

occam 2.1 Toolset Language and Libraries Reference Manual



August 1995

© SGS-THOMSON Microelectronics Limited 1995. This document may not be copied, in whole or in part, without prior written consent of SGS-THOMSON Microelectronics.

 **inmos**[®], IMS, occam and DS-Link[®] are trademarks of SGS-THOMSON Microelectronics Limited.

 **SGS-THOMSON** is a registered trademark of the SGS-THOMSON Microelectronics Group.

Windows is a trademark of Microsoft Corporation.

X Window System is a trademark of MIT.

OSF/Motif is a trademark of the Open Software Foundation, Inc.

This product incorporates innovative techniques which were developed with support from the European Commission under the ESPRIT Projects:

- P2701 PUMA (Parallel Universal Message-passing Architectures)
- P5404 GPMIMD (General Purpose Multiple Instruction Multiple Data Machines).
- P7250 TMP (Transputer Macrocell Project).
- P7267 OMI/STANDARDS.
- P6290 HAMLET (High Performance Computing for Industrial Applications)

Document Number: 72 TDS 368 02

Contents overview

Contents

Preface

Libraries

1	<i>The occam libraries</i>	Describes the libraries supplied with the toolset.
2	<i>Compiler libraries</i>	Describes the functions and procedures in the compiler library.
3	<i>Mathematics libraries</i>	Describes the maths library functions.
4	<i>Host file server library</i>	Describes the host server i/o library procedures.
5	<i>Stream i/o library</i>	Describes the stream i/o library procedures.
6	<i>String handling library</i>	Describes the string handling library functions and procedures.
7	<i>String conversion library</i>	Describes the string conversion library procedures.
8	<i>Block CRC library</i>	Describes the block CRC library functions.
9	<i>Extraordinary link handling library</i>	Describes the extraordinary link handling library procedures.
10	<i>Debugging support library</i>	Describes the interactive debugging support library procedures.

Appendices

A	<i>Language extensions</i>	Describes language extensions that are supported by the occam 2.1 compiler.
B	<i>Implementation of occam on the transputer</i>	Describes how the compiler allocates memory and gives details of type mapping, hardware dependencies and language.
C	<i>Alias and usage checking rules</i>	Describes the alias and usage checking that is implemented by the compiler.

Index

Contents

Contents overview	i
Contents	iii
Preface	v
Libraries	1
1 The occam libraries	3
1.1 Introduction	3
1.2 Using the occam libraries	3
2 Compiler libraries	5
2.1 Using compiler library routines	6
2.2 Mathematics functions	6
2.3 2D block moves	8
2.4 Bit manipulation functions	9
2.5 CRC functions	10
2.6 Floating point arithmetic support functions	12
2.7 Dynamic code loading support	14
2.8 Transputer-related procedures	17
2.9 Assertion test procedure	18
3 Mathematics libraries	19
3.1 Introduction and terminology	20
3.2 Single and double length elementary function libraries	25
3.3 T4 elementary function library	46
4 Host file server library	67
4.1 Errors and the server run time library	67
4.2 Reading real numbers	67
4.3 Procedure descriptions	68
4.4 Host i/o procedures arranged by purpose	68
4.5 Host i/o procedure definitions	72
5 Stream i/o library	101
5.1 Naming conventions	101
5.2 Stream i/o procedures	101
5.3 Stream i/o procedure definitions	104

Contents

6	String handling library	115
6.1	String handling procedures	115
6.2	String handling procedure definitions	117
7	String conversion library	125
7.1	Procedure definitions	126
8	Block CRC library	133
8.1	Example of use	134
8.2	Function definitions	134
9	Extraordinary link handling library	137
9.1	Procedure definitions	137
10	Debugging support library	141
10.1	Procedure definitions	141
	Appendices	143
A	Language extensions	145
B	Implementation of occam on the transputer	151
C	Alias and usage checking	165
	Index	175

Preface

Host versions

The documentation set which accompanies the OCCAM 2.1 toolset is designed to cover all host versions of the toolset:

- IBM 386 PC compatible running MS-DOS
- Sun 4 systems running SunOS or Solaris.

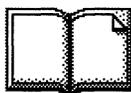
About this manual

This manual is the *occam 2.1 Language and Libraries Reference Manual* for the occam 2.1 Toolset.

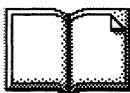
The manual provides a language reference for the toolset and implementation data. A list of the library functions provided is followed by detailed information about each function. Details of extensions to the language are given in an appendix.

About the toolset documentation set

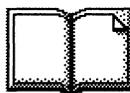
The toolset documentation set comprises the following volumes:



User Guide



Toolset Reference



Language and Libraries

- *occam 2.1 Toolset User Guide*

Describes the use of the toolset in developing programs for running on IMS T2xx, T4xx and T8xx transputers. The manual is divided into two sections; *Basics* which describes each of the main stages of the development process, how to compile, link and configure programs for the transputer. It also includes a *Getting started* tutorial. The *Advanced Techniques* section is aimed at more experienced users. The appendices contain a glossary of terms and a bibliography. Some chapters are generic to other toolsets e.g. *Developing programs for the transputer* and *Mixed language programming*.

- *Toolset Reference Manual*

Provides reference material for each tool in the occam 2.1 and ANSI C Toolsets, including command line options, syntax and error messages. Many of the tools in the Toolsets are generic to several toolset products, e.g. the occam 2.1 toolset and the ANSI C toolset, and the documentation reflects this. The appen-

Preface

ices provide details of toolset conventions, transputer types, memory configuration files and the configuration languages.

- *occam 2.1 Language and Libraries Reference Manual* (this manual)

Other documents

Other documents provided with your toolset product include:

- *Delivery manual*

This document gives installation data and is host specific.

- *occam 2.1 Reference Manual*

Transputer targets supported by this toolset

The documentation set provided with this toolset tends to be generic, i.e. it covers different transputer types and hosts. The OCCAM 2.1 toolset supports OCCAM 2.1 running on IMS T2xx, T4xx and T8xx transputers, including the ST20450. References in the documentation to other languages and transputer targets should be ignored.

The first generation of IMS T2xx, T4xx and T8xx transputers are referred to as the 'T2/T4/T8-series'. The new product family based around the IMS T9000 transputer is referred to as the 'T9-series'. The ST20450 may be called an ST20 or the T450 depending on the context, and is included in the T2/T4/T8-series family unless otherwise stated.

Documentation conventions

The following notational and typographical conventions are used in this manual:

Bold type	Used to emphasize new or special terminology.
Teletype	Used to distinguish command line examples, code fragments, and program listings from normal text.
<i>Italic type</i>	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Brackets []	Used in command syntax to denote optional items on the command line.
Braces { }	Used to denote optional items in a command syntax which may be repeated.
Ellipsis ...	Used to denote a range or the continuation of a series. For example, in syntax definitions denotes a list of one or more items.
	In command syntax, separates two mutually exclusive alternatives.

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 the compiler itself uses, and a number of *user libraries* to support common programming tasks. The compiler libraries are automatically referenced whereas user libraries must be declared in a #USE directive. Libraries, including the compiler libraries, must be specified to the linker. Table 1.1 lists the occam libraries.

Library	Description	Source provided	Uses include file
Compiler libraries			
<code>occamx.lib</code>	Multiple length integer arithmetic Floating point functions 32-bit IEEE arithmetic functions 64-bit IEEE arithmetic functions 2D block move library Bit manipulation CRC functions Supplementary floating point support Dynamic code loading support Transputer-related functions	No	
User libraries			
<code>snglmath.lib</code>	Single length mathematical functions	Yes	<code>mathvals.inc</code>
<code>dblmath.lib</code>	Double length mathematical functions	Yes	<code>mathvals.inc</code>
<code>tbmaths.lib</code>	T400/T414/T425/ST20450 optimized maths	Yes	<code>mathvals.inc</code>
<code>hostio.lib</code>	Host file server library	Yes	<code>hostio.inc</code>
<code>streamio.lib</code>	Stream I/O library	Yes	<code>streamio.inc</code>
<code>string.lib</code>	String handling library	Yes	
<code>convert.lib</code>	String conversion library	Yes	
<code>crc.lib</code>	Block CRC library	Yes	
<code>xlink.lib</code>	Extraordinary link handling library	No	
<code>debug.lib</code>	Debugging support library	No	

Table 1.1 occam libraries

1.2 Using the occam libraries

Libraries other than the compiler library must be referenced in a #USE directive. For example:

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

1.2 Using the occam libraries

Any use of a library routine must be in scope with the `#USE` directive which references the associated library. The scope of a library, like any OCCAM declaration, depends on its level of indentation within the text.

If the library uses an include file of predefined constants (see section 1.2.3), as listed in the right hand column of Table 1.1, then this must be referenced by an `#INCLUDE` directive before the associated `#USE`. For example:

```
#INCLUDE "hostio.inc"  
#USE "hostio.lib"
```

1.2.1 Linking libraries

All libraries used by a program or program module must be linked with the main program. This includes the compiler libraries even though they are automatically referenced by the compiler (see Chapter 2).

1.2.2 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 11 of the *Toolset Reference Manual* for details of `ilist`.

1.2.3 Library constants

Constants and protocols used by the libraries are defined in the include files listed in Table 1.2.

File	Description	Related libraries
<code>hostio.inc</code>	Constants for the host file server interface (hostio library)	<code>hostio.lib</code>
<code>streamio.inc</code>	Constants for the stream i/o interface (streamio library)	<code>streamio.lib</code>
<code>mathvals.inc</code>	Maths constants	<code>englmath.lib</code> <code>dblmath.lib</code> <code>tbmaths.lib</code>
<code>linkaddr.inc</code>	Addresses of transputer links	
<code>ticks.inc</code>	Rates of the two transputer clocks	

Table 1.2 Include files

Include files should always be referenced before the related library.

2 Compiler libraries

Compiler libraries contain multiple length and floating point arithmetic functions, 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 a `#USE` directive. However, they *must* be specified to the linker along with all other libraries that the program uses; this is best done using one of the linker indirect files `occam2.lnk`, `occam8.lnk`, `occama.lnk` or `occam450.lnk`, which specify the correct libraries for the transputer target.

Separate compiler libraries are supplied for different types and classes of processors. Processor types supported are:

- T2 series
- T4 series
- ST20450
- T8 series
- All 32-bit processors

The compiler selects the correct library for the transputer type specified. All error modes are supported in each library.

File	Processor types supported
<code>occam2.lib</code>	T212, T222, T225, M212
<code>occam8.lib</code>	T800, T801, T805
<code>occama.lib</code>	T400, T414, T425, TA, TB
<code>occam450.lib</code>	ST20450
<code>occamut1.lib</code>	All
<code>virtual.lib</code>	All

`occamut1.lib` contains routines which are called from within some of the other compiler libraries and `virtual.lib` is used to support software virtual routing. 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. See Section A.4 in the *Toolset Reference Manual* for details of the mechanism for locating files.

The compiler `ε` option disables all of the compiler libraries except `virtual.lib`, which can be disabled by the `γ` option.

The *occam 2.1 Reference Manual* contains formal descriptions of many of the compiler library routines.

2.1 Using compiler library routines

2.1 Using compiler library routines

Although primarily intended for use by the compiler, some compiler library routines are available to the programmer. These are listed in sections 2.2 through 2.9. They can be called directly without referencing a library via a `#USE` statement. They are disabled by the compiler `E` option.

As an example of how they may be used, consider an application which requires compliance with the IEEE standards for **NaNs** (floating point Not-a-Numbers) and **Infs** (floating point infinities). The *occam* compiler defaults to non-IEEE behavior, i.e. **NaNs** and **Infs** are treated as errors, whereas ANSI/IEEE 754-1985 requires there to be error and overflow handling. To obtain IEEE behavior the appropriate compiler library function must be called.

The following code fragments show a simple addition can be implemented by default or using IEEE-compatible functions.

If **A**, **B**, and **C** are **REAL32s** and **b** is a **BOOL**:

```
A := B + C      Default occam behavior.
```

```
A := REAL32OP (B, 0, C)      IEEE function, round to nearest only.  
The 0 indicates add.
```

```
b, A := IEEE32OP (B, 1, 0, C)  IEEE function with rounding option.  
The 1 indicates round to nearest, and the 0 indicates add.
```

2.2 Mathematics functions

Tables 2.1 to 2.3 list the compiler library mathematics functions available to the programmer. Further details can be found in Appendices J, K and L of the *occam 2.1 Reference Manual*.

Result(s)	Function name	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	SHIFLEFT	VAL INT hi.in, lo.in, places
INT, INT	SHIFRIGHT	VAL INT hi.in, lo.in, places
INT, INT, INT	NORMALISE	VAL INT hi.in, lo.in
INT	ASHIFLEFT	VAL INT argument, places
INT	ASHIFRIGHT	VAL INT argument, places
INT	ROTATELEFT	VAL INT argument, places
INT	ROTATERIGHT	VAL INT argument, places

Table 2.1 Multiple length arithmetic functions

Notes on multiple length arithmetic functions

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

ASHIFTRIGHT, **ASHIFLEFT**, **ROTATERIGHT** and **ROTATELEFT** are all invalid when the number of places to shift is negative or exceeds the word length of the transputer.

Result(s)	Function name	Parameter specifiers
REAL32	ABS	VAL REAL32 x
REAL64	DABS	VAL REAL64 x
REAL32	SQRT	VAL REAL32 x
REAL64	DSQRT	VAL REAL64 x
BOOL	ISNAN	VAL REAL32 x
BOOL	DISNAN	VAL REAL64 x
BOOL	NOTFINITE	VAL REAL32 x
BOOL	DNOTFINITE	VAL REAL64 x
REAL32	SCALEB	VAL REAL32 x, VAL INT n
REAL64	DSCALEB	VAL REAL64 x, VAL INT n
REAL32	LOGB	VAL REAL32 x
REAL64	DLOGB	VAL REAL64 x
INT, REAL32	FLOATING.UNPACK	VAL REAL32 x
INT, REAL64	DFLOATING.UNPACK	VAL REAL64 x
REAL32	MINUSX	VAL REAL32 x
REAL64	DMINUSX	VAL REAL64 x
REAL32	COPYSIGN	VAL REAL32 x, y
REAL64	DCOPYSIGN	VAL REAL64 x, y
REAL32	NEXTAFTER	VAL REAL32 x, y
REAL64	DNEXTAFTER	VAL REAL64 x, y
BOOL	ORDERED	VAL REAL32 x, y
BOOL	DORDERED	VAL REAL64 x, y
BOOL, INT32, REAL32	ARGUMENT.REDUCE	VAL REAL32 x, y, y.err
BOOL, INT32, REAL64	DARGUMENT.REDUCE	VAL REAL64 x, y, y.err
REAL32	MULBY2	VAL REAL32 x
REAL64	DMULBY2	VAL REAL64 x
REAL32	DIVBY2	VAL REAL32 x
REAL64	DDIVBY2	VAL REAL64 x
REAL32	FPINT	VAL REAL32 x
REAL64	DFPINT	VAL REAL64 x

Table 2.2 Floating point functions

2.3 2D block moves

Result(s)	Function name	Parameter specifiers
REAL32	REAL32OP	VAL REAL32 x, VAL INT op, VAL REAL32 y
REAL64	REAL64OP	VAL REAL64 x, VAL INT op, VAL REAL64 y
REAL32	REAL32REM	VAL REAL32 x, y
REAL64	REAL64REM	VAL REAL64 x, y
BOOL, REAL32	IEEE32OP	VAL REAL32 x, VAL INT rm, op, VAL REAL32 y
BOOL, REAL64	IEEE64OP	VAL REAL64 x, VAL INT rm, op, VAL REAL64 y
BOOL, REAL32	IEEE32REM	VAL REAL32 x, y
BOOL, REAL64	IEEE64REM	VAL REAL64 x, y
BOOL	REAL32EQ	VAL REAL32 x, y
BOOL	REAL64EQ	VAL REAL64 x, y
BOOL	REAL32GT	VAL REAL32 x, y
BOOL	REAL64GT	VAL REAL64 x, y
INT	IEEECOMPARE	VAL REAL32 x, y
INT	DIEEECOMPARE	VAL REAL64 x, y

Table 2.3 Full IEEE floating point arithmetic

2.3 2D block moves

This section describes compiler library block move procedures available to the programmer.

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

Table 2.4 2D block move functions

CLIP2D

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

As MOVE2D but only bytes with value zero are transferred.

This is equivalent to:

```

SEQ line = 0 FOR length
  SEQ point = 0 FOR width
  VAL temp IS Source[line+sy][point+sx] :
  IF
    temp = (0(BYTE))
    Dest[line+dy][point+dx] := 0(BYTE)
  TRUE
  SKIP

```

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. This is equivalent to:

```

SEQ line = 0 FOR length
  SEQ point = 0 FOR width
  VAL temp IS Source[line+sy][point+sx] :
  IF
    temp <> (0(BYTE))
    Dest[line+dy][point+dx] := temp
  TRUE
  SKIP

```

MOVE2D

```

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

```

This is equivalent to:

```

SEQ y = 0 FOR length
  [Dest[y+dy] FROM dx FOR width] :=
  [Source[y+sy] FROM sx FOR width]

```

2.4 Bit manipulation functions

This section describes compiler library bit-based functions available to the programmer.

Result	Function name	Parameter specifiers
INT	BITCOUNT	VAL INT Word, CountIn
INT	BITREVNBITS	VAL INT x, n
INT	BITREVEWORD	VAL INT x

Table 2.5 Bit manipulation functions

2.5 CRC functions

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.

BITREWORD

INT FUNCTION BITREWORD (VAL INT x)

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

2.5 CRC functions

This section describes compiler library CRC (cyclic redundancy check) functions available to the programmer. Two further functions **CRCFROMMSB** and **CRCFROMLSB** are described in Chapter 8, and are provided in the library `crc.lib` for calculating CRC values from byte strings.

Result	Function name	Parameter specifiers
INT	CRCWORD	VAL INT data, CRCIn, generator
INT	CRCBYTE	VAL INT data, CRCIn, generator

Table 2.6 CRC functions

Two functions are provided, **CRCWORD** and **CRCBYTE**, as listed in Table 2.6. **CRCWORD** calculates the CRC of an integer, while **CRCBYTE** calculates the CRC of the most significant byte of an integer.

A cyclic redundancy check value is the remainder from modulo 2 polynomial division. Consider bit sequences as representing the coefficients of polynomials; for example, the bit sequence 10100100 (where the leading bit is the most significant bit) corresponds to $P(x) = x^7 + x^5 + x^2$. **CRCWORD** and **CRCBYTE** calculate the remainder of the following modulo 2 polynomial division:

$$\frac{x^n H(x) + F(x)}{G(x)}$$

where: $F(x)$ corresponds to *data* (the whole word for **CRCWORD**; only the *most* significant byte for **CRCBYTE**)

$G(x)$ corresponds to `generator`

$H(x)$ corresponds to `CRCIn`

n is the word size in bits of the processor used (i.e. n is 16 or 32).

`CRCIn` can be viewed as the value that would be pre-loaded into the cyclic shift register that is part of hardware implementations of CRC generators.

When representing $G(x)$ in the word `generator`, note that there is an understood bit before the most significant bit of `generator`. For example, on a 16-bit processor, with $G(x) = x^{16} + x^{12} + x^5 + 1$, which is #11021, then `generator` must be assigned #1021, because the bit corresponding to x^{16} is understood. Thus, a value of #9603 for `generator`, corresponds to $G(x) = x^{16} + x^{15} + x^{12} + x^{10} + x^9 + x + 1$, for a 16-bit processor.

A similar situation holds on a 32-bit processor, so that:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

is encoded in `generator` as #04C11DB7.

It is possible to calculate a 16-bit CRC on a 32-bit processor. For example, if:

$$G(x) = x^{16} + x^{12} + x^5 + 1,$$

then `generator` is #10210000, because the most significant 16 bits of the 32-bit integer form a 16-bit generator.

For `CRCWORD`, the least significant 16 bits of `CRCIn` form the initial CRC value; the most significant 16 bits of `data` form the data; and the calculated CRC is the most significant 16 bits of the result.

For `CRCBYTE`, the most significant 16 bits of `CRCIn` form the initial CRC value; the next 8 bits of `CRCIn` (the third most significant byte) form the byte of data; and the calculated CRC is the most significant 16 bits of the result.

