(X)$, $R = X\tanh(X)$ (See note 3)

Generated Error

Primary Domain Error:	**MRE**	**RMS**
Single Length:	0.99 ulp	0.3 ulp
Double Length:	1.23 ulp	0.3 ulp

The Algorithm

1 If $|X| > XBig$, set $result$:= **EXP** ($|X| - \ln(2)$) .

2 Otherwise, set $temp$:= **EXP** ($|X|$) , and set $result = (temp + 1/temp)/2$.

Notes

1) $Hmax$ is the point at which $\cosh(X)$ becomes too large to be represented in the floating-point format.

2) $XBig$ is the point at which $e^{-|X|}$ becomes insignificant compared with $e^{|X|}$ (in floating-point).

3) Errors in the argument are not seriously magnified by this function, although the output does become less reliable near the ends of the range.

TANH

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

These compute: $\tanh(X)$

Domain:	[−Inf, Inf]
Range:	$[-1.0, 1.0]$
Primary Domain:	$[-Log(3)/2, Log(3)/2] =$ [−0.549, 0.549]

Exceptions

None.

Propagated Error

$A = X/\cosh^2(X)$, $R = X/\sinh(X)\cosh(X)$

Generated Error

Primary Domain Error:	MRE	RMS
Single Length:	0.52 ulp	0.2 ulp
Double Length:	4.6 ulp	2.6 ulp

The Algorithm

1 If $|X| > \ln(3)/2$, set $temp$:= **EXP** ($|X|/2$) . Then set $Rwork = 1 - 2/(1 + temp)$.

2 Otherwise compute $Rwork = \tanh(|X|)$ with a floating-point rational approximation.

3 In both cases, set $RESULT = \pm Rwork$ according to the sign of X.

Notes

1) As a floating-point number, $\tanh(X)$ becomes indistinguishable from its asymptotic values of ±1.0 for $|X| > HTmax$, where $HTmax$ is 8.4 for single-length, and 19.06 for double-length. Thus the output of **TANH** is equal to ±1.0 for such X.

2) This function is very stable and well-behaved, and errors in the argument are always diminished by it.

RAN

```
REAL32,INT32 FUNCTION RAN (VAL INT32 X)
REAL64,INT64 FUNCTION DRAN (VAL INT64 X)
```

These produce a pseudo-random sequence of integers, and a corresponding sequence of floating-point numbers between zero and one.

Domain: Integers (see note 1)
Range: [0.0, 1.0] x Integers

Exceptions

None.

The Algorithm

1 Produce the next integer in the sequence: $N_{k+1} = (aN_k + 1)_{mod\,M}$

2 Treat N_{k+1} as a fixed-point fraction in [0,1), and convert it to floating point.

3 Output the floating point result and the new integer.

Notes

1) This function has two results, the first a real, and the second an integer (both 32 bits for single-length, and 64 bits for double-length). The integer is used as the argument for the next call to **RAN**, i.e. it 'carries' the pseudo-random linear congruential sequence N_k, and it should be kept in scope for as long as **RAN** is used. It should be initialised before the first call to **RAN** but not modified thereafter except by the function itself.

2) If the integer parameter is initialised to the same value, the same sequence (both floating-point and integer) will be produced. If a different sequence is required for each run of a program it should be initialised to some 'random' value, such as the output of a timer.

3) The integer parameter can be copied to another variable or used in expressions requiring random integers. The topmost bits are the most random. A random integer in the range $[0, L]$ can conveniently be produced by taking the remainder by $(L+1)$ of the integer parameter shifted right by one bit. If the shift is not done an integer in the range $[-L, L]$ will be produced.

4) The modulus M is 2^{32} for single-length and 2^{64} for double-length, and the multipliers, a, have been chosen so that all M integers will be produced before the sequence repeats. However several different integers can produce the same floating-point value and so a floating-point output may be repeated, although the *sequence* of such will not be repeated until M calls have been made.

5) The floating-point result is uniformly distributed over the output range, and the sequence passes various tests of randomness, such as the 'run test', the 'maximum of 5 test' and the 'spectral test'.

6) The double-length version is slower to execute, but 'more random' than the single-length version. If a highly-random sequence of single-length numbers is required, this could be produced by converting the output of **DRAN** to single-length. Conversely if only a relatively crude sequence of double-length numbers is required, **RAN** could be used for higher speed and its output converted to double-length.

1.4 Host file server library

Library: `hostio.lib`

The host file server library contains routines that are used to communicate with the host file server. The routines are independent of the host on which the server is running. Using routines from this library you can guarantee that programs will be portable across all implementations of the toolset.

Constant and protocol definitions for the hostio library, including error and return codes, are provided in the include file **`hostio.inc`**. A listing of the file can be found in appendix C.

The **`result`** value from many of the routines in this library can take the value $\geq$ **`spr.operation.failed`** which is a server dependent failure result. It has been left open with the use of $\geq$ because future server implementations may give more information back via this byte.

1.4.1 Errors and the C run time library

The hostio routines use functions provided by the host file server. These are defined in appendix H. The server is implemented in C and uses routines in a C run time library, some of which are implementation dependent.

In particular, the hostio routines do not check the validity of stream identifiers, and the consequences of specifying an incorrect **`streamid`** may differ from system to system. For example, some systems may return an error tag, some may return a text message. If you use only those stream ids returned by the hostio routines that open files (**`so.open`**, **`so.open.temp`**, and **`so.popen.read`**), invalid ids are unlikely to occur.

It is also possible in rare circumstances for a program to fail altogether with an invalid streamid because of the way the C library is implemented on the system. This error can only occur if direct use of the library to perform the operation would produce the same error.

1.4.2 Inputting real numbers

Routines for inputting real numbers only accept numbers in the standard occam format for **`REAL`** numbers. Programs that allow other ways of specifying real numbers must convert to the occam format before presenting them to the library procedure.

For details of occam syntax for real numbers see the *'occam 2 Reference*

Manual'.

1.4.3 Procedure descriptions

In the procedure descriptions, `fs` is the channel *from* the host file server, and `ts` is the channel *to* the host file server. The `SP` protocol used by the host file server channels is defined in the include file `hostio.inc`, which is listed in appendix C.

The hostio routines are divided into six groups: five groups that reflect function and use, and a sixth miscellaneous group. The five specific groups are:

- File access and management
- General host access
- Keyboard input
- Screen output
- File output.

Each group of routines is described in a separate section. Each section begins with a list of the routines in the group with their formal parameters. This is followed by a description of each routine in turn.

Note: for those routines which write data to a stream (including the screen), if the data is not sent as an entire block then it cannot be guaranteed that the data arrives contiguously at its destination. This is because another process writing to the same destination may interleave its server request(s) with those of these routines.

1.4.4 File access routines

This group includes routines for managing file streams, for opening and closing files, and for reading and writing blocks of data.

Procedure	Parameter Specifiers
so.open	CHAN OF SP fs, ts, VAL []BYTE name, VAL BYTE type, mode, INT32 streamid, BYTE result
so.open.temp	CHAN OF SP fs, ts, VAL BYTE type, [so.temp.filename.length]BYTE filename, INT32 streamid, BYTE result
so.popen.read	CHAN OF SP fs, ts, VAL []BYTE filename, VAL []BYTE path.variable.name, VAL BYTE open.type, INT full.len, []BYTE full.name, INT32 streamid, BYTE result
so.close	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.read	CHAN OF SP fs, ts, VAL INT32 streamid, INT length, []BYTE data
so.write	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE data, INT length
so.gets	CHAN OF SP fs, ts, VAL INT32 streamid, INT length, []BYTE data, BYTE result
so.puts	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE data, BYTE result
so.flush	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result

Procedure	Parameter Specifiers

`so.seek`	`CHAN OF SP fs, ts,`
	`VAL INT32 streamid,`
	`VAL INT32 offset, origin, BYTE result`

`so.tell`	`CHAN OF SP fs, ts,`
	`VAL INT32 streamid,`
	`INT32 position, BYTE result`

`so.eof`	`CHAN OF SP fs, ts,` `VAL INT32 streamid, BYTE result`

`so.ferror`	`CHAN OF SP fs, ts,`
	`VAL INT32 streamid, INT32 error.no,`
	`INT length, []BYTE message,`
	`BYTE result`

`so.remove`	`CHAN OF SP fs, ts, VAL []BYTE name,` `BYTE result`

`so.rename`	`CHAN OF SP fs, ts,`
	`VAL []BYTE oldname, newname,`
	`BYTE result`

`so.test.exists`	`CHAN OF SP fs, ts,` `VAL []BYTE filename, BOOL exists`

Procedure definitions

`so.open`

```
PROC so.open  (CHAN OF SP fs, ts,
               VAL []BYTE name,
               VAL BYTE type, mode,
               INT32 streamid, BYTE result)
```

Opens the file given by `name` and returns a stream identifier `streamid` for all future operations on the file until it is closed. If `name` does not include a directory then the file is searched for in the current directory. File type is specified by `type` and the mode of opening by `mode`.

`type` can take the following values:

`spt.binary`	File contains raw bytes only.
`spt.text`	File contains text records separated by newline sequences.

mode can take the following values:

spm.input	Open existing file for reading.
spm.output	Open new file, or truncate an existing one, for writing.
spm.append	Open a new file, or append to an existing one, for writing.
spm.existing.update	Open an existing file for update (reading and writing), starting at beginning of the file.
spm.new.update	Open new file, or truncate existing one, for update.
spm.append.update	Open new file, or append to an existing one, for update.

result can take the following values:

spr.ok	The open was successful.
spr.bad.name	Null file name supplied.
spr.bad.type	Invalid file type.
spr.bad.mode	Invalid open mode.
spr.bad.packet.size	File name too large (i.e. > **sp.max.openname.size**)
≥ **spr.operation.failed**	If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.open.temp

```
PROC so.open.temp
        (CHAN OF SP fs, ts,
         VAL BYTE type,
         [so.temp.filename.length]BYTE filename,
         INT32 streamid, BYTE result)
```

Opens a temporary file in **spm.new.update** mode. The first filename tried is **temp00**. If the file already exists the **nn** suffix on the name **temp***nn* is incremented up to a maximum of 9999 until an unused number is found. If the number exceeds 2 digits the last character of **temp** is overwritten. For example: if the number exceeds 99 the **p** is overwritten , as in **tem999**; if the number exceeds 999, the **m** is overwritten, as in **te9999**. File type can be **spt.binary** or **spt.text**, as with **so.open**. The name of the file actually opened is returned in **filename**.

The result returned can take any of the following values:

spr.ok	The open was successful.
spr.notok	There are already 10,000 temporary files.
spr.bad.type	Invalid file type specified.
≥ spr.operation.failed	If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.popen.read

```
PROC so.popen.read
              (CHAN OF SP fs, ts,
               VAL []BYTE filename,
               VAL []BYTE path.variable.name,
               VAL BYTE open.type,
               INT full.len, []BYTE full.name,
               INT32 streamid, BYTE result)
```

As for **so.open**, but if the file is not found and the filename does not include a directory, the routine uses the directory path string associated with the host environment variable, given in **path.variable.name**, and performs a search in each directory in the path in turn. This corresponds to the searching rules used by the toolset, using the environment variable ISEARCH, see part 1, section 2.10.3.

File type can be **spt.binary** or **spt.text**, as with **so.open**. The mode of opening is always **spm.input**.

The name of the file opened is returned in **full.name**, and the length of the file name is returned in **full.len**. If no file is opened, **full.len** and **full.name** are undefined, and the result will not be **spr.ok**.

The result returned can take any of the following values:

`spr.ok`	The open was successful.
`spr.bad.name`	Null name supplied.
`spr.bad.type`	Invalid file type specified.
`spr.bad.packet.size`	File name too large (i.e. > `sp.max.openname.size`) or `path.variable.name` is too large (i.e. > `sp.max.getenvname.size`).
`spr.buffer.overflow`	The environment string referenced by `path.variable.name` is longer than 256 characters.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

`so.close`

```
PROC so.close (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               BYTE result)
```

Closes the stream identified by `streamid`.

The result returned can take any of the following values:

`spr.ok`	The close was successful.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.read

```
PROC so.read  (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               INT length, []BYTE data)
```

Reads a block of bytes from the specified stream up to a maximum given by the size of the array **data**. If **length** returned is not the same as the size of **data** then the end of the file has been reached or an error has occurred.

so.write

```
PROC so.write (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               VAL []BYTE data,
               INT length)
```

Writes a block of data to the specified stream. If **length** is less than the size of **data** then an error has occurred.

so.gets

```
PROC so.gets (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              INT length, []BYTE data,
              BYTE result)
```

Reads a line from the specified input stream. Characters are read until a newline sequence is found, the end of the file is reached, or **sp.max.readbuffer.size** characters have been read. The characters read are in the first **length** bytes of **data**. The newline sequence is not included in the returned array. If the read fails then either the end of file has been reached or an error has occurred.

The result returned can take any of the following values:

`spr.ok`	The read was successful.
`spr.bad.packet.size`	`data` is too large (> `sp.max.readbuffer.size`).
`spr.buffer.overflow`	The line was larger than the buffer `data` and has been truncated to fit.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.puts

```
PROC so.puts (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              VAL []BYTE data, BYTE result)
```

Writes a line to the specified output stream. A newline sequence is added to the end of the line. The size of `data` must be less than or equal to the hostio constant `sp.max.writebuffer.size`.

The result returned can take any of the following values:

`spr.ok`	The write was successful.
`spr.bad.packet.size`	`SIZE data` is too large (> `sp.max.writebuffer.size`).
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.flush

```
PROC so.flush (CHAN OF SP fs, ts,
               VAL INT32 streamid,
               BYTE result)
```

Flushes the specified output stream. All internally buffered data is written to the stream. Write and put operations that are directed to standard output are flushed automatically. The stream remains open.

The result returned can take any of the following values:

`spr.ok`	The flush was successful.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.seek

```
PROC so.seek (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              VAL INT32 offset, origin,
              BYTE result)
```

Sets the file position for the specified stream. A subsequent read or write will access data at the new position.

For a binary file the new position will be `offset` bytes from the position defined by `origin`. For a text file `offset` must be zero or a value returned by `so.tell`, in which case `origin` must be `spo.start`.

`origin` may take the following values:

`spo.start`	The start of the file.
`spo.current`	The current position in the file.
`spo.end`	The end of the file.

The result returned can take any of the following values:

`spr.ok`	The operation was successful.
`spr.bad.origin`	Invalid origin.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.tell

```
PROC so.tell (CHAN OF SP fs, ts,
              VAL INT32 streamid,
              INT32 position, BYTE result)
```

Returns the current file position for the specified stream.

The result returned can take any of the following values:

`spr.ok`	The operation was successful.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.eof

```
PROC so.eof (CHAN OF SP fs, ts,
             VAL INT32 streamid, BYTE result)
```

Tests whether the specified stream has reached the end of a file. The end of file is reached when a read operation attempts to read past the end of file.

The result returned can take any of the following values:

`spr.ok`	End of file has been reached.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2). This result will also be obtained if `eof` has not been reached.

so.ferror

```
PROC so.ferror (CHAN OF SP fs, ts,
                VAL INT32 streamid,
                INT32 error.no, INT length,
                []BYTE message, BYTE result)
```

Indicates whether an error has occurred on the specified stream. The integer `error.no` is a host defined error number. The returned message is in the first `length` bytes of `message`. `length` will be zero if no message can be provided. If the returned message is longer than 505 bytes then it is truncated to this size.

The result returned can take any of the following values:

`spr.ok`	An error has occurred on the specified stream.
`spr.buffer.overflow`	An error has occurred but the message is too large for `message` and has been truncated to fit.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2). This result will also be obtained if no error has occured on the specified stream.

`so.remove`

```
PROC so.remove (CHAN OF SP fs, ts,
                VAL []BYTE name, BYTE result)
```

Deletes the specified file.

The result returned can take any of the following values:

`spr.ok`	The delete was successful.
`spr.bad.name`	Null name supplied.
`spr.bad.packet.size`	`SIZE name` is too large (> `sp.max.removename.size`).
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

`so.rename`

```
PROC so.rename (CHAN OF SP fs, ts,
                VAL []BYTE oldname, newname,
                BYTE result)
```

Renames the specified file.

The result returned can take any of the following values:

`spr.ok`	The operation was successful.
`spr.bad.name`	Null name supplied.
`spr.bad.packet.size`	File names are too large (`SIZE name1 + SIZE name2` > `sp.max.renamename.size`).
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

`so.test.exists`

```
PROC so.test.exists (CHAN OF SP fs, ts,
                     VAL []BYTE filename,
                     BOOL exists)
```

Tests if the specified file exists. The value of **`exists`** is **`TRUE`** if the file exists, otherwise it is **`FALSE`**.

1.4.5 General host access

This group contains routines to access the host computer for system information and services.

Procedure	Parameter Specifiers
`so.commandline`	`CHAN OF SP fs, ts, VAL BYTE all, INT length, []BYTE string, BYTE result`
`so.parse.command.line`	`CHAN OF SP fs, ts, VAL [][]BYTE option.strings, VAL []INT option.parameters.required, []BOOL option.exists, [][2]INT option.parameters, INT error.len, []BYTE line`
`so.getenv`	`CHAN OF SP fs, ts, VAL []BYTE name, INT length, []BYTE value, BYTE result`
`so.time`	`CHAN OF SP fs, ts, INT32 localtime, UTCtime`

Procedure	Parameter Specifiers

`so.system`	`CHAN OF SP fs, ts,`
	`VAL []BYTE command,`
	`INT32 status, BYTE result`

`so.exit`	`CHAN OF SP fs, ts,` `VAL INT32 status`

`so.core`	`CHAN OF SP fs, ts,`
	`VAL INT32 offset,`
	`INT bytes.read,`
	`[]BYTE data, BYTE result`

`so.version`	`CHAN OF SP fs, ts,` `BYTE version, host, os, board`

Procedure definitions

`so.commandline`

```
PROC so.commandline (CHAN OF SP fs, ts,
                     VAL BYTE all, INT length,
                     []BYTE string, BYTE result)
```

Returns the command line passed to the server when it was invoked. If `all` has the value `sp.short.commandline` then all valid server options and their arguments are stripped from the command line, as is the server command name. If `all` is `sp.whole.commandline` then the command line is returned exactly as it was invoked. The returned command line is in the first `length` bytes of `string`. If the command line string is longer than 509 bytes then it is truncated to this size.

The result returned can take any of the following values:

`spr.ok`	The operation was successful.
`spr.buffer.overflow`	Command line too long for `string` and has been truncated to fit.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.parse.command.line

```
PROC so.parse.command.line
          (CHAN OF SP fs, ts,
           VAL [][]BYTE option.strings,
           VAL []INT option.parameters.required,
           []BOOL option.exists,
           [][2]INT option.parameters,
           INT error.len, []BYTE line)
```

This procedure reads the server command line and parses it for specified options and associated parameters.

The parameter **option.strings** contains a list of all the possible options and must be in upper case. Options may be any length up to 256 bytes and when entered on the command line may be either upper or lower case.

To read a parameter that has no preceding option (such as a file name) then the first option string should be empty (contain only spaces). For example, consider a program to be supplied with a file name, and any of three options 'A', 'B' and 'C'. The array **option.strings** would look like this:

```
VAL option.strings IS [" ", "A", "B", "C"]:
```

The parameter **option.parameters.required** indicates if the corresponding option (in **option.strings**) requires a parameter. The values it may take are:

spopt.never	Never takes a parameter.
spopt.maybe	Optionally takes a parameter.
spopt.always	Must take a parameter.

Continuing the above example, if the file name must be supplied and none of the options take parameters, except for 'C', which may or may not have a parameter, then **option.parameters.required** would look like this:

```
VAL option.parameters.required IS
    [spopt.always, spopt.never,
     spopt.never, spopt.maybe]:
```

If an option was present on the command line the corresponding element of **option.exists** is set to **TRUE**, otherwise it is set to **FALSE**.

If an option was followed by a parameter then the position in the array **line** where the parameter starts and the length of the parameter are given by the first and second elements respectively in the corresponding element in **option.parameters**.

If an error occurs whilst the command line is being parsed then **error.len** will be greater than zero and **line** will contain an error message of the given length. If no error occurs then **line** will contain the command line as supplied by the host file server.

Most of the possible error messages are self-explanatory, however, it is worth noting the meaning of the error '**Command line error: called incorrectly**'. This error means that either **option.strings** was null or that **SIZE option.exists**, **SIZE option.parameters** or **SIZE option.parameters.required** does not equal **SIZE option.strings**.

so.getenv

```
PROC so.getenv (CHAN OF SP fs, ts,
                VAL []BYTE name,
                INT length, []BYTE value,
                BYTE result)
```

Returns the string defined for the host environment variable **name**. The returned string is in the first **length** bytes of **value**. If **name** is not defined on the system **result** takes the value ≥ **spr.operation.failed**. If the environment variable's string is longer than 509 bytes then it is truncated to this size.

The result returned can take any of the following values:

spr.ok	The operation was successful.
spr.bad.name	The specified name is a null string.
spr.bad.packet.size	**SIZE name** is too large (> **sp.max.getenvname.size**).
spr.buffer.overflow	Environment string too large for **value** but has been truncated to fit.
≥ **spr.operation.failed**	If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.time

```
PROC so.time (CHAN OF SP fs, ts,
              INT32 localtime, UTCtime)
```

Returns the local time and Coordinated Universal Time. Both times are expressed as the number of seconds that have elapsed since midnight on 1st January, 1970. If UTC time is unavailable then it will have a value of zero. The times are given as unsigned **INT32**s.

so.system

```
PROC so.system (CHAN OF SP fs, ts,
                VAL []BYTE command,
                INT32 status, BYTE result)
```

Passes the string **command** to the host command processor for execution. If the command string is of zero length **result** takes the value **spr.ok** if there is a host command processor, otherwise an error is returned. If **command** is non-zero in length then **status** contains the host-specified value of the command, otherwise it is undefined.

The result returned can take any of the following values:

spr.ok	Host command processor exists.
spr.bad.packet.size	The array **command** is too large (> **sp.max.systemcommand.size**).
≥ **spr.operation.failed**	If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.exit

```
PROC so.exit (CHAN OF SP fs, ts,
              VAL INT32 status)
```

Terminates the server, which returns the value of **status** to its caller. If **status** has the special value **sps.success** then the server will terminate with a host specific 'success' result. If **status** has the special value **sps.failure** then the server will terminate with a host specific 'failure' result.

so.core

```
PROC so.core (CHAN OF SP fs, ts,
              VAL INT32 offset, INT bytes.read,
              []BYTE data, BYTE result)
```

Returns the contents of the root transputer's memory as peeked from the transputer when `iserver` is invoked with the analyse ('SA') option. The start of the memory segment is given by `offset` which is an offset from the base of memory (and is therefore positive). The number of bytes to be read is given by the size of the `data` vector. The number of bytes actually read into `data` is given by `bytes.read`. An error is returned if `offset` is larger than the total amount of peeked memory.

The result returned can take any of the following values:

`spr.ok`	The operation was successful.
`spr.bad.packet.size`	The array `data` is too large (> `sp.max.corerequest.size`).
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

This procedure can also be used to determine whether the memory was peeked (whether the server was invoked with the 'SA' option), by specifying a size of zero for `data` and `offset`. If the result returned is `spr.ok` the memory was peeked.

so.version

```
PROC so.version (CHAN OF SP fs, ts,
                 BYTE version, host, os, board)
```

Returns version information about the server and the host on which it is running. A value of zero for any of the items indicates that the information is unavailable.

The version of the server is given by `version`. The value should be divided by ten to yield the true version number. For example, a value of 15 means version 1.5.

The host machine type is given by `host`, and can take any of the following values:

`sph.PC`	IBM PC
`sph.S370`	IBM 370 Architecture
`sph.NECPC`	NEC PC
`sph.VAX`	DEC VAX
`sph.SUN3`	Sun Microsystems Sun 3
`sph.BOX.SUN4`	Sun Microsystems Sun 4
`sph.BOX.SUN386`	Sun Microsystems Sun 386i
`sph.BOX.APOLLO`	Apollo

Values up to 127 are reserved for use by INMOS.

The host operating system is given by **os**, and can take any of the following values:

`spo.DOS`	DOS
`spo.HELIOS`	HELIOS
`spo.VMS`	VMS
`spo.SUNOS`	SunOS
`spo.CMS`	CMS

Values up to 127 are reserved for use by INMOS.

The interface board type is given by **board**, and can take any of the following values:

`spb.B004`	IMS B004
`spb.B008`	IMS B008
`spb.B010`	IMS B010
`spb.B011`	IMS B011
`spb.B014`	IMS B014
`spb.B015`	IMS B015
`spb.B016`	IMS B016
`spb.DRX11`	DRX-11
`spb.IBMCAT`	CAT
`spb.QT0`	Caplin QT0
`spb.UDPLINK`	UDPlink

Values up to 127 are reserved for use by INMOS.

1.4.6 Keyboard input

Procedure	Parameter Specifiers
`so.pollkey`	`CHAN OF SP fs, ts, BYTE key, result`
`so.getkey`	`CHAN OF SP fs, ts, BYTE key, result`
`so.read.line`	`CHAN OF SP fs, ts, INT len, []BYTE line, BYTE result`
`so.read.echo.line`	`CHAN OF SP fs, ts, INT len, []BYTE line, BYTE result`
`so.ask`	`CHAN OF SP fs, ts, VAL []BYTE prompt, replies, VAL BOOL display.possible.replies, VAL BOOL echo.reply, INT reply.number`
`so.read.echo.int`	`CHAN OF SP fs, ts, INT n, BOOL error`
`so.read.echo.int32`	`CHAN OF SP fs, ts, INT32 n, BOOL error`
`so.read.echo.int64`	`CHAN OF SP fs, ts, INT64 n, BOOL error`
`so.read.echo.hex.int`	`CHAN OF SP fs, ts, INT n, BOOL error`
`so.read.echo.hex.int32`	`CHAN OF SP fs, ts, INT32 n, BOOL error`
`so.read.echo.hex.int64`	`CHAN OF SP fs, ts, INT64 n, BOOL error`
`so.read.echo.any.int`	`CHAN OF SP fs, ts, INT n, BOOL error`
`so.read.echo.real32`	`CHAN OF SP fs, ts, REAL32 n, BOOL error`
`so.read.echo.real64`	`CHAN OF SP fs, ts, REAL64 n, BOOL error`

Procedure definitions

so.pollkey

```
PROC so.pollkey (CHAN OF SP fs, ts,
                 BYTE key, result)
```

Reads a single character from the keyboard. If no key is available then it returns immediately with ≥ **spr.operation.failed**. The key is not echoed on the screen.

The result returned can take any of the following values:

spr.ok	A key was available and has been returned in **key**.
≥ **spr.operation.failed**	If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.getkey

```
PROC so.getkey (CHAN OF SP fs, ts,
                BYTE key, result)
```

As **so.pollkey** but waits for a key if none is available.

so.read.line

```
PROC so.read.line (CHAN OF SP fs, ts, INT len,
                   []BYTE line, BYTE result)
```

Reads a line of text from the keyboard, without echoing it on the screen. The characters read are in the first **len** bytes of **line**. The line is read until 'RETURN' is pressed at the keyboard. The line is truncated if **line** is not large enough. A newline or carriage return is not included in **line**.

The result returned can take any of the following values:

spr.ok	The read was successful.
≥ **spr.operation.failed**	If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.read.echo.line

```
PROC so.read.echo.line (CHAN OF SP fs, ts,
                        INT len, []BYTE line,
                        BYTE result)
```

As `so.read.line`, but user input (except newline or carriage return) is echoed on the screen.

so.ask

```
PROC so.ask (CHAN OF SP fs, ts,
             VAL []BYTE prompt, replies,
             VAL BOOL display.possible.replies,
             VAL BOOL echo.reply,
             INT reply.number)
```

Prompts on the screen for a user response on the keyboard. The prompt is specified by the string `prompt`, and the list of permitted relies by the string `replies`. Only single character responses are permitted, and alphabetic characters are *not* case sensitive. For example if the permitted responses are 'Y', 'N' and 'Q' then the `replies` string would contain the characters `"YNQ"`, and '`y`', '`n`' and '`q`' would also be accepted. `reply.number` indicates which response was typed, numbered from zero. " ? " is automatically output at the end of the prompt.

If `display.possible.replies` is `TRUE` the permitted replies are displayed on the screen. If `echo.reply` is `TRUE` the user's response is displayed.