CRCWORD

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

Takes the whole of the word `data` to correspond to $F(x)$ in the above formula. This implements the following algorithm:

```

INT MyData, CRCOut, OldCRC :
VALOF
SEQ
    MyData, CRCOut := data, CRCIn
    SEQ i = 0 FOR BitsPerWord -- 16 or 32
        SEQ
            OldCRC := CRCOut
            CRCOut, MyData := SHIFTLLEFT (CRCOut, MyData, 1)
            IF
                OldCRC < 0 -- MSB of CRC = 1
                    CRCOut := CRCOut >> generator
            TRUE
            SKIP
    RESULT CRCOut

```

2.6 Floating point arithmetic support functions

CRCBYTE

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

Takes the *most* significant byte of *data* to correspond to $F(x)$ in the above formula. This implements the following algorithm:

```
INT MyData, CRCOut, OldCRC :
VALOF
SEQ
    MyData, CRCOut := data, CRCIn
    SEQ i = 0 FOR 8
        SEQ
            OldCRC := CRCOut
            CRCOut, MyData := SHIFTLLEFT (CRCOut, MyData, 1)
        IF
            OldCRC < 0 -- MSB of CRC = 1
                CRCOut := CRCOut >> generator
            TRUE
                SKIP
    RESULT CRCOut
```

Note: The predefines `CRCBYTE` and `CRCWORD` can be chained together to help calculate a CRC from a string considered as one long polynomial. A simple chaining would calculate:

$$\frac{x^k H(x) + F(x)}{G(x)}$$

where $F(x)$ corresponds to the string and k is the number of bits in the string. This is not the same CRC that is calculated by `CRCFROMMSB` and `CRCFROMLSB` in `crc.lib`, as described in Chapter 8, because these latter routines shift the numerator by x^n .

2.6 Floating point arithmetic support functions

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

Table 2.7 Floating point arithmetic support functions

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 repre-

sentation. 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 -1.0 .

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, coded as shown in Table 2.8.

Type	Reason
0	<code>x</code> is zero
1	<code>x</code> is a normalized or denormalized number
2	<code>x</code> is <code>Inf</code>
3	<code>x</code> is <code>NaN</code>

Table 2.8 `unpacksn` result type codes

Examples:

```
UNPACKSN (#40490FDB) returns #C90FDB00 , #0000080, 1
UNPACKSN (#00000001) returns #00000100 , #00000001, 1
UNPACKSN (#7FC00001) returns #40000100 , #000000FF, 3
```

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 fraction (possibly not normalized), guard word and exponent, and returns the IEEE single length floating point value it represents. It takes care of all the normalization, post-normalization, rounding and packing of the result. The rounding mode used is round to nearest. The exponent should already be biased. This routine is not intended for use with `Yexp` and `Yfrac` representing an infinity or a `NaN`.

2.7 Dynamic code loading support

Examples:

```

ROUNDSN (#00000080, #C90FDB00, #00000000) returns #40490FDB
ROUNDSN (#00000080, #C90FDB80, #00000000) returns #40490FDC
ROUNDSN (#00000080, #C90FDA80, #00000000) returns #40490FDA
ROUNDSN (#00000080, #C90FDA80, #00003000) returns #40490FDB
ROUNDSN (#00000001, #00000100, #00000000) returns #00000001

```

The function normalizes and post-normalizes the number represented by **Yexp**, **Yfrac** and **Yguard** into local variables **Xexp**, **Xfrac**, and **Xguard**. 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 overflow 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**.

2.7 Dynamic code loading support

This section describes compiler library dynamic loading routines available to the programmer. The use of these routines for dynamic code loading is described in section 12.4 of the *Occam 2.1 Toolset User Guide*.

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

Table 2.9 Dynamic code loading procedures

Result(s)	Function name	Parameter specifiers
INT	WSSIZEOF	routinename
INT	VSSIZEOF	routinename

Table 2.10 Dynamic code loading functions

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 use of this procedure for dynamic code loading is described in section 12.4 of the *occam 2.1 Toolset User Guide*.

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 by visually inspecting the executable code file (`.rsc` file) to be loaded using the binary lister tool `ilist`. Alternatively, a routine can be written to read this file and pass the information to `KERNEL.RUN`. The format of the `.rsc` file is described in section 5.6 of the *Toolset Reference Manual*.

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 not less than 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 `lptr` 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.

LOAD.INPUT.CHANNEL

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

2.7 Dynamic code loading support

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

The normal protocol checking of channel parameters is suppressed; therefore channels of any protocol may be passed to this routine. The channel parameter is considered by the compiler to have been used for input.

LOAD.INPUT.CHANNEL.VECTOR

```
PROC 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).

The normal protocol checking of channel parameters is suppressed; therefore channels of any protocol may be passed to this routine. The channel parameter is considered by the compiler to have been used for input.

LOAD.OUTPUT.CHANNEL

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

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

The normal protocol checking of channel parameters is suppressed; therefore channels of any protocol may be passed to this routine. The channel parameter is considered by the compiler to have been used for input.

LOAD.OUTPUT.CHANNEL.VECTOR

```
PROC 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).

The normal protocol checking of channel parameters is suppressed; therefore channels of any protocol may be passed to this routine. The channel parameter is considered by the compiler to have been used for input.

LOAD.BYTE.VECTOR

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

The variable `here` is assigned the address of the byte array `bytes`. This can be used in conjunction with `RETTYPES` to find the address of any variable.

WSSIZEOF

```
INT FUNCTION WSSIZEOF (routine_name)
```

This function returns the number of workspace slots (words) required by the procedure or function *routine_name*. **INLINE** or predefined routines are not permitted.

VSSIZEOF

```
INT FUNCTION VSSIZEOF (routine_name)
```

This function returns the number of vectorspace slots (words) required by the procedure or function *routine_name*. **INLINE** or predefined routines are not permitted.

2.8 Transputer-related procedures

This section describes compiler library transputer-specific routines available to the programmer.

Procedure	Parameter specifiers
CAUSEERROR	none
RESCHEDULE	none

Table 2.11 Transputer-related procedures

CAUSEERROR

```
PROC CAUSEERROR()
```

Inserts instructions into the program to set the transputer error flag. If the program is in **STOP** or **UNIVERSAL** mode instructions to stop the current process are also inserted.

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 and in **STOP** mode that process will stop, leaving the transputer error flag set **TRUE**. If run-time error checking has been suppressed (e.g. by a command line option), this stop is suppressed.

The difference between **CAUSEERROR()** and the **STOP** process, is that **CAUSEERROR** guarantees to set the transputer error flag.

RESCHEDULE

```
PROC RESCHEDULE()
```

2.9 Assertion test procedure

This causes the current process to be rescheduled by inserting instructions into the program to cause the current process to be moved to the end of the current priority scheduling queue. This occurs even if the current process is a high priority process.

RESCHEDULE effectively forces a timeslice, even in high priority.

2.9 Assertion test procedure

This section describes the compiler library debugging assertion test routine available to the programmer for use when **INQUEST** is not present. Further debugging support procedures for use when debugging interactively are provided in the debugging support library described in Chapter 10.

Procedure	Parameter specifiers
ASSERT	VAL BOOL test

Table 2.12 Assertion test procedure

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. This run-time check may be disabled by means of a command line option.

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 **ASSERTS** at run-time. If possible **ASSERTS** will still be checked at compile time.

3 Mathematics libraries

Elementary mathematics and trigonometric functions are provided in three libraries, as listed in Table 3.1.

Library	Description
<code>snglmath.lib</code>	Single length library
<code>dblmath.lib</code>	Double length library
<code>tbmaths.lib</code>	T400 / T414 / T425 / ST20450 optimized library

Table 3.1 Mathematics libraries

The single and double length libraries contain the same set of mathematics functions in single and double length forms. The double length forms all begin with the letter **D**. All function names are in upper case.

The T4 optimized library is a combined single and double length library containing functions for the T4 series (IMS T400, T414, T425) and the ST20450. The functions have been optimized for speed and are implemented using fixed point floating point. The standard single or double length libraries *can* be used on T4 processors but optimum performance will be achieved by using the TB optimized library. The accuracy of the T4 optimized functions is similar to that of the standard single length functions but results returned may not be identical because different algorithms are used. If the optimized library is used in code compiled for any processor except a T4 then the compiler reports an error.

To obtain the best possible speed performance with the **occam** mathematics functions use the following strategy:

- For networks consisting of only T4 series transputers, use the `tbmaths.lib` library.
- For networks consisting of only T8 series transputers, use the `snglmath.lib` and `dblmath.lib` libraries.
- For networks consisting of a mix of T4 series and T8 series transputers use:
 - `tbmaths.lib` on the T4 series and `snglmath.lib` or `dblmath.lib` on the T8 series when a consistent level of accuracy is not required;
 - if accuracy must be the same in the T8 and T4 processes then use the `snglmath.lib` and `dblmath.lib` libraries.

Constants for the mathematics libraries are provided in the include file `mathvals.inc`.

The elementary function library is also described in Appendix M of the *occam 2.1 Reference Manual*.

3.1 Introduction and terminology

3.1 Introduction and terminology

This, and the following subsections, contain some notes on the presentation of the elementary function libraries described in section 3.2, and the TB version described in section 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.

3.1.1 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). For each function, the parameters 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-Numbers) and **Infs** (Infinities). Generally if a **NaN** is used as an argument then the result is the same **NaN**. **Infs** may or may not be in the domain. The domain is the set of arguments for which the result is a normal (or de-normalized) floating-point number.

3.1.2 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 values in Table 3.2 for these Not-a-Numbers:

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

Table 3.2 Not-a-Number values

3.1.3 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 output 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:

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

3.1 Introduction and terminology

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 (MRE)* and the *root-mean-square relative error (RMSRE)*. These measures are defined by the following formulae:

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

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 $1+\varepsilon \neq 1$). Its magnitude depends on the floating-point format: for single-length it is $2^{-23} = 1.19 \times 10^{-7}$, and for double-length it is $2^{-52} = 2.22 \times 10^{-16}$.

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.

3.1.4 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.

3.1.5 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.

3.1.6 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**.

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

RAN returns two results, of types **REAL32** and **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** (**NaN**s are copied directly to the output and are not listed as exceptions). These are all **Infs** or **NaNs**.

3.1 Introduction and terminology

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

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.

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

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 behavior of the function.

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.

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 $2^{-23} = 1.19 \times 10^{-7}$.

For the double-length format it is $2^{-52} = 2.22 \times 10^{-16}$.

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.

3.2 Single 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 optimized library described in section 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 mathematics 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

3.2 Single and double length elementary function libraries

Result(s)	Function	Parameter specifiers
REAL32	SINH	VAL REAL32 X
REAL32	COSH	VAL REAL32 X
REAL32	TANH	VAL REAL32 X
REAL32, INT32	RAN	VAL INT32 X
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

Table 3.3 Elementary functions

ACOS, DACOS

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

Compute 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

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

Step 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

Note 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.

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

3.2 Single and double length elementary function libraries

ALOG, DALOG

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

Compute $\log_e(X)$.

Domain: (0, *XMAX*]

Range: [*MinLog*, *MaxLog*] [*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 (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

Step 1 Split *X* into its exponent *N* and fraction *F*.

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

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

Notes

Note 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.

Note 2 The minimum value that can be produced, *MinLog*, is the logarithm of the smallest de-normalized floating-point number. For single length *MinLog* is -103.28, and for double length it is -744.4. The maximum value *MaxLog* is the logarithm of *XMAX*. For single-length it is 88.72, and for double-length it is 709.78.

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

Note 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

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

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

Step 1 Set $temp := \text{ALOG}(X)$.

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

Notes

Note 1 See note 1 for **ALOG**.

Note 2 The minimum value that can be produced, $MinL10$, is the base-10 logarithm of the smallest de-normalized floating-point number. For single length $MinL10$ is -44.85 , and for double length it is -323.3 . 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 .

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

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

3.2 Single and double length elementary function libraries

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

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

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

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

Step 2 Otherwise compute the result directly using the rational approximation.

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

Notes

Note 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.

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

ATAN, DATAN

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

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

Domain: $[-\text{Inf}, \text{Inf}]$
Range: $[-\text{Pi}/2, \text{Pi}/2]$
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

- Step 1 If $|X| > 1.0$, set $X_{work} = 1/|X|$, otherwise $X_{work} = |X|$.
 Step 2 If $X_{work} > 2 - \sqrt{3}$, set $F = (X_{work} * \sqrt{3} - 1) / (X_{work} + \sqrt{3})$, otherwise $F = X_{work}$.
 Step 3 Compute $R_{work} = \arctan(F)$ with a floating-point rational approximation.
 Step 4 If X_{work} was reduced in (2), set $R = \text{Pi}/6 + R_{work}$, otherwise $R = R_{work}$.
 Step 5 If X was reduced in (1), set $RESULT = \text{Pi}/2 - R$, otherwise $RESULT = R$.
 Step 6 Set the sign of the $RESULT$ according to the sign of the argument.

Notes

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

3.2 Single and double length elementary function libraries

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: $[-\text{Inf}, \text{Inf}] \times [-\text{Inf}, \text{Inf}]$

Range: $(-\text{Pi}, \text{Pi}]$

Primary domain: See note 2.

Exceptions

(0, 0) and $(\pm\text{Inf}, \pm\text{Inf})$ 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

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

Step 2 Otherwise set $Rwork := \text{ATAN}(Y/X)$. Then if $Y < 0$ set $RESULT = Rwork - \text{Pi}$, otherwise set $RESULT = \text{Pi} - Rwork$.

Notes

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

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

Note 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.

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)
 $= [-2.1 \cdot 10^8, 2.1 \cdot 10^8]$ D

Range: $[-1.0, 1.0]$

Primary domain: See note 1.

Exceptions

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

Propagated Error

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

Generated error

	[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	0.90 ulp	0.19 ulp

The Algorithm

Step 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).

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

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

Notes

Note 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

3.2 Single and double length elementary function libraries

remainder equal to the number of extra bits of agreement, and this causes a loss of accuracy in the result.

- Note 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 C_{max} 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 C_{max} prevents this also.
- Note 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.
- Note 4 The propagated error has a complex behavior. 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.
- Note 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 .
- Note 6 The output range is not exceeded. Thus the output of `COS` is always a valid argument for `ACOS`.

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** **RMSRE**

Single Length: 1.24 ulp 0.32 ulp

Double Length: 1.24 ulp 0.32 ulp

The Algorithm

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

Step 2 Otherwise, set $temp := \mathbf{EXP}(|X|)$, and set $result = (temp + 1/temp)/2$.

Notes

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

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

Note 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.

3.2 Single and double length elementary function libraries

EXP, DEXP

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

Compute e^X .

Domain: $[-\text{Inf}, \text{MaxLog}] = [-\text{Inf}, 88.72]_S, [-\text{Inf}, 709.78]_D$

Range: $[0, \text{Inf}]$ (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:	1.4 ulp	0.25 ulp

The Algorithm

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

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

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

Step 4 Increase the exponent of the result by N . If N is sufficiently negative the result must be de-normalized.

Notes

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

Note 2 For sufficiently negative arguments (below -87.34 for single-length and below -708.4 for double-length) the output is de-normalized, 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.

Note 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.

Note 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: $[0, \text{Inf}] \times [-\text{Inf}, \text{Inf}]$

Range: $(-\text{Inf}, \text{Inf})$

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:

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

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

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

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

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

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

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

Notes

Note 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.

Note 2 Results for special cases are as given in Table 3.4.

Note 3 Performing all the calculations in extended precision makes the double-precision algorithm very complex in detail, and having two arguments makes a

3.2 Single and double length elementary function libraries

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.

Note 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.

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	$-\text{Inf} \leq Y \leq -1$	0
Inf	$-1 < Y < 1$	undefined.NaN
otherwise	0	1
otherwise	1	X

Table 3.4 Power special cases

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

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

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

Step 3 Output the floating point result and the new integer.

Notes

Note 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 initialized before the first call to **RAN** but not modified thereafter except by the function itself.

Note 2 If the integer parameter is initialized 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 initialized to some 'random' value, such as the output of a timer.

Note 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.

Note 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. Several different integers can produce the same floating-point value, so a floating-point output may be repeated, although the *sequence* will not be repeated until *M* calls have been made.

Note 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*.

Note 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.

3.2 Single and double length elementary function libraries

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)
= $[-4.29*10^9, 4.29*10^9]$ 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 $\pm Inf$, which generates an **undefined.NaN**.

Propagated Error

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

Generated error (See note 1)

	Primary Domain		[0, 2 <i>Pi</i>]	
	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:	0.90 ulp	0.22 ulp	0.91 ulp	0.18 ulp

The Algorithm

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

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

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

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

Notes

Note 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.

Note 2 The propagated error has a complex behavior. 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.

Note 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 .

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

3.2 Single and double length elementary function libraries

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: $(-\text{Inf}, \text{Inf})$

Primary domain: $(-1.0, 1.0)$

Exceptions

$X < -Hmax$ gives $-\text{Inf}$, 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	MAE	RMSAE
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

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

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

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

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

Notes

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

Note 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.

Note 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.

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),
 $= [-2.1 \times 10^6, 2.1 \times 10^6]$ S(T2 special),
 $= [-2.1 \times 10^9, 2.1 \times 10^9]$ D

Range: $(-\text{Inf}, \text{Inf})$

Primary domain: $[-Pi/4, Pi/4] = [-0.785, 0.785]$

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except $\pm \text{Inf}$, 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** (See note 3)

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

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

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

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

Notes

Note 1 R is large whenever X is near 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.

Note 2 The difference between **TAN** evaluated at successive floating-point numbers is given approximately by the absolute error amplification factor, A . For arguments

3.2 Single and double length elementary function libraries

larger than S_{max} 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.

Note 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.

Note 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.

TANH, DTANH

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

Compute $\tanh(X)$.

Domain: $[-\text{Inf}, \text{Inf}]$

Range: $[-1.0, 1.0]$

Primary domain: $[-\text{Log}(3)/2, \text{Log}(3)/2] = [-0.549, 0.549]$

Exceptions

None.

Propagated Error

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

Generated Error

Primary domain error: **MRE** **RMSRE**

Single Length: 0.53 ulp 0.2 ulp

Double Length: 0.53 ulp 0.2 ulp

The Algorithm

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

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

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

Notes

Note 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 .

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

3.3 T4 elementary function library

To use this library a program header must include the line:

```
#USE "tmaths.lib"
```

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, T425 and ST20450 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 and the processor ST20450.

The tables and notes at the beginning of this chapter apply equally here. However all the functions are contained in one library.

ACOS, DACOS

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

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

Domain: [-1.0, 1.0]

Range: [0, π]

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:	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

Step 1 If $|X| > 0.5$, set $X_{work} := \text{SQRT}((1 - |X|)/2)$. Compute $R_{work} = \arcsin(2 * X_{work})$ with a floating-point rational approximation. If the argument was positive, this is the result, otherwise set the result = $\pi - R_{work}$.

Step 2 Otherwise compute R_{work} directly using the rational approximation. If the argument was positive, set result = $\pi/2 - R_{work}$, otherwise result = $\pi/2 + R_{work}$.

Notes

Note 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.

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

3.3 T4 elementary function library

ALOG, DALOG

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

These compute: $\log_2(X)$

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

$A \equiv 1, \quad R = \log_2(X)$

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

Step 1 Split X into its exponent N and fraction F .

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

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

Notes

Note 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.

Note 2 The minimum value that can be produced, $MinLog$, is the logarithm of the smallest de-normalized floating-point number. For single length $MinLog$ is -103.28 , and for double length it is -744.4 . The maximum value $MaxLog$ is the logarithm of $XMAX$. For single-length it is 88.72 , and for double-length it is 709.78 .

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

Note 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)
```

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

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

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

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

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

Notes

Note 1 See note 1 for **ALOG**.

Note 2 The minimum value that can be produced, $MinL10$, is the base-10 logarithm of the smallest de-normalized floating-point number. For single length $MinL10$ is -44.85 , and for double length it is -323.3 . 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 .

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

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

3.3 T4 elementary function library

ASIN, DASIN

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

These compute: $\sin^{-1}(X)$ (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

Step 1 If $|X| > 0.5$, set $Xwork := \text{SQRT}((1 - |X|)/2)$.

Compute $Rwork = \text{arcsine}(-2 * Xwork)$ with a floating-point rational approximation, and set the result = $Rwork + \pi/2$.

Step 2 Otherwise compute the result directly using the rational approximation.

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

Notes

Note 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.

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

ATAN, DATAN

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

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

Domain: $[-\text{Inf}, \text{Inf}]$

Range: $[-\text{Pi}/2, \text{Pi}/2]$

Primary domain: $[-z, z]$, $z = 2 - \sqrt{3} = 0.2679$

Exceptions

None.

Propagated Error

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

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

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

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

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

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

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

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

Notes

Note 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.

Note 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.

Note 3 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)
```

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

Domain: $[-\text{Inf}, \text{Inf}] \times [-\text{Inf}, \text{Inf}]$

Range: $(-\text{Pi}, \text{Pi}]$

Primary domain: See note 2.

Exceptions

$(0, 0)$ and $(\pm\text{Inf}, \pm\text{Inf})$ 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

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

Step 2 Otherwise set $Rwork := \text{ATAN}(Y/X)$. Then if $Y < 0$ set $RESULT = Rwork - \text{Pi}$, otherwise set $RESULT = \text{Pi} - Rwork$.

Notes

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

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

Note 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.

COS, DCOS

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 $\pm Inf$, which generates an **undefined.NaN**.

Propagated Error

$A = -X \sin(X), \quad R = -X \tan(X)$ (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

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

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

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

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

Notes

Note 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.

Note 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.

3.3 T4 elementary function library

- Note 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.
- Note 4 The propagated error has a complex behavior. 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.
- Note 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 .
- Note 6 The output range is not exceeded. Thus the output of COS is always a valid argument for $ACOS$.

COSH, DCOSH

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** **RMSRE**

Single Length: 0.99 ulp 0.3 ulp

Double Length: 1.23 ulp 0.3 ulp

The Algorithm

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

Step 2 Otherwise, set $temp := \text{EXP}(|X|)$, and set $result = (temp + 1/temp)/2$.

Notes

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

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

Note 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.

EXP, DEXP

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

These compute: e^X

Domain: $[-\text{Inf}, \text{MaxLog}] = [-\text{Inf}, 88.03]_S, [-\text{Inf}, 709.78]_D$

Range: $[0, \text{Inf}]$ (See note 4)

Primary domain: $[-\text{Ln}2/2, \text{Ln}2/2] = [-0.3466, 0.3466]$

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:	0.51 ulp	0.21 ulp
Double Length:	0.5 ulp	0.21 ulp

The Algorithm

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

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

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

Step 4 Increase the exponent of the result by N . If N is sufficiently negative the result must be de-normalized.

Notes

Note 1 MaxLog is $\log_e(X_{MAX})$.

Note 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 routine. 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.

Note 3 For sufficiently negative arguments (below -87.34 for single-length and below -708.4 for double-length) the output is de-normalized, 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.

Note 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.

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

3.3 T4 elementary function library

POWER, DPOWER

```
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

$A = YX^Y(1 \pm \log_2(X))$, $R = Y(1 \pm \log_2(X))$ (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:

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

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

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

(b) For double precision:

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

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

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

Notes

Note 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.

Note 2 Results for special cases are as given in Table 3.5.

Note 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.

Note 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.

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	$-\text{Inf} \leq Y \leq -1$	0
Inf	$-1 < Y < 1$	undefined.NaN
otherwise	0	1
otherwise	1	X

Table 3.5 Power special cases

RAN, DRAN

```
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

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

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

Step 3 Output the floating point result and the new integer.

Notes

Note 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 initialized before the first call to **RAN** but not modified thereafter except by the function itself.

Note 2 If the integer parameter is initialized 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 initialized to some 'random' value, such as the output of a timer.

Note 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.

Note 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.

Note 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*.

Note 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.

SIN, DSIN

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

These compute: $\text{sine}(X)$ (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 $\pm \text{Inf}$, which generates an **undefined.NaN**.

Propagated Error

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

Generated Error (See note 3)

Range:	Primary Domain $[0, 2Pi]$			
	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

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

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

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

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

Notes

Note 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.

Note 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.

3.3 T4 elementary function library

Note 3 The propagated error has a complex behavior. 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.

Note 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 .

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

SINH, DSINH

```
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: $(-\text{Inf}, \text{Inf})$

Primary domain: $(-1.0, 1.0)$

Exceptions

$X < -Hmax$ gives $-\text{Inf}$, and $X > Hmax$ gives Inf .

Propagated Error

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

Generated Error

	Primary Domain		[1.0, <i>XBig</i>] (See note 2)	
	MRE	RMSRE	MAE	RMSAE
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

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

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

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

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

Notes

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

Note 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.

Note 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.

3.3 T4 elementary function library

TAN, DTAN

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

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

Domain: $[-Tmax, Tmax] = [-6434.0, 6434.0]_S$
 $= [-1.05 \cdot 10^8, 1.05 \cdot 10^8]_D$

Range: $(-\text{Inf}, \text{Inf})$

Primary domain: $[-\pi/4, \pi/4] = [-0.785, 0.785]$

Exceptions

All arguments outside the domain generate an **inexact.NaN**, except $\pm \text{Inf}$, 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

Step 1 Set $N = \text{integer part of } X/(\pi/2)$.

Step 2 Compute the remainder of X by $\pi/2$, using extended precision arithmetic.

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

Step 4 If N is odd, take the reciprocal.

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

Notes

Note 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.

Note 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.

Note 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.

Note 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.

Note 5 Since only the remainder of X by $\pi/2$ is used in step 3, the symmetry $\tan(x + \pi) = \tan(x)$ 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.

TANH, DTANH

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

These compute: $\tanh(X)$

Domain: $[-\text{Inf}, \text{Inf}]$

Range: $[-1.0, 1.0]$

Primary domain: $[-\text{Log}(3)/2, \text{Log}(3)/2] = [-0.549, 0.549]$

Exceptions

None.

Propagated Error

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

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

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

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

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

Notes

Note 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 .

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

4 Host file server library

Library: `hostio.lib`

Constants: `hostio.inc`

The host file server library contains routines that are used to communicate with the host file server plus support routines. 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 host i/o library, including error and return codes, are provided in the include file `hostio.inc`.

The result value from many of the routines in this library can take the value not less than `spr.operation.failed`, which is a server dependent failure result. It has been left open with the use of greater values because future server implementations may give more information back via this byte.

The routines are listed in groups according to function in section 4.4 and described in full in the alphabetical list in section 4.5.

4.1 Errors and the server run time library

The host i/o routines use support provided by the host file server. The server is implemented in C and uses routines in a C run time library, some of which are implementation dependent.

In particular, the host i/o routines do not check the validity of stream identifiers given in the parameter `streamid`, and the consequences of specifying an incorrect stream identifier may differ from system to system. For example, some systems may return an error tag, while some may return a text message. If you use only those stream identifiers returned by the host i/o routines that open files (`so.open`, `so.open.temp`, and `so.popen.read`), invalid identifiers are unlikely to occur. It is also possible in rare circumstances for a program to fail altogether with an invalid stream identifier 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.

4.2 Reading real numbers

Routines for reading 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 the OCCAM syntax for real numbers see the *occam 2.1 Reference Manual*.

4.3 Procedure descriptions

4.3 Procedure descriptions

When pairs of channels are passed as parameters, by convention they are normally passed in the order input channel then output channel. Hence all the routines which interact with the host server take `fs` and `ts` (*f*rom *s*erver and *t*o *s*erver) as the first two parameters. `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`. The multiplexors take the next two parameters as *from user*, *to user* which will normally correspond with *to server*, *from server*.

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 requests with those of these routines.

4.4 Host i/o procedures arranged by purpose

The host i/o routines may be divided into eight groups:

- general host access (Table 4.1);
- keyboard input (Table 4.2);
- screen output (Table 4.3);
- file access and management (Table 4.4);
- file output (Table 4.5);
- buffers (Table 4.6);
- multiplexors (Table 4.7);
- time and date processing (Table 4.8).

4.4.1 General host access

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

Procedure	Description
<code>so.system</code>	Passes a command to the host operating system.
<code>so.exit</code>	Terminates the server.
<code>so.getenv</code>	Returns host environment variables.
<code>so.commandline</code>	Returns the server command line.
<code>so.parse.command.line</code>	Returns the server command line and interprets it.
<code>so.version</code>	Returns information about the server and host.

Table 4.1 General host access procedures

4.4.2 Keyboard input

Procedure	Description
<code>so.ask</code>	Prompts on the screen for a user response on the keyboard.
<code>so.read.echo.line</code>	Reads and echoes a line from the keyboard.
<code>so.read.line</code>	Reads a line from the keyboard without echo.
<code>so.getkey</code>	Waits for and fetches the next byte from the keyboard.
<code>so.pollkey</code>	Fetches the next byte from the keyboard if there is one.
<code>so.read.echo.any.int</code>	Reads and echoes any integer from the keyboard.
<code>so.read.echo.int</code>	Reads and echoes a decimal <code>INT</code> from the keyboard.
<code>so.read.echo.int32</code>	Reads and echoes a decimal <code>INT32</code> integer from the keyboard.
<code>so.read.echo.int64</code>	Reads and echoes a decimal <code>INT64</code> integer from the keyboard.
<code>so.read.echo.hex.int</code>	Reads and echoes a hexadecimal <code>INT</code> from the keyboard.
<code>so.read.echo.hex.int32</code>	Reads and echoes a hexadecimal <code>INT32</code> integer from the keyboard.
<code>so.read.echo.hex.int64</code>	Reads and echoes a hexadecimal <code>INT64</code> integer from the keyboard.
<code>so.read.echo.real32</code>	Reads and echoes a <code>REAL32</code> from the keyboard.
<code>so.read.echo.real64</code>	Reads and echoes a <code>REAL64</code> from the keyboard.

Table 4.2 Keyboard input procedures

4.4.3 Screen output

Procedure	Description
<code>so.write.char</code>	Displays a single character on the screen.
<code>so.write.nl</code>	Displays a new line sequence on the screen.
<code>so.write.string</code>	Displays a string on the screen.
<code>so.write.string.nl</code>	Displays a string on the screen followed by a new line.
<code>so.write.int</code>	Displays an <code>INT</code> as decimal on the screen.
<code>so.write.int32</code>	Displays an <code>INT32</code> as decimal on the screen.
<code>so.write.int64</code>	Displays an <code>INT64</code> as decimal on the screen.
<code>so.write.hex.int</code>	Displays an <code>INT</code> as hexadecimal on the screen.
<code>so.write.hex.int32</code>	Displays an <code>INT32</code> as hexadecimal on the screen.
<code>so.write.hex.int64</code>	Displays an <code>INT64</code> as hexadecimal on the screen.
<code>so.write.real32</code>	Displays a <code>REAL32</code> on the screen.
<code>so.write.real64</code>	Displays a <code>REAL64</code> on the screen.

Table 4.3 Screen output procedures

4.4.4 File access and management

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

4.4 Host i/o procedures arranged by purpose

Procedure	Description
<code>so.open</code>	Opens a file.
<code>so.open.temp</code>	Opens a temporary file.
<code>so.popen.read</code>	Searches for and opens an input file.
<code>so.close</code>	Closes a file.
<code>so.test.exists</code>	Tests whether a file exists.
<code>so.read</code>	Reads a block of bytes from a file stream.
<code>so.write</code>	Writes a block of bytes to a file stream.
<code>so.gets</code>	Reads a line from a file stream.
<code>so.puts</code>	Writes a line to a file stream.
<code>so.flush</code>	Flushes an output stream.
<code>so.seek</code>	Sets the file position.
<code>so.tell</code>	Returns the current file position.
<code>so.eof</code>	Tests whether the end of the file has been reached.
<code>so.ferror</code>	Indicates whether an error has occurred in a file stream
<code>so.remove</code>	Deletes a file.
<code>so.rename</code>	Renames a file.

Table 4.4 File access and management procedures

4.4.5 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, \geq `spr.operation.failed`.

Procedure	Description
<code>so.fwrite.char</code>	Writes a single character to a file stream.
<code>so.fwrite.nl</code>	Writes a new line sequence to a file stream.
<code>so.fwrite.string</code>	Writes a string to a file stream.
<code>so.fwrite.string.nl</code>	Writes a string followed by a new line to a file stream.
<code>so.fwrite.int</code>	Writes an <code>INT</code> to a file stream as decimal ASCII.
<code>so.fwrite.int32</code>	Writes an <code>INT32</code> to a file stream as decimal ASCII.
<code>so.fwrite.int64</code>	Writes an <code>INT64</code> to a file stream as decimal ASCII.
<code>so.fwrite.hex.int</code>	Writes an <code>INT</code> to a file stream as hexadecimal ASCII.
<code>so.fwrite.hex.int32</code>	Writes an <code>INT32</code> to a file stream as hexadecimal ASCII.
<code>so.fwrite.hex.int64</code>	Writes an <code>INT64</code> to a file stream as hexadecimal ASCII.
<code>so.fwrite.real32</code>	Writes a <code>REAL32</code> to a file stream as ASCII.
<code>so.fwrite.real64</code>	Writes a <code>REAL64</code> to a file stream as ASCII.

Table 4.5 File output procedures

4.4.6 Server protocol buffers

This group of procedures is designed to assist with buffering data exchange between the program and host. Each procedure is a complete process, containing a loop which will run until explicitly stopped by sending a signal on the `stopper` channel. The procedures are normally called once in parallel with the rest of the program.

Procedure	Description
<code>so.buffer</code>	Creates a server protocol buffer process.
<code>so.overlapped.buffer</code>	Creates an overlapping server protocol buffer process.

Table 4.6 Buffering procedures

4.4.7 Server protocol multiplexors

This group of procedures is designed to assist with multiplexing data exchange between the program and host. Each procedure is a complete process, containing a loop which will run until explicitly stopped by sending a signal on the `stopper` channel. The procedures are normally called once in parallel with the rest of the program.

Procedure	Description
<code>so.multiplexor</code>	Creates a server protocol multiplexor process.
<code>so.overlapped.multiplexor</code>	Creates an overlapping server protocol multiplexor process.
<code>so.pri.multiplexor</code>	Creates a prioritized server protocol multiplexor process.
<code>so.overlapped.pri.multiplexor</code>	Creates a prioritized overlapping server protocol multiplexor process.

Table 4.7 Multiplexing procedures

4.4.8 Host date and time

Procedure	Description
<code>so.time</code>	Returns the packed local and UTC time.
<code>so.time.to.date</code>	Converts a packed time into an array of integers.
<code>so.date.to.ascii</code>	Converts an array of integers into ASCII time.
<code>so.time.to.ascii</code>	Converts a packed time into ASCII time.
<code>so.today.date</code>	Returns local time as an array of integers.
<code>so.today.ascii</code>	Returns local time as ASCII.

Table 4.8 Time and date processing procedures

4.5 Host i/o procedure definitions

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 replies by the string `replies`. Only single character responses are permitted, and alphabetic characters are *not* case sensitive. For example if the permitted responses are `x`, `n` and `q` then the `replies` string should contain the string "YNQ", and `y`, `n` and `q` would also be accepted. `reply.number` indicates which response was typed, numbered from zero. A question mark (?) is automatically displayed 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.buffer

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

This procedure buffers data between the user processes and the host, as shown in Figure 4.1. It can be used by processes on a network to pass data to the host across intervening processors.

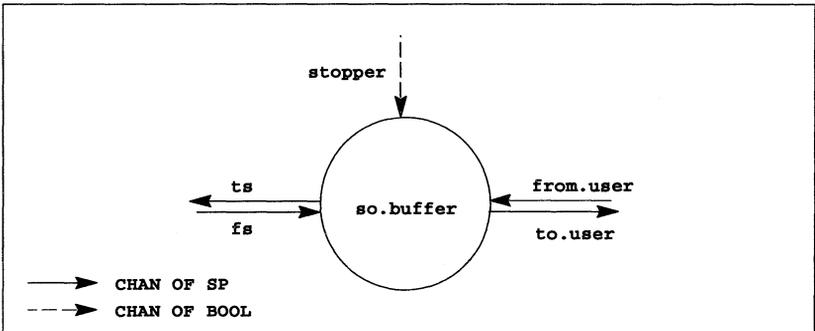


Figure 4.1 Channels of `so.buffer`

4.5 Host i/o procedure definitions

The **result** returned can take any of the following values:

spr.ok	End of file has been reached.
≥ spr.operation.failed	If result ≥ spr.operation.failed then this denotes a server returned failure. This result will also be obtained if eof has not been reached.

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.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 503 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 ≥ spr.operation.failed then this denotes a server returned failure. This result will also be obtained if no error has occurred on the specified stream.

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 ≥ spr.operation.failed` then this denotes a server returned failure.

`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.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 `0s` or `Fs` 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)
```

Writes the value `n` (of type `INT32`) 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 `0s` or `Fs` 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.

4.5 Host i/o procedure definitions

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)
```

Writes the value `n` (of type `INT64`) 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 `0s` or `Fs` 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.int

```
PROC so.fwrite.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 decimal ASCII digits, padded out with leading spaces and an optional sign to the specified field width, `width`. If the field width is too small for the number it is widened as necessary; a zero value for `width` specifies minimum width. A negative value for `width` 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, n,  
                     VAL INT width,  
                     BYTE result)
```

Writes the value `n` (of type `INT32`) to the specified stream as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified field width, `width`. If the field width is too small for the number it is widened as necessary; a zero value for `width` specifies minimum width. A negative value for `width` is an error.

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

so.fwrite.int64

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

Writes the value `n` (of type `INT64`) to the specified stream as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified field width, `width`. If the field width is too small for the number it is widened as necessary; a zero value for `width` specifies minimum width. A negative value for `width` is an error.

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

`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 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 7).

4.5 Host i/o procedure definitions

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`.

`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.

The `result` returned can take any of the following values:

<code>spr.ok</code>	The operation was successful.
<code>spr.notok</code>	Not all of the characters were written.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure

`so.fwrite.string.nl`

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

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

The `result` returned can take any of the following values:

<code>spr.ok</code>	The operation was successful.
<code>spr.notok</code>	Not all of the characters were written.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure

`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 507 bytes then it is truncated to this size.

The `result` returned can take any of the following values:

<code>spr.ok</code>	The operation was successful.
<code>spr.bad.name</code>	The specified name is a null string.
<code>spr.bad.packet.size</code>	<code>SIZE name</code> is too large (<code>> sp.max.getenvname.size</code>).
<code>spr.buffer.overflow</code>	Environment string too large for value but has been truncated to fit.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

`so.getkey`

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

Reads a single character from the keyboard. If no key is available then it waits until a key is pressed. The key is not echoed on the screen.

The `result` returned can take any of the following values:

<code>spr.ok</code>	A key has been returned in <code>key</code> .
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

`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, or `sp.max.readbuffer.size` characters have been read or the end of the file is reached. 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:

<code>spr.ok</code>	The read was successful.
<code>spr.bad.packet.size</code>	Data is too large (<code>> sp.max.readbuffer.size</code>).

4.5 Host i/o procedure definitions

- `spr.buffer.overflow` The line was larger than the buffer data and has been truncated to fit.
- `≥ spr.operation.failed` If `result ≥ spr.operation.failed` then this denotes a server returned failure.

`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, as shown in Figure 4.2.

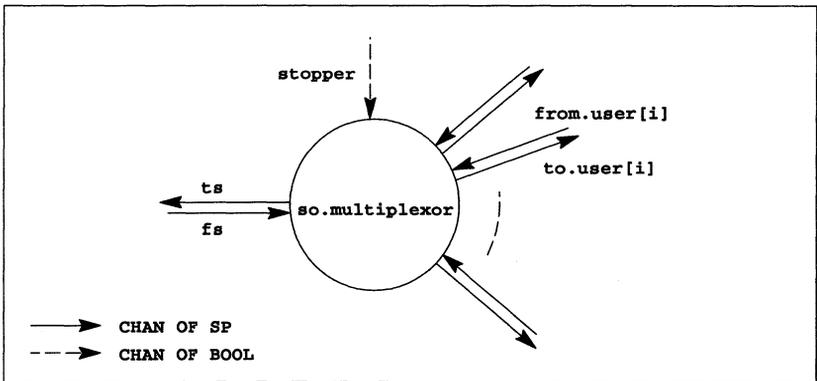


Figure 4.2 Channels of `so.multiplexor`

The channel pair `fs` and `ts` may connect to the file server or another `SP` protocol multiplexor or buffer. For n input 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.

This procedure is a complete process, containing a loop which will run until explicitly stopped by sending either a `TRUE` or `FALSE` value on the `stopper` channel. The procedure is normally called once in parallel with the rest of the program.

`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).
\geq spr.operation.failed	If result \geq spr.operation.failed then this denotes a server returned failure.

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 **temp 00**. 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 over-

4.5 Host i/o procedure definitions

written, 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:

<code>spr.ok</code>	The open was successful.
<code>spr.notok</code>	There are already 10,000 temporary files.
<code>spr.bad.type</code>	Invalid file type specified.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

`so.overlapped.buffer`

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

This procedure buffers data between the user processes and the host, as shown in Figure 4.3. It can be used by processes on a network to pass data to the host across intervening processors.

The `so.overlapped.buffer` procedure is a complete process, containing a loop which will run until explicitly stopped by sending either a `TRUE` or `FALSE` value on the `stopper` channel. The procedure is normally called once in parallel with the rest of the program.

This procedure is 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 buffer or multiplexor processes before reaching the server. It is terminated by either a `TRUE` or `FALSE` value on the channel `stopper`.

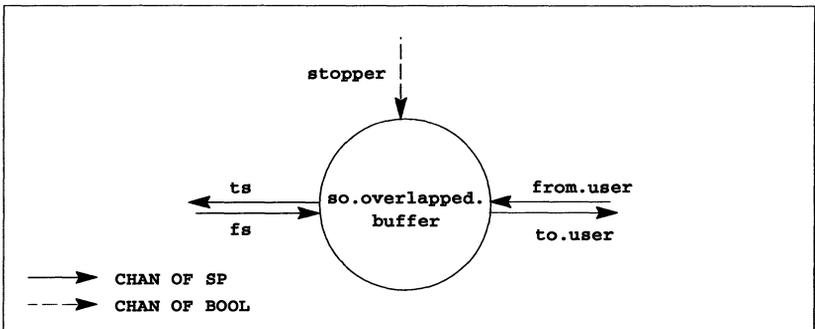


Figure 4.3 Channels of `so.overlapped.buffer`

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)

```

This procedure multiplexes any number of pairs of *SP* protocol channels onto a single pair of *SP* protocol channels, as shown in Figure 4.4. The channel pair *fs* and *ts* may connect to the file server or another *SP* protocol multiplexor or buffer. 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.