The procedure will not return until a valid response has been typed.

so.read.echo.int

```
PROC so.read.echo.int (CHAN OF SP fs, ts, INT n,
                       BOOL error)
```

Reads a decimal integer typed at the keyboard and displays it on the screen. The number must be terminated by 'RETURN'. The boolean `error` is set to `TRUE` if an invalid integer is typed, `FALSE` otherwise.

so.read.echo.int32

```
PROC so.read.echo.int32 (CHAN OF SP fs, ts,
                         INT32 n, BOOL error)
```

As `so.read.echo.int` but reads 32-bit numbers.

so.read.echo.int64

```
PROC so.read.echo.int64 (CHAN OF SP fs, ts,
                         INT64 n, BOOL error)
```

As `so.read.echo.int` but reads 64-bit numbers.

so.read.echo.hex.int

```
PROC so.read.echo.hex.int (CHAN OF SP fs, ts,
                           INT n, BOOL error)
```

As `so.read.echo.int` but reads a number in hexadecimal format. The number may be in lower or upper case but must be prefixed with either '#', or '$' which directly indicates a hexadecimal number, or '%', which means add `MOSTNEG INT` to the given hex (using modulo arithmetic). For example, on a 32-bit transputer `%70` is interpreted as `#80000070`, and on a 16-bit transputer as `#8070`. This is useful when specifying transputer addresses, which are signed and start at `MOSTNEG INT`.

so.read.echo.hex.int32

```
PROC so.read.echo.hex.int32 (CHAN OF SP fs, ts,
                             INT32 n, BOOL error)
```

As `so.read.echo.hex.int` but reads 32-bit numbers.

so.read.echo.hex.int64

```
PROC so.read.echo.hex.int64 (CHAN OF SP fs, ts,
                             INT64 n, BOOL error)
```

As `so.read.echo.hex.int` but reads 64-bit numbers.

so.read.echo.any.int

```
PROC so.read.echo.any.int (CHAN OF SP fs, ts,
                           INT n, BOOL error)
```

As `so.read.echo.int` but accepts numbers in either decimal or hexadecimal format. Hexadecimal numbers may be lower or upper case but must be prefixed with either '#' or '$' which specifies the number directly, or '%', which means add `MOSTNEG INT` to the given hex (using modulo arithmetic). For example, on a 32-bit transputer `%70` is interpreted as `#80000070`, and on a 16-bit transputer as `#8070`. This is useful when specifying transputer addresses, which are signed and start at `MOSTNEG INT`.

so.read.echo.real32

```
PROC so.read.echo.real32 (CHAN OF SP fs, ts,
                          REAL32 n, BOOL error)
```

Reads a real number typed at the keyboard and displays it on the screen. The number must conform to occam syntax and be terminated by 'RETURN'. The boolean variable `error` is set to `TRUE` if an invalid number is typed, `FALSE` otherwise.

so.read.echo.real64

```
PROC so.read.echo.real64 (CHAN OF SP fs, ts,
                          REAL64 n, BOOL error)
```

As `so.read.echo.real32` but for 64-bit real numbers.

1.4.7 Screen output

Procedure	Parameter Specifiers
`so.write.char`	`CHAN OF SP fs, ts,` `VAL BYTE char`
`so.write.nl`	`CHAN OF SP fs, ts`
`so.write.string`	`CHAN OF SP fs, ts,` `VAL []BYTE string`
`so.write.string.nl`	`CHAN OF SP fs, ts,` `VAL []BYTE string`
`so.write.int`	`CHAN OF SP fs, ts,` `VAL INT n, field`
`so.write.int32`	`CHAN OF SP fs, ts,` `VAL INT32 n, VAL INT field`
`so.write.int64`	`CHAN OF SP fs, ts,` `VAL INT64 n, VAL INT field`
`so.write.hex.int`	`CHAN OF SP fs, ts,` `VAL INT n, width`
`so.write.hex.int32`	`CHAN OF SP fs, ts,` `VAL INT32 n, VAL INT width`
`so.write.hex.int64`	`CHAN OF SP fs, ts,` `VAL INT64 n, VAL INT width`
`so.write.real32`	`CHAN OF SP fs, ts,` `VAL REAL32 r, VAL INT Ip, Dp`
`so.write.real64`	`CHAN OF SP fs, ts,` `VAL REAL64 r, VAL INT Ip, Dp`

Procedure definitions

`so.write.char`

```
PROC so.write.char (CHAN OF SP fs, ts,
                    VAL BYTE char)
```

Writes the single byte `char` to the screen.

so.write.nl

```
PROC so.write.nl (CHAN OF SP fs, ts)
```

Writes a newline sequence to the screen.

so.write.string

```
PROC so.write.string (CHAN OF SP fs, ts,
                      VAL []BYTE string)
```

Writes the string `string` to the screen.

so.write.string.nl

```
PROC so.write.string.nl (CHAN OF SP fs, ts,
                         VAL []BYTE string)
```

As `so.write.string`, but appends a newline sequence to the end of the string.

so.write.int

```
PROC so.write.int (CHAN OF SP fs, ts,
                   VAL INT n, field)
```

Writes the value `n` (of type `INT`) to the screen as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified `field` width. If the `field` width is too small for the number it is widened as necessary; a zero value for `field` specifies minimum width. A negative value for `field` is an error.

so.write.int32

```
PROC so.write.int32 (CHAN OF SP fs, ts,
                     VAL INT32 n, VAL INT field)
```

As `so.write.int` but for 32-bit integers.

so.write.int64

```
PROC so.write.int64 (CHAN OF SP fs, ts,
                     VAL INT64 n, VAL INT field)
```

As `so.write.int` but for 64-bit integers.

so.write.hex.int

```
PROC so.write.hex.int (CHAN OF SP fs, ts,
                       VAL INT n, width)
```

Writes the value **n** (of type **INT**) to the screen as hexadecimal ASCII digits, preceded by the '#' character. The number of characters printed is **width** + 1. If **width** is larger than the size of the number then the number is padded with leading '0's or 'F's as appropriate. If **width** is smaller than the size of the number, the number is truncated, from the left, to **width** digits. A negative value for **width** is an error.

so.write.hex.int32

```
PROC so.write.hex.int64 (CHAN OF SP fs, ts,
                         VAL INT32 n,
                         VAL INT width)
```

As **so.write.hex.int** but for 32-bit integers.

so.write.hex.int64

```
PROC so.write.hex.int64 (CHAN OF SP fs, ts,
                         VAL INT64 n,
                         VAL INT width)
```

As **so.write.hex.int** but for 64-bit integers.

so.write.real32

```
PROC so.write.real32 (CHAN OF SP fs, ts,
                      VAL REAL32 r,
                      VAL INT Ip, Dp)
```

Writes the value **r** (of type **REAL32**) to the screen as ASCII characters formatted using **Ip** and **Dp** as described under **REAL32TOSTRING** (see section 1.7).

Note : Due to fixed size internal buffers, this procedure will be invalid if the string representing the real number is longer than 24 characters. If this is a problem, it is suggested you write your own procedure to perform this function. The procedure should include a buffer set to the required size, a call to **REAL32TOSTRING**, followed by a call to **so.write**.

so.write.real64

```
PROC so.write.real64 (CHAN OF SP fs, ts,
                      VAL REAL64 r,
                      VAL INT Ip, Dp)
```

As so.write.real32 but for 64-bit real numbers. The formatting variables Ip and Dp are described under REAL32TOSTRING (see section 1.7).

Note : Due to fixed size internal buffers, this procedure will be invalid if the string representing the real number is longer than 30 characters. If this is a problem, it is suggested you write your own procedure to perform this function. The procedure should include a buffer set to the required size, a call to REAL64TOSTRING, followed by a call to so.write.

1.4.8 File output

These routines write characters and strings to a specified stream, usually a file. The result returned can take the values spr.ok, spr.notok or very rarely ≥ spr.operation.failed.

Procedure	Parameter Specifiers
so.fwrite.char	CHAN OF SP fs, ts, VAL INT32 streamid, VAL BYTE char, BYTE result
so.fwrite.nl	CHAN OF SP fs, ts, VAL INT32 streamid, BYTE result
so.fwrite.string	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE string, BYTE result
so.fwrite.string.nl	CHAN OF SP fs, ts, VAL INT32 streamid, VAL []BYTE string, BYTE result
so.fwrite.int	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT n, field, BYTE result
so.fwrite.int32	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT32 n, VAL INT field, BYTE result
so.fwrite.int64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT64 n, VAL INT field, BYTE result
so.fwrite.hex.int	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT n, width, BYTE result
so.fwrite.hex.int32	CHAN OF SP fs, ts, VAL INT32 streamid, n VAL INT width, BYTE result
so.fwrite.hex.int64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL INT64 n, VAL INT width, BYTE result
so.fwrite.real32	CHAN OF SP fs, ts, VAL INT32 streamid, VAL REAL32 r, VAL INT Ip, Dp, BYTE result
so.fwrite.real64	CHAN OF SP fs, ts, VAL INT32 streamid, VAL REAL64 r, VAL INT Ip, Dp, BYTE result

Procedure definitions

`so.fwrite.char`

```
PROC so.fwrite.char (CHAN OF SP fs, ts,
                     VAL INT32 streamid,
                     VAL BYTE char,
                     BYTE result)
```

Writes a single character to the specified stream. The result `spr.notok` will be returned if the character is not written.

`so.fwrite.nl`

```
PROC so.fwrite.nl (CHAN OF SP fs, ts,
                   VAL INT32 streamid,
                   BYTE result)
```

Writes a newline sequence to the specified stream.

If `result` takes a value $\geq$ `spr.operation.failed` then this denotes a server returned failure, details of which are documented in section C.1. (See also, section H.2.2).

`so.fwrite.string`

```
PROC so.fwrite.string (CHAN OF SP fs, ts,
                       VAL INT32 streamid,
                       VAL []BYTE string,
                       BYTE result)
```

Writes a string to the specified stream. The result `spr.notok` will be returned if not all the characters are written.

`so.fwrite.string.nl`

```
PROC so.fwrite.string.nl (CHAN OF SP fs, ts,
                          VAL INT32 streamid,
                          VAL []BYTE string,
                          BYTE result)
```

As `so.fwrite.string`, but appends a newline sequence to the end of the string.

The result returned can take any of the following values:

`spr.notok`	Not all of the characters were written.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

`so.fwrite.int`

```
PROC so.fwrite.int (CHAN OF SP fs, ts,
                    VAL INT32 streamid,
                    VAL INT n, field,
                    BYTE result)
```

Writes the value `n` (of type `INT`) to the specified stream as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified `field` width. If the `field` width is too small for the number it is widened as necessary; a zero value for `field` will give minimum width. A negative value for `field` is an error.

The result `spr.notok` will be returned if not all of the digits are written.

`so.fwrite.int32`

```
PROC so.fwrite.int32 (CHAN OF SP fs, ts,
                      VAL INT32 streamid,
                      VAL INT32 n, VAL INT field,
                      BYTE result)
```

As `so.fwrite.int` but for 32-bit integers.

`so.fwrite.int64`

```
PROC so.fwrite.int64 (CHAN OF SP fs, ts,
                      VAL INT32 streamid,
                      VAL INT64 n, VAL INT field,
                      BYTE result)
```

As `so.fwrite.int` but for 64-bit integers.

`so.fwrite.hex.int`

```
PROC so.fwrite.hex.int (CHAN OF SP fs, ts,
                        VAL INT32 streamid,
                        VAL INT n, width,
                        BYTE result)
```

Writes the value **n** (of type **INT**) to the specified stream as hexadecimal ASCII digits preceded by the '#' character. The number of characters printed is **width** + 1. If **width** is larger than the size of the number then the number is padded with leading '0's or 'F's as appropriate. If **width** is smaller than the size of the number, then the number is truncated, from the left, to **width** digits. A negative value for **width** is an error.

The result **spr.notok** will be returned if not all the characters are written.

so.fwrite.hex.int32

```
PROC so.fwrite.hex.int32 (CHAN OF SP fs, ts,
                          VAL INT32 streamid, n
                          VAL INT width,
                          BYTE result)
```

As **so.fwrite.hex.int** but for 32-bit integers.

so.fwrite.hex.int64

```
PROC so.fwrite.hex.int64 (CHAN OF SP fs, ts,
                          VAL INT32 streamid,
                          VAL INT64 n,
                          VAL INT width,
                          BYTE result)
```

As **so.fwrite.hex.int** but for 64-bit integers.

so.fwrite.real32

```
PROC so.fwrite.real32 (CHAN OF SP fs, ts,
                       VAL INT32 streamid,
                       VAL REAL32 r,
                       VAL INT Ip, Dp,
                       BYTE result)
```

Writes the value **r** (of type **REAL32**) to the specified stream as ASCII characters formatted using **Ip** and **Dp** as described under **REAL32TOSTRING** (see section 1.7).

The result **spr.notok** will be returned if not all the characters are written.

Note : Due to fixed size internal buffers, this procedure will be invalid if the string representing the real number is longer than 24 characters. If this is a problem, it is suggested you write your own procedure to perform this function. The procedure should include a buffer set to the required size, a call to **REAL32TOSTRING**, followed by a call to **so.write**.

so.fwrite.real64

```
PROC so.fwrite.real64 (CHAN OF SP fs, ts,
                       VAL INT32 streamid,
                       VAL REAL64 r,
                       VAL INT Ip, Dp,
                       BYTE result)
```

As **so.fwrite.real32** but for 64-bit real numbers. The formatting variables **Ip** and **Dp** are described under **REAL32TOSTRING** (see section 1.7).

Note : Due to fixed size internal buffers, this procedure will be invalid if the string representing the real number is longer than 30 characters. If this is a problem, it is suggested you write your own procedure to perform this function. The procedure should include a buffer set to the required size, a call to **REAL64TOSTRING**, followed by a call to **so.write**.

1.4.9 Miscellaneous commands

The miscellaneous group includes procedures for:

- Time and date processing
- Buffering and multiplexing

Time processing

Procedure	Parameter Specifiers
`so.time.to.date`	`VAL INT32 input.time,` `[so.date.len]INT date`

`so.date.to.ascii`	`VAL [so.date.len]INT date,`
	`VAL BOOL long.years,`
	`VAL BOOL days.first,`
	`[so.time.string.len]BYTE string`

`so.time.to.ascii`	`VAL INT32 time,`
	`VAL BOOL long.years,`
	`VAL BOOL days.first`
	`[so.time.string.len]BYTE string`

`so.today.date`	`CHAN OF SP fs, ts,` `[so.date.len]INT date`

`so.today.ascii`	`CHAN OF SP fs, ts,`
	`VAL BOOL long.years,`
	`VAL BOOL days.first,`
	`[so.time.string.len]BYTE string`

`so.time.to.date`

```
PROC so.time.to.date (VAL INT32 input.time,
                      [so.date.len]INT date)
```

Converts time (as supplied by **`so.time`**) to six integers, stored in the **`date`** array. The elements of the array are as follows:

Element of array	Data
0	Seconds past the minute
1	Minutes past the hour
2	The hour (24 hour clock)
3	The day of the month
4	The month (1 to 12)
5	The year (4 digits)

so.date.to.ascii

```
PROC so.date.to.ascii
                 (VAL [so.date.len]INT date,
                  VAL BOOL long.years,
                  VAL BOOL days.first,
                  [so.time.string.len]BYTE string)
```

Converts an array of six integers containing the date (as supplied by `so.time.to.date`) into an ASCII string of the form:

HH:MM:SS DD/MM/YYYY

If `long.years` is `FALSE` then year is reduced to two characters, and the last two characters of the year field are padded with spaces. If `days.first` is `FALSE` then the ordering of day and month is changed (to the U.S. standard).

so.time.to.ascii

```
PROC so.time.to.ascii
                 (VAL INT32 time,
                  VAL BOOL long.years,
                  VAL BOOL days.first
                  [so.time.string.len]BYTE string)
```

Converts time (as supplied by `so.time`) into an ASCII string, as described for `so.date.to.ascii`.

so.today.date

```
PROC so.today.date (CHAN OF SP fs, ts,
                    [so.date.len]INT date)
```

Gives today's date, in local time, as six integers, stored in the `date` array. The format of the array is the same as for `so.time.to.date`. If the date is unavailable all elements in `date` are set to zero.

so.today.ascii

```
PROC so.today.ascii
               (CHAN OF SP fs, ts,
                VAL BOOL long.years, days.first,
                [so.time.string.len]BYTE string)
```

Gives today's date, in local time, as an ASCII string, in the same format as procedure `so.date.to.ascii`. If the date is unavailable `string` is filled with spaces.

Buffers and multiplexors

This group of procedures are designed to assist with buffering and multiplexing data exchange between the program and host.

Procedure	Parameter Specifiers
`so.buffer`	`CHAN OF SP fs, ts, from.user, to.user, CHAN OF BOOL stopper`
`so.overlapped.buffer`	`CHAN OF SP fs, ts, from.user, to.user, CHAN OF BOOL stopper`
`so.multiplexor`	`CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper`
`so.overlapped.multiplexor`	`CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper, []INT queue`
`so.pri.multiplexor`	`CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper`
`so.overlapped.pri.multiplexor`	`CHAN OF SP fs, ts, []CHAN OF SP from.user, to.user, CHAN OF BOOL stopper, []INT queue`

so.buffer

```
PROC so.buffer (CHAN OF SP fs, ts,
                           from.user, to.user,
                CHAN OF BOOL stopper)
```

This procedure buffers data between the user and the host. It can be used by processes on a network to pass data to the host across intervening processes. It is terminated by sending either a **TRUE** or **FALSE** value on the channel **stopper**.

so.overlapped.buffer

```
PROC so.overlapped.buffer (CHAN OF SP fs, ts,
                                      from.user,
                                      to.user,
                           CHAN OF BOOL stopper)
```

Similar to **so.buffer**, but allows many host communications to occur simultaneously through a train of processes. This can improve efficiency if the communications pass through many processes before reaching the server. It is terminated by either a **TRUE** or **FALSE** value on the channel **stopper**.

so.multiplexor

```
PROC so.multiplexor (CHAN OF SP fs, ts,
                     []CHAN OF SP from.user,
                                  to.user,
                     CHAN OF BOOL stopper)
```

This procedure multiplexes any number of pairs of **SP** protocol channels onto a single pair of **SP** protocol channels, which may go to the file server or another **SP** protocol multiplexor (or buffer). It is terminated by sending either a **TRUE** or **FALSE** value on the channel **stopper**. For n channels, each channel is guaranteed to be able to pass on a message for every n messages that pass through the multiplexor. This is achieved by cycling the selection priority from the lowest index of **from.user**. However, **stopper** always has highest priority.

so.overlapped.multiplexor

```
PROC so.overlapped.multiplexor
                (CHAN OF SP fs, ts,
                 []CHAN OF SP from.user, to.user,
                 CHAN OF BOOL stopper,
                 []INT queue)
```

Similar to **so.multiplexor**, but can pipeline server requests. The number of requests than can be pipelined is determined by the size of **queue**, which must provide one word for each request that can be pipelined. If **SIZE queue** is zero then the routine simply waits for input from **stopper**. Pipelining improves efficiency if the server requests have to pass through many processes on the way to and from the server. It is terminated by sending either a **TRUE** or **FALSE** value on the channel **stopper**.

The multiplexing is done in the same cyclic manner as in **so.multiplexor**. **stopper** has higher priority than any of **from.user**.

so.pri.multiplexor

```
PROC so.pri.multiplexor
                (CHAN OF SP fs, ts,
                 []CHAN OF SP from.user, to.user,
                 CHAN OF BOOL stopper)
```

As **so.multiplexor** but the multiplexing is *not* done in a cyclic manner; rather there is a hierarchy of priorities amongst the channels **from.user**: **from.user [i]** is of higher priority than **from.user [j]**, for i < j. Also **stopper** is of lower priority than any of **from.user**.

so.overlapped.pri.multiplexor

```
PROC so.overlapped.pri.multiplexor
                (CHAN OF SP fs, ts,
                 []CHAN OF SP from.user, to.user,
                 CHAN OF BOOL stopper,
                 []INT queue)
```

As **so.overlapped.multiplexor** but the multiplexing is done in the same prioritized manner as in **so.pri.multiplexor**. **stopper** has higher priority than any of **from.user**.

1.5 Streamio library

Library: `streamio.lib`

The streamio library contains routines for reading and writing to files and to the terminal at a higher level of abstraction than the hostio library. The file `streamio.inc` defines the `KS` and `SS` protocols and constants used by the streamio library routines. The `result` value from many of the routines in this library can take a value ≥ `spr.operation.failed` which is a server dependent failure result. It has been left open with the use of ≥ because future server implementations may give more failure information back via this byte. Names for result values can be found in the file `hostio.inc`.

The streamio routines can be classified into three main groups:

- Stream processes
- Stream input procedures
- Stream output procedures.

Stream input and output procedures are used to input and output characters in keystream `KS` and screen stream `SS` protocols. `KS` and `SS` protocols must be converted to the server protocol before communicating with the host.

Stream processes convert streams from keyboard or screen protocol to the server protocol `SP` or to related data structures. They are used to transfer data from the stream input and output routines to the host. Stream processes can be run as parallel processes serving stream input and output routines called in sequential code. For example, the following code clears the screen of a terminal supporting ANSI escape sequences:

```
CHAN OF SS scrn :
PAR
  so.scrstream.to.ANSI(fs, ts, scrn)
  SEQ
   ss.goto.xy(scrn, 0, 0)
   ss.clear.eos(scrn)
   ss.write.endstream(scrn)
```

The key stream and screen stream protocols are identical to those used in the IMS D700 Transputer Development System (TDS) and facilitate the porting of programs between the TDS and the toolset.

1.5.1 Naming conventions

Procedure names always begin with a prefix derived from the first parameter. Stream processes, where the SP channel (listed first) is used in combination with either the KS or SS protocols, are prefixed with 'so.'. Stream input routines, which use only the KS protocol are prefixed with 'ks.', and stream output routines, which use only the SS protocol, are prefixed with 'ss.'. The single KS to SS conversion routine, which uses both protocols, is prefixed with 'ks.'.

1.5.2 Stream processes

Procedure	Parameter Specifiers
so.keystream.from.kbd	CHAN OF SP fs, ts, CHAN OF KS keys.out, CHAN OF BOOL stopper, VAL INT ticks.per.poll
so.keystream.from.file	CHAN OF SP fs, ts, CHAN OF KS keys.out, VAL []BYTE filename, BYTE result
so.keystream.from.stdin	CHAN OF SP fs, ts, CHAN OF KS keys.out, BYTE result
ks.keystream.sink	CHAN OF KS keys
ks.keystream.to.scrstream	CHAN OF KS keyboard, CHAN OF SS scrn
ss.scrstream.sink	CHAN OF SS scrn
so.scrstream.to.file	CHAN OF SP fs, ts, CHAN OF SS scrn, VAL []BYTE filename, BYTE result
so.scrstream.to.stdout	CHAN OF SP fs, ts, CHAN OF SS scrn, BYTE result

Procedure	Parameter Specifiers
`ss.scrstream.to.array`	`CHAN OF SS scrn, []BYTE buffer`
`ss.scrstream.from.array`	`CHAN OF SS scrn, VAL []BYTE buffer`
`ss.scrstream.fan.out`	`CHAN OF SS scrn, screen.out1, screen.out2`
`ss.scrstream.copy`	`CHAN OF SS scrn.in, scrn.out`
`so.scrstream.to.ANSI`	`CHAN OF SP fs, ts, CHAN OF SS scrn`
`so.scrstream.to.TVI920`	`CHAN OF SP fs, ts, CHAN OF SS scrn`
`ss.scrstream.multiplexor`	`[]CHAN OF SS screen.in, CHAN OF SS screen.out, CHAN OF INT stopper`

Procedure definitions

so.keystream.from.kbd

```
PROC so.keystream.from.kbd
                          (CHAN OF SP fs, ts,
                           CHAN OF KS keys.out,
                           CHAN OF BOOL stopper,
                           VAL INT ticks.per.poll)
```

Reads characters from the keyboard and outputs them one at a time as integers on the channel `keys.out`. It is terminated by sending either a `TRUE` or `FALSE` on the boolean channel `stopper`. The procedure polls the keyboard at an interval determined by the value of `ticks.per.poll`, in transputer clock cycles, unless keys are available, in which case they are read at full speed. It is an error if `ticks.per.poll` is less than or equal to zero.

After `FALSE` is sent on the channel `stopper` the procedure sends the negative value `ft.terminated` on `keys.out` to mark the end of the file.

so.keystream.from.file

```
PROC so.keystream.from.file
                    (CHAN OF SP fs, ts,
                     CHAN OF KS keys.out,
                     VAL []BYTE filename,
                     BYTE result)
```

Reads lines from the specified text file and outputs them on **keys.out**. Terminates automatically on error or when it has reached the end of the file and all the characters have been output on the **keys.out** channel. A '*c' is output to terminate a text line. The negative value **ft.terminated** is sent on the channel **keys.out** to mark the end of the file. The result returned can take any of the following values:

spr.ok	The operation was successful.
spr.bad.packet.size	Filename too large i.e. **SIZE filename** > **sp.max.openname.size**.
spr.bad.name	Null file name.
≥ spr.operation.failed	The open failed or reading the file failed. If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

so.keystream.from.stdin

```
PROC so.keystream.from.stdin
                    (CHAN OF SP fs, ts,
                     CHAN OF KS keys.out,
                     BYTE result)
```

As **so.keystream.from.file**, but reads from the standard input stream. The standard input stream is normally assigned to the keyboard, but can be redirected by the host operating system. End of file from keyboard will terminate this routine. The result returned may take any of the following values:

`spr.ok`	The operation was successful.
≥ `spr.operation.failed`	Reading standard input failed. If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

ks.keystream.sink

```
PROC ks.keystream.sink  (CHAN OF KS keys)
```

Reads word length quantities until `ft.terminated` is received, then terminates.

ks.keystream.to.scrstream

```
PROC ks.keystream.to.scrstream (CHAN OF KS
                                        keyboard,
                                 CHAN OF SS scrn)
```

Converts key stream protocol to screen stream protocol. The value `ft.terminated` on `keyboard` terminates the procedure.

ss.scrstream.sink

```
PROC ss.scrstream.sink (CHAN OF SS scrn)
```

Reads screen stream protocol and ignores it except for the stream terminator from `ss.write.endstream` which terminates the procedure.

so.scrstream.to.file

```
PROC so.scrstream.to.file (CHAN OF SP fs, ts,
                           CHAN OF SS scrn,
                           VAL []BYTE filename,
                           BYTE result)
```

Creates a new file with the specified name and writes the data sent on channel `scrn` to it. The `scrn` channel uses the screen stream protocol which is used by all the stream output library routines (and is the same as the inmos TDS screen stream protocol). It terminates on receipt of the stream terminator from `ss.write.endstream`, or on an error condition. The result returned can take any of the following values:

`spr.ok`	The data sent on `scrn` was successfully written to the file.
`spr.bad.packet.size`	Filename too large i.e. `SIZE filename` > `sp.max.openname.size`.
`spr.bad.name`	Null filename.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

If used in conjunction with **so.scrstream.fan.out** this procedure may be used to file a copy of everything sent to the screen.

so.scrstream.to.stdout

```
PROC so.scrstream.to.stdout (CHAN OF SP fs, ts,
                             CHAN OF SS scrn,
                             BYTE result)
```

Performs the same operation as **so.scrstream.to.file**, but writes to the standard output stream. The standard output stream goes to the screen, but can be redirected to a file by the host operating system. The result returned can take any of the following values:

`spr.ok`	The data sent on `scrn` was successfully written to standard output.
≥ `spr.operation.failed`	If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

ss.scrstream.to.array

```
PROC ss.scrstream.to.array (CHAN OF SS scrn,
                            []BYTE buffer)
```

Buffers a screen stream whose total size does not exceed the capacity of **buffer**, for debugging purposes or subsequent onward transmission using **so.scrstream.from.array**. The procedure terminates on receipt of the stream terminator from **ss.write.endstream**.

ss.scrstream.from.array

```
PROC ss.scrstream.from.array (CHAN OF SS scrn,
                              VAL []BYTE buffer)
```

Regenerates a screen stream buffered in **buffer** by a previous call of **so.scrstream.to.array**. Terminates when all buffered data has been sent.

ss.scrstream.fan.out

```
PROC ss.scrstream.fan.out
                    (CHAN OF SS scrn,
                                screen.out1,
                                screen.out2)
```

Sends copies of everything received on the input channel **scrn** to two output channels. The procedure terminates on receipt of the stream terminator from **ss.write.endstream** without passing on the terminator.

ss.scrstream.copy

```
PROC ss.scrstream.copy (CHAN OF SS scrn.in,
                                   scrn.out)
```

Copies screen stream protocol input on **scrn.in** to **scrn.out**. Terminates on receipt of the endstream terminator from **ss.write.endstream**, which is not passed on.

so.scrstream.to.ANSI

```
PROC so.scrstream.to.ANSI (CHAN OF SP fs, ts,
                           CHAN OF SS scrn)
```

Converts screen stream protocol into a stream of BYTEs according to the requirements of ANSI terminal screen protocol. Not all of the screen stream commands are supported.

The following tags are ignored:

st.ins.char, **st.reset**, **st.terminate**, **st.help**, **st.initialise**, **st.key.raw**, **st.key.cooked**, **st.release**, **st.claim**.

The procedure terminates on receipt of the stream terminator from **ss.write.endstream**.

`so.scrstream.to.TVI920`

```
PROC so.scrstream.to.TVI920 (CHAN OF SP fs, ts,
                             CHAN OF SS scrn)
```

Converts screen stream protocol into a stream of BYTEs according to the requirements of TVI 920 (and compatible) terminals. Not all of the screen stream commands are supported. The following tags are ignored:

`st.reset`, `st.terminate`, `st.help`, `st.initialise`, `st.key.raw`, `st.key.cooked`, `st.release`, `st.claim`.

The procedure terminates on receipt of the stream terminator from `ss.write.endstream`.

`ss.scrstream.multiplexor`

```
PROC ss.scrstream.multiplexor
                  ([]CHAN OF SS screen.in,
                   CHAN OF SS screen.out,
                   CHAN OF INT stopper)
```

This procedure multiplexes up to 256 screen stream channels onto a single screen stream channel. Each change of input channel directs output to the next line of the screen, and each such line is annotated at the left with the array index of the channel used followed by '>'. The tag `st.endstream` is ignored. The procedure is terminated by the receipt of any integer on the channel `stopper`. For n channels, each channel is guaranteed to be able to pass on a message for every n messages that pass through the multiplexor. This is achieved by cycling from the lowest index of `screen.in`. However, `stopper` always has highest priority.

1.5.3 Stream input

These routines read characters and strings from the input stream, in **KS** protocol.