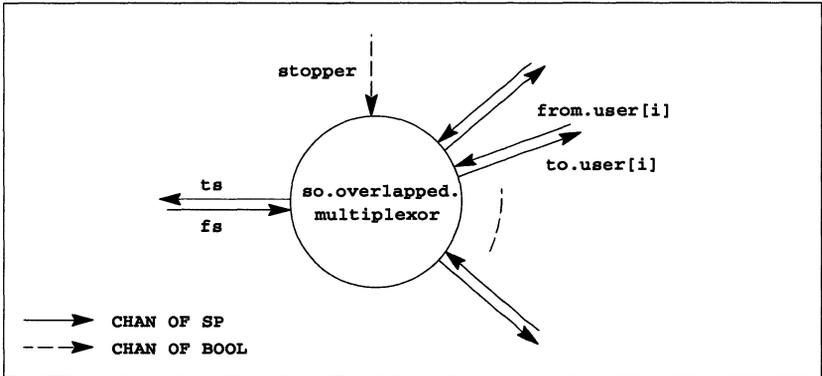


Figure 4.4 Channels of **so.overlapped.multiplexor**

so.overlapped.multiplexor is a complete process, containing a loop which will run until explicitly stopped by sending either a **TRUE** or **FALSE** value on the **stopper** channel. The procedure is normally called once in parallel with the rest of the program.

This procedure is 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**.

4.5 Host i/o procedure definitions

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)
```

This procedure multiplexes any number of pairs of *SP* protocol channels onto a single pair of *SP* protocol channels, as shown in Figure 4.5. The channel pair *fs* and *ts* may connect to the file server or another *SP* protocol multiplexor or buffer.

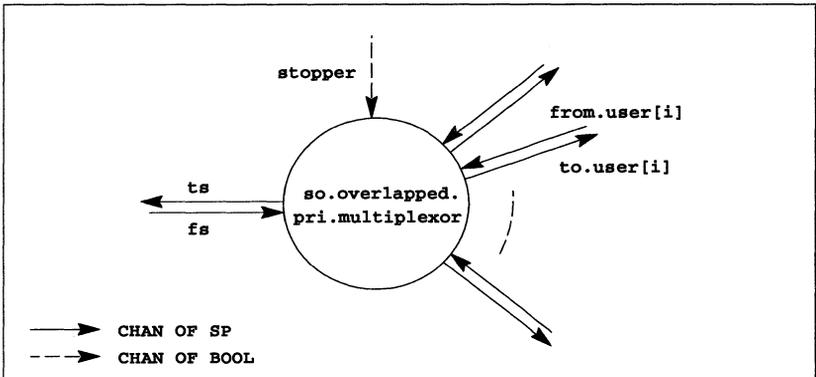


Figure 4.5 Channels of `so.overlapped.pri.multiplexor`

This procedure is a complete process, containing a loop which will run until explicitly stopped by sending either a `TRUE` or `FALSE` value on the `stopper` channel. The procedure is normally called once in parallel with the rest of the program.

The multiplexing is *not* cyclic but uses a hierarchy of priorities between the input channels. Within the array `from.user`, `from.user[i]` is of higher priority than `from.user[j]`, for $i < j$. The channel `stopper` has higher priority than any channel in the array `from.user`.

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,  
    [][]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. Because all of the strings in `option.strings` must be the same length, trailing spaces should be used to pad.

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 whether the corresponding option (in `option.strings`) requires a parameter. The values it may take are:

<code>spopt.never</code>	Never takes a parameter.
<code>spopt.maybe</code>	Optionally takes a parameter.
<code>spopt.always</code>	Must take a parameter.

Continuing the above example, if the file name must be supplied and none of the options take parameters, except for option **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 were 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. The error message **Command line error: called incorrectly** means that either:

- `option.strings` was null, or
- `SIZE option.exists`, `SIZE option.parameters` or `SIZE option.parameters.required` does not equal `SIZE option.strings`.

4.5 Host i/o 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 \geq `spr.operation.failed`. The key is not echoed on the screen.

The `result` returned can take any of the following values:

<code>spr.ok</code>	A key was available and has been returned in <code>key</code> .
\geq <code>spr.operation.failed</code>	If <code>result</code> \geq <code>spr.operation.failed</code> then this denotes a server returned failure.

`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)
```

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. 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`, as described in section 4.10.2 in the *Occam 2.1 Toolset User Guide*.

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 file type is specified by `type` and the mode of opening by `mode`. The mode of opening is always `sprm.input`. `type` can take the following values:

<code>spr.binary</code>	File contains raw bytes only.
<code>spr.text</code>	File contains text records separated by newline sequences.

The `result` returned can take any of the following values:

<code>spr.ok</code>	The open was successful.
<code>spr.bad.name</code>	Null name supplied.

<code>spr.bad.type</code>	Invalid file type specified.
<code>spr.bad.packet.size</code>	File name is too large (i.e. > <code>sp.max.open-name.size</code>) or <code>path.variable.name</code> is too large (i.e. > <code>sp.max.getenvname.size</code>).
<code>spr.buffer.overflow</code>	The environment string referenced by <code>path.variable.name</code> is longer than 507 characters.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

`so.pri.multiplexor`

```
PROC so.pri.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, as shown in Figure 4.6.

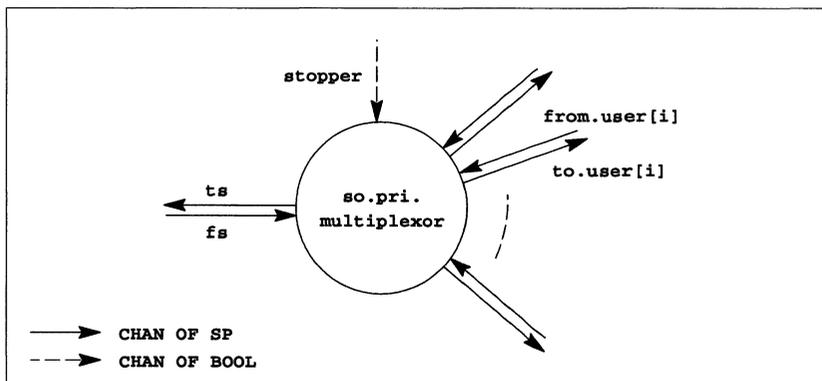


Figure 4.6 Channels of `so.pri.multiplexor`

The channel pair `fs` and `ts` may connect to the file server or another `SP` protocol multiplexor or buffer. 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 procedure is a complete process, containing a loop which will run until explicitly stopped by sending either a `TRUE` or `FALSE` value on the channel `stopper`. The procedure is normally called once in parallel with the rest of the program.

The multiplexing is *not* cyclic but uses a hierarchy of priorities between the input channels. Within the array `from.user`, `from.user[i]` is of higher priority than `from.user[j]`, for $i < j$. The channel `stopper` is of lower priority than any channel in the array `from.user`.

4.5 Host i/o procedure definitions

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 host i/o 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 ≥ spr.operation.failed then this  
denotes a server returned failure.
```

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.

Note: `so.read` reads in multiples of the packet size defined by `sp.max.readbuffer.size`. For greatest efficiency, read requests should be made in multiples of this size.

so.read.echo.any.int

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

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

`so.read.echo.any.int` accepts numbers in either decimal or hexadecimal format. Hexadecimal numbers may be lower or upper case but must be prefixed with either `#` or `$` to indicate a hexadecimal number, or `%`, which means add `MOSTNEG INT` to the given hexadecimal, using modulo arithmetic. For example, on a 32-bit transputer `%70` is interpreted as `#8000070`, 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.int

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

Reads a hexadecimal integer typed at the keyboard, displays it on the screen and interprets it as an **INT**. The number must be terminated by a return character. The boolean **error** is set to **TRUE** if an invalid integer is typed, **FALSE** otherwise.

The hexadecimal digits may be in lower or upper case but must be prefixed with either **#**, or **\$** to indicate a hexadecimal number, or **%**, which means add **MOSTNEG INT** to the given hexadecimal, 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.

Reads a hexadecimal integer typed at the keyboard, displays it on the screen and interprets it as an **INT32**. The number must be terminated by a return character. The boolean **error** is set to **TRUE** if an invalid integer is typed, **FALSE** otherwise.

The hexadecimal digits may be in lower or upper case but must be prefixed with either **#**, or **\$** to indicate a hexadecimal number, or **%**, which means add **MOSTNEG INT** to the given hexadecimal, 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.int64

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

Reads a hexadecimal integer typed at the keyboard, displays it on the screen and interprets it as an **INT64**. The number must be terminated by a return character. The boolean **error** is set to **TRUE** if an invalid integer is typed, **FALSE** otherwise.

4.5 Host i/o procedure definitions

The hexadecimal digits may be in lower or upper case but must be prefixed with either #, or \$ to indicate a hexadecimal number, or %, which means add **MOSTNEG INT** to the given hexadecimal, using modulo arithmetic. For example, on a 32-bit transputer %70 is interpreted as #8000070, 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.int

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

Reads a decimal integer typed at the keyboard, displays it on the screen and interprets it as an **INT**. The number must be terminated by a return character. 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)
```

Reads a decimal integer typed at the keyboard, displays it on the screen and interprets it as an **INT32**. The number must be terminated by a return character. The boolean **error** is set to **TRUE** if an invalid integer is typed, **FALSE** otherwise.

so.read.echo.int64

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

Reads a decimal integer typed at the keyboard, displays it on the screen and interprets it as an **INT64**. The number must be terminated by a return character. The boolean **error** is set to **TRUE** if an invalid integer is typed, **FALSE** otherwise.

so.read.echo.line

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

Reads a line of text from the keyboard, echoing it on the screen. The characters read are placed in the first **len** bytes of **line**. The line is read until the Return key is

4.5 Host i/o procedure definitions

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 ≥ spr.operation.failed then this denotes a server returned failure.

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 oldname + SIZE newname) > sp.max.renamename.size).
≥ spr.operation.failed	If result ≥ spr.operation.failed then this denotes a server returned failure.

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:

<code>spr.start</code>	The start of the file.
<code>spr.current</code>	The current position in the file.
<code>spr.end</code>	The end of the file.

The `result` returned can take any of the following values:

<code>spr.ok</code>	The operation was successful.
<code>spr.bad.origin</code>	Invalid origin.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

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:

<code>spr.ok</code>	Host command processor exists.
<code>spr.bad.packet.size</code>	The array command is too large (<code>> sp.max.systemcommand.size</code>).
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

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:

<code>spr.ok</code>	The operation was successful.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

so.test.exists

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

4.5 Host i/o procedure definitions

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

`so.time`

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

Returns the local host 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 `INT32s`.

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

`so.time.to.date`

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

Converts host time (as supplied by `so.time`) to six integers, stored in the `date` array. The elements of the array are given in Table 4.9.

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)

Table 4.9 Elements of date array

`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.

`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 array `date`. The format of the array is given in Table 4.9. If the date is unavailable all elements in `date` are set to zero.

`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. This function is obsolete and has not been updated with new host, operating system and board types.

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:

<code>sph.unknown</code>	unknown host type
<code>sph.PC</code>	IBM PC
<code>sph.S370</code>	IBM 370 Architecture
<code>sph.NECPC</code>	NEC PC
<code>sph.VAX</code>	DEC VAX
<code>sph.SUN3</code>	Sun Microsystems Sun 3
<code>sph.BOX.SUN4</code>	Sun Microsystems Sun 4
<code>sph.BOX.SUN386</code>	Sun Microsystems Sun 386i
<code>sph.BOX.APOLLO</code>	Apollo
<code>sph.BOX.ATARI</code>	Atari ST or TT

Values up to 127 are reserved for use by SGS-THOMSON Microelectronics Limited.

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

<code>spo.unknown</code>	unknown OS type
<code>spo.DOS</code>	DOS

4.5 Host i/o procedure definitions

<code>spo.HELIOS</code>	HELIOS
<code>spo.VMS</code>	VMS
<code>spo.SUNOS</code>	SunOS
<code>spo.CMS</code>	CMS
<code>spo.TOS</code>	TOS

Values up to 127 are reserved for use by SGS-THOMSON Microelectronics Limited.

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

<code>spb.unknown</code>	unknown board type
<code>spb.B004</code>	IMS B004
<code>spb.B008</code>	IMS B008
<code>spb.B010</code>	IMS B010
<code>spb.B011</code>	IMS B011
<code>spb.B014</code>	IMS B014
<code>spb.B015</code>	IMS B015
<code>spb.B016</code>	IMS B016
<code>spb.DRX11</code>	DRX-11
<code>spb.IBMCAT</code>	CAT
<code>spb.QT0</code>	Caplin QT0
<code>spb.UDPLINK</code>	UDP link
<code>spb.TCPLINK</code>	TCP link
<code>spb.ACSILA</code>	ACSILA

Values up to 127 are reserved for use by SGS-THOMSON Microelectronics Limited.

`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.

Note: `so.write` writes in multiples of the packet size defined by `sp.max.writebuffer.size`. For greatest efficiency, write requests should be made in multiples of this size.

`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.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 `0s` or `Fs` 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)
```

Writes the value `n` (of type `INT32`) 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 `0s` or `Fs` 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.int64

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

Writes the value `n` (of type `INT64`) 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 `0s` or `Fs` 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.int

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

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, `width`. If the

4.5 Host i/o procedure definitions

field width is too small for the number it is widened as necessary; a zero value for `width` specifies minimum width. A negative value for `width` is an error.

`so.write.int32`

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

Writes the value `n` (of type `INT32`) to the screen as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified field width, `width`. If the field width is too small for the number it is widened as necessary; a zero value for `width` specifies minimum width. A negative value for `width` is an error.

`so.write.int64`

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

Writes the value `n` (of type `INT64`) to the screen as decimal ASCII digits, padded out with leading spaces and an optional sign to the specified field width, `width`. If the field width is too small for the number it is widened as necessary; a zero value for `width` specifies minimum width. A negative value for `width` is an error.

`so.write.nl`

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

Writes a new line sequence to the screen.

`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 7).

Note: Since the internal buffers are of fixed size, 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)
```

Writes the value *r* (of type **REAL64**) to the screen as ASCII characters formatted using *Ip* and *Dp* as described under **REAL32TOSTRING** in Chapter 7.

Note: Since the internal buffers are of fixed size, 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**.

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)
```

Writes the string *string* to the screen, followed by a newline sequence.

5 Stream i/o library

Library: `streamio.lib`

Constants: `streamio.inc`

The stream i/o library contains routines for reading and writing to files and to the terminal at a higher level of abstraction than the host i/o library. The file `streamio.inc` defines the `KS` and `SS` protocols and constants used by the stream i/o library routines.

The routines are listed in groups according to function in section 5.2 and described in full in the alphabetical list in section 5.3.

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 `KS`-to-`SS` conversion routine, which actually uses both protocols, is prefixed for convenience with `ks`.

5.2 Stream i/o procedures

The stream i/o routines may be divided into three groups:

- stream processes (Table 5.1);
- key stream input (Table 5.2);
- screen stream output (Table 5.3).

Stream input and output procedures are used to input and output characters in key stream `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)
```

5.2 Stream i/o procedures

The result value from many of the routines in this library can take a value equal to or greater than `spr.operation.failed`, which is a server dependent failure result. Future server implementations may give more failure information back via this byte. Names for result values can be found in the file `hostio.inc`.

5.2.1 Stream processes

This group of procedures is designed to assist with handling key and screen streams. Each procedure is a complete process, containing a loop which will run until explicitly stopped. The procedures are normally called once in parallel with the rest of the program.

Procedure	Description
<code>so.keystream.from.kbd</code>	Creates a process to read keys and convert to a key stream.
<code>so.keystream.from.file</code>	Creates a process to read a file and convert to a key stream.
<code>so.keystream.from.stdin</code>	Creates a process to read from standard in and convert to a key stream.
<code>ks.keystream.sink</code>	Creates a process to soak up a key stream.
<code>ks.keystream.to.scrstream</code>	Creates a process to convert a key stream to a screen stream.
<code>ss.scrstream.sink</code>	Creates a process to soak up a screen stream.
<code>so.scrstream.to.file</code>	Creates a process to send a screen stream to a file.
<code>so.scrstream.to.stdout</code>	Creates a process to send a screen stream to standard out.
<code>ss.scrstream.to.array</code>	Creates a process to send a screen stream to an array.
<code>ss.scrstream.from.array</code>	Creates a process to read a screen stream from an array.
<code>ss.scrstream.fan.out</code>	Creates a process to copy screen stream input to two output channels.
<code>ss.scrstream.copy</code>	Creates a process to copy screen stream input to one output channel.
<code>so.scrstream.to.ANSI</code>	Creates a process to convert a screen stream to ANSI format.
<code>so.scrstream.to.TVI920</code>	Creates a process to convert a screen stream to TVI920 format.
<code>ss.scrstream.multiplexor</code>	Creates a process to merge multiple screen streams.

Table 5.1 Stream process procedures

5.2.2 Key stream input

Procedure	Description
<code>ks.read.char</code>	Returns the next byte from a key stream.
<code>ks.read.line</code>	Returns the next line from a key stream.
<code>ks.read.int</code>	Returns the next <code>INT</code> from a key stream.
<code>ks.read.int64</code>	Returns the next <code>INT64</code> from a key stream.
<code>ks.read.real32</code>	Returns the next <code>REAL32</code> from a key stream.
<code>ks.read.real64</code>	Returns the next <code>REAL64</code> from a key stream.

Table 5.2 Key stream input procedures

These procedures, as listed in Table 5.2, read characters and strings from the input stream, in **KS** protocol.

5.2.3 Stream output

These routines write text, numbers and screen control codes to an output stream in **SS** protocol.

Procedure	Description
ss.write.char	Write a character to a screen stream.
ss.write.nl	Write a newline to screen stream.
ss.write.string	Write a string to a screen stream.
ss.write.endstream	Write an end of stream character to a screen stream.
ss.write.text.line	Write a string to a screen stream followed by return and next line.
ss.write.int	Write an INT to a screen stream as decimal ASCII characters.
ss.write.int64	Write an INT64 to a screen stream as decimal ASCII characters.
ss.write.hex.int	Write an INT to a screen stream as hexadecimal characters.
ss.write.hex.int64	Write an INT64 to a screen stream as hexadecimal characters.
ss.write.real32	Write an REAL32 to a screen stream as ASCII characters.
ss.write.real64	Write an REAL64 to a screen stream as ASCII characters.
ss.goto.xy	Send control signals to a screen stream to move the cursor.
ss.clear.eol	Send control signals to a screen stream to clear to the end of the line.
ss.clear.eos	Send control signals to a screen stream to clear to the end of the screen.
ss.beep	Send a bell character to a screen stream.
ss.up	Send a cursor up character to a screen stream.
ss.down	Send a cursor down character to a screen stream.
ss.left	Send a cursor left character to a screen stream.
ss.right	Send a cursor right character to a screen stream.
ss.insert.char	Send control signals to a screen stream to insert a character.
ss.ins.line	Send control signals to a screen stream to insert a line.
ss.delete.chr	Send control signals to a screen stream to delete the current cursor character.
ss.delete.chl	Send control signals to a screen stream to delete the character to the left of the cursor.
ss.del.line	Send control signals to a screen stream to delete a line.

Table 5.3 Screen stream output procedures

5.3 Stream i/o procedure definitions

ks.keystream.sink

PROC ks.keystream.sink (CHAN OF KS keys)

Reads word length quantities from the channel **keys** until **ft.terminated** is received, then terminates.

This procedure is a complete process, containing a loop which will run until explicitly stopped by sending **ft.terminated** on the channel **keys**. The procedure is normally called once in parallel with the rest of the program.

ks.keystream.to.scrstream

**PROC ks.keystream.to.scrstream (CHAN OF KS keyboard,
CHAN OF SS scrn)**

Converts the key stream protocol received on the input channel **keyboard** to the screen stream protocol sent on the output channel **scrn**. The procedure terminates when it receives the value **ft.terminated** on **keyboard**.

This procedure is a complete process, containing a loop which will run until explicitly stopped by sending **ft.terminated** on the channel **keyboard**. The procedure is normally called once in parallel with the rest of the program.

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.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 initialized 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)**

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 **INT64** in **number**. **char** must be

initialized 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.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.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 `REAL32` value in `number`. `char` must be initialized to the first character of the input. If there is an error in the syntax of the real, if it is \pm 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)
```

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 `REAL64` value in `number`. `char` must be initialized to the first character of the input. If there is an error in the syntax of the real, if it is \pm infinity, or if more than 30 characters read then `char` is returned as `ft.number.error`.