Procedure	Parameter Specifiers
`ks.read.char`	`CHAN OF KS source, INT char`
`ks.read.line`	`CHAN OF KS source, INT len, []BYTE line, INT char`
`ks.read.int`	`CHAN OF KS source, INT number, char`
`ks.read.int64`	`CHAN OF KS source, INT64 number, INT char`
`ks.read.real32`	`CHAN OF KS source, REAL32 number, INT char`
`ks.read.real64`	`CHAN OF KS source, REAL64 number, INT char`

Procedure definitions

ks.read.char

```
PROC ks.read.char (CHAN OF KS source, INT char)
```

Returns in **char** the next word length quantity from **source**.

ks.read.line

```
PROC ks.read.line (CHAN OF KS source, INT len,
                   []BYTE line, INT char)
```

Reads text into the array **line** up to but excluding **'*c'**, or up to and excluding any error code. Any **'*n'** encountered is thrown away. **len** gives the number of characters in **line**. If there is an error its code is returned as **char**, otherwise the value of **char** will be **INT '*c'**. If the array is filled before a **'*c'** is encountered all further characters are ignored.

ks.read.int

```
PROC ks.read.int (CHAN OF KS source,
                  INT number, char)
```

Skips input up to a digit, #, + or –, then reads a sequence of digits to the first non-digit, returned as **char**, and converts the digits to an integer in **number**. **char** must be initialised to the first character of the input. If the first significant character is a '#' then a hexadecimal number is input, thereby allowing the user the option of which number base to use. The hexadecimal may be in upper or lower case. **char** is returned as **ft.number.error** if the number overflows the **INT** range.

ks.read.int64

```
PROC ks.read.int64 (CHAN OF KS source,
                    INT64 number, INT char)
```

As **ks.read.int**, but for 64-bit integers.

ks.read.real32

```
PROC ks.read.real32 (CHAN OF KS source,
                     REAL32 number, INT char)
```

Skips input up to a digit, + or –, then reads a sequence of digits with optional decimal point and exponent) up to the first invalid character, returned as **char**. Converts the digits to a floating point value in **number**. **char** must be initialised to the first character of the input. If there is an error in the syntax of the real, if it is ± infinity, or if more than 24 characters read then **char** is returned as **ft.number.error**.

ks.read.real64

```
PROC ks.read.real64 (CHAN OF KS source,
                     REAL64 number, INT char)
```

As **ks.read.real32**, but for 64-bit real numbers. Allows for reading up to 30 characters.

1.5.4 Stream output

These routines write text, numbers and screen control codes to an output stream in SS protocol.

Procedure	Parameter Specifiers
`ss.write.char`	`CHAN OF SS scrn,` `VAL BYTE char`
`ss.write.nl`	`CHAN OF SS scrn`
`ss.write.string`	`CHAN OF SS scrn,` `VAL []BYTE str`
`ss.write.endstream`	`CHAN OF SS scrn`
`ss.write.text.line`	`CHAN OF SS scrn,` `VAL []BYTE str`
`ss.write.int`	`CHAN OF SS scrn,` `VAL INT number, field`

`ss.write.int64`	`CHAN OF SS scrn,`
	`VAL INT64 number,`
	`VAL INT field`

`ss.write.hex.int`	`CHAN OF SS scrn,` `VAL INT number, field`

`ss.write.hex.int64`	`CHAN OF SS scrn,`
	`VAL INT64 number,`
	`VAL INT field`

Procedure	Parameter Specifiers

`ss.write.real32`	`CHAN OF SS scrn,`
	`VAL REAL32 number,`
	`VAL INT Ip, Dp`

`ss.write.real64`	`CHAN OF SS scrn,`
	`VAL REAL64 number,`
	`VAL INT Ip, Dp`

`ss.goto.xy`	`CHAN OF SS scrn,` `VAL INT x, y`
`ss.clear.eol`	`CHAN OF SS scrn`
`ss.clear.eos`	`CHAN OF SS scrn`
`ss.beep`	`CHAN OF SS scrn`
`ss.up`	`CHAN OF SS scrn`
`ss.down`	`CHAN OF SS scrn`
`ss.left`	`CHAN OF SS scrn`
`ss.right`	`CHAN OF SS scrn`
`ss.insert.char`	`CHAN OF SS scrn, VAL BYTE ch`
`ss.delete.chr`	`CHAN OF SS scrn`
`ss.delete.chl`	`CHAN OF SS scrn`
`ss.ins.line`	`CHAN OF SS scrn`
`ss.del.line`	`CHAN OF SS scrn`

Procedure definitions

`ss.write.char`

```
PROC ss.write.char (CHAN OF SS scrn,
                    VAL BYTE char)
```

Sends the ASCII value `char` on `scrn`, in `scrstream` protocol, to the current position in the output line.

`ss.write.nl`

```
PROC ss.write.nl (CHAN OF SS scrn)
```

Sends `"*c*n"` to `scrn`.

ss.write.string

```
PROC ss.write.string (CHAN OF SS scrn,
                      VAL[]BYTE str)
```

Sends all characters in **str** to **scrn**.

ss.write.endstream

```
PROC ss.write.endstream (CHAN OF SS scrn)
```

Sends a special stream terminator value to **scrn**.

ss.write.text.line

```
PROC ss.write.text.line (CHAN OF SS scrn,
                         VAL []BYTE str)
```

Sends all of **str** to **scrn** ensuring that, whether or not the last character of **str** is '***c**', the last two characters sent are "***c*n**".

ss.write.int

```
PROC ss.write.int (CHAN OF SS scrn,
                   VAL INT number, field)
```

Converts **number** into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified **field** width if necessary. If the number cannot be represented in **field** characters it is widened as necessary, a zero value for **field** will give minimum width. The converted number is sent to **scrn**. A negative value for **field** is an error.

ss.write.int64

```
PROC ss.write.int64 (CHAN OF SS scrn,
                     VAL INT64 number,
                     VAL INT field)
```

As **ss.write.int** but for 64-bit integers.

ss.write.hex.int

```
PROC ss.write.hex.int (CHAN OF SS scrn,
                       VAL INT number, field)
```

Converts **number** into a sequence of ASCII hexadecimal digits, using upper case letters, preceded by '#'. The total number of characters sent is always **field + 1**, padding out with '0' or 'F' on the left if necessary. The number is truncated at the left if the field is too narrow, thereby allowing the less significant part of any number to be printed. The converted number is sent to **scrn**. A negative value for **field** is an error.

ss.write.hex.int64

```
PROC ss.write.hex.int64 (CHAN OF SS scrn,
                         VAL INT64 number,
                         VAL INT field)
```

As **ss.write.hex.int** but for 64-bit integer values.

ss.write.real32

```
PROC ss.write.real32 (CHAN OF SS scrn,
                      VAL REAL32 number,
                      VAL INT Ip, Dp)
```

Converts **number** into an ASCII string formatted using **Ip** and **Dp**, as described for **REAL32TOSTRING**, (see section 1.7). The converted number is sent to **scrn**. If the formatted form of **number** is larger than 24 characters then this procedure acts as an invalid process.

ss.write.real64

```
PROC ss.write.real64 (CHAN OF SS scrn,
                      VAL REAL64 number,
                      VAL INT Ip, Dp)
```

As for **ss.write.real32** but for 64-bit real values. See section 1.7, **REAL32TOSTRING** for the details of the formatting effect of **Ip** and **Dp**. If the formatted form of **number** is larger than 30 characters then this procedure acts as an invalid process.

`ss.goto.xy`

```
PROC ss.goto.xy (CHAN OF SS scrn, VAL INT x, y)
```

Sends the cursor to screen position (x,y). The origin (0,0) is at the top left corner of the screen.

`ss.clear.eol`

```
PROC ss.clear.eol (CHAN OF SS scrn)
```

Clears screen from the cursor position to the end of the current line.

`ss.clear.eos`

```
PROC ss.clear.eos (CHAN OF SS scrn)
```

Clears screen from the cursor position to the end of the current line and all lines below.

`ss.beep`

```
PROC ss.beep (CHAN OF SS scrn)
```

Sends a bell code to the terminal.

`ss.up`

```
PROC ss.up (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one line up the screen.

`ss.down`

```
PROC ss.down (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one line down the screen.

`ss.left`

```
PROC ss.left (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one place left.

`ss.right`

```
PROC ss.right (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one place right.

`ss.insert.char`

```
PROC ss.insert.char (CHAN OF SS scrn,
                     VAL BYTE ch)
```

Sends a command to the terminal to move the character at the cursor and all those to the right of it one place to the right and inserts **`char`** at the cursor. The cursor moves one place right.

`ss.delete.chr`

```
PROC ss.delete.chr (CHAN OF SS scrn)
```

Sends a command to the terminal to delete the character at the cursor and move the rest of the line one place to the left. The cursor does not move.

`ss.delete.chl`

```
PROC ss.delete.chl (CHAN OF SS scrn)
```

Sends a command to the terminal to delete the character to the left of the cursor and move the rest of the line one place to the left. The cursor also moves one place left.

`ss.ins.line`

```
PROC ss.ins.line (CHAN OF SS scrn)
```

Sends a command to the terminal to move all lines below the current line down one line on the screen, losing the bottom line. The current line becomes blank.

`ss.del.line`

```
PROC ss.del.line (CHAN OF SS scrn)
```

Sends a command to the terminal to delete the current line and move all lines below it up one line. The bottom line becomes blank.

1.6 String handling library

Library: `string.lib`

This library contains functions and procedures for handling strings and scanning lines of text. They assist with the manipulation of character strings such as names, commands, and keyboard responses.

The library provides routines for:

- Identifying characters
- Comparing strings
- Searching strings
- Editing strings
- Scanning lines of text

Result	Function	Parameter Specifiers
BOOL	is.in.range	VAL BYTE char, bottom, top
BOOL	is.upper	VAL BYTE char
BOOL	is.lower	VAL BYTE char
BOOL	is.digit	VAL BYTE char
BOOL	is.hex.digit	VAL BYTE char
BOOL	is.id.char	VAL BYTE char
INT	compare.strings	VAL []BYTE str1, str2
BOOL	eqstr	VAL []BYTE s1,s2
INT	string.pos	VAL []BYTE search, str
INT	char.pos	VAL BYTE search, VAL []BYTE str
INT, BYTE	search.match	VAL []BYTE possibles, str
INT, BYTE	search.no.match	VAL []BYTE possibles, str

Procedure	Parameter Specifiers
str.shift	[]BYTE str, VAL INT start, len, shift, BOOL not.done
delete.string	INT len, []BYTE str, VAL INT start, size, BOOL not.done
insert.string	VAL []BYTE new.str, INT len, []BYTE str, VAL INT start, BOOL not.done
to.upper.case	[]BYTE str
to.lower.case	[]BYTE str
append.char	INT len, []BYTE str, VAL BYTE char
append.text	INT len, []BYTE str, VAL []BYTE text
append.int	INT len, []BYTE str, VAL INT number, field
append.int64	INT len, []BYTE str, VAL INT64 number, VAL INT field
append.hex.int	INT len, []BYTE str, VAL INT number, field
append.hex.int64	INT len, []BYTE str, VAL INT64 number, VAL INT width
append.real32	INT len, []BYTE str, VAL REAL32 number, VAL INT Ip, Dp
append.real64	INT len, []BYTE str, VAL REAL64 number, VAL INT Ip, Dp

Procedure	Parameter Specifiers

`next.word.from.line`	`VAL []BYTE line,`
	`INT ptr, len,`
	`[]BYTE word, BOOL ok`

`next.int.from.line`	`VAL []BYTE line,` `INT ptr, number, BOOL ok`

1.6.1 Character identification

`is.in.range`

```
BOOL FUNCTION is.in.range (VAL BYTE char, bottom,
                                         top)
```

Returns `TRUE` if the value of `char` is in the range defined by `bottom` and `top` inclusive.

`is.upper`

```
BOOL FUNCTION is.upper (VAL BYTE char)
```

Returns `TRUE` if `char` is an ASCII upper case letter.

`is.lower`

```
BOOL FUNCTION is.lower (VAL BYTE char)
```

Returns `TRUE` if `char` is an ASCII lower case letter.

`is.digit`

```
BOOL FUNCTION is.digit (VAL BYTE char)
```

Returns `TRUE` if `char` is an ASCII decimal digit.

`is.hex.digit`

```
BOOL FUNCTION is.hex.digit (VAL BYTE char)
```

Returns `TRUE` if `char` is an ASCII hexadecimal digit. Upper or lower case letters A–F are allowed.

is.id.char

```
BOOL FUNCTION is.id.char (VAL BYTE char)
```

Returns **TRUE** if **char** is an ASCII character which can be part of an occam name.

1.6.2 String comparison

These two procedures allow strings to be compared for order or for equality.

compare.strings

```
INT FUNCTION compare.strings (VAL []BYTE str1,
                                          str2)
```

This general purpose ordering function compares two strings according to the lexicographic ordering standard. (Lexicographic ordering is the ordering used in dictionaries etc., using the ASCII values of the bytes). It returns one of the 5 results 0, 1, –1, 2, –2 as follows.

0 The strings are exactly the same in length and content.

1 **str2** is a leading substring of **str1**

–1 **str1** is a leading substring of **str2**

2 **str1** is lexicographically later than **str2**

–2 **str2** is lexicographically later than **str1**

So if **s** is 'abcd':

```
compare.strings ("abc", [s FROM 0 FOR 3])  =  0
compare.strings ("abc", [s FROM 0 FOR 2])  =  1
compare.strings ("abc", s)                 = -1
compare.strings ("bc", s)                  =  2
compare.strings ("a4", s)                  = -2
```

eqstr

```
BOOL FUNCTION eqstr (VAL []BYTE s1,s2)
```

This is an optimised test for string equality. It returns **TRUE** if the two strings are the same size and have the same contents, **FALSE** otherwise.

1.6.3 String searching

These procedures allow a string to be searched for a match with a single byte or a string of bytes, for a byte which is one of a set of possible bytes, or for a byte which is not one of a set of bytes. Searches insensitive to alphabetic case should use `to.upper.case` or `to.lower.case` on both operands before using these procedures.

`string.pos`

```
INT FUNCTION string.pos (VAL []BYTE search, str)
```

Returns the position in `str` of the first occurrence of a substring which exactly matches `search`. Returns −1 if there is no such match.

`char.pos`

```
INT FUNCTION char.pos (VAL BYTE search,
                       VAL []BYTE str)
```

Returns the position in `str` of the first occurrence of the byte `search`. Returns −1 if there is no such byte.

`search.match`

```
INT, BYTE FUNCTION search.match
                   (VAL []BYTE possibles, str)
```

Searches `str` for any one of the bytes in the array `possibles`. If one is found its index and identity are returned as results. If none is found then −1, 255(`BYTE`) are returned.

`search.no.match`

```
INT, BYTE FUNCTION search.no.match
                   (VAL []BYTE possibles, str)
```

Searches `str` for a byte which does not match any one of the bytes in the array `possibles`. If one is found its index and identity are returned as results. If none is found then −1, 255(`BYTE`) are returned.

1.6.4 String editing

These procedures allow strings to be edited. The string to be edited is stored in an array which may contain unused space. The editing operations supported are: deletion of a number of characters and the closing of the gap created;

insertion of a new string starting at any position within a string, which creates a gap of the necessary size.

These two operations are supported by a lower level procedure for shifting a consecutive substring left or right within the array. The lower level procedure does exhaustive tests against overflow.

str.shift

```
PROC str.shift ([]BYTE str, VAL INT start,
                 len, shift, BOOL not.done)
```

Takes a substring **[str FROM start FOR len]**, and copies it to a position **shift** places to the right. Any implied actions involving bytes outside the string are not performed and cause the error flag **not.done** to be set **TRUE**. Negative values of **shift** cause leftward moves.

delete.string

```
PROC delete.string (INT len, []BYTE str,
                    VAL INT start, size,
                    BOOL not.done)
```

Deletes **size** bytes from the string **str** starting at **str[start]**. There are initially **len** significant characters in **str** and it is decremented appropriately. If **start** is outside the string, or **start+size** is greater than **len**, then no action occurs and **not.done** is set **TRUE**.

insert.string

```
PROC insert.string (VAL []BYTE new.str, INT len,
                    []BYTE str, VAL INT start,
                    BOOL not.done)
```

Creates a gap in **str** after **str[start]** and copies the string **new.str** into it. There are initially **len** significant characters in **str** and **len** is incremented by the length of **new.str** inserted. Any overflow of the declared SIZE of **str** results in truncation at the right and setting **not.done** to **TRUE**. This procedure may be used for simple concatenation on the right by setting **start = len** or on the left by setting **start = 0**. This method of concatenation differs from that using the **append.** procedures in that it can never cause the program to stop.

to.upper.case

```
PROC to.upper.case ([]BYTE str)
```

Converts all alphabetic characters in **str** to upper case.

to.lower.case

```
PROC to.lower.case ([]BYTE str)
```

Converts all alphabetic characters in **str** to lower case.

append.char

```
PROC append.char (INT len, []BYTE str,
                  VAL BYTE char)
```

Writes a byte **char** into the array **str** at **str[len]**. **len** is incremented by 1. Behaves like **STOP** if the array overflows.

append.text

```
PROC append.text (INT len, []BYTE str,
                  VAL []BYTE text)
```

Writes a string **text** into the array **str**, starting at **str[len]** and computing a new value for **len**. Behaves like **STOP** if the array overflows.

append.int

```
PROC append.int (INT len, []BYTE str,
                 VAL INT number, field)
```

Converts **number** into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified **field** width if necessary. If the number cannot be represented in **field** characters it is widened as necessary. A zero value for **field** will give minimum width. The converted number is written into the array **str** starting at **str[len]** and **len** is incremented. Behaves like **STOP** if the array overflows or if **field** $<$ 0.

append.int64

```
PROC append.int64 (INT len, []BYTE str,
                   VAL INT64 number,
                   VAL INT field)
```

As **append.int** but for 64-bit integers.

append.hex.int

```
PROC append.hex.int (INT len, []BYTE str,
                     VAL INT number, width)
```

Converts **number** into a sequence of ASCII hexadecimal digits, using upper case letters, preceded by '#'. The total number of characters sent is always **width+1**, padding out with '0' or 'F' on the left if necessary. The number is truncated at the left if the field is too narrow, thereby allowing the less significant part of any number to be printed. The converted number is written into the array **str** starting at **str[len]** and **len** is incremented. Behaves like **STOP** if the array overflows or if **width** < 0.

append.hex.int64

```
PROC append.hex.int64 (INT len, []BYTE str,
                       VAL INT64 number,
                       VAL INT width)
```

As **append.hex.int** but for 64-bit integers.

append.real32

```
PROC append.real32 (INT len, []BYTE str,
                    VAL REAL32 number,
                    VAL INT Ip, Dp)
```

Converts **number** into a sequence of ASCII characters formatted using **Ip** and **Dp** as described under **REAL32TOSTRING** (see section 1.7).

The converted number is written into the array **str** starting at **str[len]** and **len** is incremented. Behaves like **STOP** if the array overflows.

append.real64

```
PROC append.real64 (INT len, []BYTE str,
                    VAL REAL64 number,
                    VAL INT Ip, Dp)
```

As **append.real32**, but for 64-bit real values. The formatting variables **Ip** and **Dp** are described under **REAL32TOSTRING**, (see section 1.7).

1.6.5 Line parsing

Depending on the initial value of the variable **ok** these two procedures either read a line serially, returning the next word and next integer respectively, or the procedures act almost like a **SKIP** (see below). The user should initialise the variable **ok** as appropriate.

next.word.from.line

```
PROC next.word.from.line (VAL []BYTE line,
                          INT ptr, len,
                          []BYTE word,
                          BOOL ok)
```

If **ok** is passed in as **TRUE**, on entry to the procedure, skips leading spaces and horizontal tabs and reads the next word from the string **line**. The value of **ptr** is the starting point of the search. A word continues until a space or tab or the end of the string **line** is encountered.

If the end of the string is reached without finding a word, the boolean **ok** is set to **FALSE**, and **len** is 0. If a word is found but is too large for **word**, then **ok** is set to **FALSE**, but **len** will be the length of the word that was found; otherwise the found word will be in the first **len** bytes of **word**.

The index **ptr** is updated to be that of the space or tab immediately after the found word, or is **SIZE line**.

If **ok** is passed in as **FALSE**, **len** is set to 0, **ptr** and **ok** remain unchanged, and **word** is undefined.

next.int.from.line

```
PROC next.int.from.line (VAL []BYTE line,
                         INT ptr, number,
                         BOOL ok)
```

If **ok** is passed in as **TRUE**, on entry to the procedure, skips leading spaces and horizontal tabs and reads the next integer from the string **line**. The value of **ptr** is the starting point of the search. The integer is considered to start with the first non-space, non-tab character found and continues until a space or tab or the end of the string **line** is encountered.

If the first sequence of non-space, non-tab characters does not exist, does not form an integer, or forms an integer that overflows the **INT** range then **ok** is set to **FALSE**, and **number** is undefined; otherwise **ok** remains **TRUE**, and **number** is the integer read. A + or - may be the first character of the integer.

The index **ptr** is updated to be that of the space or tab immediately after the found integer, or is **SIZE line**.

If **ok** is passed in as **FALSE**, then **ptr** and **ok** remain unchanged, and **number** is undefined.

1.7 Type conversion library

Library: convert.lib

This library contains procedures for converting numeric variables to strings and vice versa.

String to numeric conversions return two results, the converted value and a boolean error indication. Numeric to string conversions return the converted string and an integer which represents the number of significant characters written into the string.

Procedure	Parameter Specifiers
INTTOSTRING	INT len, []BYTE string, VAL INT n
INT16TOSTRING	INT len, []BYTE string, VAL INT16 n
INT32TOSTRING	INT len, []BYTE string, VAL INT32 n
INT64TOSTRING	INT len, []BYTE string, VAL INT64 n
HEXTOSTRING	INT len, []BYTE string, VAL INT n
HEX16TOSTRING	INT len, []BYTE string, VAL INT16 n
HEX32TOSTRING	INT len, []BYTE string, VAL INT32 n
HEX64TOSTRING	INT len, []BYTE string, VAL INT64 n
REAL32TOSTRING	INT len, []BYTE string, VAL REAL32 X, VAL INT Ip, Dp
REAL64TOSTRING	INT len, []BYTE string, VAL REAL64 X, VAL INT Ip, Dp
BOOLTOSTRING	INT len, []BYTE string, VAL BOOL b
STRINGTOINT	BOOL Error, INT n, VAL []BYTE string
STRINGTOINT16	BOOL Error, INT16 n, VAL []BYTE string
STRINGTOINT32	BOOL Error, INT32 n, VAL []BYTE string
STRINGTOINT64	BOOL Error, INT64 n, VAL []BYTE string
STRINGTOHEX	BOOL Error, INT n, VAL []BYTE string
STRINGTOHEX16	BOOL Error, INT16 n, VAL []BYTE string
STRINGTOHEX32	BOOL Error, INT32 n, VAL []BYTE string
STRINGTOHEX64	BOOL Error, INT64 n, VAL []BYTE string
STRINGTOREAL32	BOOL Error, REAL32 X, VAL []BYTE string
STRINGTOREAL64	BOOL Error, REAL64 X, VAL []BYTE string
STRINGTOBOOL	BOOL Error, b, VAL []BYTE string

1.7.1 Procedure definitions

INTTOSTRING

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

Converts an integer value to a string. The procedure returns the decimal representation of `n` in `string` and the number of characters in the representation, in `len`. If `string` is not long enough to hold the representation then this routine acts as an invalid process.

Similar procedures are provided for the types `INT16`, `INT32` and `INT64`.

INT16TOSTRING

```
PROC INT16TOSTRING (INT len, []BYTE string,
                    VAL INT16 n)
```

As `INTTOSTRING` but for 16-bit integers.

INT32TOSTRING

```
PROC INT32TOSTRING (INT len, []BYTE string,
                    VAL INT32 n)
```

As `INTTOSTRING` but for 32-bit integers.

INT64TOSTRING

```
PROC INT64TOSTRING (INT len, []BYTE string,
                    VAL INT64 n)
```

As `INTTOSTRING` but for 64-bit integers.

HEXTOSTRING

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

The procedure returns the hexadecimal representation of `n` in `string` and the number of characters in the representation, in `len`. All the words of `n`, (in 4-bit wide word lenghts) are output so that leading zeroes are included. The number of characters will be the number of bits in an `INT` divided by four. A '`#`' is not output by the `HEXTOSTRING` procedure. If `string` is not long enough to hold the representation then this routine acts as an invalid process.

Similar procedures are provided for the types **HEX16**, **HEX32** and **HEX64**.

HEX16TOSTRING

```
PROC HEX16TOSTRING (INT len, []BYTE string,
                    VAL INT16 n)
```

As **HEXTOSTRING** but for 16-bit integers.

HEX32TOSTRING

```
PROC HEX32TOSTRING (INT len, []BYTE string,
                    VAL INT32 n)
```

As **HEXTOSTRING** but for 32-bit integers.

HEX64TOSTRING

```
PROC HEX64TOSTRING (INT len, []BYTE string,
                    VAL INT64 n)
```

As **HEXTOSTRING** but for 64-bit integers.

REAL32TOSTRING

```
PROC REAL32TOSTRING (INT len, []BYTE string,
                     VAL REAL32 X,
                     VAL INT Ip, Dp)
```

Converts a 32-bit real number (represented in single precision IEEE format) to a string of ASCII characters. **len** is the number of characters (**BYTES**) of **string** used for the formatted decimal representation of the number. (The following description applies to and notes the differences between this procedure and **REAL64TOSTRING**).

Depending on the value of **X** and the two formatting variables **Ip** and **Dp** the procedure will use either a fixed or exponential format for the output string. These formats are defined as follows:

Fixed : First, either a minus sign or space (an explicit plus sign is not used), followed by a fraction in the form <digits> . <digits>. Padding spaces are added to the left of the sign indicator, as necessary. (`Ip` gives the number of places before the point and `Dp` the number of places after the point).

Exponential : First, either a minus sign or space (again, an explicit plus sign is not used), followed by a fraction in the form <digit> . <digits>, the exponential symbol (E), the sign of the exponent (explicitly plus or minus), then the exponent, which is two digits for a `REAL32` and three digits for a `REAL64`. (`Dp` gives the number of digits in the fraction (1 before the decimal point and the others after)).

Possible combinations of `Ip` and `Dp` fall into three categories, described below. **Note** the term 'Free format' means that the procedure may adopt either fixed or exponential format, depending on the actual value of `X`.

1 If `Ip=0, Dp=0`, then free format is adopted. Exponential format is used if the absolute value of `X` is less than 10^{-4}, but non-zero, or greater than 10^9 (for `REAL32`), or greater than 10^{17} (for `REAL64`); otherwise fixed format is used.

   The value of `len` is dependent on the actual value of `X` with trailing zeroes suppressed. The maximum length of the result is 15 or 24, depending on whether it is `REAL32` or `REAL64` respectively.

   If `X` is 'Not-a-Number' or infinity then the string will contain one of the following: '`Inf`', '`-Inf`' or '`NaN`', (excluding the quotes).

2 If `Ip>0, Dp>0`, fixed format is used, unless the value needs more than `Ip` significant digits before the decimal point, in which case, exponential format is used. If exponential does not fit either, then a signed string '`Ov`' is produced. The length is always `Ip + Dp + 2` when `Ip>0, Dp>0`.

   If `X` is 'Not-a-Number' or infinity then the string will contain one of the following: '`Inf`', '`-Inf`' or '`NaN`', (excluding the quotes) and padded out by spaces on the right to fill the field width.

3 If `Ip=0, Dp>0`, then exponential format is always used. The length of the result is `Dp + 6` or `Dp + 7`, depending on whether `X` is a `REAL32` or `REAL64`, respectively.

   If `Ip=0, Dp=1`, then a special result is produced consisting of a sign, a blank, a digit and the exponent. The length is 7 or 8 depending on whether **X** is a **REAL32** or **REAL64**. **Note**: this result does not conform to the occam format for a **REAL**.

If **X** is 'Not-a-Number' or infinity then the string will contain one of the following: '**Inf**', '**-Inf**' or '**NaN**', (excluding the quotes) and padded out by spaces on the right to fill the field width.

All other combinations of **Ip** and **Dp** are errors.

If **string** is not long enough to hold the requested formatted real number as a string then these routines act as invalid processes.

REAL64TOSTRING

```
PROC REAL64TOSTRING (INT len, []BYTE string,
                     VAL REAL64 X,
                     VAL INT Ip, Dp)
```

As **REAL32TOSTRING** but for 64-bit numbers.

BOOLTOSTRING

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

Converts a boolean value to a string. The procedure returns '**TRUE**' in **string** if **b** is **TRUE** and '**FALSE**' otherwise. **len** contains the number of characters in the string returned. If **string** is not long enough to hold the representation then this routine acts as an invalid process.

STRINGTOINT

```
PROC STRINGTOINT (BOOL Error, INT n,
                  VAL []BYTE string)
```

Converts a string to a decimal integer. The procedure returns in **n** the value represented in **string**. **error** is set to **TRUE** if a non-numeric character is found in **string** or if **string** is empty. + or a - are allowed in the first character position. **n** will be the value of the portion of **string** up to any illegal characters, with the convention that the value of an empty string is 0. **error** is also set to **TRUE** 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.

Similar procedures are provided for the types **INT16**, **INT32** and **INT64**.

STRINGTOINT16

```
PROC STRINGTOINT16 (BOOL Error, INT16 n,
                    VAL []BYTE string)
```

As **STRINGTOINT** but converts to a 16-bit integer.

STRINGTOINT32

```
PROC STRINGTOINT32 (BOOL Error, INT32 n,
                    VAL []BYTE string)
```

As **STRINGTOINT** but converts to a 32-bit integer.