`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

5.3 Stream i/o procedure definitions

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:

This procedure is a complete process, containing a loop which will run until it an error occurs or it reaches the end of the file. The procedure is normally called once in parallel with the rest of the program.

<code>spr.ok</code>	The operation was successful.
<code>spr.bad.packet.size</code>	Filename too large i.e. <code>SIZE filename > spr.max.openname.size</code> .
<code>spr.bad.name</code>	Null file name.
<code>≥spr.operation.failed</code>	The open failed or reading the file failed. If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

`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.

This procedure is a complete process, containing a loop which will run until explicitly stopped by sending `FALSE` on the channel `stopper`. When `FALSE` is received on the channel `stopper` the procedure sends the value `ft.terminated` on `keys.out`. The procedure is normally called once in parallel with the rest of the program.

`so.keystream.from.stdin`

```
PROC so.keystream.from.stdin (CHAN OF SP fs, ts,  
                             CHAN OF KS keys.out,  
                             BYTE result)
```

Reads lines from the standard input stream and outputs them on `keys.out`. The standard input stream is normally assigned to the keyboard, but can be redirected by the host operating system. An end of file character from the keyboard will terminate this routine. '*'c is output to terminate each text line.

This procedure is a complete process, containing a loop which will run until explicitly stopped by sending an end of file character. When this character is received the procedure sends the value `ft.terminated` on `keys.out`. The procedure is normally called once in parallel with the rest of the program.

The result returned may take any of the following values:

<code>spr.ok</code>	The operation was successful.
<code>≥ spr.operation.failed</code>	Reading standard input failed. If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure.

`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.claim
st.key.raw st.key.cooked st.release st.initialise
```

The procedure terminates on receipt of the stream terminator from `ss.write.endstream`.

`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. 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:

<code>spr.ok</code>	The data sent on <code>scrn</code> was successfully written to the file.
<code>spr.bad.packet.size</code>	Filename too large i.e. <code>SIZE filename > sp.max.openname.size</code> .
<code>spr.bad.name</code>	Null file name.
<code>≥ spr.operation.failed</code>	If <code>result ≥ spr.operation.failed</code> then this denotes a server returned failure

5.3 Stream i/o procedure definitions

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 ≥ spr.operation.failed` then this denotes a server returned failure.

`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.beep`

```
PROC ss.beep (CHAN OF SS scrn)
```

Sends a bell code to the terminal.

`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.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.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.

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.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.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.

5.3 Stream i/o procedure definitions

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.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.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 end-stream terminator from `ss.write.endstream`, which is not passed on.

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, as shown in Figure 5.1. The procedure terminates on receipt of the stream terminator from `ss.write.endstream` without passing on the terminator.

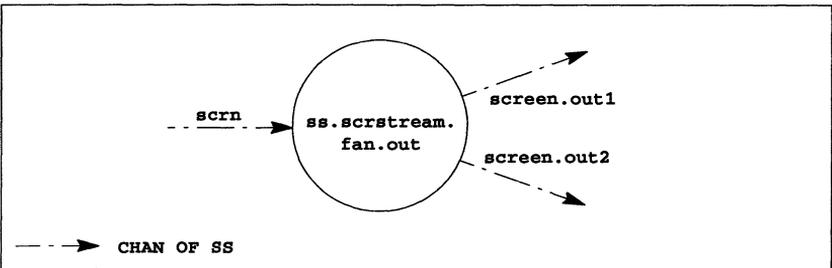


Figure 5.1 Channels of `ss.scrstream.fan.out`

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.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, as shown in Figure 5.2. 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.

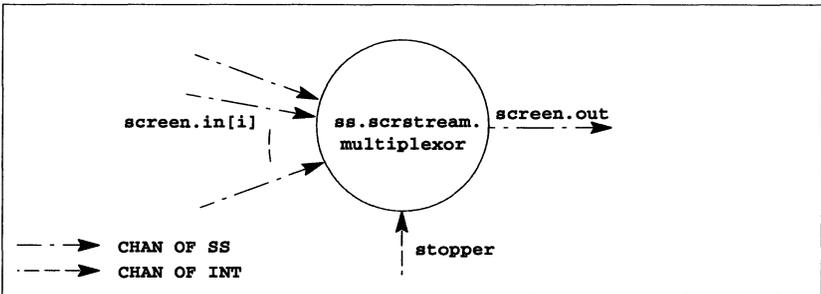


Figure 5.2 Channels of `ss.scrstream.multiplexor`

This procedure is a complete process, containing a loop which will run until explicitly stopped by sending an `INT` value on the channel `stopper`. The procedure is normally called once in parallel with the rest of the program.

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.

5.3 Stream i/o procedure definitions

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.up

```
PROC ss.up (CHAN OF SS scrn)
```

Sends a command to the terminal to move the cursor one line up the screen.

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.endstream

```
PROC ss.write.endstream (CHAN OF SS scrn)
```

Sends a special stream terminator value to *scrn*.

ss.write.hex.int

```
PROC ss.write.hex.int (CHAN OF SS scrn,  
                      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 sent to *scrn*. A negative value for *width* is an error.

ss.write.hex.int64

```
PROC ss.write.hex.int64 (CHAN OF SS scrn,  
                        VAL INT64 number,  
                        VAL INT width)
```

As *ss.write.hex.int* but for 64-bit integer values.

ss.write.int

```
PROC ss.write.int (CHAN OF SS scrn,  
                  VAL INT number, width)
```

Converts *number* into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified field width, *width*, if necessary. If the number cannot be represented in *width* characters it is widened as necessary; a zero value for *width* will give minimum width. The converted number is sent to *scrn*. A negative value for *width* is an error.

ss.write.int64

```
PROC ss.write.int64 (CHAN OF SS scrn,  
                    VAL INT64 number,  
                    VAL INT width)
```

As *ss.write.int* but for 64-bit integers.

ss.write.nl

```
PROC ss.write.nl (CHAN OF SS scrn)
```

Sends **c*n* to *scrn*.

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* in Chapter 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)
```

Converts *number* into an ASCII string formatted using *Ip* and *Dp*, as described for *REAL32TOSTRING* in Chapter 7. The converted number is sent to *scrn*. If the formatted form of *number* is larger than 30 characters then this procedure acts as an invalid process.

5.3 Stream i/o procedure definitions

ss.write.string

```
PROC ss.write.string (CHAN OF SS scrn,  
                     VAL [ ]BYTE str)
```

Sends all characters in `str` 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`".

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 routines are listed in groups according to function in section 6.1 and described in full in the alphabetical list in section 6.2.

6.1 String handling procedures

The library provides routines for:

- identifying characters;
- comparing strings;
- editing strings;
- searching strings;
- scanning lines of text.

6.1.1 Character identification

Result	Function	Parameter specifiers
BOOL	<code>is.in.range</code>	VAL BYTE char, bottom, top
BOOL	<code>is.upper</code>	VAL BYTE char
BOOL	<code>is.lower</code>	VAL BYTE char
BOOL	<code>is.digit</code>	VAL BYTE char
BOOL	<code>is.hex.digit</code>	VAL BYTE char
BOOL	<code>is.id.char</code>	VAL BYTE char

Table 6.1 Character identification functions

6.1.2 String comparison

These two functions allow strings to be compared for order or for equality.

Result	Function	Parameter specifiers
INT	<code>compare.strings</code>	VAL []BYTE str1, str2
BOOL	<code>eqstr</code>	VAL []BYTE str1, str2

Table 6.2 String comparison functions

6.1 String handling procedures

6.1.3 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.

Procedure	Parameter specifiers
<code>str.shift</code>	<code>[]BYTE str, VAL INT start, len, shift, BOOL not.done</code>
<code>delete.string</code>	<code>INT len, []BYTE str, VAL INT start, size, BOOL not.done</code>
<code>insert.string</code>	<code>VAL []BYTE new.str, INT len, []BYTE str, VAL INT start, BOOL not.done</code>
<code>to.upper.case</code>	<code>[]BYTE str</code>
<code>to.lower.case</code>	<code>[]BYTE str</code>
<code>append.char</code>	<code>INT len, []BYTE str, VAL BYTE char</code>
<code>append.text</code>	<code>INT len, []BYTE str, VAL []BYTE text</code>
<code>append.int</code>	<code>INT len, []BYTE str, VAL INT number, width</code>
<code>append.int64</code>	<code>INT len, []BYTE str, VAL INT64 number, VAL INT width</code>
<code>append.hex.int</code>	<code>INT len, []BYTE str, VAL INT number, width</code>
<code>append.hex.int64</code>	<code>INT len, []BYTE str, VAL INT64 number, VAL INT width</code>
<code>append.real32</code>	<code>INT len, []BYTE str, VAL REAL32 number, VAL INT Ip, Dp</code>
<code>append.real64</code>	<code>INT len, []BYTE str, VAL REAL64 number, VAL INT Ip, Dp</code>

Table 6.3 String editing procedures

6.1.4 String searching

These functions 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.

Result	Function	Parameter specifiers
INT	<code>string.pos</code>	<code>VAL []BYTE search, str</code>
INT	<code>char.pos</code>	<code>VAL BYTE search, VAL []BYTE str</code>
INT, BYTE	<code>search.match</code>	<code>VAL []BYTE possibles, str</code>
INT, BYTE	<code>search.no.match</code>	<code>VAL []BYTE possibles, str</code>

Table 6.4 String searching functions

6.1.5 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`, as described in section 6.2. The user should initialize the variable `ok` as appropriate.

Procedure	Parameter specifiers
<code>next.word.from.line</code>	<code>VAL []BYTE line, INT ptr, len, []BYTE word, BOOL ok</code>
<code>next.int.from.line</code>	<code>VAL []BYTE line, INT ptr, number, BOOL ok</code>

Table 6.5 Line parsing procedures

6.2 String handling procedure definitions

`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.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 set 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 then `len` is incremented by the number of characters added to the string. 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)
```

Converts `number` into a sequence of ASCII hexadecimal digits, using upper case letters, preceded by '#'. The total number of characters set is always `width+1`,

6.2 String handling procedure definitions

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 then `len` is incremented by the number of characters added to the string. Behaves like `STOP` if the array overflows or if `width < 0`.

`append.int`

```
PROC append.int (INT len, [BYTE str,  
                VAL INT number, width)
```

Converts `number` into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified field width, `width`, if necessary. If the number cannot be represented in `width` characters it is widened as necessary. A zero value for `width` will give minimum width. The converted number is written into the array `str` starting at `str[len]` and then `len` is incremented by the number of characters added to the string. This procedure behaves like `STOP` if the array overflows or if `width` is negative.

`append.int64`

```
PROC append.int64 (INT len, [BYTE str,  
                            VAL INT64 number,  
                            VAL INT width)
```

Converts `number` into a sequence of ASCII decimal digits padded out with leading spaces and an optional sign to the specified field width, `width`, if necessary. If the number cannot be represented in `width` characters it is widened as necessary. A zero value for `width` will give minimum width. The converted number is written into the array `str` starting at `str[len]` and then `len` is incremented by the number of characters added to the string. This procedure behaves like `STOP` if the array overflows or if `width` is negative.

`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` in Chapter 7.

The converted number is written into the array `str` starting at `str[len]` and `len` is incremented by the number of characters added. This procedure behaves like `STOP` if the array overflows.

append.real64

```
PROC append.real64 (INT len, []BYTE str,
                   VAL REAL64 number,
                   VAL INT Ip, Dp)
```

Converts `number` into a sequence of ASCII characters formatted using `Ip` and `Dp` as described under `REAL32TOSTRING` in Chapter 7.

The converted number is written into the array `str` starting at `str[len]` and then `len` is incremented by the number of characters added to the string. This procedure 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 then `len` is incremented by the number of characters added to the string. This procedure behaves like `STOP` if the array overflows.

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.

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, or -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`

6.2 String handling procedure definitions

So if `s` is "abcd" then:

<code>compare.strings ("abc", [s FROM 0 FOR 3])</code>	returns the value 0
<code>compare.strings ("abc", [s FROM 0 FOR 2])</code>	returns the value 1
<code>compare.strings ("abc", s)</code>	returns the value -1
<code>compare.strings ("bc", s)</code>	returns the value 2
<code>compare.strings ("a4", s)</code>	returns the value -2

`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 to `TRUE`.

`eqstr`

```
BOOL FUNCTION eqstr (VAL [BYTE s1,s2)
```

This is an optimized test for string equality. It returns `TRUE` if the two strings are the same size and have the same contents, `FALSE` otherwise.

`insert.string`

```
PROC insert.string (VAL [BYTE new.str, INT len,
                    [BYTE str, VAL INT start,
                    BOOL not.done)
```

Creates a gap in `str` starting at `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.

`is.digit`

```
BOOL FUNCTION is.digit (VAL BYTE char)
```

Returns `TRUE` if `char` is an ASCII decimal digit, otherwise returns `FALSE`.

is.hex.digit

BOOL FUNCTION is.hex.digit (VAL BYTE char)

Returns **TRUE** if **char** is an ASCII hexadecimal digit, otherwise returns **FALSE**. Upper or lower case letters A to 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; otherwise returns **FALSE**.

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, otherwise returns **FALSE**.

is.lower

BOOL FUNCTION is.lower (VAL BYTE char)

Returns **TRUE** if **char** is an ASCII lower case letter, otherwise returns **FALSE**.

is.upper

BOOL FUNCTION is.upper (VAL BYTE char)

Returns **TRUE** if **char** is an ASCII upper case letter, otherwise returns **FALSE**.

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

6.2 String handling procedure definitions

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.

`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.

`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.

`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 and 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 and 255 (BYTE) are returned.

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 to **TRUE**. Negative values of **shift** cause leftward moves.

to.lower.case

```
PROC to.lower.case ([]BYTE str)
```

Converts all alphabetic characters in **str** to lower case. All other characters are left unaltered.

to.upper.case

```
PROC to.upper.case ([]BYTE str)
```

Converts all alphabetic characters in **str** to upper case. All other characters are left unaltered.

7 String conversion library

Library: `convert.lib`

This library contains procedures for converting numeric values to strings and vice versa, as listed in Table 7.1. 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.

These routines are also described in appendix N of the *occam 2.1 Reference Manual*.

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

Table 7.1 String conversion procedures

7.1 Procedure definitions

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.

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 is returned in **len**. The hexadecimal value generated represents the entire word length of **n**. 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.

See also: **HEX16TOSTRING**, **HEX32TOSTRING**, **HEX64TOSTRING**

HEX16TOSTRING

```
PROC HEX16TOSTRING (INT len, []BYTE string,  
                   VAL INT16 n)
```

The procedure returns the hexadecimal representation of **n** in **string**, and the number of characters in the representation is returned in **len**. The hexadecimal value generated represents the entire 16 bits of **n**. The number of characters will be four. A **#** is not output by the **HEX16TOSTRING** procedure. If **string** is not long enough to hold the representation then this routine acts as an invalid process.

See also: **HEXTOSTRING**, **HEX32TOSTRING**, **HEX64TOSTRING**

HEX32TOSTRING

```
PROC HEX32TOSTRING (INT len, []BYTE string,  
                   VAL INT32 n)
```

The procedure returns the hexadecimal representation of **n** in **string**, and the number of characters in the representation is returned in **len**. The hexadecimal

value generated represents the entire 32 bits of *n*. The number of characters will be eight. A # is not output by the `HEX32TOSTRING` procedure. If *string* is not long enough to hold the representation then this routine acts as an invalid process.

See also: `HEXTOSTRING`, `HEX16TOSTRING`, `HEX64TOSTRING`

`HEX64TOSTRING`

```
PROC HEX64TOSTRING (INT len, [BYTE string,  
                    VAL INT64 n)
```

The procedure returns the hexadecimal representation of *n* in *string*, and the number of characters in the representation is returned in *len*. The hexadecimal value generated represents the entire 64 bits of *n*. The number of characters will be sixteen. A # is not output by the `HEX64TOSTRING` procedure. If *string* is not long enough to hold the representation then this routine acts as an invalid process.

See also: `HEXTOSTRING`, `HEX16TOSTRING`, `HEX32TOSTRING`

`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 is returned in *len*. If *string* is not long enough to hold the representation then this routine acts as an invalid process.

See also: `INT16TOSTRING`, `INT32TOSTRING`, `INT64TOSTRING`

`INT16TOSTRING`

```
PROC INT16TOSTRING (INT len, [BYTE string,  
                    VAL INT16 n)
```

Converts an `INT16` value to a string. The procedure returns the decimal representation of *n* in *string* and the number of characters in the representation is returned in *len*. If *string* is not long enough to hold the representation then this routine acts as an invalid process.

See also: `INTTOSTRING`, `INT32TOSTRING`, `INT64TOSTRING`

`INT32TOSTRING`

```
PROC INT32TOSTRING (INT len, [BYTE string,  
                    VAL INT32 n)
```

Converts an `INT32` value to a string. The procedure returns the decimal representation of *n* in *string* and the number of characters in the representation is returned

7.1 Procedure definitions

in `len`. If `string` is not long enough to hold the representation then this routine acts as an invalid process.

See also: `INTTOSTRING`, `INT16TOSTRING`, `INT64TOSTRING`

`INT64TOSTRING`

```
PROC INT64TOSTRING (INT len, [BYTE string,  
                    VAL INT64 n)
```

Converts an `INT64` value to a string. The procedure returns the decimal representation of `n` in `string` and the number of characters in the representation is returned in `len`. If `string` is not long enough to hold the representation then this routine acts as an invalid process.

See also: `INTTOSTRING`, `INT16TOSTRING`, `INT32TOSTRING`

`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`).

The string must match the format for OCCAM real literals, for example, `1.2E+2`, as defined in the *OCCAM 2.1 Reference Manual* on p 146.

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 I_p and D_p 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 $I_p=0$, $D_p=0$, 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 $I_p>0$, $D_p>0$, fixed format is used, unless the value needs more than I_p 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 $I_p + D_p + 2$ when $I_p>0$, $D_p>0$.

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 $I_p=0$, $D_p>0$, then exponential format is always used. The length of the result is $D_p + 6$ or $D_p + 7$, depending on whether x is a **REAL32** or **REAL64**, respectively.

If $I_p=0$, $D_p=1$, 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 I_p and D_p 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.

7.1 Procedure definitions

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'.

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.

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.

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

7.1 Procedure definitions

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.

8 Block CRC library

Library: `crc.lib`

The block CRC library provides two functions for calculating cyclic redundancy check values from byte strings. Such values can be of use in, for example, the generation of the frame check sequence (FCS) in data communications. These functions are in addition to the compiler library functions `CRCWORD` and `CRCBYTE`, described in Chapter 2.

A cyclic redundancy check value is the remainder from modulo 2 polynomial division. Consider bit sequences as representing the coefficients of polynomials; for example, the bit sequence 10100100 (where the leading bit is the most significant bit (msb)) corresponds to $P(x) = x^7 + x^5 + x^2$. The routines in the library calculate the remainder of the modulo 2 polynomial division:

$$\frac{x^{k+n} H(x) + x^n F(x)}{G(x)}$$

where: $F(x)$ corresponds to `InputString`

$G(x)$ corresponds to `PolynomialGenerator`

$H(x)$ corresponds to `OldCRC`

k is the number of bits in `InputString`

n is the word size in bits of the processor used (i.e. n is 16 or 32).

(`OldCRC` can be viewed as the value that would be pre-loaded into the cyclic shift register that is part of hardware implementations of CRC generators.)

When representing $G(x)$ in the word `PolynomialGenerator`, note that there is an understood bit before the msb of `PolynomialGenerator`. For example, on a 16-bit processor, with $G(x) = x^{16} + x^{12} + x^5 + 1$, which is #11021, then `PolynomialGenerator` must be assigned #1021, because the bit corresponding to x^{16} is understood. Thus, a value of #9603 for `PolynomialGenerator`, corresponds to $G(x) = x^{16} + x^{15} + x^{12} + x^{10} + x^9 + x + 1$, for a 16-bit processor.

A similar situation holds on a 32-bit processor, so that:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

is encoded in `PolynomialGenerator` as #04C11DB7.

It is possible, however, to calculate a 16-bit CRC on a 32-bit processor. For example, if $G(x) = x^{16} + x^{12} + x^5 + 1$, then `PolynomialGenerator` is #10210000. This is because the most significant 16 bits of the 32-bit integer form a 16-bit generator; the least significant 16 bits of `OldCRC` form the initial CRC value; and the calculated CRC is the

8.1 Example of use

most significant 16 bits of the result from `CRCFROMMSB` and the least significant 16 bits of the result from `CRCFROMLSB`.

8.1 Example of use

Suppose it is required to transmit information between two 32-bit transputers, and the message that is to be transmitted is the byte array:

```
[data FROM 4 FOR size.message]
```

where: there are `size.message` bytes in the message.

Both the transmitter and receiver use the same 32-bit generating polynomial and `OldCRC` value. There are two methods for the receiver to check messages:

First `CRCFROMMSB` is given the message as an input string, the result is placed into the first four bytes of `data` and the message is sent. The receiver can either:

- Give the received `data` (which is `(size.message + 4)` bytes long) to `CRCFROMMSB` and expect a result of zero, or
- Give the received `[data FROM 4 FOR size.message]` to `CRCFROMMSB` and check that the result is equal to the `INT` contained in the received `[data FROM 0 FOR 4]`.

These methods of checking are equivalent. If the check fails then the transmitted data was corrupted and re-transmission can be requested; if the check passes then it is most probable that the data was transmitted without corruption - just how probable depends on many factors, associated with the transmission media.

Note: The `OCCAM` predefines `CRCBYTE` and `CRCWORD` can be chained together to help calculate a CRC from a byte string, and this is indeed the use to which they are put in `CRCFROMMSB` and `CRCFROMLSB`. However, because these latter routines shift the polynomial $F(x)$ corresponding to `InputString` by x^n , these routines should not be chained together over segments of a byte string to find its CRC; the whole string must be used in a single call to `CRCFROMMSB` or `CRCFROMLSB`.

8.2 Function definitions

`CRCFROMMSB`

```
INT FUNCTION CRCFROMMSB (VAL []BYTE InputString,  
                        VAL INT PolynomialGenerator,  
                        VAL INT OldCRC)
```

This routine is intended for strings in normal transputer format (little-endian). The most significant bit of the given string is taken to be bit 7 of the most significant byte `InputString[(SIZE InputString) - 1]`.

`PolynomialGenerator`, `OldCRC` and the result are all also in normal transputer format (little-endian).

CRCFROMLSB

```
INT FUNCTION CRCFROMLSB (VAL []BYTE InputString,  
                        VAL INT PolynomialGenerator,  
                        VAL INT OldCRC)
```

This routine accommodates strings in big-endian format. The most significant bit of `InputString` is taken to be bit 0 of `InputString[0]`. The generated CRC is given in big-endian format. `PolynomialGenerator` and `OldCRC` are taken to be in little-endian format.

9 Extraordinary link handling library

Library: `xlink.lib`

The extraordinary link handling library contains routines for handling communication failures on a link, as listed in Table 9.1.

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. These procedures are intended to be used as input/output pairs, in order to provide secure communication at both ends of the link. A fifth procedure allows the channel to be reset. Use of these routines is described in section 12.5 of the *occam 2.1 Toolset User Guide*.

These procedures must not be used on virtual channels implemented in software.

Procedure	Parameter specifiers
<code>InputOrFail.c</code>	<code>CHAN OF ANY c, []BYTE message,</code> <code>CHAN OF INT kill, BOOL aborted</code>
<code>InputOrFail.t</code>	<code>CHAN OF ANY c, []BYTE message,</code> <code>TIMER t, VAL INT time, BOOL aborted</code>
<code>OutputOrFail.c</code>	<code>CHAN OF ANY c, VAL []BYTE message,</code> <code>CHAN OF INT kill, BOOL aborted</code>
<code>OutputOrFail.t</code>	<code>CHAN OF ANY c, VAL []BYTE message,</code> <code>TIMER t, VAL INT time, BOOL aborted</code>
<code>Reinitialise</code>	<code>CHAN OF ANY c</code>

Table 9.1 Extraordinary link handling procedures

CAUTION:

Use of the routines in `xlink.lib` during *interactive debugging* will lead to undefined results. In addition these routines may only be used for channel communication on a link; they must not be used for communications between processes on the same processor.

9.1 Procedure definitions

The first four of these procedures take as parameters a link channel `c` (on which the communication is to take place), a byte vector `mess` (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. `aborted` is set to `TRUE` if the communication times out or is aborted; otherwise it is set to `FALSE`.

Note: In rare circumstances `aborted` may be set to `TRUE` even though the communication is successful. This happens if the communication terminates successfully in the interval after the timeout or abort signal and before the channel is initialized again. The likelihood of this event is very small.

9.1 Procedure definitions

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.

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 determined 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`. If the timeout occurs then the channel `c` is reset and this procedure terminates.

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.

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 determined 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`. If the timeout occurs then the channel `c` is reset and this procedure terminates.

Reinitialise

PROC Reinitialise (CHAN OF ANY c)

This procedure may be used to reinitialize the link channel *c* after it is known that all activity on the link has ceased.

reinitialise must only be used to reinitialize 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 behavior is undefined.

10 Debugging support library

Library: `debug.lib`

The debugging support library provides procedures, listed in Table 10.1, to assist with debugging when the INQUEST interactive debugger is being used. Two procedures are provided to stop a process, one on a specified condition. Another procedure is used to insert debugging messages. The procedure `ASSERT`, for use when INQUEST is not present, is described in section 2.9.

Procedure	Parameter Specifiers
<code>DEBUG.ASSERT</code>	<code>VAL BOOL assertion</code>
<code>DEBUG.MESSAGE</code>	<code>VAL []BYTE message</code>
<code>DEBUG.STOP</code>	none

Table 10.1 Debugging support procedures

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. This procedure may be used in place of `ASSERT` when using the interactive debugger.

If `assertion` evaluates to `FALSE` and the INQUEST debugger is present, then `DEBUG.ASSERT` stops the process and sends process data to the debugger. If `assertion` evaluates to `TRUE` then no action is taken.

If the program is not being run with the interactive 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 and stops the thread with event type 6. The interactive debugger can then locate the source of the message and execution can be resumed. The first 80 characters of the message is displayed with the normal program output.

If the program is not being run with the interactive debugger then the procedure has no effect.

10.1 Procedure definitions

DEBUG . STOP

PROC DEBUG . STOP ()

If the program is being run with the interactive debugger, this procedure stops the process and sends process data to the debugger.

If the program is not being run with the interactive debugger then the procedure stops the process or processor, depending on the error mode that the processor is in.

Appendices

A Language extensions

This appendix describes language extensions that are supported by the OCCAM 2.1 compiler.

Note: These extensions are compiler-dependent and do not extend the syntax of the OCCAM 2.1 language as defined in the *OCCAM 2.1 Reference Manual*. The extensions to the OCCAM 2 language given by OCCAM 2.1 are listed in Appendix P of the *OCCAM 2.1 Reference Manual*.

A.1 Syntax

A.1.1 Compiler keywords

The following additional keywords are supported by the OCCAM 2.1 compiler:

```
ASM IN INLINE VECSPACE WORKSPACE
```

The keyword `GUY` is also recognized by the compiler, but is no longer supported by SGS-THOMSON. This compiler supports in-line assembly code, as described in section 12.3 and Appendix B of the *OCCAM 2.1 Toolset User Guide*, is introduced by the keyword `ASM`.

A.1.2 Compiler directives

The following directives are supported by the OCCAM 2.1 compiler:

```
#INCLUDE #USE #COMMENT #IMPORT #OPTION #PRAGMA
```

For more information see section 3.13 of the *Toolset Reference Manual*.

A.1.3 String escape characters

The syntax of the non-printable character '*', as defined in Appendix G of the *OCCAM 2.1 Reference Manual*, has been extended. The first character of a literal string may now take the value *1 or *L, which is used to represent the length of the string, excluding the character itself. For example, the following statements define the same string:

```
VAL string1 is "*1Fred" :
VAL string2 is "**#04Fred" :
```

*1 (or *L) is illegal if the string (excluding the *1) is longer than 255 bytes, and will be reported as an error.

The characters *, ' and " may be used in the form given in Table A.1.

A.2 Retyping channels to and from data items

*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		

Table A.1 Special character constants

Any byte value can be represented by ***#** followed by two hexadecimal digits.

A.1.4 Tabs

The compiler expands tabs in source files to be every eighth character position. Tabs are permitted anywhere in a line but are not expanded within strings or character constants.

A.1.5 Relaxations on syntax

There is no limit on the number of significant characters in identifiers, and the case of characters is significant.

A.2 Retyping channels to and from data items

Channels may be retyped:

- to and from data items
- between protocols of different types.

Retyping between protocols allows the protocol on a channel to be changed, as described in section 10.2 of the *Occam 2.1 Reference Manual*. This facility should be used with care. This section describes retyping to and from data items.

Retyping a channel to or from a data item depends upon the implementation of channels. With the implementation given by this toolset, retyping maps a data item onto a pointer to the channel word. This can be used, for example, to determine the address of the channel word, or to create an array of channels pointing at particular addresses:

```
CHAN OF protocol c :
  VAL INT x RETYPES c : -- Must be a VAL RETYPE
  ... use x as the address of the channel word
```

This operation (retyping a channel to a data item) can be achieved more portably by means of the `LOAD.INPUT.CHANNEL` predefine. See Chapter 12 of the *Occam 2.1 Toolset User Guide* and section 2.7 in this manual.

The following code demonstrates how to create a channel array whose channels point at arbitrary addresses.

```
[10]INT x :
SEQ
... initialise elements of x to the addresses of ...
the channel words
[10]CHAN OF protocol c RETYPES x :
... use channel array c
```

A.3 Low level programming

A.3.1 ASM

The keyword **ASM** introduces a section of transputer assembly code; see section 12.3 and Appendix B of the *Occam 2.1 Toolset User Guide*.

A.3.2 PLACE statements

The **PLACE** statement in **OCCAM** allows a channel, a variable, an array, or a **PORT**, to be placed in workspace, in vector space or at an absolute location in memory.

The syntax of the supported **PLACE** statements extends the definition of an allocation as defined in the *Occam 2.1 Reference Manual*:

```
allocation = PLACE name AT expression
            | PLACE name AT WORKSPACE expression
            | PLACE name IN WORKSPACE
            | PLACE name IN VECSPACE
```

The **PLACE** statement must be inserted immediately following the declaration of the variable to which it refers e.g.

```
int x, y, z :
PLACE x .....
PLACE y .....          is correct

int x :
int y :
PLACE x .....          is incorrect
```

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)
```

A.3 Low level programming

The numbers used as **PLACE** addresses are word offsets from the bottom of address space. For example, **PLACE scalar channel AT n** places the channel word at that address, and **PLACE array of channels AT n**, places the array of pointers at that address.

The **PLACE name IN WORKSPACE** and **PLACE name IN VECSPACE** statements place variables explicitly in program workspace and vector space respectively. The allocation of workspace and vectorspace by the compiler is described in Appendix B of this manual.

The **PLACE name AT WORKSPACE expression** statement places a variable at a specific offset in workspace. An example use of this is given in the *occam 2.1 Toolset User Guide*, section 12.1.2.

Chapter 12 of the *occam 2.1 Toolset User Guide* also describes allocation.

A.3.3 **INLINE** keyword

INLINE may be used immediately before the **PROC** or **FUNCTION** keyword of any procedure or function declaration. It 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. Use of **INLINE** procedures or functions may increase the size of the object module but will also avoid the overheads incurred in executing extra calls.

The **INLINE** statement extends the syntax of a definition as defined in the *occam 2.1 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:
```

Examples:

```
INT INLINE FUNCTION sum3 (VAL INT x, y, z,) IS x + (y + z):

INLINE PROC seterror ()
  error := TRUE
:
```

A call to the `FUNCTION sum3`:

```
so.write.int(fs, ts, sum3(p,q,r),0)
```

would be expanded by the compiler thus:

```
so.write.int(fs, ts, p + (q + r),0)
```

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, marked with the `INLINE` keyword, in a separate file, and include this file with an `#INCLUDE` directive.

B 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.

B.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 in Figure B.1.

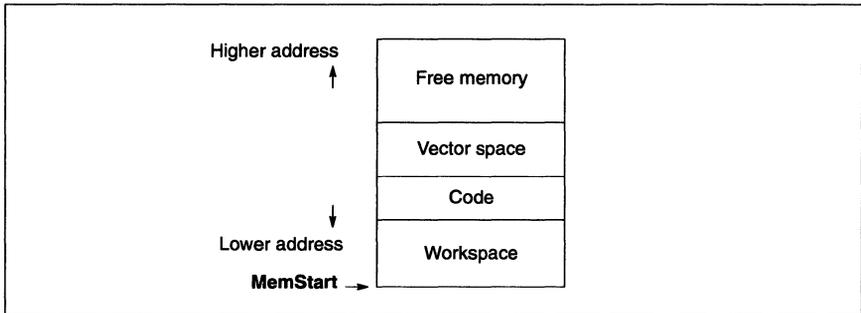


Figure B.1 Program memory layout

B.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 example, the memory layout for the following code is shown in Figure B.2:

```
PROC P()  
  PROC Q()  
    ... code for Q  
  :  
  PROC R()  
    ... code for R  
  :  
  ... code for P  
  :
```

B.1 Memory allocation by the compiler

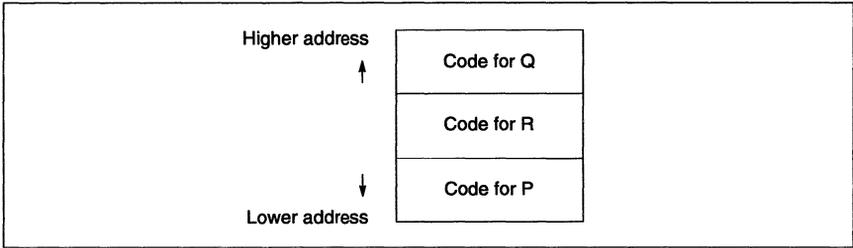


Figure B.2 Code layout for procedures

B.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.

The compiler will attempt to optimize floating point routines, such as `REAL32OP`, by giving them a high priority. This can be overridden by using the compiler directive `#PRAGMA LINKAGE` in conjunction with the linker directive `#section`.

B.1.3 Workspace

Workspace is placed lowest in memory, before the arithmetic handling library, so that it has priority usage of the on-chip RAM, if the processor is configured to have any. 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. Figure B.3 shows the data layout in memory at the point labelled `here`.

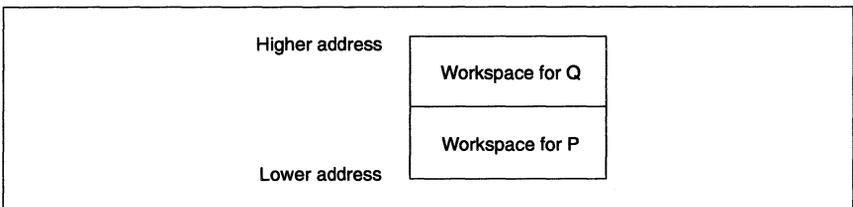


Figure B.3 Workspace layout for procedures

In a **PAR** or **PRI PAR** construct the last textually defined process is allocated the lowest addressed workspace. For example, Figure B.4 shows the workspace layout for the following parallel processes:

```
PAR
... P1
... P2
... P3
```

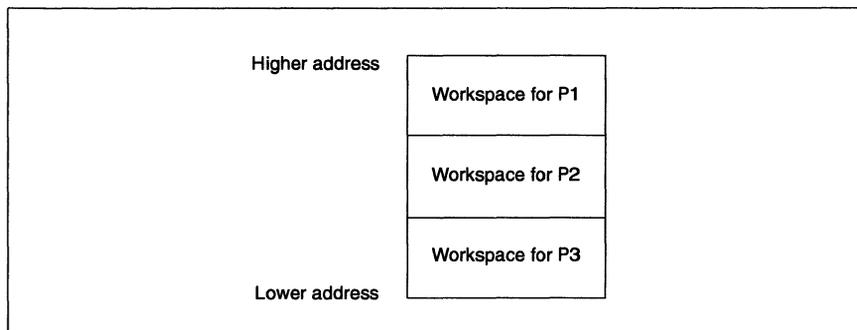


Figure B.4 Workspace layout for parallel processes

In a replicated **PAR** construct, the instance with the highest replication count is allocated the lowest workspace address. For example, Figure B.5 shows the workspace layout for the following parallel processes:

```
PAR i = 0 FOR 3
P [i]
```

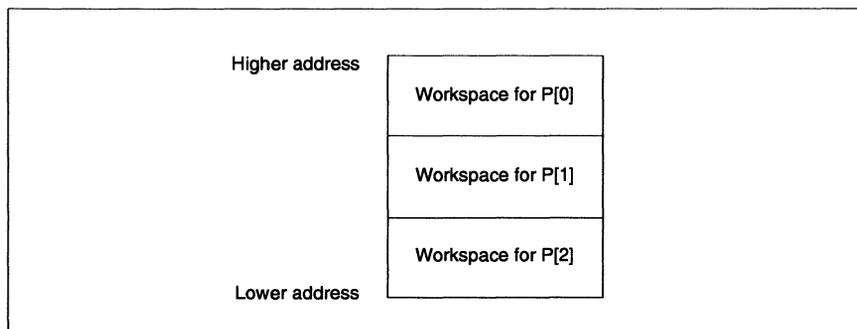


Figure B.5 Workspace layout for replicated parallel processes

Unless separate vector space is disabled, most arrays are allocated in a separate data space, known as vector space, as described in section B.1.4. The allocation is done in

B.1 Memory allocation by the compiler

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.

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.

B.1.4 Vectorspace

By default, arrays larger than 8 bytes are allocated into a separate stack known as the *vectorspace*. This scheme optimizes use of the workspace, creating more compact and quicker code. It can also make better use of a transputer's on-chip RAM. The default scheme may be overridden by an option on the command line, a directive in the source code, or, for specific variables only, by a `PLACE` statement.

This can be overridden per compilation unit by the `v` command line switch or `#OPTION "v"` directive. This will force all variables into the workspace. Secondly, the current 'default' may be overridden on an array-by-array basis by using extra allocations as follows:

```
[100]BYTE a :  
PLACE a IN WORKSPACE :  -- forces a to reside in workspace  
  
[100]BYTE b :  
PLACE b IN VECSPACE :   -- forces b to reside in vectorspace
```

Only arrays may be placed in vectorspace; scalar variables must reside in workspace. Arrays smaller than 8 bytes may be explicitly placed in vectorspace.