STRINGTOINT64

```
PROC STRINGTOINT64 (BOOL Error, INT64 n,
                    VAL []BYTE string)
```

As **STRINGTOINT** but converts to a 64-bit integer.

STRINGTOHEX

```
PROC STRINGTOHEX (BOOL Error, INT n,
                  VAL []BYTE string)
```

The procedure returns in **n** the value represented by the hexadecimal **string**. No '**#**' is allowed in the input and hex digits must be in upper case (A to F) rather than lower case (a to f). **error** is set to **TRUE** if a non-hexadecimal character is found in **string**, or if **string** is empty.

n will be the value of 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 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 **HEX16**, **HEX32** and **HEX64**.

STRINGTOHEX16

```
PROC STRINGTOHEX16 (BOOL Error, INT16 n,
                    VAL []BYTE string)
```

As **STRINGTOHEX** but converts to a 16-bit integer.

STRINGTOHEX32

```
PROC STRINGTOHEX32 (BOOL Error, INT32 n,
                    VAL []BYTE string)
```

As **STRINGTOHEX** but converts to a 32-bit integer.

STRINGTOHEX64

```
PROC STRINGTOHEX64 (BOOL Error, INT64 n,
                    VAL []BYTE string)
```

As **STRINGTOHEX** but converts to a 64-bit integer.

STRINGTOREAL32

```
PROC STRINGTOREAL32 (BOOL Error, REAL32 X,
                     VAL []BYTE string)
```

Converts a string to a 32-bit real number. This procedure takes a string containing a decimal representation of a real number and converts 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 valid. If there are 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**.

STRINGTOREAL64

```
PROC STRINGTOREAL64 (BOOL Error, REAL64 X,
                     VAL []BYTE string)
```

As **STRINGTOREAL32** but converts to a 64-bit number.

STRINGTOBOOL

```
PROC STRINGTOBOOL (BOOL Error, b,
                   VAL []BYTE string)
```

Converts a string to a boolean value. The procedure returns **TRUE** in b if the first four characters of **string** are '**TRUE**' and **FALSE** if the first five characters are '**FALSE**'; **b** is undefined in other cases. **TRUE** is returned in **error** if **string** is not exactly '**TRUE**' or '**FALSE**'.

1.8 Block CRC library

Library: crc.lib

The block CRC library provides two functions for generating CRC codes from byte strings. `OldCRC` is some agreed initialisation value e.g. zero or the polynomial generator. It may be, however, that the string that you want the CRC of, is not all available at once. In this case, although an initialisation is still required once, the value of the CRC from one segment of the string is used for `OldCRC` on the next segment, until all segments of the string are exhausted.

For further information about CRC functions see '*INMOS Technical note 26: Notes on graphics support and performance improvements on the IMS T800*'.

Result	Function	Parameter Specifiers
INT	CRCFROMMSB	VAL []BYTE InputString, VAL INT PolynomialGenerator, VAL INT OldCRC
INT	CRCFROMLSB	VAL []BYTE InputString VAL INT PolynomialGenerator, VAL INT OldCRC

1.8.1 Function definitions

CRCFROMMSB

```
INT FUNCTION CRCFROMMSB (VAL []BYTE InputString,
                         VAL INT PolynomialGenerator,
                         VAL INT OldCRC)
```

The string of bytes is polynomially divided by the generator, starting at the most significant bit of the most significant byte.

CRCFROMLSB

```
INT FUNCTION CRCFROMLSB (VAL []BYTE InputString,
                         VAL INT PolynomialGenerator,
                         VAL INT OldCRC)
```

The string of bytes is polynomially divided by the generator, starting at the least significant bit of the least significant byte.

1.9 Extraordinary link handling library

Library: `xlink.lib`

The extraordinary link handling library contains routines for handling communication failures errors on a link. Four procedures are provided to allow failures on input and output channels to be handled by timeout or by signalling the failure on another channel. A fifth procedure allows the channel to be reset. The use of these routines is described in part 1, section 10.5.

Procedure	Parameter Specifiers
`InputOrFail.t`	`CHAN OF ANY c, []BYTE mess,` `TIMER t, VAL INT time, BOOL aborted`
`OutputOrFail.t`	`CHAN OF ANY c, VAL []BYTE mess,` `TIMER t, VAL INT time, BOOL aborted`
`InputOrFail.c`	`CHAN OF ANY c, []BYTE mess` `CHAN OF INT kill, BOOL aborted`
`OutputOrFail.c`	`CHAN OF ANY c, VAL []BYTE mess,` `CHAN OF INT kill, BOOL aborted`
`Reinitialise`	`CHAN OF ANY c`

CAUTION:

Use of the routines in `xlink.lib` during interactive debugging will lead to undefined results.

1.9.1 Procedure definitions

The four procedures take as parameters a link channel `c` (on which the communication is to take place), a byte vector `mess` (which is the object of the communication), and the boolean variable `aborted`. The choice of a byte vector for the message allows an object of any type to be passed along the channel providing it is retyped first.

`InputOrFail.t`

```
PROC InputOrFail.t (CHAN OF ANY c, []BYTE mess,
                     TIMER t, VAL INT time,
                     BOOL aborted)
```

This procedure is used for communication where failure is detected by a timeout. It takes a timer parameter `t`, and an absolute time `time`. The procedure treats the communication as having failed when the time as measured by the timer `t` is **AFTER** the specified time `time`.

`OutputOrFail.t`

```
PROC OutputOrFail.t (CHAN OF ANY c,
                     VAL []BYTE mess,
                     TIMER t, VAL INT time,
                     BOOL aborted)
```

This procedure is used for communication where failure is detected by a timeout. It takes a timer parameter `t`, and an absolute time `time`. The procedure treats the communication as having failed when the time as measured by the timer `t` is **AFTER** the specified time `time`.

`InputOrFail.c`

```
PROC InputOrFail.c (CHAN OF ANY c, []BYTE mess,
                    CHAN OF INT kill,
                    BOOL aborted)
```

This procedure provides, through an abort control channel, for communication failure on a channel expecting an input. This is useful if failure cannot be detected by a simple timeout. Any integer on the channel `kill` will cause the channel `c` to be reset and this procedure to terminate.

`OutputOrFail.c`

```
PROC OutputOrFail.c (CHAN OF ANY c,
                     VAL []BYTE mess,
                     CHAN OF INT kill,
                     BOOL aborted)
```

This procedure provides, through an abort control channel, for communication failure on a channel attempting to output. This is useful if failure cannot be detected by a simple timeout. Any integer on the channel `kill` will cause the channel `c` to be reset and this procedure to terminate.

`Reinitialise`

```
PROC Reinitialise (CHAN OF ANY c)
```

This procedure may be used to reinitialise the link channel **`c`** after it is known that all activity on the link has ceased.

`Reinitialise` must only be used to reinitialise a link channel after communication has finished. If the procedure is applied to a link channel which is being used for communication the transputer's error flag will be set and subsequent behaviour is undefined.

1.10 Debugging support library

Library: `debug.lib`

The debugging support library provides four procedures. Two procedures are provided to stop a process, one on a specified condition. The third procedure is used to insert debugging messages and the fourth procedure is a timer process for analysing deadlocks.

Procedure	Parameter Specifiers
DEBUG.ASSERT	VAL BOOL assertion
DEBUG.MESSAGE	VAL []BYTE message
DEBUG.STOP	()
DEBUG.TIMER	CHAN OF INT stop

1.10.1 Procedure definitions

DEBUG.ASSERT

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

If a condition fails this procedure stops a process and notifies the debugger.

If `assertion` evaluates `FALSE`, `DEBUG.ASSERT` stops the process and sends process data to the debugger. If `assertion` evaluates `TRUE` no action is taken.

If the program is not being run within the breakpoint debugger and the assertion fails, then the procedure behaves like `DEBUG.STOP`.

DEBUG.MESSAGE

```
PROC DEBUG.MESSAGE (VAL []BYTE message)
```

This procedure sends a message to the debugger which is displayed along with normal program output. **Note:** that only the first 83 characters of the message are displayed.

If the program is not being run within the breakpoint debugger the procedure has no effect.

DEBUG.STOP

```
PROC DEBUG.STOP ()
```

This procedure stops the process and sends process data to the debugger.

If the program is not being run within the breakpoint debugger then the procedure stops the process or processor, depending on the error mode that the processor is in.

DEBUG.TIMER

```
PROC DEBUG.TIMER (CHAN OF INT stop)
```

A timer process for use when analysing deadlocks in occam programs. This procedure supports all current 16 and 32 bit transputers. The procedure remains on the timer queue until receipt of any integer value on the channel `stop`, whereupon it will terminate. For an example of this process see part 1, section 7.17.5.

1.11 Mixed languages support library

Library: `callc.lib`

This library provides four occam procedures for initialising static and heap areas and terminating them after use. They are provided to support the incorporation of code written in other languages such as C and FORTRAN into occam programs. Only code which is in the standard TCOFF format, used by this toolset may be incorporated using these procedures.

Procedure	Parameter Specifiers
`init.static`	`[] INT static.area, INT required.size, gsb`
`init.heap`	`VAL INT gsb, []INT heap.area`
`terminate.heap.use`	`VAL INT gsb`
`terminate.static.use`	`VAL INT gsb`

1.11.1 Procedure definitions

`init.static`

```
PROC init.static  ([] INT static.area,
                    INT required.size, gsb)
```

This function is used to set aside and initialise an area of memory for use as a static area.

The static area is an integer array declared in the occam calling program. Two integer values are obtained, as follows:

`required.size` : The number of words of static space required.

`gsb` : A pointer to the base of the array which will act as the global static base.

Note: the number of words of static space required is equivalent to the size of the integer array. One element of the integer array is equivalent to one word of memory.

If an error occurs when initialising the static area then the value MOSTPOS INT is returned instead of the required size.

`init.heap`

```
PROC init.heap  (INT gsb, VAL []INT heap.area)
```

This procedure is used to set aside an area of memory for use as a heap. The first argument is the gsb pointer returned by **`init.static`**, which is required because the memory allocation routines make use of static data.

As for the static area the heap area is declared as an integer array. This array must be large enough to accommodate all calls to C memory allocation functions. The number of words of heap area required is equivalent to the size of the integer array. One element of the integer array is equivalent to one word of memory.

If the heap is used by a function before **`init.heap`** has been called the memory allocation functions will fail with their normal error returns.

`terminate.heap.use`

```
PROC terminate.heap.use (VAL INT gsb)
```

`terminate.heap.use` should be called when the heap is no longer required. It provides a clean way of terminating the use of the heap.

Once the heap terminate procedure has been called the state of the heap is undefined and further calls to memory allocation functions will fail.

`terminate.heap.use` must be called *before* terminating the static area because the heap requires static variables for its operation.

`terminate.static.use`

```
PROC terminate.static.use (VAL INT gsb)
```

`terminate.static.use` should be called when the static area is no longer required, usually when no further calls to other languages will be made. It provides a clean way of ending the use of the static area.

Once the static terminate procedure has been called the state of the static area is undefined.

1.12 DOS specific hostio library

Library: `msdos.lib`

The MSDOS host file server library allows programs to use some facilities specific to the IBM PC. A set of constants for the library are provided in the include file `msdos.inc`, which is listed in appendix C.

Caution: Programs that use this DOS specific library will not be portable to versions of the toolset on other hosts.

Procedure	Parameter Specifiers

`dos.receive.block`	`CHAN OF SP fs, ts,`
	`VAL INT32 location,`
	`INT bytes.read, []BYTE block,`
	`BYTE result`

`dos.send.block`	`CHAN OF SP fs, ts,`
	`VAL INT32 location,`
	`VAL []BYTE block,`
	`INT len, BYTE result`

`dos.call.interrupt`	`CHAN OF SP fs, ts,`
	`VAL INT16 interrupt,`
	`VAL [dos.interrupt.regs.size]`
	`BYTE register.block.in,`
	`BYTE carry.flag,`
	`[dos.interrupt.regs.size] BYTE`
	`register.block.out,`
	`BYTE result`

`dos.read.regs`	`CHAN OF SP fs, ts,`
	`[dos.read.regs.size] BYTE`
	`registers,`
	`BYTE result`

`dos.port.read`	`CHAN OF SP fs, ts,`
	`VAL INT16 port.location,`
	`BYTE value, result`

`dos.port.write`	`CHAN OF SP fs, ts,`
	`VAL INT16 port.location,`
	`VAL BYTE value, BYTE result`

1.12.1 Procedure definitions

dos.receive.block

```
PROC dos.receive.block (CHAN OF SP fs, ts,
                        VAL INT32 location,
                        INT bytes.read,
                        []BYTE block,
                        BYTE result)
```

Reads a block of data, starting at `location`, from host memory. `location` is arranged as the segment in the top two bytes and the offset in the lower two bytes, both unsigned.

The number bytes requested is `SIZE block`; the number of bytes read is returned in `bytes.read`. The result returned can take any of the following values:

`spr.ok`	The read operation was successful.
`spr.bad.packet.size`	Too many bytes were requested to be read: (`SIZE block`) > `dos.max.receive.block.buffer.size`.
≥ `spr.operation.failed`	The read failed, so `bytes.read` = 0. If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

dos.send.block

```
PROC dos.send.block (CHAN OF SP fs, ts,
                     VAL INT32 location,
                     VAL []BYTE block,
                     INT len, BYTE result)
```

Writes a block of data to host memory, starting at `location`. The `location` is arranged as the segment in the top two bytes and the offset in the lower two bytes, both unsigned.

The number of bytes, requested to be written is `SIZE block`; the number of bytes written is returned in `len`. The result returned can take any of the following values:

`spr.ok`	The write operation was successful.
`spr.bad.packet.size`	Too many bytes were requested to be written: (`SIZE block`) > `dos.max.send.block.buffer.size`.
≥ `spr.operation.failed`	The write failed. If `result` takes a value ≥ `spr.operation.failed` then this denotes a server returned failure. (See sections C.1 and H.2.2).

dos.call.interrupt

```
PROC dos.call.interrupt
     (CHAN OF SP fs, ts,
     VAL INT16 interrupt,
     VAL [dos.interrupt.regs.size] BYTE register.block.in,
     BYTE carry.flag,
     [dos.interrupt.regs.size] BYTE register.block.out,
     BYTE result)
```

Invokes an interrupt call on the host PC, with the processor's registers initialised to requested values. On return from the interrupt the values stored in the processor's registers are returned in `register.block.out`, along with the value of the carry flag on the PC, which is stored in `carry.flag`.

The interrupt number is specified by `interrupt`. The registers are represented by a block of bytes called `register.block.in`. This block stores the values to be written to the registers. Each register value occupies 4 bytes of a block. On the IBM PC the 2 most significant bytes are ignored as this machine has only 2 byte registers (16 bit registers). The layout of registers in the block is as follows:

Register	Start position in block (least significant byte)	End position in block (most significant byte)
ax	0	3
bx	4	7
cx	8	11
dx	12	15
di	16	19
si	20	23
cs	24	27
ds	28	31
es	32	35
ss	36	39

Note, however, that the cs and ss registers cannot be set.

The result returned can take any of the following values:

spr.ok	The interrupt was successful.
≥ spr.operation.failed	The interrupt failed. If result takes a value ≥ spr.operation.failed then this denotes a server returned failure. (See sections C.1 and H.2.2).

dos.read.regs

```
PROC dos.read.regs
              (CHAN OF SP fs, ts,
              [dos.read.regs.size] BYTE registers,
              BYTE result)
```

Reads the current values of some registers of the PC. The values of the registers are returned as a block of bytes, each register occupying 4 bytes of the block:

Register	Start position in block (least significant byte)	End position in block (most significant byte)
ax	0	3
bx	4	7
cx	8	11
dx	12	15

On the IBM PC the 2 most significant bytes are ignored as this machine has only 2 byte registers (16 bit registers).

The result returned can take any of the following values:

spr.ok	The read was successful.
≥ **spr.operation.failed**	The read failed. If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

dos.port.read

```
PROC dos.port.read (CHAN OF SP fs, ts,
                    VAL INT16 port.location,
                    BYTE value, result)
```

Reads the value at the port, specified by the port address, **port.location**. The port address being in the input/output space of the PC is an unsigned number between 0 and 64K.

No check is made to ensure that the value received from the port (if any) is valid. The value returned in **value** is that of the given address at the moment the port is read by the host file server.

The result returned can take any of the following values:

spr.ok	The read was successful.
≥ **spr.operation.failed**	The read failed. If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

dos.port.write

```
PROC dos.port.write (CHAN OF SP fs, ts,
                     VAL INT16 port.location,
                     VAL BYTE value, BYTE result)
```

Writes the given **value** to the port specified by the port address, **port.location**. The port address being in the input/output space of the PC is an unsigned number between 0 and 64K.

No check is made to ensure that the value written to the port has been correctly read by the device connected to the port (if any).

The result returned can take any of the following values:

spr.ok	The write was successful.
≥ **spr.operation.failed**	The write failed. If **result** takes a value ≥ **spr.operation.failed** then this denotes a server returned failure. (See sections C.1 and H.2.2).

Appendices

A Names defined by the software

All names which may appear in occam source text and which are defined either by the language, the compiler or the libraries are given below in alphabetical order.

Toolset constants are not included; for listings of the constants files see appendix C.

The names in this table are grouped into the following classes:

1 *Language keyword.* Keyword defined in the language reference manual.

2 *Compiler keyword.* Keyword defined by the current compiler implementation.

3 *Compiler predefine.* A procedure or function which is predeclared by the compiler. On some processors these are implemented by a routine in a library with the name indicated, on others they are implemented as in line code.

4 *System library.* Library routines for special transputer system operations. Consists of the libraries **crc.lib** and **xlink.lib**.

5 *Maths library.* A function in the elementary function libraries. The library name depends on the version required (single or double length).

6 *Maths support.* Supporting functions for routines in **tbmaths.lib**.

7 *I/O library.* A procedure or function in the input/output and supporting libraries (**hostio.lib**, **streamio.lib**, **string.lib**, **process.lib**, and **convert.lib**). The library name which must be used to access it is also shown.

8 *Debug library.* Routines to help with interactive debugging.

9 *Compiler directive.* A word in occam source code recognised by the compilation system for special action at compile time. The word is preceded in occam source either by the character '**#**' or by '**#PRAGMA**'.

Any name which is not a language keyword may be redeclared as an identifier in an occam program. However, redefining a name of a compiler library procedure or function can have unexpected consequences and it is strongly recommended that all the names in these tables are reserved for the uses specified.

Name	Class	Library	Notes
ABS	compiler predefine		
ACOS	maths library	snglmath	also tbmaths
AFTER	language keyword		
ALOG	maths library	snglmath	also tbmaths
ALOG10	maths library	snglmath	also tbmaths
ALT	language keyword		
AND	language keyword		
ANY	language keyword		
append.char	io library	string	
append.hex.int64	io library	string	
append.hex.int	io library	string	
append.int64	io library	string	
append.int	io library	string	
append.real32	io library	string	
append.real64	io library	string	
append.text	io library	string	
ARGUMENT.REDUCE	compiler predefine		
ASHIFTLEFT	compiler predefine		
ASHIFTRIGHT	compiler predefine		
ASIN	maths library	snglmath	also tbmaths
ASM	compiler keyword		
ASSERT	compiler predefine		
AT	language keyword		
ATAN	maths library	snglmath	also tbmaths
ATAN2	maths library	snglmath	also tbmaths
BITAND	language keyword		
BITCOUNT	compiler predefine		
BITNOT	language keyword		
BITOR	language keyword		
BITREVNBITS	compiler predefine		
BITREVWORD	compiler predefine		
BOOL	language keyword		
BOOLTOSTRING	io library	convert	
BYTE	language keyword		
CASE	language keyword		
CAUSEERROR	compiler predefine		
CHAN	language keyword		
char.pos	io library	string	
CLIP2D	compiler predefine		
COMMENT	compiler directive		
compare.strings	io library	string	

Name	Class	Library	Notes
COPYSIGN	compiler predefine		
COS	maths library	snglmath	also tbmaths
COSH	maths library	snglmath	also tbmaths
CRCBYTE	compiler predefine		
CRCFROMLSB	system library	crc	
CRCFROMMSB	system library	crc	
CRCWORD	compiler predefine		
DABS	compiler predefine		
DACOS	maths library	dblmath	also tbmaths
DALOG	maths library	dblmath	also tbmaths
DALOG10	maths library	dblmath	also tbmaths
DARGUMENT.REDUCE	compiler predefine		
DASIN	maths library	dblmath	also tbmaths
DATAN	maths library	dblmath	also tbmaths
DATAN2	maths library	dblmath	also tbmaths
DCOPYSIGN	compiler predefine		
DCOS	maths library	dblmath	also tbmaths
DCOSH	maths library	dblmath	also tbmaths
DDIVBY2	compiler predefine		
DEBUG.ASSERT	debug library	debug	
DEBUG.MESSAGE	debug library	debug	
DEBUG.STOP	debug library	debug	
DEBUG.TIMER	debug library	debug	
delete.string	io library	string	
DEXP	maths library	dblmath	also tbmaths
DFLOATING.UNPACK	compiler predefine		
DFPINT	compiler predefine		
DIEEECOMPARE	compiler predefine		
DISNAN	compiler predefine		
DIVBY2	compiler predefine		
DLOGB	compiler predefine		
DMINUSX	compiler predefine		
DMULBY2	compiler predefine		
DNEXTAFTER	compiler predefine		
DNOTFINITE	compiler predefine		
DORDERED	compiler predefine		
DPOWER	maths library	dblmath	also tbmaths
DRAN	maths library	dblmath	also tbmaths
DRAW2D	compiler predefine		
DSCALEB	compiler predefine		

Name	Class	Library	Notes
DSIN	maths library	dblmath	also tbmaths
DSINH	maths library	dblmath	also tbmaths
DSQRT	compiler predefine		
DTAN	maths library	dblmath	also tbmaths
DTANH	maths library	dblmath	also tbmaths
ELSE	language keyword		
eqstr	io library	string	
EXP	maths library	snglmath	also tbmaths
EXTERNAL	compiler directive		
FALSE	language keyword		
FLOATING.UNPACK	compiler predefine		
FOR	language keyword		
FPINT	compiler predefine		
FRACMUL	compiler predefine		
FROM	language keyword		
FUNCTION	language keyword		
GUY	compiler keyword		
HEX16TOSTRING	io library	convert	
HEX32TOSTRING	io library	convert	
HEX64TOSTRING	io library	convert	
HEXTOSTRING	io library	convert	
IEEE32OP	compiler predefine		
IEEE32REM	compiler predefine		
IEEE64OP	compiler predefine		
IEEE64REM	compiler predefine		
IEEECOMPARE	compiler predefine		
IF	language keyword		
IMPORT	compiler directive		
IN	language keyword		
INCLUDE	compiler directive		
INLINE	compiler keyword		
InputOrFail.c	system library	xlink	
InputOrFail.t	system library	xlink	
insert.string	io library	string	
INT	language keyword		
INT16	language keyword		
INT16TOSTRING	io library	convert	
INT32	language keyword		
INT32TOSTRING	io library	convert	
INT64	language keyword		

Name	Class	Library	Notes
INT64TOSTRING	io library	convert	
INTTOSTRING	io library	convert	
IS	language keyword		
is.digit	io library	string	
is.hex.digit	io library	string	
is.id.char	io library	string	
is.in.range	io library	string	
is.lower	io library	string	
is.upper	io library	string	
ISNAN	compiler predefine		
KERNEL.RUN	compiler predefine		
ks.keystream.sink	io library	streamio	
ks.keystream.to.scrstream	io library	streamio	
ks.read.char	io library	streamio	
ks.read.int	io library	streamio	
ks.read.int64	io library	streamio	
ks.read.line	io library	streamio	
ks.read.real32	io library	streamio	
ks.read.real64	io library	streamio	

Name	Class	Libr	Notes
LINKAGE	compiler directive		
LOAD.BYTE.VECTOR	compiler predefine		
LOAD.INPUT.CHANNEL	compiler predefine		
LOAD.INPUT.CHANNEL.VECTOR	compiler predefine		
LOAD.OUTPUT.CHANNEL	compiler predefine		
LOAD.OUTPUT.CHANNEL.VECTOR	compiler predefine		
LOGB	compiler predefine		
LONGADD	compiler predefine		
LONGDIFF	compiler predefine		
LONGDIV	compiler predefine		
LONGPROD	compiler predefine		
LONGSUB	compiler predefine		
LONGSUM	compiler predefine		
MINUS	language keyword		
MINUSX	compiler predefine		
MOSTNEG	language keyword		
MOSTPOS	language keyword		
MOVE2D	compiler predefine		
MULBY2	compiler predefine		
next.int.from.line	io library	string	
next.word.from.line	io library	string	
NEXTAFTER	compiler predefine		
NORMALISE	compiler predefine		
NOT	language keyword		
NOTFINITE	compiler predefine		
OF	language keyword		
OPTION	compiler directive		
OR	language keyword		
ORDERED	compiler predefine		
OutputOrFail.c	system library	xlink	
OutputOrFail.t	system library	xlink	
PAR	language keyword		
PLACE	language keyword		
PLACED	language keyword		
PLUS	language keyword		
PORT	language keyword		

Name	Class	Library	Notes
POWER	maths library	snglmath	also tbmaths
PRAGMA	compiler directive		
PRI	language keyword		
PROC	language keyword		
PROCESSOR	language keyword		
PROTOCOL	language keyword		
RAN	maths library	snglmath	also tbmaths
REAL32	language keyword		
REAL32EQ	compiler predefine		
REAL32GT	compiler predefine		
REAL32OP	compiler predefine		
REAL32REM	compiler predefine		
REAL32TOSTRING	io library	convert	
REAL64	language keyword		
REAL64EQ	compiler predefine		
REAL64GT	compiler predefine		
REAL64OP	compiler predefine		
REAL64REM	compiler predefine		
REAL64TOSTRING	io library	convert	
Reinitialise	system library	xlink	
REM	language keyword		
RESCHEDULE	compiler predefine		
RESULT	language keyword		
RETYPES	language keyword		
ROTATELEFT	compiler predefine		
ROTATERIGHT	compiler predefine		
ROUND	language keyword		
ROUNDSN	compiler predefine		not T2s
SC	compiler directive		
SCALEB	compiler predefine		
search.match	io library	string	
search.no.match	io library	string	
SEQ	language keyword		
SHIFTLEFT	compiler predefine		
SHIFTRIGHT	compiler predefine		
SIN	maths library	snglmath	also tbmaths
SINH	maths library	snglmath	also tbmaths

Name	Class	Library
SIZE	language keyword	
SKIP	language keyword	
so.ask	io library	hostio
so.buffer	io library	hostio
so.close	io library	hostio
so.commandline	io library	hostio
so.core	io library	hostio
so.date.to.ascii	io library	hostio
so.eof	io library	hostio
so.exit	io library	hostio
so.ferror	io library	hostio
so.flush	io library	hostio
so.fwrite.char	io library	hostio
so.fwrite.hex.int	io library	hostio
so.fwrite.hex.int32	io library	hostio
so.fwrite.hex.int64	io library	hostio
so.fwrite.int	io library	hostio
so.fwrite.int32	io library	hostio
so.fwrite.int64	io library	hostio
so.fwrite.nl	io library	hostio
so.fwrite.real32	io library	hostio
so.fwrite.real64	io library	hostio
so.fwrite.string	io library	hostio
so.fwrite.string.nl	io library	hostio
so.getenv	io library	hostio
so.getkey	io library	hostio
so.gets	io library	hostio
so.keystream.from.file	io library	streamio
so.keystream.from.kbd	io library	streamio
so.keystream.from.stdin	io library	streamio
so.multiplexor	io library	hostio
so.open	io library	hostio
so.open.temp	io library	hostio
so.overlapped.buffer	io library	hostio
so.overlapped.multiplexor	io library	hostio
so.overlapped.pri.multiplexor	io library	hostio

Name	Class	Library	Notes
so.parse.command.line	io library	hostio	
so.pollkey	io library	hostio	
so.popen.read	io library	hostio	
so.pri.multiplexor	io library	hostio	
so.puts	io library	hostio	
so.read	io library	hostio	
so.read.echo.any.int	io library	hostio	
so.read.echo.hex.int	io library	hostio	
so.read.echo.hex.int32	io library	hostio	
so.read.echo.hex.int64	io library	hostio	
so.read.echo.int	io library	hostio	
so.read.echo.int32	io library	hostio	
so.read.echo.int64	io library	hostio	
so.read.echo.line	io library	hostio	
so.read.echo.real32	io library	hostio	
so.read.echo.real64	io library	hostio	
so.read.line	io library	hostio	
so.remove	io library	hostio	
so.rename	io library	hostio	
so.scrstream.to.ANSI	io library	streamio	
so.scrstream.to.file	io.library	streamio	
so.scrstream.to.stdout	io library	streamio	
so.scrstream.to.TVI920	io library	streamio	
so.seek	io library	hostio	
so.system	io library	hostio	
so.tell	io library	hostio	
so.test.exists	io library	hostio	
so.time	io library	hostio	
so.time.to.ascii	io library	hostio	
so.time.to.date	io library	hostio	
so.today.ascii	io library	hostio	
so.today.date	io library	hostio	
so.version	io library	hostio	
so.write	io library	hostio	
so.write.char	io library	hostio	
so.write.hex.int	io library	hostio	
so.write.hex.int32	io library	hostio	
so.write.hex.int64	io library	hostio	
so.write.int	io library	hostio	

Name	Class	Library
so.write.int32	io library	hostio
so.write.int64	io library	hostio
so.write.nl	io library	hostio
so.write.real32	io library	hostio
so.write.real64	io library	hostio
so.write.string	io library	hostio
so.write.string.nl	io library	hostio
sp.buffer	io library	hostio
sp.close	io library	hostio
sp.commandline	io library	hostio
sp.core	io library	hostio
sp.eof	io library	hostio
sp.exit	io library	hostio
sp.ferror	io library	hostio
sp.flush	io library	hostio
sp.getenv	io library	hostio
sp.getkey	io library	hostio
sp.gets	io library	hostio
sp.multiplexor	io library	hostio
sp.open	io library	hostio
sp.overlapped.buffer	io library	hostio
sp.overlapped.multiplexor	io library	hostio
sp.overlapped.pri.multiplexor	io library	hostio
sp.pollkey	io library	hostio
sp.pri.multiplexor	io library	hostio
sp.puts	io library	hostio
sp.read	io library	hostio
sp.receive.packet	io library	hostio
sp.remove	io library	hostio
sp.rename	io library	hostio
sp.seek	io library	hostio
sp.send.packet	io library	hostio
sp.system	io library	hostio
sp.tell	io library	hostio
sp.time	io library	hostio
sp.version	io library	hostio
sp.write	io library	hostio
SQRT	compiler predefine	
ss.beep	io library	streamio

Name	Class	Library	Notes
ss.clear.eol	io library	streamio	
ss.clear.eos	io library	streamio	
ss.del.line	io library	streamio	
ss.delete.chl	io library	streamio	
ss.delete.chr	io library	streamio	
ss.down	io library	streamio	
ss.goto.xy	io library	streamio	
ss.ins.line	io library	streamio	
ss.insert.char	io library	streamio	
ss.left	io library	streamio	
ss.right	io library	streamio	
ss.scrstream.copy	io library	streamio	
ss.scrstream.fan.out	io library	streamio	
ss.scrstream.from.array	io library	streamio	
ss.scrstream.multiplexor	io library	streamio	
ss.scrstream.sink	io library	streamio	
ss.scrstream.to.array	io library	streamio	
ss.up	io library	streamio	
ss.write.char	io library	streamio	
ss.write.endstream	io library	streamio	
ss.write.hex.int	io library	streamio	
ss.write.hex.int64	io library	streamio	
ss.write.int	io library	streamio	
ss.write.int64	io library	streamio	
ss.write.nl	io library	streamio	
ss.write.real32	io library	streamio	
ss.write.real64	io library	streamio	
ss.write.string	io library	streamio	
ss.write.text.line	io library	streamio	
STOP	language keyword		
str.shift	io library	string	
string.pos	io library	string	
STRINGTOBOOL	io library	convert	

Name	Class	Library	Notes
STRINGTOHEX	io library	convert	
STRINGTOHEX16	io library	convert	
STRINGTOHEX32	io library	convert	
STRINGTOHEX64	io library	convert	
STRINGTOINT16	io library	convert	
STRINGTOINT32	io library	convert	
STRINGTOINT64	io library	convert	
STRINGTOINT	io library	convert	
STRINGTOREAL32	io library	convert	
STRINGTOREAL64	io library	convert	
TAN	maths library	snglmath	also tbmaths
TANH	maths library	snglmath	also tbmaths
TIMER	language keyword		
TIMES	language keyword		
to.lower.case	io library	string	
to.upper.case	io library	string	
TRANSLATE	compiler directive		
TRUE	language keyword		
TRUNC	language keyword		
UNPACKSN	compiler predefine		not T2s
USE	compiler directive		
VAL	language keyword		
VALOF	language keyword		
VECSPACE	compiler keyword		
WHILE	language keyword		
WORKSPACE	compiler keyword		

B Transputer instruction set support

This appendix contains the list of transputer instructions supported by the toolset restricted code insertion facility, and gives the mnemonic for each instruction. All the instructions listed can be inserted into occam programs using the **ASM** construct. The appendix ends with a summary of the differences between the **ASM** and **GUY** constructs and describes the restrictions placed on the use of **GUY** code.