It may be desirable to change the default vectorspace allocation for various reasons. Using vectorspace can actually slow down execution, since an extra parameter is passed to each subroutine which requires it. However, this cost is normally overwhelmed by the reduction in workspace size, and the associated compaction in the number of prefix instructions required to address local variables. In certain circumstances it may be useful to place a commonly used array into workspace, particularly if it is heavily used in array assignment (block moves). Alternatively it may be useful to place most arrays in workspace, but move any large arrays into vectorspace.

B.2 Type mapping

This section describes how the OCCAM types are represented on the target processor. The correct operation of an OCCAM program should not depend on implementation details unless they are part of the language, as described in the *occam 2.1 Reference Manual*. Future compiler releases may use different implementations.

Table B.1 summarizes the type implementation. For further information on data types see Chapter 3 of the *occam 2.1 Reference Manual*.

Type	Storage	Range of values
BOOL	1 byte	FALSE, TRUE
BYTE	1 byte	0 to 255
INT16	2 bytes	-32768 to 32767
INT32	4 bytes	-2,147,483,648 to 2,147,483,647
INT64	8 bytes	-2 ⁶³ to (2 ⁶³ -1)
INT (On 32-bit processors)	4 bytes	-2,147,483,648 to 2,147,483,647
INT (On 16-bit processors)	2 bytes	-32768 to 32767
REAL32	4 bytes	IEEE single precision format
REAL64	8 bytes	IEEE double precision format
PACKED RECORD	Sum of fields	
Named types	As for underlying type	
CHAN (On 32-bit processors)	8 bytes	Channels are implemented as a pointer to a channel word.
CHAN (On 16-bit processors)	4 bytes	
PORT OF D	As for D	
TIMER	None	

Table B.1 OCCAM type implementation

All objects are word aligned, i.e. 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. Any named type is stored in the same way as the underlying type.

Arrays are packed, i.e. there are no spaces between the elements. (**Note:** that an array of type **BOOL** has one byte for each element).

The compiler uses reordering and padding to implement **RECORDS**, and the current implementation may be changed in future releases. If the implementation is significant then **PACKED RECORDS** should be used. The operators **OFFSETOF** and **BYTESIN** may be used to identify the position of a field and the size of a record.

For example, consider the following definition of **RECORD** type **REC.TYPE**, compiled for a 32-bit transputer:

B.3 Parameter passing

```
DATA TYPE REC.TYPE
RECORD
  BYTE  b1:
  INT16 i16:
  REAL64 r64:
  BYTE  b2:
  BYTE  b3:
:
```

In this example the compiler allocates one 32-bit word for the three **BYTE** fields, one word for the **INT16** field and two words for the **REAL64** field.

The fields of a **PACKED RECORD** type are stored and communicated in the same order as they are listed in the type definition with no padding added between the fields. The first field is stored at the lowest address and communicated first. The total size of a **PACKED RECORD** must be a whole number of words. Within a **PACKED RECORD**, the following alignment rules apply to the start of each field:

- **BYTE** and **BOOL** scalar and array fields may start on any byte,
- **INT16** scalar and array fields must be 2-byte aligned, and
- other integer, real, array and record fields must be word aligned.

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 **RETYPE**s must be aligned to the natural alignment for that data type; **BYTES** and **BOOLS** 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 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
```

B.3 Parameter passing

Parameters are divided into **VAL** and non-**VAL** parameters. These have different semantics and may be passed differently.

B.3.1 VAL parameters

Scalar values that fit within the word length of the target machine are represented as items one word long containing the value of the parameter.

In the case of **BYTE** and **BOOL**, the value is found in the low-order byte of the word and the high order bytes are guaranteed to be zero.

In the case of an **INT16** parameter on a 32-bit processor, the value resides in the low-order 2 bytes of the word. The high-order bytes are undefined.

If the parameter is a primitive type, or a record, that will not fit into a processor word, then it is passed as a pointer to the actual value.

If the parameter is an array, a pointer to its base is passed, i.e. the address of the first element in the array. If any of its strides are undefined in the source then extra parameters are passed containing the integer values of the missing strides. These parameters, of type **INT**, are placed immediately after the pointer to the base of the array and appear in the same order as the missing strides appear in the source.

Timers, channels and ports can never be **VAL** parameters.

B.3.2 Non-VAL parameters

Since these parameters change the actual parameters passed, the formal parameters are always represented as pointers to the actual parameters (except for timers - see below).

If the parameter is an array, then it is treated the same way as a **VAL** parameter with any missing strides following the address of the first element.

If the parameter is a timer, it occupies no storage and so no parameter slot is reserved for it.

B.3.3 Alignment

The compiler assumes that all parameters passed to a routine are aligned to the requirements described in Table B.2.

Type	Alignment	Type	Alignment
BOOL	byte	BYTE	byte
INT	word	INT16	even byte
INT32	word	INT64	word
REAL32	word	REAL64	word

Table B.2 Data alignment

B.4 Calling sequence

This section describes the state on entry to the called routine.

B.4.1 Registers

All registers are undefined. In particular, you may not assume that **Arg** is defined to be the return address, because the entry may be via a jump which has timesliced. If some

B.4 Calling sequence

occam code calls a separately compiled routine, then the call is done by means of a *call* instruction to a *stub*, which then does a *j* to the true entry point. Calls to procedures and functions defined within the same compilation module cannot timeslice.

On all processors with a floating point unit, the floating point registers are never assumed to have any values.

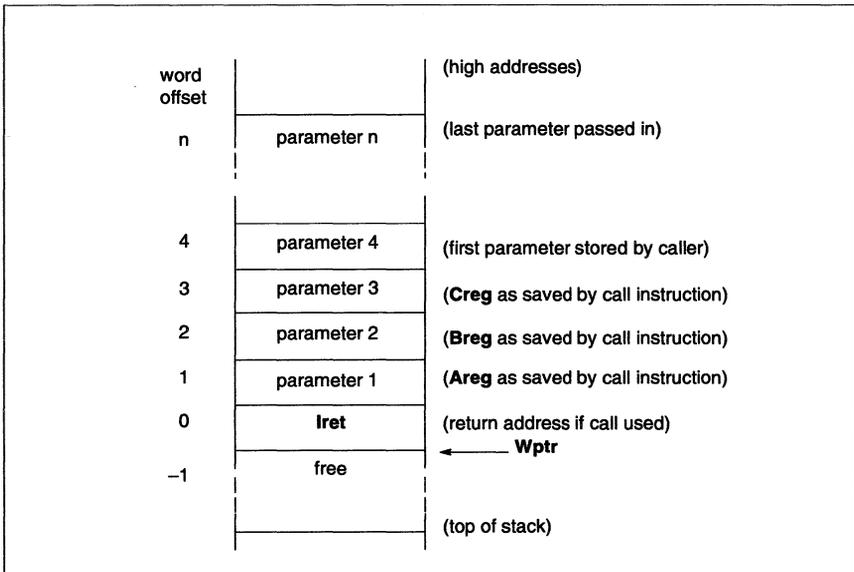
Iptr addresses the first instruction of the invoked procedure or function.

Wptr addresses the invocation stack frame (see next section). It must be word aligned.

B.4.2 Invocation stack

The invocation stack at entry to a procedure or function is addressed by non-negative offsets from **Wptr**. Negative offsets are in the free (unused) part of the invocation stack.

Most parameters map one-to-one onto words of stack space. However, as described in section B.3, **TIMERS** occupy no storage, and arrays with open dimensions have extra hidden parameters passed to indicate their real dimensions. The following diagram assumes that all parameters are a single word long.



B.4.3 Iret

In all cases, the value of **Iret** has been stored before the first instruction of the called routine is executed.

B.4.4 Parameters

The source code parameters to the procedure are *found* at successive word offsets from **Wptr** + *k* (words), where $1 \leq k \leq 3$. There may be up to two hidden parameters inserted before the source code parameters. The hidden parameters being static link address and vector space address:

k = 1 if neither the vector space address or static link parameters are used;

k = 2 if one but not both of these parameters are used;

k = 3 if both parameters are used.

In some cases the vector space address may be the last parameter - see below. Each parameter occupies just one word. (See section B.3 for exceptions to this.) The source code parameters are placed in the appropriate number of parameter words (see below for details) in the lexical order in the source program.

The pointers to indicate the **FUNCTION** return value positions (if there are any) come before all the source parameters (see below).

In some cases, the procedure needs the address of the outer level stack frame (the static link). In this case, it is the first parameter to the procedure or function. (**Note:** that this is never true for externally visible functions).

If the procedure being called allocates arrays in the vector space, the current value of the vector space pointer is also passed in as a parameter. For calls to externally visible procedures or functions the parameter is the last one of all. Otherwise, the parameter is the first following the static link (if any). This parameter is optional.

The compiler is free to assume that locations (**Wptr** + 0) to (**Wptr** + 3) (the slots corresponding to the registers stored by the *call* instruction) have been allocated for its own use. It may corrupt any of the locations, after (if necessary), reading any actual parameters stored therein. If the routine is called via a *gcall* instruction, or by any other method, it is up to the caller to ensure that these locations are available. In effect, there must always be at least 3 actual parameters.

B.5 Implementation of channels

A channel is implemented as a *pointer* to a channel word. When mapping channels to specific transputer links, the channel word is placed at the specified address for scalar channels. Arrays of channels are mapped as arrays of pointers to channel words.

As a result of this **PLACE**ing arrays of channels is implemented such that:

```
PLACE array.of.channels AT n:
```

places the array of *pointers* at that address.

```
PLACE scalar.channel AT n:
```

B.6 Transputer timers (clocks)

places the channel word at that address.

An example of the placement of channels on links is given in Chapter 12 of the *occam 2.1 Toolset User Guide*.

The effect of **PLACE** on scalar channels differs from the effect of **PLACE** on channel arrays. For scalar channels, the *channel word* is **PLACED**, but for channel arrays, the *array of pointers* is **PLACED**.

Arrays of channels may be constructed out of a list of other channels. For example:

```
PROC p (CHAN OF protocol a, [2]CHAN OF protocol b)
  [3]CHAN OF protocol c IS [a, b[0], b[1]] :
  -- channel constructor

  ALT i = 0 FOR SIZE c
    c[i] ? data
    ...
  :
```

Channel constructors extend the facilities for manipulating channels; further information is given in section 12.2 of the *occam 2.1 Toolset User Guide*.

B.6 Transputer timers (clocks)

The transputer has two timers which can be accessed by the programmer. They are used for real time programming, timing events, data logging, timing out, delays and so on.

Two timers are provided to give low and high resolution timing. The timers themselves are word length registers which are incremented regularly and related to the speed of the input clock. The low resolution timer goes 64 times slower than the high resolution timer, as shown in Table B.3. These speeds are independent of transputer model, processor speed, and word length.

Priority	Low	High
Time between ticks	64 μ seconds	1 μ second
Ticks per second	15625	1000000
Approximate cycle time (16-bit)	4 seconds	65 milliseconds
Approximate cycle time (32-bit)	76 hours	1 hour 10mins

Table B.3 Timer parameters

The high resolution timer is always used by high priority processes, so is often called the high priority timer. The low resolution timer is always used by low priority processes.

B.6.1 TIMER variables

TIMER variables in *occam* give access to the transputer timers. The syntax of timer input is similar to channel input.

Timers can have only one of two possible values, corresponding to the high and low priority transputer clocks. The clock which is read depends on the priority of the enclosing process. When comparing clock values the same timer variable should be used, and the value *must* be input from the same process or from a process of the same priority on the same processor. If the same timer is used in processes with different priorities or on different processors, the results are undefined.

A common use of timers is to time a process, for example, a channel input:

```
TIMER clock:
SEQ
  clock ? start
  chan ? y
  clock ? end
  delay := end MINUS start
```

The **MINUS** operator performs a modulo subtraction and is used to give a relative difference, because the transputer timers are registers which wrap-around. The maximum period that can be timed is limited to half the clock cycle time, e.g. about 33 milliseconds at high priority and 2 seconds at low priority on 16-bit transputers. If the period being measured is likely to be greater than this then a delay period based on multiples of clock cycles should be built in.

B.6.2 TIMERS as formal parameters

When calling **OCCAM** from **C**, any formal **TIMER** parameter in the **OCCAM** procedure or function must be ignored by the calling **C** code, and no actual parameter should be passed. When calling **OCCAM** routines from other **OCCAM** routines an actual parameter should be passed in the normal way.

B.7 CASE statement

The **CASE** statement is 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 optimize 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.

B.8 ALT statement

No assumption can be made about the relative priority of the guards of an **ALT** statement; if priority is required, a **PRI ALT** must be used.

B.9 Checking IF statements

The compiler checks subscripts in all expressions at compile-time, including those inside code which will never be executed. Hence the following code is illegal:

B.10 Implementation of PORTs

```
PROC p (INT x, [2]INT array)
  IF
    FALSE
    x := array[100]
    TRUE
    x := array[1]
  :
```

B.10 Implementation of PORTs

OCCAM provides **PORTs** to access memory mapped devices. These should be used to access any devices which do not behave as normal memory, i.e. devices which should not be cached, or those which are sensitive to the numbering and/or order of successive read and write operations.

B.11 Formal parameters

If a name appears more than once in a single formal parameter list, the *last* definition is used.

B.12 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 15 625 ticks per second, or one tick = 64 microseconds for all transputer types.
- The high priority clock increments at a rate of 1 000 000 ticks per second, or one tick = 1 microsecond for all transputer types.
- Because the high and low priority clocks tick at different speeds, the same **TIMER** should not be read from within both high and low priority branches of a **PRI PAR**.
- **TIMER** channels cannot be placed in memory with a **PLACE** statement.

B.13 Summary of implementation restrictions

- **FUNCTIONs** may not return arrays of variable size.
- Multiple assignment of arrays of unknown size is not permitted.
- A replicated **PAR** count must be constant.

- There must be exactly two branches in a **PRI PAR**.
- Replicated **PRI PARS** are not permitted.
- Nested **PRI PARS** are not permitted.
- **PLACE** statements must immediately follow the declaration of the variable to which they refer. The only statements which may intervene are other **PLACE** statements, e.g.:

```
INT x, y, z :
PLACE x .....
PLACE y .....
```

- Compiler pragmas **SHARED** and **PERMITALIASES** must immediately follow the declaration of the variable to which they refer.
- 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.
- **FUNCTIONS** which use **ALT** replicator variables as free variables may not be called in the guard of the same **ALT**. An error is reported at compile time if this occurs. Example:

```
PROC P([10]CHAN OF INT c
  ALT i = 0 FOR 10
    INT FUNCTION f() IS 9 - i :
    INT x:
    c[f()] ? x    -- call of 'f()' is illegal
  SEQ
  ...
  ALT
    c[f()] ? x    -- this is legal.
```

This can be resolved by passing the replicator variable into the **FUNCTION** as a parameter:

```
PROC P([10]CHAN OF INT c
  ALT i = 0 FOR 10
    INT FUNCTION f(VAL INT i) IS 9 - i :
    INT x:
    c[f(i)] ? x    -- call of 'f(i)' is legal
  SKIP
```

B.13 Summary of implementation restrictions

- Maximum array size is 64 Kbytes on 16-bit processors, 2 Gbytes on 32-bit processors. No dimension of any array may exceed `MOSTPOS INT`.
- Maximum filename length is 128 characters.
- Maximum 256 tags allowed in `PROTOCOLS`.
- Maximum number of lexical levels is 254. (Applies to nested `PROCS` and replicated `PARS`).
- The compiler can compile a maximum of 240 Kbytes of object code in a single compilation unit. (This limit can be changed by using the compiler command line option `CODE`).
- 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.
 - No abbreviations containing function calls or `VALOFS` 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.
```

- The compiler does not permit the following syntax, which refers to the zeroth element of a table of one element, type converted to a named data type:

```
x := mytype [myelement][0]
```

This restriction is necessary because the compiler cannot distinguish this syntax from an element of a two dimensional array.

- The compiler does not permit the following statement, where the length of the counted array is the `SIZE` of an array of channels :

```
c ! SIZE [n]CHAN OF INT :: a
```

This restriction is necessary because the compiler cannot distinguish this syntax from sending the `SIZE` of an array of channels with a counted array protocol, as it is not clear where the operand of `SIZE` ends. This ambiguity can be resolved by adding parentheses:

```
c ! (SIZE [n]CHAN OF INT) :: a
```

C Alias and usage checking

C.1 Alias checking

This section describes the alias checking that is implemented by the compiler. Alias and usage checking are described in the *occam 2.1 Reference Manual*.

In the following text 'assigned to' means 'assigned to by assignment or input'.

C.1.1 Introduction

Alias checking is the name given to the series of checks made by the compiler on OCCAM abbreviations to ensure that an object is known by a single name within a given scope i.e. no *aliases* exist. In practice this means that:

- named abbreviations must be used correctly within the code
- expressions used to define abbreviations (plus any subscripts within them) must be valid and within range
- variables in defining expressions must not be changed.

The same rules apply to retyped names, since retyping is simply another form of abbreviation.

Alias checking is governed by a strict set of rules. This enables many checks to be done at compile time, reducing the need for run-time checking code. Some checks, however, can only be performed at run-time.

Alias checking can be disabled in a compilation unit or for specific variables. It is then up to the programmer to ensure that the rules are complied with, or the behavior of the program is undefined.

The formal rules used for alias checking are set out below. For further details see the *occam 2.1 Reference Manual*.

C.1.2 Rules

Scalar variables

Rule 1 If a scalar variable appears in the abbreviated expression of a **VAL** abbreviation, for example **x** in:

```
VAL a IS x + 2 :
```

then that variable may not be assigned to or abbreviated by a non-**VAL** abbreviation anywhere within the scope of the **VAL** abbreviation.

C.1 Alias checking

Rule 2 If a scalar variable is abbreviated in a non-**VAL** abbreviation, for example **x** in:

```
a IS x :
```

then that variable may not be referenced anywhere within the scope of the abbreviation.

Arrays

The aliasing rules for arrays attempt to treat each element of the array as an individual scalar variable. They allow the maximum freedom possible without introducing run-time checking code except at points of abbreviation. In the following text the word 'constant' means any expression that can be evaluated at compile-time.

If an array is referenced in the expression of a **VAL** abbreviation, for example **x** in:

```
VAL a IS x[i] :
```

then rules 3 to 5 apply to the use of the array within the scope of the abbreviation:

Rule 3 If the subscript is constant then elements of the array may be assigned to as long as they are only subscripted by constant values different from the abbreviated subscript. Any element of the array may also appear anywhere in the expression of a subsequent **VAL** abbreviation. Any other elements of the array may be non-**VAL** abbreviated, and run-time checking code is generated if subscripts used in the abbreviation are not constant.

Rule 4 If the subscript is not constant then no element of the array may be assigned to unless it is first non-**VAL** abbreviated. The non-**VAL** abbreviation may generate run-time code to check that it does not overlap the **VAL** abbreviation. The array may be used in the expression of a subsequent **VAL** abbreviation.

Rule 5 Elements of the array may be read anywhere within the scope of the abbreviation except where restricted by further abbreviations.

If an array is abbreviated in a non-**VAL** abbreviation, for example **x** in:

```
a IS x[i] :
```

then rules 6 to 8 apply to the use of the array within the scope of the abbreviation:

Rule 6 If the subscript is constant then elements of the array may be read and assigned to as long as they are accessed by constant subscripts different from the abbreviated subscript. Other elements of the array may be abbreviated in further **VAL** and non-**VAL** abbreviations, and run-time checking code is generated if subscripts used in the abbreviation are not constants.

Rule 7 If the subscript is not constant then the array may not be referenced at all except in abbreviations, in which case run-time code may be generated to check that the abbreviations do not overlap.

Rule 8 Variables used in subscripts of the array being abbreviated act as if they have been **VAL** abbreviated. In the above example **i** acts as if it has been **VAL** abbreviated and cannot be altered in the scope of the abbreviation. Where elements of the array being abbreviated are used in the subscript of the array then the abbreviation is checked as if the subscript expression was **VAL** abbreviated just before the non-**VAL** abbreviation. For example:

```
a IS x[x[2]] :
```

is checked as if it was written:

```
VAL subscript IS x[2] :
a IS x[subscript] :
```

which (by Rule 7 above) will generate run-time checking code.

C.1.3 Alias checking disabled

Note: the compiler will be able to generate the most efficient code when all alias checks are *enabled*.

When alias checking is disabled, either on a compilation unit or on specific variables, the compiler cannot guarantee no aliasing on the affected variables. It is then the programmer's responsibility to ensure that the rules described in this section are not broken.