The instructions described are available when the compiler is targetted to an IMS T212, M212, T222, T225, T400, T414, T425, T800, T801, or T805 unless otherwise indicated. Instructions that are only supported when the compiler is targetted to certain processor types, are given in separate sections. The reader is referred to the *'Transputer instruction set: a compiler writer's guide'* for further details of the instruction set. Details of the instructions listed in section B.8 are given in *'The transputer databook'*.

B.1 Pseudo-instructions

Pseudo-instructions are instructions to the assembler, rather than true transputer instructions.

Expressions used in *load* or *store* pseudo instructions must be word sized or smaller. To load a floating point value, use a **LD** to load its address, then a **FPLDNLSN** or **FPLDNLDB** as required.

The following pseudo-instructions are implemented:

BYTE	This instruction takes as an argument a list of constant values in the range 0 to 255, or a list of (constant) byte arrays or strings. The assembler copies the literal bytes into the instruction stream.
LD	Loads a value into the **Areg**.
LDAB	Loads values into the **Areg** and **Breg**. The left hand expression is placed in **Areg**.
LDABC	Loads values into **Areg**, **Breg** and **Creg**. The leftmost expression is placed in **Areg**.
LDLABELDIFF	Loads the difference between the addresses of two labels into **Areg**.

`ST`	Stores the value from the **Areg**.
`STAB`	Stores values into the **Areg** and **Breg**. The leftmost element receives **Areg**.
`STABC`	Stores values into the **Areg**, **Breg**, and **Creg**. The leftmost element receives **Areg**.
`WORD`	Generates constants of the target-machine word length. This instruction takes as an argument a list of INTs or INT (constant) arrays.

The `LD`, `LDAB`, `ST`, and `STAB` instructions may use other registers and/or temporaries. `LDABC` and `STABC` may use temporaries.

B.2 Prefixing instructions

The transputer instruction set is built up from 16 *direct* instructions, each with a 4-bit argument field. The direct instructions include *prefix* instructions which augment the 4-bit field in a direct instruction which follows them by their own 4-bit argument field, effectively allowing the argument to be extended to 32 bits. Normally, the assembler will compute the prefix instructions required for operand values greater than 4 bits automatically.

`PFIX`	prefix
`NFIX`	negative prefix

B.3 Direct instructions

The direct instructions form the core of the transputer instruction set. Each direct instruction has a single operand, normally an integer constant, which will be encoded in the instruction itself and, if it is larger than will fit into the 4-bit argument field of the direct instruction, into a series of **PFIX** and **NFIX** instructions as well.

The transputer architecture is based around a three-register *evaluation stack* and a single base register **Wreg**. The load and store 'local' instructions access a word in memory at a displacement from **Wreg** given by the operand value used. The displacement is scaled by the word size. The load and store 'non-local' instructions use the top evaluation stack register (**Areg**) as the base instead of **Wreg**, allowing computed base addresses to be used.

The operand of the **J**, **CJ** and **CALL** instructions is interpreted as a byte displacement from the instruction pointer (program counter) register **Iptr**. **LDPI** is similar but takes its operand from **Areg**.

ADC	Add constant operand value to **Areg**
AJW	Adjust workspace pointer **Wreg** by constant operand value (scaled by word length)
CALL	Call
CJ	Conditional jump i.e. 'jump if zero otherwise pop **Areg**'. As with **JUMP**, a label identifier may be used as argument to this instruction.
EQC	Test if **Areg** equals constant; **Areg** gets 1/0 result
J	Jump: the argument may be an identifier indicating a label for the jump to go to; the assembler will compute the displacement required.
LDC	Load constant
LDL	Load local word
LDLP	Load pointer to local word
LDNL	Load non-local word
LDNLP	Load pointer to non-local word
OPR	'operate': the argument to this instruction is a code indicating a zero-operand *indirect* instruction to be executed. Most of the transputer instruction set is made up of these indirect instructions. Normally you would use the mnemonic for the specific indirect instruction which you require: the assembler will encode this as an **opr** instruction on your behalf. However, it is possible to use **opr** explicitly, for example to synthesise the instruction sequence for a new indirect instruction not supported by the T414 and T800 transputers.
STL	Store local word
STNL	Store non-local word

B.4 Operations

The instructions in this section are all *indirect* instructions built out of the OPR instruction. None of these instructions take an argument; instead, they work solely with the transputer evaluation stack.

The arithmetic instructions take their operands from the top of the evaluation stack (**Areg**, **Breg**) and push the result value back on the stack in **Areg**.

B.4.1 Short indirect instructions

ADD	Add
BSUB	Byte subscript (**Areg** = **Areg** + **Breg**)
DIFF	Difference
GT	Greater than (result 'true' or 'false', placed in **Areg**)
LB	Load byte
PROD	Product
REV	Reverse top two stack elements
SUB	Subtract
WSUB	Word subscript (**Areg** = **Areg** + 4****Breg**) (32-bit)
	Word subscript (**Areg** = **Areg** + 2****Breg**) (16-bit)

B.4.2 Long indirect instructions

AND	Bit-wise and
BCNT	Byte count
CCNT1	Check count from 1
CSNGL	Check single
CSUB0	Check subscript from 0
CWORD	Check word
DIV	Divide
FMUL	Fractional multiply (32-bit processors only)
LADD	Long add
LDIFF	Long difference
LDIV	Long divide

LDPI	Load pointer to instruction (**Areg** is byte displacement from **Iptr**)
LDPRI	Load current priority
LDTIMER	Load timer
LMUL	Long multiply
LSHL	Long shift left
LSHR	Long shift right
LSUB	Long subtract
LSUM	Long sum
MINT	Minimum integer
MOVE	Move block of memory (src: **Creg** dest: **Breg** len: **Areg**)
MUL	Multiply
NORM	Normalise
NOT	Bit-wise not
OR	Bit-wise inclusive or
REM	Remainder
SB	Store byte
SETERR	Set error
SHL	Shift left
SHR	Shift right
STTIMER	Store timer
SUM	Sum
TESTERR	Test error false and clear
TESTHALTERR	Test halt-on-error
TESTPRANAL	Test processor analysing
WCNT	Word count
XDBLE	Extend to double
XOR	Bit-wise exclusive or
XWORD	Extend to word

B.5 Additional instructions for the T400, T414, T425 and TB

The indirect instructions in this section may only be executed on a T400, T414 or T425 processor.

`CFLERR`	Check single-length floating-point infinity or not-a-number
`LDINF`	Load single-length infinity
`POSTNORMSN`	Post-normalise correction of single-length floating-point number
`ROUNDSN`	Round single-length floating-point number
`UNPACKSN`	Unpack single-length floating-point number

B.6 Additional instructions for the IMS T800, T801 and T805

The instructions in this section may only be executed on T800, T801 and T805 processors.

B.6.1 Floating-point instructions

The indirect instructions in this section provide access to the T8 series built-in floating-point processor. Note that the instructions beginning with '`FPU...`' are doubly indirect: they are accessed by loading an *entry code* constant with a `LDC` instruction, then executing an `FPENTRY` instruction, which is itself indirect. As with ordinary indirect instructions, this indirection is handled transparently by the assembler, although the `FPENTRY` instruction is also available.

The floating point load and store instructions use the *integer* **Areg** as a pointer to the operand location.

`FPADD`	Floating-point add
`FPB32TOR64`	Convert 32-bit unsigned integer to 64-bit real
`FPCHKERR`	Check floating error
`FPDIV`	Floating-point divide
`FPDUP`	Floating duplicate
`FPENTRY`	Floating point unit entry: used to synthesise the '`FPU...`' instructions.
`FPEQ`	Floating point equality
`FPGT`	Floating point greater than

FPI32TOR32	Convert 32-bit integer to 32-bit real
FPI32TOR64	Convert 32-bit integer to 64-bit real
FPINT	Round to floating integer
FPLDNLADDDB	Floating load non-local and add double
FPLDNLADDSN	Floating load non-local and add single
FPLDNLDB	Floating load non-local double
FPLDNLDBI	Floating load non-local indexed double
FPLDNLMULDB	Floating load non-local and multiply double
FPLDNLMULSN	Floating load non-local and multiply single
FPLDNLSN	Floating load non-local single
FPLDNLSNI	Floating load non-local indexed single
FPLDZERODB	Fload zero double
FPLDZEROSN	Load zero single
FPMUL	Floating-point multiply
FPNAN	Floating point not-a-number
FPNOTFINITE	Floating point finite
FPORDERED	Floating point orderability
FPREMFIRST	Floating-point remainder first step
FPREMSTEP	Floating-point remainder iteration step
FPREV	Floating reverse
FPRTOI32	Convert floating to 32-bit integer
FPSTNLDB	Floating store non-local double
FPSTNLI32	Store non local int32
FPSTNLSN	Floating store non-local single
FPSUB	Floating-point subtract
FPTESTERR	Test floating error false and clear
FPUABS	Floating-point absolute
FPUCHKI32	Check in range of 32-bit integer
FPUCHKI64	Check in range of 64-bit integer
FPUCLRERR	Clear floating error
FPUDIVBY2	Divide by 2.0
FPUEXPDEC32	Divide by 2^{32}
FPUEXPINC32	Multiply by 2^{32}
FPUMULBY2	Multiply by 2.0
FPUNOROUND	Convert 64-bit real to 32-bit real without rounding

FPUR32TOR64	Convert single to double
FPUR64TOR32	Convert double to single
FPURM	Set rounding mode to round minus
FPURN	Set rounding mode to round nearest
FPURP	Set rounding mode to round positive
FPURZ	Set rounding mode to round zero
FPUSETERR	Set floating error
FPUSQRTFIRST	Floating-point square root first step
FPUSQRTLAST	Floating-point square root end
FPUSQRTSTEP	Floating-point square root step

B.7 Additional instructions for the IMS T225, T400, T425, T800, T801 and T805

The indirect instructions in this section supplement the T414's integer instruction set.

BITCNT	Count the number of bits set in a word
BITREVNBITS	Reverse bottom n bits in a word
BITREVWORD	Reverse bits in a word
CRCBYTE	Calculate CRC on byte
CRCWORD	Calculate Cyclic Redundancy Check (CRC) on word
DUP	Duplicate top of stack
WSUBDB	Form double-word subscript

The following 2-dimensional block move instructions apply to the **IMS T400, T425, T800, T801 and T805** only:

MOVE2DALL	2-dimensional block copy
MOVE2DINIT	Initialise data for 2-dimensional block move
MOVE2DNONZERO	2-dimensional block copy non-zero bytes
MOVE2DZERO	2-dimensional block copy zero bytes

B.8 Additional instructions for the IMS T225, T400, T425, T801 and T805

The indirect instructions listed in this section provide debugging and general support functions.

CLRJ0BREAK	Clear jump 0 break enable flag
SETJ0BREAK	Set jump 0 break enable flag
TESTJ0BREAK	Test if jump 0 break flag is set
TIMERDISABLEH	Disable high priority timer interrupt
TIMERDISABLEL	Disable low priority timer interrupt
TIMERENABLEH	Enable high priority timer interrupt
TIMERENABLEL	Enable low priority timer interrupt
LDMEMSTARTVAL	Load value of **MemStart** address
POP	Pop processor stack
LDDEVID	Load device identity

B.9 Differences between ASM and GUY

The `ASM` construct has very different semantics to `GUY` code. This means that simply changing the word '`GUY`' to '`ASM`' within your code, will usually break the code.

The following list summarises the differences between `GUY` and `ASM` code and outlines the restrictions now placed on using `GUY` constructs.

- A primary instruction in `ASM` code always generates that primary instruction in the object file; this was not always the case with the `GUY` construct.
- There are differences in the instructions used to perform *load* and *store* operations depending on whether a `GUY` or `ASM` construct is used.
  - The statements `LDL x` and `LDNL x` in `GUY` code both generate code which behaves as `LD x` in `ASM` code. They may generate one or more transputer instructions.
  - The statements `LDLP x` and `LDNLP x` in `GUY` code both generate code which behaves as `LD ADDRESSOF x` in `ASM` code. They may generate one or more transputer instructions.
  - The statements `STL x` and `STNL x` in `GUY` code both generate code which behaves as `ST x` in `ASM` code. They may generate one or more transputer instructions.

  - If a **GUY** construct is changed to an **ASM** construct then changes of loads and stores using any of these primary operations to the corresponding pseudo-operations should always be performed.
  - Use of these primary operations directly in **ASM** code is not usually necessary or desirable. In **ASM** code each primary load or store statement will generate a single, possibly prefixed, transputer instruction, whose operand is the offset within workspace of the variable named. Whether the location at this offset is a value or a pointer depends on whether the name is of a local variable (or value parameter) or not.

- References to labels in **GUY** code are preceded by a full stop whereas in **ASM** they are preceded by a colon.

- Symbolic access to channels is not permitted in **GUY** code although it was in previous releases of the toolset (i.e. the IMS D705/D605/D505 products). This is due to the fact that the internal representation of channels has changed; the base data type of a channel is now 'pointer to channel'. (See part 1 section 10.1.3).

  In **ASM** code, **LD** c will return the address of the channel, whereas **LD ADDRESSOF** c will return the address of a pointer to the channel.

C Constants

This appendix lists the constants provided with the occam libraries. The constants are supplied in text files and are given the extension `.inc` (for 'include'). These files should be placed on the path specified by the `ISEARCH` environment variable.

There are six separate files containing toolset constants, as follows:

File	Contents
`hostio.inc`	Hostio values and protocols
`streamio.inc`	Streamio values and protocols
`mathvals.inc`	Mathematical constants
`linkaddr.inc`	Transputer link addresses
`ticks.inc`	Rates of the two transputer clocks
`msdos.inc`	DOS specific constants

To use any of these files in a program, incorporate the file into the source using the `#INCLUDE` directive as follows:

```
#INCLUDE "hostio.inc"
```

Constants must be declared before they are used in a program or library.

C.1 Hostio constants

```
-- hostio.inc
-- Copyright 1989 INMOS Limited
-- updated for iserver v1.42 apart from buffer size 5-June-90 SRH
-- SP protocol
PROTOCOL SP IS INT16::[]BYTE :

-- Command tags
-- values up to 127 are reserved for use by INMOS
--  File command tags
VAL sp.open.tag     IS 10(BYTE) :
VAL sp.close.tag    IS 11(BYTE) :
VAL sp.read.tag     IS 12(BYTE) :
VAL sp.write.tag    IS 13(BYTE) :
VAL sp.gets.tag     IS 14(BYTE) :
VAL sp.puts.tag     IS 15(BYTE) :
VAL sp.flush.tag    IS 16(BYTE) :
VAL sp.seek.tag     IS 17(BYTE) :
VAL sp.tell.tag     IS 18(BYTE) :
VAL sp.eof.tag      IS 19(BYTE) :
VAL sp.ferror.tag   IS 20(BYTE) :
VAL sp.remove.tag   IS 21(BYTE) :
VAL sp.rename.tag   IS 22(BYTE) :
```

```
VAL sp.getblock.tag IS 23(BYTE) :
VAL sp.putblock.tag IS 24(BYTE) :
VAL sp.isatty.tag   IS 25(BYTE) :

--  Host command tags
VAL sp.getkey.tag  IS 30(BYTE) :
VAL sp.pollkey.tag IS 31(BYTE) :
VAL sp.getenv.tag  IS 32(BYTE) :
VAL sp.time.tag    IS 33(BYTE) :
VAL sp.system.tag  IS 34(BYTE) :
VAL sp.exit.tag    IS 35(BYTE) :

--  Server command tags
VAL sp.commandline.tag IS 40(BYTE) :
VAL sp.core.tag        IS 41(BYTE) :
VAL sp.version.tag     IS 42(BYTE) :

--  OS specific command tags
--  These OS specific tags will be followed by another tag
--  indicating which OS specific function is required

VAL sp.DOS.tag    IS 50(BYTE) :
VAL sp.HELIOS.tag IS 51(BYTE) :
VAL sp.VMS.tag    IS 52(BYTE) :
VAL sp.SUNOS.tag  IS 53(BYTE) :


-- Packet and buffer Sizes
VAL sp.max.packet.size IS 512 :
-- bytes transferred, includes length & data
VAL sp.min.packet.size IS   8 :
-- bytes transferred, includes length & data

VAL sp.max.packet.data.size IS sp.max.packet.size - 2 :
-- INT16 length
VAL sp.min.packet.data.size IS sp.min.packet.size - 2 :
-- INT16 length

--  Individual command maxima
VAL sp.max.openname.size      IS sp.max.packet.data.size - 5 :
-- 5 bytes extra
VAL sp.max.readbuffer.size    IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
-- ditto for gets
VAL sp.max.writebuffer.size   IS sp.max.packet.data.size - 7 :
-- 7 bytes extra
-- ditto for puts
VAL sp.max.removename.size    IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
VAL sp.max.renamename.size    IS sp.max.packet.data.size - 5 :
-- 5 bytes extra
VAL sp.max.getenvname.size    IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
VAL sp.max.systemcommand.size IS sp.max.packet.data.size - 3 :
-- 3 bytes extra
```

```
VAL sp.max.corerequest.size   IS sp.max.packet.data.size - 3 :
-- 3 bytes extra

VAL sp.max.buffer.size IS sp.max.writebuffer.size :
-- smaller of read & write

-- Result values          (spr.)

VAL spr.ok                    IS   0(BYTE) :
-- success

VAL spr.not.implemented       IS   1(BYTE) :
VAL spr.bad.name              IS   2(BYTE) :
-- filename is null
VAL spr.bad.type              IS   3(BYTE) :
-- open file type is incorrect
VAL spr.bad.mode              IS   4(BYTE) :
-- open file mode is incorrect
VAL spr.invalid.streamid      IS   5(BYTE) :
-- never opened that streamid
VAL spr.bad.stream.use        IS   6(BYTE) :
-- reading an output file, or vice versa
VAL spr.buffer.overflow       IS   7(BYTE) :
-- buffer too small for required data
VAL spr.bad.packet.size       IS   8(BYTE) :
-- data too big or small for packet
VAL spr.bad.origin            IS   9(BYTE) :
-- seek origin is incorrect
VAL spr.full.name.too.short IS  10(BYTE) :
-- a truncation of a filename would be required
VAL spr.notok                 IS 127(BYTE) :
-- a general fail result

-- anything 128 or above is a server dependent 'failure' result
VAL spr.operation.failed      IS 128(BYTE) :
-- general failure
VAL spr.failed.operation      IS 129(BYTE) :
-- identical in meaning to spr.operation.failed due
-- to historical accident

-- Predefined streams     (spid.)
VAL spid.stdin  IS 0(INT32) :
VAL spid.stdout IS 1(INT32) :
VAL spid.stderr IS 2(INT32) :

-- Open types             (spt.)
VAL spt.binary IS 1(BYTE) :
VAL spt.text   IS 2(BYTE) :

-- Open modes             (spm.)
VAL spm.input              IS 1(BYTE) :
VAL spm.output             IS 2(BYTE) :
VAL spm.append             IS 3(BYTE) :
VAL spm.existing.update    IS 4(BYTE) :
VAL spm.new.update         IS 5(BYTE) :
```

```
VAL spm.append.update   IS 6(BYTE) :

-- Status values         (sps.)
VAL sps.success IS  999999999(INT32) :
VAL sps.failure IS -999999999(INT32) :

-- Seek origins          (spo.)
VAL spo.start   IS 1(INT32) :
VAL spo.current IS 2(INT32) :
VAL spo.end     IS 3(INT32) :

-- Version information   (sph., spo., spb.)
--  Host types            (sph.)
-- values up to 127 are reserved for use by INMOS
VAL sph.PC         IS 1(BYTE) :
VAL sph.NECPC      IS 2(BYTE) :
VAL sph.VAX        IS 3(BYTE) :
VAL sph.SUN3       IS 4(BYTE) :
VAL sph.S370       IS 5(BYTE) :
VAL sph.BOX.SUN4   IS 6(BYTE) :
VAL sph.BOX.SUN386 IS 7(BYTE) :
VAL sph.BOX.APOLLO IS 8(BYTE) :

--  OS types              (spo.)
VAL spo.DOS    IS 1(BYTE) :
VAL spo.HELIOS IS 2(BYTE) :
VAL spo.VMS    IS 3(BYTE) :
VAL spo.SUNOS  IS 4(BYTE) :
VAL spo.CMS    IS 5(BYTE) :
-- values up to 127 are reserved for use by INMOS

--  Interface Board types (spb.)
--  This determines the interface between the link and the host
VAL spb.B004    IS 1(BYTE) :
VAL spb.B008    IS 2(BYTE) :
VAL spb.B010    IS 3(BYTE) :
VAL spb.B011    IS 4(BYTE) :
VAL spb.B014    IS 5(BYTE) :
VAL spb.DRX11   IS 6(BYTE) :
VAL spb.QT0     IS 7(BYTE) :
VAL spb.B015    IS 8(BYTE) :
VAL spb.IBMCAT  IS 9(BYTE) :
VAL spb.B016    IS 10(BYTE) :
VAL spb.UDPLINK IS 11(BYTE) :
-- values up to 127 are reserved for use by INMOS

-- Command line
VAL sp.short.commandline IS BYTE 0 :
-- remove  server's own arguments
VAL sp.whole.commandline IS BYTE 1 :
-- include server's own arguments

-- values for so.parse.commandline indicate whether
-- an option requires a following parameter
VAL spopt.never  IS 0 :
```

```
VAL spopt.maybe  IS 1 :
VAL spopt.always IS 2 :

-- Time string and date lengths
VAL so.time.string.len IS 19 :
-- enough for "HH:MM:SS DD/MM/YYYY"
VAL so.date.len        IS  6 :
-- enough for DDMMYY (as integers)

-- Temp filename length
VAL so.temp.filename.length IS 6 :
-- six chars will work on anything!
```

C.2 Streamio constants

```
-- streamio.inc
-- Copyright 1989 INMOS Limited
-- Updated to match TDS3 strmhdr list; 4-Feb-91 SRH
VAL st.max.string.size IS 256 :
VAL ft.terminated   IS  -8 : -- used to terminate a keystream
VAL ft.number.error IS -11 :
PROTOCOL KS IS INT:
PROTOCOL SS
  CASE
    st.reset
    st.up
    st.down
    st.left
    st.right
    st.goto; INT32; INT32
    st.ins.char; BYTE
    st.del.char
    st.out.string; INT32::[]BYTE
    st.clear.eol
    st.clear.eos
    st.ins.line
    st.del.line
    st.beep
    st.spare
    st.terminate
    st.help
    st.initialise
    st.out.byte; BYTE
    st.out.int; INT32
    st.key.raw
    st.key.cooked
    st.release
    st.claim
    st.endstream
    st.set.poll; INT32
:
```

C.3 Maths constants

```
-- mathvals.inc
-- Copyright 1989 INMOS Limited
-- Appended the error condition NaNs for the implementation of
-- the maths libraries; 4/Oct/90 SRH
--{{{  Maths constants

--{{{  REAL32 Constants
VAL REAL32 INFINITY RETYPES #7F800000(INT32) :
VAL REAL32 MINREAL  RETYPES #00000001(INT32) :
-- 1.40129846E-45
VAL REAL32 MAXREAL  RETYPES #7F7FFFFF(INT32) :
-- 3.40282347E+38
VAL REAL32 E        RETYPES #402DF854(INT32) :
-- 2.71828174E+00
VAL REAL32 PI       RETYPES #40490FDB(INT32) :
-- 3.14159274E+00
VAL REAL32 LOGE2    RETYPES #3F317218(INT32) :
-- 6.93147182E-01
VAL REAL32 LOG10E   RETYPES #3EDE5BD9(INT32) :
-- 4.34294492E-01
VAL REAL32 ROOT2    RETYPES #3FB504F3(INT32) :
-- 1.41421354E+00
VAL LOGEPI IS 1.1447298858(REAL32) :
-- log to the base e of pi
VAL RADIAN IS 57.295779513(REAL32) :
-- the number of degrees in 1 radian
VAL DEGREE IS 1.74532925199E-2(REAL32) :
-- the number of radians in 1 degree
VAL GAMMA  IS 0.5772156649(REAL32) :
-- Euler's constant

--{{{  implementation defined NaNs
VAL REAL32 undefined.NaN RETYPES #7F800010(INT32) :
VAL REAL32 unstable.NaN  RETYPES #7F800008(INT32) :
VAL REAL32 inexact.NaN   RETYPES #7F800004(INT32) :
--}}}
--}}}

--{{{  REAL64 Constants
VAL REAL64 DINFINITY RETYPES #7FF0000000000000(INT64):
VAL REAL64 DMINREAL  RETYPES #0000000000000001(INT64):
-- 4.9406564584124654E-324
VAL REAL64 DMAXREAL  RETYPES #7FEFFFFFFFFFFFFF(INT64):
-- 1.7976931348623157E+308
VAL REAL64 DE        RETYPES #4005BF0A8B145769(INT64):
-- 2.7182818284590451E+000
VAL REAL64 DPI       RETYPES #400921FB54442D18(INT64):
-- 3.1415926535897931E+000
VAL REAL64 DLOGE2    RETYPES #3FE62E42FEFA39EF(INT64):
-- 6.9314718055994529E-001
VAL REAL64 DLOG10E   RETYPES #3FDBCB7B1526E50E(INT64):
-- 4.3429448190325182E-001
VAL REAL64 DROOT2    RETYPES #3FF6A09E667F3BCD(INT64):
```

```
-- 1.4142135623730951E+000
VAL DLOGEPI IS 1.1447298858494001741(REAL64) :
-- log to the base e of pi
VAL DRADIAN IS 57.295779513082320877(REAL64) :
-- the number of degrees in 1 radian
VAL DDEGREE IS 1.7453292519943295769E-2(REAL64) :
-- the number of radians in 1 degree
VAL DGAMMA  IS 0.57721566490153286061(REAL64) :
-- Euler's constant

--{{{  implementation defined NaNs
VAL REAL64 Dundefined.NaN RETYPES #7FF0000200000000(INT64) :
VAL REAL64 Dunstable.NaN  RETYPES #7FF0000100000000(INT64) :
VAL REAL64 Dinexact.NaN   RETYPES #7FF0000080000000(INT64) :
--}}}
--}}}
--}}}
```

C.4 Transputer link addresses

```
-- linkaddr.inc
-- Copyright 1989 INMOS Limited

-- Transputer link addresses

VAL link0.in  IS 4:
VAL link0.out IS 0:

VAL link1.in  IS 5:
VAL link1.out IS 1:

VAL link2.in  IS 6:
VAL link2.out IS 2:

VAL link3.in  IS 7:
VAL link3.out IS 3:


-- Transputer event address

VAL event.in  IS 8:
```

C.5 Rates of the transputer clocks

```
-- ticks.inc
-- V1.0, 09/May/90
-- Copyright 1990 INMOS Limited
-- These values are not for the A revision of the T414
-- (which is no longer supported ).
```

```
-- these values are the rates at which the two priority clocks
-- increment on the transputer
VAL lo.ticks.per.second IS   15625 ( INT32 ) :
VAL hi.ticks.per.second IS 1000000 ( INT32 ) :

VAL lo.tick.in.micro.seconds IS 64 ( INT ) :
--   1000000 / lo.ticks.per.second
VAL hi.tick.in.micro.seconds IS  1 ( INT ) :
--   1000000 / hi.ticks.per.second
```

C.6 DOS specific constants

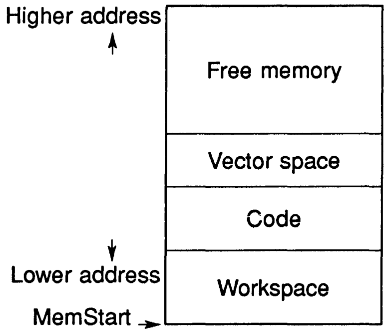
```
-- msdos.inc
-- Copyright 1989 INMOS Limited
--   DOS command tags
VAL dos.send.block.tag      IS  0(BYTE) :
VAL dos.receive.block.tag   IS  1(BYTE) :
VAL dos.call.interrupt.tag  IS  2(BYTE) :
VAL dos.read.regs.tag       IS  3(BYTE) :
VAL dos.port.write.tag      IS  4(BYTE) :
VAL dos.port.read.tag       IS  5(BYTE) :

--   call.interrupt register layout
VAL dos.interrupt.regs.size IS 40 :

VAL dos.interrupt.regs.ax   IS  0 :
VAL dos.interrupt.regs.bx   IS  4 :
VAL dos.interrupt.regs.cx   IS  8 :
VAL dos.interrupt.regs.dx   IS 12 :
VAL dos.interrupt.regs.di   IS 16 :
VAL dos.interrupt.regs.si   IS 20 :
VAL dos.interrupt.regs.cs   IS 24 :
VAL dos.interrupt.regs.ds   IS 28 :
VAL dos.interrupt.regs.es   IS 32 :
VAL dos.interrupt.regs.ss   IS 36 :

--   read.regs register layout
VAL dos.read.regs.size IS 16 :

VAL dos.read.regs.cs   IS  0 :
VAL dos.read.regs.ds   IS  4 :
VAL dos.read.regs.es   IS  8 :
VAL dos.read.regs.ss   IS 12 :

--   buffer sizes (These depend on sp.max.packet.data.size)
VAL dos.max.send.block.buffer.size IS
    sp.max.packet.data.size - 8 :
VAL dos.max.receive.block.buffer.size IS
    sp.max.packet.data.size - 3 :

-- this is the smaller of send & receive
VAL dos.max.block.buffer.size IS
    dos.max.send.block.buffer.size :
```

D Implementation of occam on the transputer

This appendix defines the toolset implementation of occam on the transputer. It describes how the compiler allocates memory and gives details of type mapping, hardware dependencies and language. The appendix ends with the syntax definition of the language extensions implemented by the occam compiler.

D.1 Memory allocation by the compiler

The code for a whole program occupies a contiguous section of memory. When a program is loaded onto a transputer in a network, memory is allocated in the following order starting at `MemStart`: workspace; code; separate vector space. this is shown below:

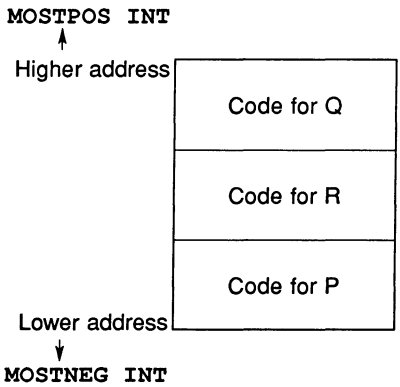


D.1.1 Procedure code

The compiler places the code for any nested procedures at higher addresses (nearer `MOSTPOS INT`) than the code for the enclosing procedure. Nested procedures are placed at increasingly lower addresses in the order in which their definitions are completed. For the code in the following example:

```
PROC P()
  PROC Q ()
    ...  code for Q
  :
  PROC R ()
    ...  code for R
  :
  ...  code for P
:
```

the layout of the code in memory is:



Note: this is a change from the previous release of the occam compiler in the IMS D705/D605/D505 products.

D.1.2 Compilation modules

The order in which compilation modules are placed in memory, including those referenced by a **#PRAGMA LINKAGE** directive, is controlled by a linker directive. Modules are placed in priority order, with the highest priority module being placed at the lowest available address.

Note: the compiler will attempt to optimise floating point routines such as **REAL32OP** and **REAL32OPERR** by giving them a high priority. This can be overridden by using the compiler directive **#PRAGMA LINKAGE** in conjunction with the linker directive **#SECTION**.

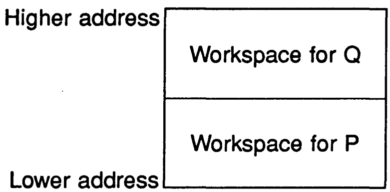
D.1.3 Workspace

Workspace is given priority usage of the on-chip RAM, before the arithmetic handling library.

Workspace is allocated from higher to lower address (i.e. the workspace for a called procedure is nearer **MOSTNEG INT** than the workspace for the caller). For example:

```
PROC P ()
  ...  code
  here
  ...  code
:
PROC Q ()
  P ()
:
```

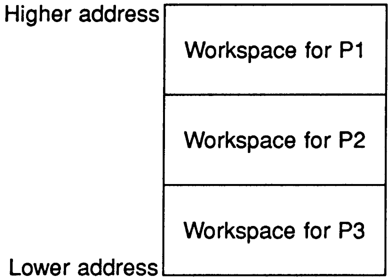
In the above example when Q is called, it will in turn call P. At the point labelled **here**, the data layout in memory will be:



In a **PAR** or **PRI PAR** construct the last textually defined process is allocated the lowest addressed workspace. For example:

```
PAR
  ... P1
  ... P2
  ... P3
```

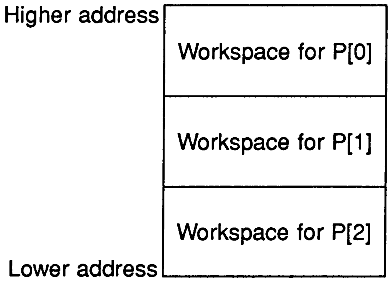
the workspace layout for the parallel processes will be:



In a replicated **PAR** construct the instance with the highest replication count is allocated the lowest workspace address. For example:

```
PAR i = 0 FOR 3
  P [i]
```

the workspace layout for the parallel processes will be:



Unless separate vector space is disabled, arrays larger than 8 bytes (apart from those explicitly placed in the workspace) are allocated in a separate data space, known as vector space. The allocation is done in a similar way to the allocation of workspace, except that the data space for a called procedure is at a *higher* address than the data space of its caller.

Arrays whose elements are word-sized or longer, and which occupy 8 bytes or less, remain in workspace eg.

```
[2]INT32
```

will be placed in workspace.

The variables within a single procedure or parallel process are allocated on the basis of their estimated usage. The variables which the compiler estimates will be used the most, are allocated lower addresses in the current workspace.

From within a called procedure the parameters appear immediately above the local variables. When an unsized vector is declared as a formal procedure parameter an extra **VAL INT** parameter is also allocated to store the size of the array passed as the actual parameter. This size is the number of elements in the array. One extra parameter is supplied for each dimension of the array unsized in the call, in the order in which they appear in the declaration.

If a procedure requires separate vector space, it is supplied by the calling procedure. A pointer to the vector space supplied is given as an additional parameter. If the procedure is at the outer level of a compilation unit, the vector space pointer is supplied after all the actual parameters. Otherwise it is supplied before all the actual parameters.

D.2 Type mapping

This section defines all the occam types and how they are represented on the each target processor.

All objects are word aligned, ie. the lowest byte of the object is on a word boundary. For objects of type **BOOL** and **BYTE**, the padding above the object is guaranteed to be all bits zero: for all other objects, the value of any padding bytes is undefined.

Arrays are packed, ie. there are no spaces between the elements. (**Note**: that an object of type **BOOL** has one byte for each element).

Table D.1 summarizes the type mapping, for further information on data types see Section 3 of the occam *2 Reference Manual.*

Protocol tags are represented by 8-bit values. The compiler allocates tag values for each protocol from 0 **(BYTE)** upwards in order of declaration.

Values accessed through **RETYPES** must be aligned to the natural alignment for that data type; **BYTE**s and **BOOL**s may be aligned to any byte; **INT16**s on a 32 bit processor must be aligned to a half-word boundary and all other data types must be aligned to a word boundary. This will be checked at run-time if it cannot be checked at compile time. For example:

```
[20]BYTE array:                       -- This will be word aligned
INT32 x RETYPES [array FROM 1 FOR 4] : -- Run-time check is
                                       -- inserted
INT32 y RETYPES [array FROM i FOR 4] : -- Run-time check is
                                       -- inserted
INT32 z RETYPES [array FROM 8 FOR 4] : -- No run-time check
                                       -- inserted
```

Type	*Storage*	*Range of values*
BOOL BYTE	1 byte 1 byte	FALSE, TRUE 0 to 255
INT16 INT32 INT64	2 bytes 4 bytes 8 bytes	-32768 to 32767 -2,147,483,648 to 2,147,483,647 -2^{63} to $(2^{63}-1)$
INT On T400/T414/T425 T800/T801/T805	4 bytes	-2,147,483,648 to 2,147,483,647
INT On T212/T222/T225 M212	2 bytes	-32768 to 32767
REAL32 REAL64	4 bytes 8 bytes	IEEE single precision format IEEE double precision format
CHAN on T400/T414/T425 T800/T801/T805 CHAN on T212/T222/T225 M212	8 bytes 4 bytes	Channels are implemented as a pointer to a channel word.
PORT OF D TIMER	as for D none	

Table D.1 occam data types

Channels may be **RETYPE**d. This allows the protocol on a channel to be changed, in order to pass it as a parameter to another routine. This facility should be used with care.

D.3 Hardware dependencies

- The number of priorities supported by the transputer is 2, (i.e. high and low), so a **PRI PAR** may have two component processes. The compiler does not permit a **PRI PAR** statement to be nested inside the high priority branch of another. This is checked at compile time, even across separately compiled units.
- The low priority clock increments at a rate of 15625 ticks per second, or

one tick = 64 microseconds (IMS T212, T222, T225, M212, T400, T414, T425, T800, T801 and T805).

- The high priority clock increments at a rate of 1 000 000 ticks per second, or one tick = 1 microsecond (IMS T212, T222, T225, M212, T400, T414, T425, T800, T801 and T805).
- `TIMER` channels cannot be placed in memory with a `PLACE` statement.

D.4 Language

- The following directives are supported: `#INCLUDE`, `#USE`, `#COMMENT`, `#IMPORT`, `#OPTION` and `#PRAGMA`. For more information about compiler directives see part 1, section 25.10.
- The following statements are supported: `PLACE` *name* `IN VECSPACE`, `PLACE` *name* `IN WORKSPACE` and `PLACE` *name* `AT WORKSPACE` *n*
- The address used in a `PLACE` allocation is converted to a transputer address by considering the address to be a word offset from `MOSTNEG INT`.

  For example, in order to access a `BYTE` memory mapped peripheral located at machine address `#1234`, on a 32-bit processor:

```
PORT OF BYTE peripheral :
PLACE peripheral AT (#1234 >< (MOSTNEG INT)) >> 2 :
peripheral ! 0 (BYTE)
```

- The numbers used as `PLACE` addresses are word offsets from the bottom of address space.

  `PLACE` *scalar channel* `AT` *n*, places the channel word at that address.

  `PLACE` *array of channels* `AT` *n*, places the the array of pointers at that address.

  Note: `PLACE` *array of channels* `AT` *n* maps an array of pointers to channels. This is a change from D705/D605/D505 releases of the occam compiler where this allocation was used to place an array of channels.

- A channel declared as `CHAN OF ANY` can be passed as an actual parameter in place of a formal channel parameter of *any* protocol. A channel of a specific protocol *cannot* be passed in place of a formal channel parameter of `CHAN OF ANY`. Communications on a channel declared as `CHAN OF ANY` must be identical at both ends of the channel.

- The keywords **GUY** and **ASM** introduce a section of transputer assembly code.

- The keyword **INLINE** may be used immediately before the **PROC** or **FUNCTION** keyword of any procedure or function declaration. This will cause the body of the procedure or function to be expanded inline in any call, and the declaration will not be compiled as a normal routine. **Note:** the declaration is marked with the keyword, but the call is affected. This means that you cannot inline expand procedures and functions which have been declared by a **#USE** directive; to achieve that effect you may put the source of the routine in an include file, marked with the **INLINE** keyword, and include it with an **#INCLUDE** directive.

  Examples:

```
INT INLINE FUNCTION sum3 (VAL INT x, y, z)
IS  x + (y + z) :


INLINE PROC seterror ()
  error := TRUE
:
```

- The compiler accepts the string escape characters as described in section I of the occam *2 Reference Manual.* The compiler also accepts '`*l`' or '`*L`' as the first character of a string literal . This is expanded to be the length of the string excluding the character itself. For example `string1` and `string2` are identical:

```
VAL string1 is ''*lFred'' :
VAL string2 is ''#04Fred'' :
```

  The use of '`*l`' is illegal if the string (excluding the '`*l`') is longer than 255 bytes, and will be reported as an error.

- Multidemensional arrays defined by a **RETYPES** definition may have one element whose value is not explicitly stated. This may be any one of the elements. For example:

```
[6]INT a, f :
[2][ ]INT b RETYPES a :
[ ][3]INT c RETYPES f :

[24]INT d :
[2][ ][6] e RETYPES d :
```

  Note this is a change from the previous implementation of the compiler in the IMS D705/D605/D505 products, and removes the restriction that the inner-most element of the array could not be left unspecified.

- The compiler places restrictions on the syntax which is permitted at the outermost level of a compilation unit; i.e. not enclosed by any function or procedure.

  - No variable declarations are permitted.

  - The file must contain at least one **PROC** or **FUNCTION**; a null source file is illegal.

  - No abbreviations containing function calls or **VALOF**s are allowed, even if they are actually constant. For example:

```
VAL x IS (VALOF
            SKIP
            RESULT 99
         ) :                -- This is illegal.
VAL m IS max (27, 52) :     -- This is also
                            -- illegal.
```

- There is no limit on the number of significant characters in identifiers, and the case of characters is significant.

- **CASE** statements are implemented as a combination of explicit test, binary searches, and jump tables, depending on the relative density of the selection values. The choice has been made to optimise the general case where each selection is equally probable. The compiler does not make any use of the order of the selections as they are written in the source code.

- No assumption can be made about the relative priority of the guards of an **ALT** statement; if priority is required, you must use a **PRI ALT**.

- The compiler expands tabs in source files to be every eight character position. Tabs are permitted anywhere in a line except within strings or character constants.

- If a name is used more than once in a single formal parameter list, the *last* definition is used.

D.5 Summary of implementation restrictions

- **FUNCTION**s may not return arrays, not even with fixed sizes.
- Multiple assignment of arrays of unknown size is not permitted.
- Replicated **PAR** count must be constant.
- There must be exactly two branches in a **PRI PAR**.
- Replicated **PRI PAR**s are not allowed.
- Nested **PRI PAR**s are not permitted.

  The D705/D605/D505 releases of the occam compiler did not check this condition correctly, allowing some erroneous programs. Such code should be modified as follows:

  ```
  PRI PAR
    ... high priority process
    ... code which includes a PRI PAR
  ```

  This can be re-written as the following:

  ```
  PAR
    PRI PAR
      ... high priority process
      SKIP
    ... code which includes a PRI PAR
  ```

- Table sizes must be known at compile time, for example:

  ```
  PROC p ([]INT a, []INT b)
    VAL [] []INT x IS [a] :   -- this is
                              -- illegal
    VAL    []INT y IS b :     -- this is
                              -- legal
  :
  ```

- Constant arrays which are indexed by replicator variables are not considered to be constants for the purposes of compiler constant folding, even if the start and limit of the replicator are also constant. This restriction does not apply during usage checking.
- Maximum number of nested include files is 20.
- Maximum filename length is 128 characters.

- Maximum 256 tags allowed in **PROTOCOL**s.
- Maximum number of lexical levels is 254. (Nested **PROC**s and replicated **PAR**s).
- Maximum number of variables in a procedure or function is 2048.

  If this limit is reached, it should be remembered that any occam code can be 'wrapped up' into a separate procedure, and can still access 'non-local' variables correctly. This will reduce the complexity of an enclosing procedure and should allow the program to be compiled.

  For example, suppose that the following program reaches this limit:

```
PROC p ()
  ...  variable declarations in here
  SEQ
    ...  lots more variable declarations
    SEQ
      ...  first block of code

    ...  lots more variable declarations
    SEQ
      ...  second block of code
:
```

This could be modified to read as follows:

```
PROC p ()
  ...  variable declarations in here
  SEQ
    PROC local0 ()
      ...  lots more variable declarations
      SEQ
        ...  first block of code
    :
    local0()

    PROC local1 ()
      ...  lots more variable declarations
      SEQ
        ...  second block of code
    :
    local1()
:
```

D.6 Syntax of language extensions

This section describes the syntax of the following extensions to occam:

- **ASM**
- **PLACE** *name* **AT WORKSPACE** *n*
- **PLACE** *name* **IN WORKSPACE**
- **PLACE** *name* **IN VECSPACE**
- **INLINE**
- The non-printable character '`*l`' or '`*L`'.

D.6.1 ASM statement

The syntax of the **ASM** construct takes the following format:

```
process          =    asm.construct

asm.construct    =    ASM
                        { asm.directive }

asm.directive    =    primary.op constant.expression
                 |    load.or.store.op name
                 |    branch.op :label
                 |    secondary.op
                 |    pseudo.op
                 |    labeldef

labeldef         =    : label

primary.op       =    direct instruction
                 |    prefixing instruction
                 |    OPR

load.or.store.op =    LDL | LDNL | LDLP | LDNLP
                 |    STL | STNL

branch.op        =    J | CJ | CALL

secondary-op     =    any transputer operation

pseudo-op        =    LD asm.exp
                 |    LDAB asm.exp , asm.exp
                 |    LDABC asm.exp , asm.exp , asm.exp
                 |    ST element
                 |    STAB element , element
                 |    STABC element , element , element
                 |    BYTE {,  constant.expression }
                 |    WORD {,  constant.expression }
                 |    LDLABELDIFF :label - :label

asm.exp          =    ADDRESSOF element
                 |    expression
```

Appendix B lists the transputer instructions and operations supported by the restricted code insertion facility. All the instructions listed can be inserted into occam programs using the **ASM** construct. **Note**: instructions should be specified in upper-case.

D.6.2 PLACE statements

The syntax of the **PLACE** statements extends the definition of an allocation as defined in the '*occam* 2 Reference Manual':

```
allocation  =    PLACE name AT expression
               | PLACE name AT WORKSPACE expression
               | PLACE name IN WORKSPACE
               | PLACE name IN VECSPACE
```

D.6.3 INLINE statement

The **INLINE** statement extends the syntax of a definition as defined in the '*occam* 2 Reference Manual':

```
definition  =    PROTOCOL name IS simple.protocol:
               | PROTOCOL name IS sequential.protocol:
               | PROTOCOL name
                   CASE
                     { tagged.protocol }
                 :
               | [INLINE] PROC name ( {0 , formal })
                   procedure.body
                 :
               | {1 , primitive type } [INLINE] FUNCTION name
                   ( {0 , formal } ) function.body
                 :
               | {1 , primitive type } [INLINE] FUNCTION name
                   ( {0 , formal } ) IS expression.list :
               | specifier name RETYPES element :
               | VAL specifier name RETYPES expression :
```

D.6.4 *l or *L character

The syntax of the non-printable character '*', as defined in section I of the '*occam* 2 Reference Manual' has been extended. The first character of a literal string may now take the value '*l' or '*L', which is used to represent the length of the string, excluding the character itself.

The characters *, ' and " may be used in the following form:

*c	*C	carriage return	=	*#0D
*l	*L	string length	≤	*#FF
*n	*N	newline	=	*#0A
*t	*T	tab	=	*#08
*s	*S	space	=	*#20
*'		quotation mark		
*"		double quotation mark		
**		asterisk		

Any byte value can be represented by *# followed by two hexadeximal digits.

E Configuration language definition

This appendix defines the syntax of the occam configuration language.

A configuration program file contains a sequence of specifications. These specifications should include one hardware description and one software description. There will in general be at least one node declaration, and optionally edge declarations and arc declarations. An optional mapping may appear either before of after the software configuration, but after the declaration of any nodes, edges or arcs which it references. These rules are applications of the normal occam scope rules.

This syntax should be considered as extending the syntax of occam.

The **#INCLUDE** mechanism may be used to incorporate hardware descriptions, software descriptions, or any other source text from other files.

E.1 New types and specifications

specification	=	*hardware.description*
	|	*software.description*
	|	*mapping*
	|	*node.declaration*
	|	*edge.declaration*
	|	*arc.declaration*
	|	*channel.allocation*

node.declaration	=	$\{_0$ [*expression*] $\}$ **NODE** *node.name* :
edge.declaration	=	$\{_0$ [*expression*] $\}$ **EDGE** *declared.edge.name* :
arc.declaration	=	$\{_0$ [*expression*] $\}$ **ARC** *arc.name* :

The syntax adds the new primitive types **NODE**, **EDGE** and **ARC**, and structures **CONFIG**, **NETWORK** and **MAPPING** to the occam language.

NODE declarations introduce processors (*nodes* of a graph). These processors are *physical* if their type and memory size attributes are defined as part of the hardware description, and *logical* otherwise.

EDGE declarations introduce external connections of the hardware description.

ARC declarations introduce named connections (*arcs* of a graph). Each arc connects two edges, which may be attributes of nodes, or declared edges. Connections need only be named if it is required to force a particular mapping of channels, or if names are required to aid debugging.

E.2 Software description

A **CONFIG** declaration introduces the software description as an occam process. Additional specifications and processes are added to occam: The processor name in a **PROCESSOR** statement may be a physical processor name or the name of a logical processor which is mapped onto a physical processor. A channel allocation may allocate up to two channels onto a named arc of the network.

```
software.description  =  { specification }
                          CONFIG [config.name]
                            process
                          :

specification         =   channel.allocation
                        | node.declaration
channel.allocation    =   PLACE {1 , channel.name {0 [ subscript ] }} ON
                            arc
process               =   PROCESSOR processor.name {0 [ subscript ] }
                            process
arc                   =   arc.name {0 [ subscript ] }
```

E.3 Hardware description

The **NETWORK** keyword introduces a hardware description, an optionally named structure which describes the types, connectivity and attributes of previously declared processor nodes. Connections are defined in **CONNECT** statements. Attributes are given values in **SET** statements. The attributes of a processor node include an array of edges which are its links, a string which defines its processor type, and an integer which is the memory size in bytes.

Connections and attribute settings may be combined in any order using the **DO** constructor, including replication and conditionals. For each node which has a type defined to be a processor the attributes with predefined names **type** and **memsize** must be set once only. The connections connect declared edges and edges of nodes, which have the predefined attribute name **link**. The boolean attribute **root** may be set to TRUE for only one node in a network without a connection to the predefined edge HOST. The attribute **romsize** defines the size in bytes of read only memory on a node. Attributes are referenced by subscripting node names with attribute names in brackets.

```
hardware.description  =  { specification }
                         NETWORK [ network.name ]
                           network.item
                         :

specification  =    node.declaration
                  | edge.declaration
                  | arc.declaration

network.item  =     connection.item
                    setting.item
                  | DO
                      { network.item }
                  | DO replicator
                      network.item
                  | conditional.network.item
                  | SKIP
                  | STOP
                  | abbreviation
                    network.item

conditional.network.item  =   IF
                                { network.choice }
network.choice            =   guarded.network.choice
                            | conditional.network.item
guarded.network.choice    =   boolean
                                network.item

connection.item  =   CONNECT edge TO edge [ with.clause ]
with.clause      =   WITH arc.name
edge             =   declared.edge.name {0 [ subscript ] }
                   | node.name {0 [ subscript ] } [ attribute.name ]
                       {0 [ subscript ] }

setting.item          =  SET node.name {0 [ subscript ] }
                           ( attribute.assignment )
attribute.assignment  =  {1 , attribute } := {1 , attribute.value }
attribute             =  attribute.name {0 , [ subscript ] }
attribute.value       =  expression
```

E.4 Mapping structure

The keyword **MAPPING** introduces an optionally named mapping structure which may be either before or after the software description.

A mapping may be used to associate logical processors with physical processors and channels with arcs of the hardware network. Mapping of channels is optional except in the case where one end of the arc is an external edge. The configurer will normally choose a mapping from its knowledge of the connectivity of the hardware and the implied connectivity derived from the use of channels as in the software description.

The mapping may include code mappings and channel mappings. A logical processor may appear on the left hand side of only one mapping item. A physical processor may appear on the right hand side of one or more mapping items. A code mapping may include a priority clause which will determine the priority at which the process will run. The arc in a channel mapping must connect the nodes onto which the processes using the channels are mapped. The effect of channel mappings is identical to the corresponding channel allocations which may appear in the software description.

```
mapping        =   { specification }
                   MAPPING [ mapping.name ]
                     map.item
                   :

specification  =   node.declaration

map.item                  =   code.mapping
                            | channel.mapping
                            | DO
                                { map.item }
                            | DO replicator
                                map.item
                            | conditional.map.item
                            | SKIP
                            | STOP
                            | abbreviation
                              map.item
                            | setting.item
conditional.map.item      =   IF
                                { mapping.choice }
mapping.choice            =   guarded.mapping.choice
                            | conditional.map.item
guarded.mapping.choice    =   boolean
                                map.item
```

code.mapping	=	`MAP` *processor.list* `ONTO` *node [priority.clause]*
priority.clause	=	`PRI` *expression*
processor.list	=	{ $_1$, *processid* }
processid	=	*processor.name* { $_0$ [*subscript*] }
processor.name	=	*node.name*
node	=	*node.name* { $_0$ [*subscript*] }

channel.mapping	=	`MAP` *channel.list* `ONTO` *arc*
channel.list	=	{ $_1$, *channelid* }
channelid	=	*channel.name* { $_0$ [*subscript*] }
arc	=	*arc.name* { $_0$ [*subscript*] }

setting.item	=	`SET` *node.name* { $_0$ [*subscript*] } (*attribute.assignment*)
attribute.assignment	=	{ $_1$, *attribute* } := { $_1$, *attribute.value* }
attribute	=	*attribute.name* { $_0$, [*subscript*] }
attribute.value	=	*expression*

E.5 Constraints

The following constraints apply to all configurations:

- All physical processors whose `type`s are set must be connected to each other.
- Any physical processor whose `type` is set must have its `memsize` set.
- Logical processors may only be mapped onto physical processors whose `type` has been set.
- Channels connecting processors of different word size must not use protocols based on the type `INT`.
- A priority expression must evaluate to 0 (high) or 1 (low).

E.6 Changes from the IMS D705/D605/D505 products

The following changes are necessary to convert a configuration from the language used by previous INMOS configurers:

Channel allocations to physical hardware link addresses should be removed.
PROCESSOR statements should be modified to reference (logical or physical) processor names, instead of processor numbers.
Each physical processor in the configuration should be declared in a **NODE** declaration.
Each external connection from the network should be declared in an **EDGE** declaration.
A hardware description setting attributes of all hardware processors and defining connections between them must be written.
If logical processors have been introduced then a mapping of these onto the physical processors must be written.
Arcs connecting to external edges should be declared. Channels using these arcs should be mapped.
Check that **#USE** lines refer to files containing linked code.

F Bootstrap loaders

F.1 Introduction

Special loading procedures can be created for the program and used in place of, or in addition to, the standard INMOS bootstrap. The file containing the new bootstrap is specified by invoking the collector with the '`B`' option.

User defined bootstraps must perform all the necessary operations to initialise the transputer, load the network, and set up the software environment for the application program.

Bootstraps are output to the program bootable file as the first section of code in the bootable file. The bootstrap, consisting of the primary and secondary bootstrap sequences, is followed by the standard INMOS network loader program, which is output in small packets, each packet consisting of a maximum of 60 bytes. The last packet of the network loader is followed by a length byte of zero.

In most cases a custom bootstrap will interface directly with the standard INMOS Network Loader, which places various pieces of code and data within the transputer memory in a controlled way. However it is possible to skip the standard loader by sinking its code packets and following the commands used by the network loader that are output after the network loader.

The general format of a custom bootstrap is a concatenated sequence of bootstrap code segments each preceded by a length byte. The sequence can be any length. The bootstrap program must be contained in a single file.

F.1.1 The example bootstrap

The example bootstrap loader provided on the toolset **`examples`** directory is a combination of several files used in the standard INMOS bootstrap scheme. The files have been combined into a single file to illustrate how to create a user-defined bootstrap; the functionality is the same as that used in the the standard INMOS scheme based on multiple files.

The program is written in transputer code and consists of two parts:

> *Primary bootstrap* – performs processor setup operations such as initialising the transputer links
>
> *Secondary bootstrap* – sets up the software environment and interfaces to the Network Loader.

Transfer of control

The calling sequence in the standard INMOS scheme is as follows:

The primary loader calls the secondary loader, which then calls the Network Loader. When the Network Loader has completed its work control returns to the secondary loader, which calls the application program via data set up by the Network Loader.

Custom bootstraps should follow the same sequence.

F.1.2 Writing bootstrap loaders

Bootstrap loader programs should be written to perform the same operations as the standard scheme, that is, hardware initialisation, setting up the software environment, and calling the Network Loader. If you skip the Network Loader by sinking its code bytes then you must ensure its function is reproduced in your own code. If you do use the Network Loader you must ensure the interface to it is correct by setting up the invocation stack. The method by which this is achieved can be deduced from the example program listing.

If you wish to make only a few small changes to the standard loader, for example, insert code to initialise some D-to-A convertors, then the example code can be used and the required code can be inserted between the Primary and Secondary Loader code as an additional piece of bootstrap code in the sequence of bootstraps. The rest of the code can be used as it stands.

If you decide to devise your own loading scheme and rewrite the Primary and Secondary Loaders then you should be familiar with the design of the Transputer and its instruction set. For engineering data about the transputer consult the '*Transputer Databook*' and for information about how to use the instruction set see the '*Transputer Instruction Set: a compiler writer's guide*'.

F.2 Example user bootstrap

```
--
-- (c) Inmos 1989
--
-- Assembly file for the Generic Primary bootstrap TA HALT mode
--
--
--
-- VAL   BASE            IS  1 :        -- loop index
-- VAL   COUNT           IS  2 :        -- loop count
--
-- VAL   LOAD_START      IS  0 :        -- start of loader
-- VAL   LOAD_LENGTH     IS  1 :        -- loader block length
-- VAL   NEXT_ADDRESS    IS  2 :        -- start of next block to load
-- VAL   BOOTLINK        IS  3 :        -- link booted from
-- VAL   NEXT_WPTR       IS  4 :        -- work space of loaded code
-- VAL   RETURN_ADDRESS  IS  5 :        -- return address from loader
-- VAL   TEMP_WORKSPACE  IS  RETURN_ADDRESS : -- workspace used by both
--                                      -- preamble and loader
-- VAL   NOTPROCESS      IS  6 :        -- copy of MinInt
-- VAL   LINKS           IS  NOTPROCESS : -- 1st param to loader (MinInt)
-- VAL   BOOTLINK_IN_PARAM  IS  7 :        -- 2nd parameter to loader
-- VAL   BOOTLINK_OUT_PARAM IS  8 :        -- 3nd parameter to loader
-- VAL   MEMORY          IS  9 :        -- 4th parameter to loader
-- VAL   EXTERNAL_ADDRESS IS 10 :        -- 5th parameter to loader
-- VAL   ENTRY_POINT     IS 11 :        -- 6th parameter to loader
-- VAL   DATA_POINT  IS 12 :        -- 7th parameter to loader
-- VAL   ENTRY_ADDRESS   IS 13 :        -- referenced from entry point
-- VAL   DATA_ADDRESS    IS 14 :        -- referneced from Data point
-- VAL   MEMSTART        IS 15 :        -- start of boot part 2
--
--
-- The initial workspace requirement is found by reading the workspace
-- requirement from the loader \occam\ and subtracting the size of the workspace
-- used by both the loader and the bootstrap (\verb|temp.workspace|). This value
-- is incremented by 4 to accommodate the workspace adjustment by the call
-- instruction used to preserve the processor registers.
--
-- initial.adjustment := (loader.workspace + 4) - temp.workspace
-- occam work space, + 4 for call to save registers, - adjustment made
-- when entering occam. Must be at least 4
-- IF
--   initial.adjustment < 4
--     initial.adjustment := 4
--   TRUE
--     SKIP
--
-- set up work space, save registers,
-- save MemStart and NotProcess

    align

    byte   (Endprimary-Primary)  -- Length of the primary bootstrap

Primary:

global Primary

    ajw    INITIAL_ADJUSTMENT -- see above (is 20)
    call   0                  -- save registers

    ldc    _Start - Addr0    -- distance to start byte
    ldpi                     -- address of start
Addr0:
    stl    MEMSTART          -- save for later use

    mint
    stl    NOTPROCESS        -- save for later use
```

```
-- initialise process queues and clear error
    ldl    NOTPROCESS
    stlf                          -- reset low priority queue

    ldl    NOTPROCESS
    sthf                          -- reset high priority queue

-- use clrhalterr here to create bootstrap for REDUCED application

    sethalterr                    -- set halt on error
    testerr                       -- read and clear error bit

-- initialise T8 error and rounding
    ldl    MEMSTART               -- Check if processor has floating point unit by
    ldl    NOTPROCESS             --   checking if (memstart >< mint) >= #70
    xor
    ldc    #70                    -- Memstart for T5, T8
    rev                           -- B = #70, A = (Memstart >< MINT)
    gt
    eqc    0
    cj     Nofpu

    fptesterr                -- floating check and clear error instruction

Nofpu:

-- initialise link and event words
    ldc    0
    stl    BASE                   -- index to words to initialise
    ldc    11                     -- no. words to initialise
    stl    COUNT                  -- count of words left
Startloop:
    ldl    NOTPROCESS
    ldl    BASE                   -- index
    ldl    NOTPROCESS
    wsub                          -- point to next address
    stnl   0                      -- put NotProcess into addressed word
    ldlp   BASE                   -- address of loop control info
    ldc    Endloop - Startloop -- return jump
    lend                          -- go back if more
Endloop:

-- set up some loader parameters. See the parameter
-- structure of the loader
    ldl    MEMSTART       -- clear data and entry addresses
    stl    DATA_ADDRESS
    ldl    MEMSTART
    stl    ENTRY_ADDRESS

    ldlp   DATA_ADDRESS      -- address of entry word
    stl    DATA_POINT        -- store in param 7

    ldlp   ENTRY_ADDRESS     -- address of entry word
    stl    ENTRY_POINT       -- store in param 6

    ldl    NOT_PROCESS
    stl    EXTERNAL_ADDRESS      -- buffer offset in param 5

    ldl    MEMSTART          -- start of memory
    stl    MEMORY            -- store in param 4

    ldl    BOOTLINK          -- copy of bootlink
    stl    BOOTLINK_IN_PARAM   -- store in param 2

--  Now find the corresponding output link and place in the parameter

    ldl   BOOTLINK
    ldnlp  -4                    -- Calculate the output link address
    stl    BOOTLINK_OUT_PARAM  -- store in param 3
```

```
-- load bootloader over bootstrap
-- code must be 2 bytes shorter than bootstrap
    ldlp   LOAD_LENGTH       -- packet size word
    ldl    BOOTLINK          -- address of link
    ldc    1                 -- bytes to load
    in                       -- input length byte

    ldl    MEMSTART          -- area to load bootloader
    ldl    BOOTLINK          -- address of link
    ldl    LOAD_LENGTH       -- message length
    in                       -- input bootloader

-- enter code just loaded

    pfix   0                 -- For the next bootstrap to be 2 bytes bigger
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0
    pfix   0

    ldl    MEMSTART          -- start of loaded code
    gcall                    -- enter bootloader

    align

Endprimary:

--
-- (c) Inmos 1989
-- Assembly file for the generic secondary loader TA IGNORE mode
--
--
--
-- VAL   BASE               IS  1 :         -- loop index
-- VAL   COUNT              IS  2 :         -- loop count
--
-- VAL   LOAD_START         IS  0 :         -- start of loader
-- VAL   LOAD_LENGTH        IS  1 :         -- loader block length
-- VAL   NEXT_ADDRESS       IS  2 :         -- start of next block to load
-- VAL   BOOTLINK           IS  3 :         -- link booted from
-- VAL   NEXT_WPTR          IS  4 :         -- work space of loaded code
-- VAL   RETURN_ADDRESS     IS  5 :         -- return address from loader
-- VAL   TEMP_WORKSPACE     IS  RETURN_ADDRESS : -- workspace used by both
--                                          -- preamble and loader
-- VAL   NOTPROCESS         IS  6 :         -- copy of MinInt
-- VAL   LINKS              IS  NOTPROCESS : -- 1st param to loader (MinInt)
-- VAL   BOOTLINK_IN_PARAM  IS  7 :           -- 2nd parameter to loader
-- VAL   BOOTLINK_OUT_PARAM IS  8 :           -- 3nd parameter to loader
-- VAL   MEMORY             IS  9 :         -- 4th parameter to loader
-- VAL   BUFFER             IS 10 :         -- 5
-- VAL   NEXT_POINT         IS 11 :         -- 6th parameter to loader
-- VAL   ENTRY_POINT        IS 12 :         -- 7th parameter to loader
-- VAL   DATA_POINT         IS 13 :         -- 8th parameter to loader
-- VAL   ENTRY_ADDRESS      IS 14 :         -- referenced from entry point
-- VAL   DATA_ADDRESS       IS 15 :         -- referenced from Data point
-- VAL   NEXT_ADDRESS       IS 16 :         -- referenced from Nexat point
-- VAL   MEMSTART           IS 17 :         -- start of boot part 2
--
--
-- VAL   PACKET_LENGTH      IS 120 :
-- VAL   OCCAM_WORKSPACE    IS 18 :
```

```
    byte   (Endsecondary-Secondary)  -- Length of the secondary boostrap

Secondary:

global Secondary

-- initialise bootloader workspace

    ldc     PACKET_LENGTH     -- buffer size
    ldlp    MEMSTART+1        -- buffer start address
    bsub                      -- end of buffer address
    stl     NEXT_ADDRESS      -- start of area to load loader

    ldl     NEXT_ADDRESS

    ldlp    MEMSTART+1        -- buffer start address
    stl     MEMORY            -- Earliest place to load

    ldlp    TEMP_WORKSPACE    -- pointer to loader's work space zero
    stl     NEXT_WPTR         -- work space pointer of loaded code

    ldc     0
    stl     BUFFER            -- Buffer offset from Buffer start

    ldc     0
    stl     LOAD_LENGTH       -- clear bytes to load

Loadcode:
    ldl     NEXT_ADDRESS      -- address to load loader
    stl     LOAD_START        -- current load point

-- load code until terminator
Startload:
    ldlp    LOAD_LENGTH       -- packet length
    ldl     BOOTLINK          -- address of link
    ldc     1                 -- bytes to load
    in                        -- input length byte

    ldl     LOAD_LENGTH       -- message length
    cj      Endload           -- quit if 0 bytes

    ldl     NEXT_ADDRESS      -- start of area to load loader
    ldl     BOOTLINK          -- address of link
    ldl     LOAD_LENGTH       -- message length
    in                        -- input code block
    ldl     LOAD_LENGTH       -- message length
    ldl     NEXT_ADDRESS      -- area to load
    bsub                      -- new area to load
    stl     NEXT_ADDRESS      -- save area to load

    j       Startload         -- go back for next block
Endload:

-- initialise return address and enter loaded code
    ldc     Return - Addrl    -- offset to return address
    ldpi                      -- return address
Addrl:
    stl     RETURN_ADDRESS    -- save in W0


    ldl     BOOTLINK          -- Get bootlink and save for later
    stl     OCCAM_WORKSPACE   -- Save in area that will not be used
                              -- by network loader

    ldl     NEXT_WPTR         -- wspace of loaded code
    gajw                      -- set up his work space
    ldnl    LOAD_START        -- address of first load packet
    gcall                     -- enter loaded code

Return:
```

```
-- Now set up invocation stack for the Init_system

    ajw    (TEMP_WORKSPACE + 4)-- reset work space after return

    ldl    OCCAM_WORKSPACE     -- get back boot link
    stl    BOOTLINK

    ldl    DATA_ADDRESS     -- get address of processor structure
    ldl    MEMORY
    bsub
    stl    DATA_POINT


    ldl    ENTRY_ADDRESS -- convert to real entry address
    ldl    MEMORY
    bsub
    stl    LOAD_START

    ldl    NOTPROCESS
    stl    NEXT_POINT

    ldl    MEMORY          -- make DATA base offset and CODE base offset the same
    stl    BUFFER          --

    ldl    ENTRY_ADDRESS   --
    stl    TEMP_WORKSPACE  -- Set up entry point

    ldl    NEXT_ADDRESS     -- convert returned address of next sequence to
    ldl    MEMORY           -- a real address
    bsub
    stl    NEXT_ADDRESS

    ldc    0
    stl    LOAD_LENGTH       -- clear bytes to load

    ldlp   NOT_PROCESS      -- Top of temp workspace used by bootloader
    stl    NEXT_WPTR

-- start clock

    ldc    0
    sttimer

    j      Startload        -- Go back for more and over write the network loader


    align

Endsecondary:
```

F.3 The INMOS Network Loader

The following code, written in occam, represents the standard network loader program used by INMOS.

```
--------------------------------------------------------------------------------
--                                                                            --
-- This generic loader is written and should be compiled with out any processor type  --
-- dependencies. That is the same object code is used even if the processor is one of --
-- the sixteen bit variety                                                    --
--
--------------------------------------------------------------------------------
PROC Loader ([4]CHAN OF ANY   links,
             CHAN OF ANY      bootlink.in, bootlink.out,
             [4]BYTE          memory,
             VAL INT          Buffer.address,
             INT              Next.address,
             INT              Entry.point,
             INT              Data.point)


  --{{{  constants
  VAL   data.field          IS   #3F :
  VAL   data.field.bits     IS   6 :
  VAL   tag.field           IS   #C0 :
  VAL   tag.field.bits      IS   2 :
  VAL   message             IS   0 :
  VAL   number              IS   1 :
  VAL   operate             IS   2 :
  VAL   prefix              IS   3 :
  VAL   tag.prefix          IS   prefix << data.field.bits :
  VAL   message.length      IS   60 :

  VAL   load                IS   0 :
  VAL   pass                IS   1 :
  VAL   open                IS   2 :
  VAL   operate.open        IS   BYTE ((operate << data.field.bits)
                                        \/ open)  :
  VAL   close               IS   3 :
  VAL   operate.close       IS   BYTE ((operate << data.field.bits)
                                        \/ close)  :
  VAL   address             IS   4 :
  VAL   execute             IS   5 :
  VAL   Data.position       IS   6 :
  VAL   operate.execute     IS   BYTE ((operate << data.field.bits)
                                        \/ execute)  :

  VAL   operate.data.postion IS BYTE ((operate << data.field.bits)
                                        \/ Data.position)  :
  VAL   code.load           IS   7 :
  VAL   operate.code.load   IS   BYTE ((operate << data.field.bits)
                                        \/ code.load)  :

  VAL   code.address        IS   8 :
  VAL   operate.code.address IS BYTE ((operate << data.field.bits)
                                        \/ code.address)  :

  VAL  data.load            IS   9 :
  VAL  operate.data.load    IS   BYTE ((operate << data.field.bits)
                                        \/ data.load)  :

  VAL  data.address         IS   10 :
  VAL  operate.data.address  IS BYTE ((operate << data.field.bits)
                                        \/ data.address)  :

  VAL  Entry.position       IS   11 :
  VAL  operate.entry.position IS BYTE ((operate << data.field.bits)
                                        \/ Entry.position)  :
```

```
VAL  Bootstrap.load      IS   12 :
VAL  Operate.bootstrap.load IS BYTE ((operate << data.field.bits)
                                    \/ Bootstrap.load) :

VAL  Bootstrap.end       IS   13 :
VAL  Operate.bootstrap.end IS BYTE ((operate << data.field.bits)
                                    \/ Bootstrap.end) :

--{{{  VARIABLES
BYTE   command :
INT    Bootstrap.depth, links.to.load, last.address, output.link :
BOOL   loading :
SEQ
  bootlink.in ? command
  WHILE command <> operate.execute
    INT    tag, operand :
    --{{{  process command
    SEQ
      tag := (INT command) >> data.field.bits
      operand := (INT command) /\ data.field
      IF
        --{{{  tag = message
        tag = message
          INT   load.address :
          SEQ
            IF
              --{{{  loading
              loading
                SEQ
                  load.address := last.address
                  last.address := load.address PLUS operand
              --{{{  passing on
              TRUE
                load.address := Buffer.address
            --{{{  read in message
            IF
              operand <> 0
                bootlink.in ? [memory FROM load.address FOR operand]
              TRUE
                SKIP
            --{{{  send message to outputs
            SEQ i = 0 FOR 4
              IF
                (links.to.load /\ (1 << i)) <> 0
                  SEQ
                    links[i] ! command
                    IF
                      operand <> 0
                        links[i] ! [memory FROM load.address FOR operand]
                      TRUE
                        SKIP
                TRUE
                  SKIP
        --{{{  tag = operate
        tag = operate
          IF
            --{{{  operand = load
            operand = load
              SEQ
                loading := TRUE
                links.to.load := 0
            --{{{  operand = data.load
            operand = data.load
              SEQ
                loading := TRUE
                links.to.load := 0
            --{{{  operand = Code.load
            operand = code.load
              SEQ
```

```
    loading := TRUE
    links.to.load := 0
--{{{  operand = pass
operand = pass
  SEQ
    loading := FALSE
    links.to.load := 0
--{{{  operand = open
operand = open
  INT  depth :
  SEQ
    depth := 1
    WHILE depth <> 0
      SEQ
        bootlink.in ? command
        IF
          command = operate.open
            depth := depth + 1
          command = operate.close
            depth := depth - 1
          TRUE
            SKIP
        IF
          depth <> 0
            links[output.link] !  command
          TRUE
            SKIP
--{{{  operand = address
operand = address
  SEQ
    --{{{  read in load offset
    BOOL  more :
    SEQ
      last.address := 0
      more := TRUE
      WHILE more
        SEQ
          last.address := last.address << data.field.bits
          bootlink.in ? command
          last.address := last.address PLUS
                          ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
    --{{{  entry address
    Next.address := last.address
operand = Data.position
  SEQ
    --{{{  read in data position offset
    BOOL  more :
    SEQ
      Data.point := 0
      more := TRUE
      WHILE more
        SEQ
          Data.point := Data.point << data.field.bits
          bootlink.in ? command
          Data.point := Data.point PLUS
                        ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
operand = Entry.position
  SEQ
    --{{{  read in data position offset
    BOOL  more :
    SEQ
      Entry.point := 0
      more := TRUE
      WHILE more
        SEQ
          Entry.point := Entry.point << data.field.bits
```

```
          bootlink.in ? command
          Entry.point := Entry.point PLUS
                          ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
    --{{{  entry address
operand = code.address
  SEQ
    --{{{  read in load offset
    BOOL   more :
    SEQ
      last.address := 0
      more := TRUE
      WHILE more
        SEQ
          last.address := last.address << data.field.bits
          bootlink.in ? command
          last.address := last.address PLUS
                          ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
    Entry.point := last.address
operand = data.address
  SEQ
    --{{{  read in load offset
    BOOL   more :
    SEQ
      last.address := 0
      more := TRUE
      WHILE more
        SEQ
          last.address := last.address << data.field.bits
          bootlink.in ? command
          last.address := last.address PLUS
                          ((INT command) /\ data.field)

          more := (INT command) >= tag.prefix
    --{{{  entry address
    Data.point := last.address
operand = Bootstrap.load
  INT   load.address :
  INT   Bootstrap.length :
  BOOL   more :
  SEQ
    Bootstrap.depth := 0
    Bootstrap.length := 0
    load.address := Buffer.address
    more := TRUE
    bootlink.in ? command
    more := (INT command) >= data.field
    WHILE more
      SEQ
        Bootstrap.depth := Bootstrap.depth PLUS 1
        SEQ i = 0 FOR 4
          IF
            (links.to.load /\ (1 << i)) <> 0
              SEQ
                links[i] ! command
            TRUE
              SKIP
        bootlink.in ? command
        more := (INT command) >= data.field

    operand := (INT command) /\ data.field

    IF
      Bootstrap.depth > 0
        --{{{  read in message
        SEQ
          IF
```

```
                operand <> 0
                  bootlink.in ? [memory FROM load.address FOR operand]
                TRUE
                  SKIP
              --{{{  send message to outputs
              SEQ i = 0 FOR 4
                IF
                  (links.to.load /\ (1 << i)) <> 0
                    SEQ
                      links[i] ! command
                      IF
                        operand <> 0
                          links[i] ! [memory FROM load.address
                                                   FOR operand]
                        TRUE
                          SKIP
                  TRUE
                    SKIP
          TRUE
            SEQ
              more := TRUE
              -- The next processor(s) are to be booted !!! --
              -- so build a bootable packet and output down link --
              WHILE more
                SEQ
                  bootlink.in ? [memory FROM load.address FOR operand]
                  load.address := load.address PLUS operand
                  Bootstrap.length := Bootstrap.length PLUS operand
                  bootlink.in ? command
                  -- Stop building when a proper command
                  -- is received This should be when a
                  -- 'Bootstrap.end' is received
                  more := (INT command) < data.field
                  operand := (INT command) /\ data.field

              SEQ i = 0 FOR 4
                IF
                  (links.to.load /\ (1 << i)) <> 0
                    SEQ
                      links[i] ! (BYTE Bootstrap.length)
                      IF
                        Bootstrap.length <> 0
                          links[i] ! [memory FROM Buffer.address
                                                   FOR Bootstrap.length]
                        TRUE
                          SKIP
                  TRUE
                    SKIP
      operand = Bootstrap.end
        SEQ
          SEQ ii = 0 FOR Bootstrap.depth
            SEQ
              -- Pass on all the other bootstrap ends
              bootlink.in ? command
              SEQ i = 0 FOR 4
                IF
                  (links.to.load /\ (1 << i)) <> 0
                    links[i] ! command
                  TRUE
                    SKIP
          Bootstrap.depth := 0

  --{{{  tag = number
  TRUE
    SEQ
      output.link := operand
      links.to.load := links.to.load \/ (1 << output.link)
bootlink.in ? command
```

G ITERM

G.1 Introduction

This appendix describes the format of ITERM files; it is included for people who need to write their own ITERM because they are using terminals that are not supported by the standard ITERM file supplied with the toolset. You may of course wish to tailor a standard ITERM to suit your own needs.

ITERMs are ASCII text files that describe the control sequences required to drive terminals. Screen oriented applications that use ITERM files are terminal independent.

ITERM files are similar in function to the UNIX *termcap* database and describe input from, as well as output to, the terminal. They allow applications that use function keys to be terminal independent and configurable.

Within the toolset, the ITERM file is only used by the debugger tool `idebug` and the T425 simulator tool `isim`.

G.2 The structure of an ITERM file

An ITERM file consists of three sections. These are the *host*, *screen* and *keyboard* sections. Sections are introduced by a line beginning with the section letters '`H`', '`S`' or '`K`'. Case is unimportant and the rest of the line is ignored. Sections consist of a number of lines beginning with a digit. A section is terminated by a line beginning with the letter '`E`'. The *host* section must appear first; other sections may appear in any order in the file. Sections must be separated by at least one blank line.

The syntax of the lines that make up the body of a section is best described in an example:

```
3:34,56,23,7.   comments
```

Each line starts with the index number followed by a colon and a list of numbers separated by commas. Each line is terminated by a full stop ('`.`') and anything following it is treated as a comment. Spaces are not allowed in the data string and an entry cannot be split across more than one line.

Comment lines, beginning with the character '`#`', may be placed anywhere in an ITERM file. Extra blank lines in the file are ignored.

The index numbers in each section correspond to an agreed meaning for the data. In the following sections the meaning of the data in each of the three sections is described in detail.

G.3 The host definitions

G.3.1 ITERM version

This item identifies an ITERM file by version. It provides some protection against incompatible future upgrades.

e.g. `1:2.`

G.3.2 Screen size

This item allows applications to find out the size of the terminal at startup time. The data items are the number of columns and rows, in that order, available on the current terminal.

e.g. `2:80,25.`

Screen locations should be numbered from 0, 0 by the application. Terminals which use addressing from 1, 1 can be compensated for in the definition of goto X, Y.

G.4 The screen definitions

The lists of values in the screen section represent control codes that perform certain operations; the data values are ASCII codes to send to the display device.

ITERM version 2 defines the indices given in table G.1. These definitions are used in the example ITERM file; for a complete listing of the file see section G.7.

For example, an entry like: '`8:27,91,75.`' indicates that an application should output the ASCII sequence '`ESC [ K`' to the terminal output stream to clear to end of line.

Index	Screen operation	Index	Screen operation
1	cursor up	9	clear to end of screen
2	cursor down	10	insert line
3	cursor left	11	delete line
4	cursor right	12	ring bell
5	goto x y	13	home and clear screen
6	insert character	20	enhance on (not used)
7	delete character at cursor	21	enhance off (not used)
8	clear to end of line		

Table G.1 ITERM screen operations

G.4.1 Goto X Y processing

The entry for 5, 'goto X Y', requires further interpretation by the application. A typical entry for 'goto X Y' might be:

```
5:27,-11,32,-21,32
```

The negative numbers relate to the arguments required for X and Y.

..., *-ab*, *nn*, ...

where: *a* is the argument number (i.e. 1 for X, 2 for Y).

b controls the data output format.
If *b*=1 output is an ASCII byte (e.g. 33 is output as `!`).
If *b*=2 output is an ASCII number (e.g. 33 is output as `3 3`).

nn is added to the argument before output.

As a complete example, consider the following ITERM entry in the screen section:

```
5:27,91,-22,1,59,-12,1,72. ansi cursor control
```

This would instruct an application wishing to move the terminal cursor to X=14, Y=8 (relative to 0,0) to output the following bytes to the screen:

```
Bytes in decimal: 27   91  57  59  49  53  72
Bytes in ASCII:   ESC  [   9   ;   1   5   H
```

G.5 The keyboard definitions

Each index represents a single keyboard operation. The data specified after each index defines the keystroke associated with that operation.

Multiple entries for the same index indicate alternative keystrokes for the operation.

ITERM version 2 defines the indices given in table G.2. These definitions are used in the example ITERM file; for a complete listing of the file see section G.7.

Index	Function	Index	Function
2	delete character	39	goto line
6	cursor up	40	backtrace
7	cursor down	41	inspect
8	cursor left	42	channel
9	cursor right	43	top
12	delete line	44	retrace
14	start of line	45	relocate
15	end of line	46	info
18	line up	47	modify
19	line down	48	resume
20	page up	49	monitor
21	page down	50	word left
26	enter file	51	word right
27	exit file	55	top of file
28	refresh	56	end of file
29	change file	62	toggle hex
31	finish	65	continue from
34	help	66	toggle breakpoint
36	get address	67	search

Table G.2 ITERM key operations

G.6 Setting up the ITERM environment variable

To use an ITERM the application has to find and read the file. An environment variable (or logical name on VMS) called **ITERM** should be set up with the pathname of the file as its value. For example, under MS-DOS the command would be:

```
C:\> set ITERM=C:\ITOOLS\TOOLS\PCBANSI.ITM
```

Under UNIX you would set an environment variable. For example, the command for **csh** users might be:

```
%  setenv ITERM ~/.iterm
```

Under VMS you would define a logical name. For example:

```
$ DEFINE ITERM SYS$LOGIN:VT100.ITM
```

For more details about setting environment variables see the Delivery Manual that accompanies the release.

G.7 An example ITERM

This is the toolset ITERM file for the IBM PC using the ANSI screen driver.

```
#-----------------------------------------------------------
#
#  IBM PC (BANSI) ITERM data file (derived from TDS3 ITERM)
#  Support for idebug and isim
#  IDEBUG version for BANSI.SYS driver:
#  Special care needed on screen codes 6, 7, 9, 10, 11
#
#  V1.1 - 10 July 90  (NH)  Updated idebug and isim support
#
#-----------------------------------------------------------

host section
1:2.                           version
2:80,25.                       screen size
end of host section

#  screen control characters

screen section
#                              DEBUGGER       SIMULATOR
1:27,91,65.                                   cursor up
2:27,91,66.                                   cursor down
3:27,91,68.                    cursor left    cursor left
4:27,91,67.                                   cursor right
5:27,91,-22,1,59,-12,1,72.     goto x y       goto x y
6:27,91,64.                    insert char    insert char
7:27,91,80.                    delete char    delete char
8:27,91,75.                    clear to eol   clear to eol
9:27,91,74.                    clear to eos   clear to eos
10:27,91,76.                   insert line    insert line
11:27,91,77.                   delete line    delete line
12:7.                          bell           bell
13:27,91,50,74.                clear screen   clear screen
end of screen section

keyboard section
#              KEY             DEBUGGER       SIMULATOR
#
2:8.           # BACKSPACE     del char
6:0,72.        # UP            cursor up      cursor up
7:0,80.        # DOWN          cursor down    cursor down
8:0,75.        # LEFT          cursor left    cursor left
9:0,77.        # RIGHT         cursor right   cursor right
12:0,110.      # ALT F7        delete line
```

```
12:21.          # CTRL U        delete line
12:24.          # CTRL X        delete line
14:0,65.        # F7            start of line   start of line
15:0,66.        # F8            end of line     end of line
18:0,67.        # F9            line up
19:0,68.        # F10           line down
20:0,112.       # ALT F9        page up         page up
21:0,113.       # ALT F10       page down       page down
26:0,71.        # NUM 7         enter file
27:0,73.        # NUM 9         exit file
28:27.          # ESC           refresh         refresh
29:0,87.        # SHIFT F4      change file
31:0,117.       # CTRL NUM 1    finish
34:0,59.        # F1            help            help
36:0,63.        # F5            get address
39:0,64.        # F6            goto line
40:0,129.       # ALT 0         backtrace
41:0,120.       # ALT 1         inspect
42:0,121.       # ALT 2         channel
43:0,122.       # ALT 3         top
44:0,123.       # ALT 4         retrace
45:0,124.       # ALT 5         relocate
46:0,125.       # ALT 6         info
47:0,126.       # ALT 7         modify
48:0,127.       # ALT 8         resume
49:0,128.       # ALT 9         monitor
50:0,90.        # SHIFT F7      word left
50:6.           # CTRL F        word left
50:0,115.       # CTRL NUM 4    word left
51:0,91.        # SHIFT F8      word right
51:7.           # CTRL G        word right
51:0,116.       # CTRL NUM 6    word right
55:0,92.        # SHIFT F9      top of file
55:20.          # CTRL T        top of file
56:0,93.        # SHIFT F10     end of file
56:2.           # CTRL B        end of file
62:0,108.       # ALT F5        toggle hex
65:0,105.       # ALT F2        continue from
66:0,99.        # CTRL F6       toggle break
67:0,88.        # SHIFT F5      search

end of keyboard stuff

#  idebug key that isn't really part of iterm but its here
all the same !
#
#               INTERRUPT       CTRL A    --   IDEBUG

#  THAT'S ALL FOLKS
```

H Host file server protocol

This appendix describes the protocol of the host file server **iserver**.

H.1 The host file server iserver

The host file server **iserver** is implemented in C which facilitates porting to other machines. This provides an easy method of porting the toolset (or programs written under the toolset) to new hosts. The server can, at a cost to portability, be extended to accomodate new host features.

The source of the server and of the libraries used to communicate with the server is supplied with the toolset.

H.2 The server protocol

Every communication to and from the server is a packet consisting of a counted array of bytes. The count gives the length of the message and is sent in the first two bytes of the packet as a signed 16 bit number. The structure of a server packet is illustrated in figure H.1.

This protocol has been given the name SP, and is defined in occam as follows:

```
PROTOCOL SP IS INT16::[]BYTE :
```

H.2.1 Packet size

There is a maximum packet size of 1024 bytes and a minimum packet size of 8 bytes in the to-server direction (i.e. a minimum message length of 6 bytes). The server may take advantage of this knowledge.

The packet size must always be an even number of bytes. If the number of

b0	b1	message of length b0 + (256 * b1)

Figure H.1 SP protocol packet

bytes is odd a dummy byte is added to the end of the packet and the packet byte count rounded up by one.

The hostio library contains routines that ensure that the size restrictions are met when sending a packet to the server (see section H.3).

H.2.2 Protocol operation

Every request sent to the server receives a reply of the same protocol, in strict sequence, and no further requests are accepted until the reply has been sent.

Unless otherwise stated all integer types used by the protocol are signed. Numbers are transmitted as sequences of bytes (2 bytes for 16 bit numbers, 4 bytes for 32 bit numbers) with the least significant byte first. Negative integers are represented in 2s complement. Strings and other variable length blocks are introduced by a 16 bit signed count.

All server calls return a result byte as the first item in the return packet. If the operation succeeds the result byte is zero and if the operation fails the result byte is non-zero. The result is one (1) in the special case where the operation fails because the function is not implemented[1]. If the result is non-zero, some or all of the return values may not be present, resulting in a smaller return packet than if the call was successful.

H.3 The server libraries

The hostio library `hostio.lib` contains all the routines provided in the toolset for communicating with the server. It contains a set of basic routines, hidden from the user, from which the more complex user visible routines are built.

A naming convention has been adopted for the server libraries. The basic library routines use the server protocol directly and map directly to server functions. These have the prefix '`sp.`'. Routines which use the basic routines and are visible to the user have the prefix '`so.`'. The '`so.`' routines documented in this manual use underlying '`sp.`' routines, and in some cases the mapping is one to one.

The source of the hostio library is provided with the toolset and serves as an example of how to use the SP protocol.

[1]Result values between 2 and 127 are defined to have particular meanings by occam server libraries. Result values of 128 or above are specific to the implementation of a server.

If you add your own libraries for server functions you are recommended to keep to the naming convention.

There are two '**sp.**' library routines included to help you extend the set of available routines. These are **sp.send.packet** and **sp.receive.packet**. These are described below.

sp.send.packet

```
PROC sp.send.packet (CHAN OF SP ts,
                     VAL []BYTE packet,
                     BOOL error)
```

This procedure sends a packet on the channel **ts**, provided that it meets the requirements for a **SP** protocol packet. If the requirements are not met then the packet is not sent and **error** is set to **TRUE**.

sp.receive.packet

```
PROC sp.receive.packet (CHAN OF SP fs,
                        INT16 length,
                        []BYTE packet,
                        BOOL error)
```

This procedure receives a packet on the channel **fs**. The received packet is in the first **length** bytes of **packet**. The value **error** is set to **TRUE** if the size of the packet received exceeds **sp.max.packet.data.size**; otherwise it is **FALSE**.

H.3.1 Problems with packet size

The maximum packet size which may be handled by **iserver** is 1024, this causes a potential problem, however, for some routines in **hostio.lib**. This is because the **hostio** routines have a maximum packet size of 512 bytes. The **hostio** routines which may be affected are:

- **so.getenv**
- **so.commandline**
- **so.ferror**
- **so.buffer**
- **so.overlapped.buffer**

- `so.multiplexor`
- `so.overlapped.multiplexor`
- `so.pri.multiplexor`
- `so.overlapped.pri.multiplexor`

Should any of these routines receive a packet larger than 512 bytes, they will act as invalid processes.

Care should be taken that the multiplexor and buffer routines listed above are not used by any routines which are likely to exceed the 512 byte limit.

H.4 Porting the server

In order to port the `iserver` to a new machine you must have a C compiler for that machine. A number of Makefiles that can assist with porting to a new machine are supplied in the toolset 'source' subdirectory.

The hostio library expects all the functions described below to be provided by `iserver`.

H.5 Defined protocol

The functions provided by the `iserver` are split into three groups:

1 File commands, for interacting with files

2 Host commands, for interacting with the host

3 Server commands, for interacting with the server.

In the descriptions that follow, the arguments and results of server calls are listed in the order that they appear in the data part of the packet. The size of a packet is the aggregated size of all the items in the packet, rounded up to an even number of bytes. occam types are used to define data items within the packet.

H.5.1 Reserved values

INMOS reserves the following values for its own use:

- Function tags in the range 0 to 127 inclusive.

- Result values in the range 0 to 255 inclusive.
- Stream identifiers 0, 1 and 2.

Some commands may return particular values, which may be reserved. The range of reserved values is given with each command as appropriate.

H.5.2 File commands

Open files are identified with 32 bit descriptors. There are three predefined open files:

0 – standard input
1 – standard output
2 – standard error

If one of these is closed then it may not be reopened.

Fopen – Open a file

```
Synopsis:     StreamId = Fopen( Name, Type, Mode )

To server:     BYTE             Tag = 10
               INT16::[]BYTE    Name
               BYTE             Type = 1 or 2
               BYTE             Mode = 1...6

From server:   BYTE             Result
               INT32            StreamId
```

Fopen opens the file `Name` and, if successful, returns a stream identifier `StreamId`.

`Type` can take one of two possible values:

1 Binary. The file will contain raw binary bytes.

2 Text. The file will be stored as text records. Text files are host-specified.

`Mode` can have 6 possible values:

1 Open an existing file for input.

2 Create a new file, or truncate an existing one, for output.

3 Create a new file, or append to an existing one, for output.

4 Open an existing file for update (both reading and writing), starting at the beginning of the file.

5 Create a new file, or truncate an existing one, for update.

6 Create a new file, of append to an existing one, for update.

When a file is opened for update (one of the last three modes above) then the resulting stream may be used for input or output. There are restrictions, however. An output operation may not follow an input operation without an intervening Fseek, Ftell or Fflush operation.

The number of streams that may be open at one time is host-specified, but will not be less than eight (including the three predefines).

Fclose – Close a file

`Synopsis:`	`Fclose( StreamId )`	
`To server:`	`BYTE`	`Tag = 11`
	`INT32`	`StreamId`
`From server:`	`BYTE`	`Result`

Fclose closes a stream `StreamId` which should be open for input or output. Fclose flushes any unwritten data and discards any unread buffered input before closing the stream.

Fread – Read a block of data

```
Synopsis:      Data = Fread( StreamId, Count )

To server:     BYTE            Tag = 12
               INT32           StreamId
               INT16           Count

From server:   BYTE            Result
               INT16::[]BYTE   Data
```

This function is obsolete. See the definition of FGetBlock for its replacement.

Fread reads `Count` bytes of binary data from the specified stream. Input stops when the specified number of bytes are read, or the end of file is reached, or an error occurs. If `Count` is less than one then no input is done. The stream is left positioned immediately after the data read. If an error occurs the stream position is undefined.

`Result` is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

Fwrite – Write a block of data

```
Synopsis:      Written = Fwrite( StreamId, Data )

To server:     BYTE              Tag = 13
               INT32             StreamId
               INT16::[]BYTE     Data

From server:   BYTE              Result
               INT16             Written
```

This function is obsolete. See the definition of FPutBlock for its replacement.

Fwrite writes a given number of bytes of binary data to the specified stream, which should be open for output. If the length of `Data` is less than zero then no output is done. The position of the stream is advanced by the number of bytes actually written. If an error occurs then the resulting position if undefined.

Fwrite returns the number of bytes actually output in `Written`. `Result` is always zero. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

If the `StreamId` is `1` (standard output) then the write is automatically flushed.

Fgets – Read a line

```
Synopsis:      Data = Fgets( StreamId, Count )

To server:     BYTE              Tag = 14
               INT32             StreamId
               INT16             Count

From server:   BYTE              Result
               INT16::[]BYTE     Data
```

Fgets reads a line from a stream which must be open for input. Characters are read until end of file is reached, a newline character is seen or the number of characters read is not less than `Count`.

If the input is terminated because a newline is seen then the newline sequence is *not* included in the returned array.

If end of file is encountered and nothing has been read from the stream then Fgets fails.

Fputs – Write a line

`Synopsis:`	`Fputs( StreamId, String )`	
`To server:`	`BYTE`	`Tag = 15`
	`INT32`	`StreamId`
	`INT16::[]BYTE`	`String`
`From server:`	`BYTE`	`Result`

Fputs writes a line of text to a stream which must be open for output. The host-specified convention for newline will be appended to the line and output to the file. The maximum line length is host-specified.

Fflush – Flush a stream

`Synopsis:`	`Fflush( StreamId )`	
`To server:`	`BYTE`	`Tag = 16`
	`INT32`	`StreamId`
`From server:`	`BYTE`	`Result`

Fflush flushes the specified stream, which should be open for output. Any internally buffered data is written to the destination device. The stream remains open.

Fseek – Set position in a file

```
Synopsis:     Fseek( StreamId, Offset, Origin )

To server:    BYTE          Tag = 17
              INT32         StreamId
              INT32         Offset
              INT32         Origin

From server:  BYTE          Result
```

Fseek sets the file position for the specified stream. A subsequent read or write will access data at the new position.

For a binary file the new position will be `Offset` characters from `Origin` which may take one of three values:

1 `Set`, the beginning of the file

2 `Current`, the current position in the file

3 End, the end of the file.

For a text stream, `Offset` must be zero or a value returned by Ftell. If the latter is used then `Origin` must be set to 1.

Ftell – Find out position in a file

```
Synopsis:     Position = Ftell( StreamId )

To server:    BYTE          Tag = 18
              INT32         StreamId

From server:  BYTE          Result
              INT32         Position
```

Ftell returns the current file position for `StreamId`.

Feof – Test for end of file

Synopsis:	Feof(StreamId)	
To server:	BYTE	Tag = 19
	INT32	StreamId
From server:	BYTE	Result

Feof succeeds if the end of file indicator for `StreamId` is set.

Ferror – Get file error status

Synopsis:	ErrorNo, Message = Ferror(StreamId)	
To server:	BYTE	Tag = 20
	INT32	StreamId
From server:	BYTE	Result
	INT32	ErrorNo
	INT16::[]BYTE	Message

Ferror succeeds if the error indicator for `StreamId` is set. If it is, Ferror returns a host-defined error number and a (possibly null) message corresponding to the last file error on the specified stream.

Remove – Delete a file

Synopsis:	Remove(Name)	
To server:	BYTE	Tag = 21
	INT16::[]BYTE	Name
From server:	BYTE	Result

Remove deletes the named file.

Rename – Rename a file

```
Synopsis:      Rename( OldName, NewName )

To server:     BYTE              Tag = 22
               INT16::[]BYTE     OldName
               INT16::[]BYTE     NewName

From server:   BYTE              Result
```

Rename changes the name of an existing file `OldName` to `NewName`.

FGetBlock – Read a block of data and return status

```
Synopsis:  Data,Result = FGetBlock(StreamId,Count)

To server:     BYTE              Tag = 23
               INT32             StreamId
               INT16             Count

From server:   BYTE              Result
               INT16::[]BYTE     Data
```

FGetBlock reads `Count` bytes of binary data from the specified stream. Input stops when the specified number of bytes are read, or the end of file is reached, or an error occurs. If `Count` is less than one then no input is done. The stream is left positioned immediately after the data read. If an error occurs the stream position is undefined.

The actual number of bytes returned may be less than requested. In the case of `Result` indicating a failure Feof and Ferror should be used to determine the cause of the error.

This function is preferred over the *Fread* function, which should no longer be used.

FPutBlock – Write a block of data and return status

`Synopsis:`	`Written,Result = FPutBlock(StreamId,Data)`	
`To server:`	`BYTE`	`Tag = 24`
	`INT32`	`StreamId`
	`INT16::[]BYTE`	`Data`
`From server:`	`BYTE`	`Result`
	`INT16`	`Written`

FPutBlock writes a given number of bytes of binary data to the specified stream, which should be open for output. If the length of `Data` is less than one then no output is done. The position of the stream is advanced by the number of bytes actually written. If an error occurs then the resulting position if undefined.

FPutBlock returns the number of bytes actually output in `Written`. The actual number of bytes returned may be less than requested and Feof and Ferror should be used to check for status.

If the `StreamId` is `1` (standard output) then the write is automatically flushed.

This function is preferred over the *Fwrite* function, which should no longer be used.

H.5.3 Host commands

Getkey – Get a keystroke

`Synopsis:`	`Key = GetKey()`	
`To server:`	`BYTE`	`Tag = 30`
`From server:`	`BYTE`	`Result`
	`BYTE`	`Key`

GetKey gets a single character from the keyboard. The keystroke is waited on indefinitely and will not be echoed. The effect on any buffered data in the standard input stream is host-defined.

Pollkey – Test for a key

```
Synopsis:      Key = PollKey()

To server:     BYTE                Tag = 31

From server:   BYTE                Result
               BYTE                Key
```

PollKey gets a single character from the keyboard. If a keystroke is not available then PollKey returns immediately with a non-zero result. If a keystroke is available it will not be echoed. The effect on any buffered data in the standard input stream is host-defined.

Getenv – Get environment variable

```
Synopsis:      Value = Getenv( Name )

To server:     BYTE                Tag = 32
               INT16::[]BYTE       Name

From server:   BYTE                Result
               INT16::[]BYTE       Value
```

Getenv returns a host-defined environment string for **Name**. If Name is undefined then **Result** will be non-zero. If the resultant environment string for **Name** is longer than the space available in the packet buffer, then it will be truncated.

Time – Get the time of day

```
Synopsis:      LocalTime, UTCTime = Time()

To server:     BYTE                Tag = 33

From server:   BYTE                Result
               INT32               LocalTime
               INT32               UTCTime
```

Time returns the local time and Coordinated Universal Time if it is available. Both times are expressed as the number of seconds that have elapsed since midnight on 1st January, 1970. If UTC time is unavailable then it will have a value of zero. The times are given as unsigned `INT32`s.

System – Run a command

```
Synopsis:      Status = System( Command )

To server:     BYTE              Tag = 34
               INT16::[]BYTE     Command

From server:   BYTE              Result
               INT32             Status
```

System passes the string `Command` to the host command processor for execution. If Command is zero length then System will succeed if there is a command processor. If Command is not null then `Status` is the return value of the command, which is host-defined.

H.5.4 Server commands

Exit – Terminate the server

```
Synopsis:      Exit( Status )

To server:     BYTE              Tag = 35
               INT32             Status

From server:   BYTE              Result
```

Exit terminates the server, which exits returning `Status` to its caller.

If `Status` has the special value 999999999 then the server will terminate with a host-specific 'success' result.

If `Status` has the special value -999999999 then the server will terminate with a host-specific 'failure' result.

CommandLine – Retrieve the server command line

`Synopsis:`	`String = CommandLine( All )`	
`To server:`	`BYTE`	`Tag = 40`
	`BYTE`	`All`
`From server:`	`BYTE`	`Result`
	`INT16::[]BYTE`	`String`

CommandLine returns the command line passed to the server on invocation. On certain operating systems it is possible to quote arguments on the command line. The quotes themselves have been removed by the time the arguments are passed on to the server. When building the command line to pass on to the application the server replaces the quotes.

If `All` is zero the returned string is the command line, with options and their arguments that the server recognised at startup removed, as well as the server command.

If `All` is non-zero then the string returned is the entire command vector as passed to the server on startup, including the name of the server command itself.

Core – Read peeked memory

`Synopsis`	`Data = Core( Offset, Length )`	
`To server:`	`BYTE`	`Tag = 41`
	`INT32`	`Offset`
	`INT16`	`Length`
`From server:`	`BYTE`	`Result`
	`INT16::[]BYTE`	`Core`

Core returns the contents of the root transputer's memory, as peeked from the transputer when the server was invoked with the analyse option.

Core fails if `Offset` is larger than the amount of memory peeked from the transputer or if the transputer was not analysed.

If `Offset + Length` is larger than the total amount of memory that was peeked then as many bytes as are available from the given offset are returned.

Version – Find out about the server

`Synopsis:`	`Id = Version()`	
`To server:`	`BYTE`	`Tag = 42`
`From server:`	`BYTE`	`Result`
	`BYTE`	`Version`
	`BYTE`	`Host`
	`BYTE`	`OS`
	`BYTE`	`Board`

Version returns four bytes containing identification information about the server and the host it is running on.

If any of the bytes has the value 0 then that information is not available.

`Version` identifies the server version. The byte value should be divided by ten to yield the version number.

`Host` identifies the host machine and can be any of the following:

1 PC

2 NEC-PC

3 VAX

4 Sun 3

5 370 Architecture

6 Sun 4

7 Sun 386i

8 Apollo

`OS` identifies the host environment and can be any of the following:

1 DOS

2 Helios

3 VMS

4 SunOS

5 CMS

`Board` identifies the interface board and can be any of the following:

1 B004

2 B008

3 B010

4 B011

5 B014

6 DRX-11

7 QT0

8 B015

9 CAT

10 B016

11 UDPlink

Values of `Host`, `OS` and `Board` from 0 to 127, inclusive, are reserved for use by INMOS.

I Glossary

Alias check A program compilation check that ensures that names are unique within a given scope.

Analyse To assert a signal to a transputer forcing it to halt at the next descheduling point, to allow the state of the processor to be read. In the context of 'analysing a network', to analyse all processors in the network.

Also refers to one of the system control functions on transputers and the pin on which the function is asserted.

Backtrace Within the debugger and simulator tools, to move from a position within a procedure or function body to the call of that procedure or function.

Bootable code Self-starting program code, that can be loaded onto a transputer or transputer network down a transputer link and run. Bootable code is produced by `icollect` from linked units (single transputer programs) or configuration binary files (configured programs).

Bootstrap A transputer program, loaded from a ROM or over a link after the transputer has been reset or analysed, which initialises the processor and loads a program for execution (which may be another loader).

Compiler library A group of occam library routines that are used by the compiler to implement extended arithmetic and transputer system operations.

Configuration The association of components of a program with a set of physical resources. Used in this manual to refer to the specific case of allocating software processes to processors in a network, and channels to links between processors. The term is also used, depending on the context, to describe the act of deciding on these allocations for a program, the configuration code which describes such a set of allocations, and the act of applying the configurer to a network description.

Configurer The tool which assigns processes and channels on a specified configuration of transputers. The output from the tool is a configuration binary file for input to `icollect`.

Deadlock A state in which one or more concurrent processes can no longer proceed because of a communication interdependency.

Error mode The compilation mode of a program that determines what happens when a program error (such as an array bounds violation) occurs. A program compiled using the toolset may be compiled in one of three error modes: HALT, STOP, or UNIVERSAL.

Error signal In the transputer, an external signal used to indicate that an error has occurred in a running program. Also refers to one of the system control functions on transputers. Error signals can be OR-ed together on transputer boards to indicate an error has occurred in one of the transputers in the network.

Extended data types occam data types `INT16`, `INT32`, `INT64`, `REAL32` and `REAL64`.

Hard channels Channels which are mapped onto links between processors in a transputer network (cf. *Soft channels*).

Host The computer which is running the toolset host file server and providing the filing system and terminal i/o.

Host file server A file server which provides access to the filing system and terminal i/o of a host operating system, which may be used when running standalone programs. The toolset host file server is distinct from that used to run the Transputer Development System (TDS).

Include file A file containing source code which is incorporated into a program using the `#INCLUDE` directive.

Library A collection of separately compiled procedures or functions, created by the toolset librarian `ilibr`, which may be shared between parts of a program or between different programs.

Library build file A file containing a list of input files for the librarian tool `ilibr`. Each file forms a separately loadable module in the library. Library build files must have the `.lbb` extension.

Library usage file A file listing the libraries and separately compiled units used by another library. Library usage files must have the `.liu` extension.

Link In the context of transputer hardware, the serial communication link between processors. Used as a verb in the context of program compilation, to collect together all the code for a program or compilation unit, resolving all references and recompiling where necessary, and place the collected code into a single file.

Linker The program or tool which links a program or compilation unit.

Loader Depending on the context, refers to the part of the host file server which loads a transputer network or to a small program which is loaded into a transputer, and which then distributes code to other transputers and loads a larger program on top of itself.

Makefile An input file for a Make program. A Makefile contains details of file dependencies and directions for rebuilding the object code. Makefiles are created for the toolset using **`imakef`**.

Network A set of transputers connected together using links as a connected graph, that is, in such a way that there is a path, via links and other transputers, from each transputer to every other transputer in the set.

Newline sequence The sequence of ASCII characters, defined within the host file server, that directs a new line to be started on the terminal display or within a file. Defined for the toolset as the sequence 'CR LF'.

Object code Intermediate code between source and bootable files. Object code cannot be directly loaded onto a transputer and run. The compiler and linker tools generate object code.

Peek and poke To read and write locations in a transputer's memory, by communication over a link, while the transputer is waiting for a bootstrap.

Preamble The part of a transputer loader program that initialises the state of the processor.

Priority In the transputer, the priority level at which the currently executing process is being run. INMOS transputers support two levels of priority, known as 'high' and 'low'.

Process Self-contained, independently executable occam code.

Protocol The pattern of communications between two processes, often including communications on more than one channel. When appearing as `PROTOCOL`, refers to a specific communication structure on an occam channel (see the *'occam 2 Reference Manual'*).

Reset The transputer system initialisation control signal. Also refers to the pin on which the signal is asserted.

Root transputer (or Root processor) The processor in a transputer network which is physically connected to the host computer, and through which the network is loaded or analysed.

Separate compilation A self-contained part of a program may be separately compiled, so that only those parts of a program which have changed since the last compilation need to be recompiled.

Server A program running in the host computer attached to a transputer network, which provides access to the filing system and terminal i/o of the host computer. The server can also be used to load the program onto the network.

Soft channels Channels declared and used within a process running on a single transputer. (cf. *Hard channels*). Soft channels are implemented by a single word in memory.

Standard error The host system error handler. Errors directed to standard error are displayed in a host-defined way, for example, on the terminal screen. For details of how to modify standard error on the system, consult the operating system documentation.

Standard input The host system input handler. Specifies the standard input device, for example the terminal keyboard or a disk file. For details of how to modify standard input on the system, consult the operating system documentation.

Standard output The host system output handler. Specifies the standard output device, for example, the terminal screen or a disk file. For details of how to modify standard input on the system, consult the operating system documentation.

Subsystem In transputer board architecture, the combination of the Reset, Analyse and Error signals which allows the board to control another board on its subsystem port.

Target transputer The transputer on which the code is intended to run. The transputer type, or a restricted set of types defined in a transputer class, is defined when the program is compiled, using command line options.

Usage check A compilation check that ensures no variables are shared between parallel processes, and that enforces rules about the use of channels as unidirectional point-to-point connections.

Vector space The data space required for the storage of vectors (arrays) within an occam program.

Workspace The data space required by an occam process; when used in contrast to *Vector space*, refers to the data space required for scalars within the process.

J Bibliography

This appendix contains a list of some transputer-related publications which may be of interest to the reader.

J.1 INMOS publications

D Pountain and D May
A tutorial introduction to occam programming
Blackwell Scientific 1987.

INMOS
occam 2 Reference Manual
Prentice Hall 1988.

INMOS, A B Fontaine (tr)
occam 2 Manuel de reference
Masson 1989?
(In French)

INMOS
occam
Keigaku Shuppan Publishing Company 1984
(In Japanese)

INMOS
Transputer instruction set: a compiler writer's guide
Prentice Hall 1988

INMOS Ltd
The Transputer Databook (Second Edition 1989)
INMOS 1989

INMOS Ltd
The Transputer Applications Notebook: Architecture and Software (First Edition 1989)
INMOS 1989

INMOS Ltd
The Transputer Applications Notebook: Systems and Performance (First Edition 1989)
INMOS 1989

INMOS Ltd
The Transputer Development and iq Systems Databook (Second Edition 1991)
INMOS 1991

J.2 INMOS technical notes

P Moore
IMS B010 NEC add-in board
Technical note 8
72 TCH 008

S Ghee
IMS B004 IBM PC add-in board
Technical note 11
72 TCH 011

G Harriman
Notes on graphics support and performance improvements on the IMS T800
Technical note 26
72 TCH 026

S Redfern
Implementing data structures and recursion in occam
Technical note 38
72 TCH 038

J.3 References

W J Cody and W M Waite
Software Manual for the Elementary Functions
Prentice Hall 1980

D E Knuth
The Art of Computer Programming
2nd edition, Volume 2: Seminumerical Algorithms
Addison-Wesley 1981

IEEE
IEEE Standard for Binary Floating-Point Arithmetic
ANSI-IEEE Std 754-1985

External occam 2 and transputer publications

K C Bowler, R D Kenway, G S Pawley and D Roweth
An introduction to occam 2 programming
Chartwell-Bratt 1987 ISBN 0-86-238-137-1

A Burns
Programming in occam 2
Addison-Wesley 1988 ISBN 0-201-17371-9

J Gallently
occam 2
Pitman 1989

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

J Wexler
Concurrent programming in occam 2
Ellis Horwood 1989 ISBN 745-80394-6

Index

inmos

Worldwide Headquarters

INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
UNITED KINGDOM
Telephone (0454) 616616
Fax (0454) 617910

Worldwide Business Centres

USA

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
2225 Executive Circle
PO Box 16000
Colorado Springs
Colorado 80935-6000
Telephone (719) 630 4000
Fax (719) 630 4325

SGS-THOMSON Microelectronics Inc.
Sales and Marketing Headquarters (USA)
1000 East Bell Road
Phoenix
Arizona 85022
Telephone (602) 867 6100
Fax (602) 867 6102

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
Lincoln North
55 Old Bedford Road
Lincoln
Massachusetts 01773
Telephone (617) 259 0300
Fax (617) 259 4420

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
9861 Broken Land Parkway
Suite 320
Columbia
Maryland 21045
Telephone (301) 995 6952
Fax (301) 290 7047

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
200 East Sandpointe
Suite 650
Santa Ana
California 92707
Telephone (714) 957 6018
Fax (714) 957 3281

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
2055 Gateway Place
Suite 300
San Jose
California 95110
Telephone (408) 452 9122
Fax (408) 452 0218

INMOS Business Centre
SGS-THOMSON Microelectronics Inc.
1310 Electronics Drive
Carrollton
Texas 75006
Telephone (214) 466 8844
Fax (214) 466 7352

ASIA PACIFIC

Japan

INMOS Business Centre
SGS-THOMSON Microelectronics K.K.
Nisseki Takanawa Building, 4th Floor
18–10 Takanawa 2-chome
Minato-ku
Tokyo 108
Telephone (03) 3280 4125
Fax (03) 3280 4131

Singapore

INMOS Business Centre
SGS-THOMSON Microelectronics Pte Ltd.
28 Ang Mo Kio Industrial Park 2
Singapore 2056
Telephone (65) 482 14 11
Fax (65) 482 02 40

EUROPE

United Kingdom

INMOS Business Centre
SGS-THOMSON Microelectronics Ltd.
Planar House
Parkway Globe Park
Marlow
Bucks SL7 1YL
Telephone (0628) 890 800
Fax (0628) 890 391

France

INMOS Business Centre
SGS-THOMSON Microelectronics SA
7 Avenue Gallieni
BP 93
94253 Gentilly Cedex
Telephone (1) 47 40 75 75
FAX (1) 47 40 79 10

West Germany

INMOS Business Centre
SGS-THOMSON Microelectronics GmbH
Bretonischer Ring 4
8011 Grasbrunn
Telephone (089) 46 00 60
Fax (089) 46 00 61 40

Italy

INMOS Business Centre
SGS-THOMSON Microelectronics SpA
V.le Milanofiori
Strada 4
Palazzo A/4/A
20090 Assago (MI)
Telephone (2) 89213 1
Fax (2) 8250449