The set of *aliasable* variables consists of either:

- the set of *all* variables declared in the compilation unit (if the whole compilation unit has alias checking turned off by the **A** command line option or the **#OPTION "A"** directive),

or

- the set of all variables which have been marked by the pragma **PERMITALIASES**, together with those variables which have been deduced to be aliasable by the compiler according to the rules described below.

(See Chapter 3 in the *Toolset Reference Manual* for a description of the pragma **PERMITALIASES**).

In the presence of aliasable variables, the behavior of a program is defined in the intuitive model, where all reads and assignments to aliasable variables are executed strictly in the order in which they are written; they may not be re-ordered by an optimizer. Note also that because of retyping it cannot be assumed that a write to an aliasable variable of one type will not affect an aliasable variable of another type.

When variables are aliasable the following rules apply.

VAL abbreviations

Suppose we have a **VAL** abbreviation of the form:

C.1 Alias checking

VAL *name* **IS** *expression* :

If *name* is aliasable, then

all component variables of *expression* are automatically inferred to be aliasable by the compiler and may be modified in the scope of the abbreviation,

and therefore:

name must not be used after any component variable of *expression* is modified.

If this constraint is not met, the behavior of the program is undefined, and results will be implementation-dependent.

For example:

```
VAL x IS a[i] :
#PRAGMA PERMITALIASES x
SEQ
... use x                -- This is OK
... modify i or a[i]    -- This is OK if x is aliasable
... use x                -- This is undefined
```

Non-VAL abbreviations

Suppose we have a non-VAL abbreviation of the form:

name **IS** *element* :

If *name* is aliasable, then:

any variable used in a subscript to select a component or components of an array reference in *element* may be modified in the scope of the abbreviation. (All such variables are automatically inferred to be aliasable by the compiler.)

and therefore:

name must not be used after a variable used in a subscript to select a component or components of an array referenced in *element* has been modified.

If this constraint is not met, the behavior of the program is undefined, and results will be implementation-dependent.

If the base variable of *element* is aliasable then:

the actual element referred to by such an abbreviation *may* be modified within the scope of the abbreviation. *name* is automatically inferred to be aliasable by the compiler, and the abbreviation is implemented as though *name* is a pointer to the *element*.

An alias may subsequently be created, either by referring to *element* explicitly in the scope of the abbreviation, or by using another abbreviation within the scope.

Multiple assignment

In a multiple assignment, the destinations of the assignment are written to as though they are in parallel. Therefore:

variables listed as the left hand side of a multiple assignment *must not* be aliased.

It is up to the programmer to ensure that this rule is complied with. If it is not, the behavior of the program is undefined.

Note that it is perfectly legal for the destinations of an assignment to alias expressions used on the right hand side of an assignment. Thus destination variables *may* be aliased with actual parameters to a **FUNCTION**.

Procedure parameters

It is assumed that only (non-**VAL**) formal parameters of a procedure which are aliasable may be aliased, either with each other, with the **VAL** parameters, or with aliasable free variables.

If an actual parameter to a procedure aliases another actual parameter, or aliases a free variable used by the procedure, then:

the *formal* parameter must be marked as aliasable. Note that this also applies across separately compiled units.

Since the rules for procedure parameters derive from those for abbreviations, the following constraint applying to **VAL** abbreviations also applies. That is:

any variable used in an actual parameter corresponding to a **VAL** formal parameter must not be modified in the body of the procedure prior to the final read of the formal parameter.

It is up to the programmer to ensure that these rules are complied with. If they are not, the behavior of the program is undefined.

Interaction with usage checking

Since the usage checking algorithms rely on lack of aliasing, any aliasable variable is automatically inferred to be shared; see section C.2.7.

C.2 Usage checking

This section describes the usage checking that is implemented by the compiler. The usage rules are given in section E.1 of the *occam 2.1 Reference Manual*.

C.2.1 Introduction

Usage checking is the name given to the series of checks made by the compiler to ensure that parallel processes do not share variables, channels span only two

C.2 Usage checking

processes, and communication down channels is unidirectional. Using a set of rules means that many checks can be done at compile time, reducing the need for run-time checking code.

C.2.2 Checking of non-array elements

Variables and channels which are not elements of arrays are checked according to the rules of OCCAM 2.1 given in section E.1 of the *OCCAM 2.1 Reference Manual*.

C.2.3 Checking of arrays of variables and channels

Where possible, the compiler treats each element of an array as an independent variable. This makes it possible to assign to the first and second elements of an array in parallel.

For usage checking to operate in this way, it must be possible for the compiler to evaluate all possible subscript values of an array. The compiler is capable of evaluating expressions consisting entirely of constant values and operators (but not function calls). Where a replicator is used in an expression the compiler can evaluate the expression for all values of the index provided that the base and count of the replicator can be evaluated. **Note:** however, that as each iteration of the routine is checked, this can slow the compiler down.

Where an array subscript contains variables, a function call, or the index of a replicator where the base or the count cannot be evaluated, the compiler assumes that all possible subscripts of the array may be used. This may cause a spurious error. For example, consider the following program fragment:

```
x := 1
PAR
  a[0] := 1
  a[x] := 2
```

The compiler reports the assignment to a[x] as a usage error. The fragment could be changed to:

```
VAL x IS 1:
PAR
  a[0] := 1
  a[x] := 2
```

This would be accepted by the compiler because x can be evaluated at compile time.

The compiler checks segments of arrays similarly to simple subscripts. Where the base and count of a segment can be evaluated, each segment is treated as though it has been used individually. Where the base or count cannot be evaluated, the compiler behaves as if the whole array has been used. For example, the following code is accepted without generating an error:

```

PAR
  [a FROM 4 FOR 4] := x
  a[8] := 2
  [a FROM 9 FOR 3] := y

```

C.2.4 Arrays as procedure parameters

Any variable array which is the parameter of a procedure is treated as a single entity. That is, if any element of the array is referenced, the compiler treats the whole array as being referenced. Similarly, if any free variable array, or element of a free variable array is assigned in a procedure then the compiler treats it as if every element were assigned. For example, the compiler reports an error in the following code because it considers every element of `a` to have been assigned when `p(a)` occurred.

```

PROC p([ ]INT a)
  a[1] := 2
:
PAR
  p(a)
  a[0] := 2

```

Similarly, where one element of an array of channels is used for input or output within a procedure, the compiler treats the array as if all elements were used in the same way. For example, the compiler reports an error in the following code because it considers an output has been performed on every element of `c` when `p()` occurred.

```

PROC p()
  c[1] ! 2    -- c free in p
:
PAR
  p()
  c[0] ! 1

```

C.2.5 Abbreviating variables and channels

The compiler treats an element which is abbreviated in an element abbreviation as if it had been assigned to, whether or not it is actually updated. If this causes an apparently correct program to be rejected the program should be altered to use a `VAL` abbreviation. For example, the compiler reports an error in the following code because it considers that `b` is assigned to in the first component of the `PAR`.

```

PAR
  a IS b :
  x := a
  y := b

```

This could be changed to:

```

PAR
  VAL a IS b :
  x := a
  y := b

```

C.2 Usage checking

Where a channel is an abbreviation of a channel array element, the compiler behaves as if the whole of the channel array had been used unless the element is an array element with constant subscripts, a constant segment of an array (i.e. with constant base and count) or a constant segment with constant subscripts.

C.2.6 Channels

A channel formal parameter, or a free channel of a procedure, may not be used for both input and output in a procedure. This check is disabled if usage checking is disabled.

C.2.7 Usage checking disabled

Note: the compiler will be able to generate the most efficient code when all usage checks are *enabled*.

The compiler supports two switches which can be used to disable either Usage checking only, or both Alias and Usage checking together. Usage checking can be disabled on a compilation unit by the `N` command line switch or argument to the `#OPTION` compiler directive. Usage checking can also be turned off on specific variables using the `SHARED` pragma. See Chapter 3 in the *Toolset Reference Manual* for more details.

When usage checking is disabled it is the programmer's responsibility to ensure that variables and channels are used correctly according to certain rules. If they are not the behavior of the program is undefined.

The set of shared variables consists of either:

- the set of *all* variables declared in the compilation unit, if the whole compilation unit has usage checking disabled, or
- the set of all variables which have been marked by the pragma `SHARED`, together with those variables which have been deduced to be shared by the compiler because they are aliasable (see section C.1.3).

The effects of disabling usage checking are best defined by what happens at a synchronization point – a point where the relative progress of two processes is known. A synchronization point is defined to be one of the following:

- A communication (on a channel, timer, or port)
- The beginning or end of a `PAR` construct.

A program using shared variables is valid provided that:

- If between two synchronization points of a process, a process reads a shared variable, then the variable is not updated by any other process at any time between these two points.

(This means that an implementation is at liberty to read the shared variable from memory after the first synchronization point, and then to cache a local copy of the variable, if it wishes.)

- If between two synchronization points of a process, a process updates a shared variable, then the variable is neither read nor updated by any other process at any time between these two points.

(This means that an implementation is at liberty to read the shared variable from memory after the first synchronization point, and then to cache a local copy of the variable, if it wishes, as long as it ensures that the variable is written back to memory before the second synchronization point.)

Each element of an array is considered to be a separate variable for the purpose of these rules.

If either of these constraints is broken, the behavior of the program is undefined. It is up to the programmer to ensure that these constraints are met.

Channels may be considered shared in the same way that variables may be. A program using shared channels is valid provided that:

- If a process communicates on a shared channel, then the channel is not used for communication in the same direction by any other process at any time between the previous and the following synchronization points.
- No branch of a **PAR** may use a channel for both input and output.

Each element of an array is considered to be a separate channel for the purpose of these rules.

If either of these constraints is broken, the behavior of the program is undefined. It is up to the programmer to ensure that these constraints are met.

If any variable or channel which is shared is passed as an actual parameter to a procedure or function, then the corresponding formal parameter must also be marked as shared. **Note:** that this also applies across separately compiled units. It is up to the programmer to ensure that this is done, otherwise the behavior of the program is undefined.

Index

Symbols

#COMMENT, 145
#IMPORT, 145
#INCLUDE, 145
#OPTION, 145
#PRAGMA, 145
 LINKAGE, 152
 SHARED, 172
#section, 152
#USE, 3, 145

Numbers

2D block move, 5

A

Abbreviation, checking, 171
Accuracy of floating point arithmetic, 21–23
ACOS, 27, 47
Alias checking, 165
 arrays, 166
 effect of disabling, 167
 rules, 165
Alignment, 156
ALOG, 28, 48
ALOG10, 29, 49
ALT, 161
ANSI screen protocol, 107
ANSI–IEEE standard 754, 20
Apollo, 95
append.char, 117
append.hex.int, 117
append.hex.int64, 117
append.int, 118
append.int64, 118
append.real32, 118
append.real64, 119

append.text, 119
Argument reduction, 21–23
Arithmetic functions
 floating point support, 12
 IEEE behavior, 6
 occam, 6
Array
 alias checking, 166
 channel, 159
 constant, 163
 of pointers, 159
 unknown size, 162
 usage checking, 170
ASIN, 30, 50
ASM, 145
ASSERT, 18
ATAN, 31, 51
ATAN2, 32, 52

B

Binary byte stream, 81, 86
Bit manipulation, 5, 9
BITCOUNT, 10
BITREVNBITS, 10
BITREWORD, 10
Block CRC library, 133
BOOL, 155
BOOLTOSTRING, 126
Buffers, 71
BYTE, 155

C

C run time library, 67
Caplin QT0, 96
CASE, 161
CAT, 96
CAUSEERROR, 17
Channel
 array, 159

- implementation, 159
 - place, 148
 - retyping, 146
 - usage checking, 170
- char.pos**, 119
- Character identification, 115
- CLIP2D**, 8
- Clock rate, 162
- CMS, 96
- Code, allocation in memory, 151
- Command line, server, 73, 85
- Communication. *See* Channel
- compare.strings**, 119
- Compiler
 - directives, 145
 - implementation restrictions, 162
 - language keywords, 145
 - libraries, 5
 - user functions, 6
 - memory allocation, 151
- Constant arrays, 163
- Constants, 4
- convert.lib**, 3, 125
- COS**, 33, 53
- COSH**, 35, 55
- CRC functions, 10, 133
- crc.lib**, 3
- CRCBYTE**, 12
- CRCFROMLSB**, 135
- CRCFROMMSB**, 134
- CRCWORD**, 11
- D**
- DACOS**, 27, 47
- DALOG**, 28, 48
- DALOG10**, 29, 49
- DASIN**, 30, 50
- DATAN**, 31, 51
- DATAN2**, 32, 52
- Date, 95
- dblmath.lib**, 3, 19, 25
- DCOS**, 33, 53
- DCOSH**, 35, 55
- DEBUG.ASSERT**, 141
- debug.lib**, 141
- debug.lib**, 3
- DEBUG.MESSAGE**, 141
- DEBUG.STOP**, 142
- Debugging support library, 141
- DEC VAX**, 95
- delete.string**, 120
- DEXP**, 36, 56
- Domain, 24
- DOS, 95
- DPOWER**, 37, 58
- DRAN**, 39, 60
- DRAW2D**, 9
- DRX-11**, 96
- DSIN**, 40, 61
- DSINH**, 42, 63
- DTAN**, 43, 64
- DTANH**, 45, 66
- Dynamic code loading, procedures, 14
- E**
- Echoed keyboard input, 90
- Elementary functions, 19, 25, 46
- End of file, 73
- Environment variables, 78
- eqstr**, 120
- Error, floating point, 21
- Escape characters, 145
- Examples, **INLINE**, 148
- Exceptions, 23
- EXP**, 36, 56
- Extended data types, 5
- Extensions, to occam, 145
- Extraordinary link handling library, 137

F

File

- access, 69
- access errors, 74
- deletion, 92
- name, length, 164
- output, 70
- positioning, 92
- renaming, 92

Floating point

- accuracy, 21–23
- arithmetic, 5
- compiler optimization, 152
- representation, 20

FMUL, 46

FRACMUL, 12

G

Generated error, 21, 24

GUY, 145

H

HELIOS, 96

HEX16TOSTRING, 126

HEX32TOSTRING, 126

HEX64TOSTRING, 127

HEXTOSTRING, 126

Host

- access, 68
- system call, 93
- versions, v

Host file server, library, 67

hostio.lib, 3

Hyperbolic functions, 42

I

IBM 370, 95

IEEE floating point arithmetic, 5

IF, 161

i1ist, 4

Implementation

- channels, 159

- occam, 151
- restrictions, 162

IMS B004, 96

IMS B008, 96

IMS B010, 96

IMS B011, 96

IMS B014, 96

IMS B015, 96

IMS B016, 96

IN, 145

inexact.NaN, 21

Infinity, 20

INLINE, 145, 148

InputOrFail.c, 138

InputOrFail.t, 138

insert.string, 120

INT, 155

INT16, 155

INT16TOSTRING, 127

INT32, 155

INT32TOSTRING, 127

INT64, 155

INT64TOSTRING, 128

INTTOSTRING, 127

is.digit, 120

is.hex.digit, 121

is.id.char, 121

is.in.range, 121

is.lower, 121

is.upper, 121

ISEARCH, 86

K

KERNEL.RUN, 14, 15

Keyboard

- input, 69, 106
- polling, 86

Keystream

- input, 102
- protocol, 101

ks, 101

ks.keystream.sink, 104

`ks.keystream.to.scrstream`, 104
`ks.read.char`, 104
`ks.read.int`, 104
`ks.read.int64`, 104
`ks.read.line`, 105
`ks.read.real32`, 105
`ks.read.real64`, 105

L

Language extensions, 145
Lexical levels, 164
Library, 3
 block CRC, 133
 compiler, 5
 debugging support, 141
 displaying, 4
 extraordinary link handling, 137
 host file server, 67
 maths, 19
 optimized T4 series, 19, 46
 string handling, 115
 type conversion, 125
Line parsing, 117
Link, failure, 137
Link handling library, 137
`LOAD.BYTE.VECTOR`, 14, 16
`LOAD.INPUT.CHANNEL`, 14, 15
`LOAD.INPUT.CHANNEL.VECTOR`, 14, 16
`LOAD.OUTPUT.CHANNEL`, 14, 16
`LOAD.OUTPUT.CHANNEL.VECTOR`, 14, 16
Long reals, 25
Lower case, 121, 123

M

MAE, 22
Maths
 functions, 6
 libraries, 19
Maximum error. *See* MAE; MRE
Memory, allocation, 151
`MemStart`, 151
`MOSTNEG INT`, 152

`MOSTPOS INT`, 151, 164
`MOVE2D`, 9
Moving the cursor, 112
MRE, 22
Multiplexors, 71

N

NaN, 20
NEC PC, 95
`next.int.from.line`, 121
`next.word.from.line`, 122
Not a number. *See* NaN

O

occam
 implementation, 151
 language extensions, 145
 libraries, 3
`OutputOrFail.c`, 138
`OutputOrFail.t`, 138

P

`PAR`, 153, 162
Parsing, 117
Parsing command line, 84
`PLACE`, 147, 159, 162
 syntax, 147
Placement
 array of channels, 148
 at address, 147
 channels, 148
 in vector space, 148
 in workspace, 148
 variables, 148
Pointer to channel, 159
Port, 162
 place at address, 147
`POWER`, 37, 58
`PRI ALT`, 161
`PRI PAR`, 153, 162, 163
 nested, 163
 replicated, 163
Primary domain, 24

Priority, 162
 Propagated error, 22, 24
 PROTOCOL, 164
 Protocol, tag, 156

R

RAN, 39, 60
 Random number generation, 39
 Range, 24
 Range reduction, 21
 REAL, 67
 Real numbers, 67
 REAL32TOSTRING, 128
 REAL64TOSTRING, 129
 Reinitialise, 139
 Replicated PAR, 153, 162
 RESCHEDULE, 17
 Resetting links, 137
 Restrictions, implementation, 162
 RETYPES, 156
 RMSAE, 22
 RMSRE, 22
 Root mean square error. *See* RMSAE; RMSRE
 ROUNDSN, 13

S

Scalar channels, 159
 Screenstream
 output, 103
 protocol, 101
 Search path, 86
 search.match, 122
 search.no.match, 123
 Server
 command line, 73, 85
 termination, 74
 SIN, 40, 61
 SINH, 42, 63
 singlmath.lib, 3, 19, 25
 so, 101
 so.ask, 72
 so.buffer, 72
 so.close, 73
 so.commandline, 73
 so.eof, 73
 so.exit, 74
 so.ferror, 74
 so.flush, 74
 so.fwrite.char, 75
 so.fwrite.hex.int, 75
 so.fwrite.hex.int32, 75
 so.fwrite.hex.int64, 76
 so.fwrite.int, 76
 so.fwrite.int32, 76
 so.fwrite.int64, 76
 so.fwrite.nl, 77
 so.fwrite.real32, 77
 so.fwrite.real64, 77
 so.fwrite.string, 78
 so.fwrite.string.nl, 78
 so.getenv, 78
 so.getkey, 79
 so.gets, 79
 so.keystream.from.file, 105
 so.keystream.from.kbd, 106
 so.keystream.from.stdin, 106
 so.multiplexor, 80
 so.open, 80
 so.open.temp, 81
 so.overlapped.buffer, 82
 so.overlapped.multiplexor, 83
 so.overlapped.pri.multiplexor, 84
 so.parse.command.line, 84
 so.pollkey, 86
 so.popen.read, 86
 so.pri.multiplexor, 87
 so.puts, 88
 so.read, 88
 so.read.echo.any.int, 88
 so.read.echo.hex.int, 89
 so.read.echo.hex.int32, 89

`so.read.echo.hex.int64`, 89
`so.read.echo.int`, 90
`so.read.echo.int32`, 90
`so.read.echo.int64`, 90
`so.read.echo.line`, 90
`so.read.echo.real32`, 91
`so.read.echo.real64`, 91
`so.read.line`, 91
`so.remove`, 92
`so.rename`, 92
`so.scrstream.to.ANSI`, 107
`so.scrstream.to.file`, 107
`so.scrstream.to.stdout`, 108
`so.scrstream.to.TVI920`, 108
`so.seek`, 92
`so.system`, 93
`so.tell`, 93
`so.test.exists`, 93
`so.time`, 94
`so.time.to.ascii`, 94
`so.time.to.date`, 94
`so.today.ascii`, 94
`so.today.date`, 95
`so.version`, 95
`so.write`, 96
`so.write.char`, 96
`so.write.hex.int`, 97
`so.write.hex.int32`, 97
`so.write.hex.int64`, 97
`so.write.int`, 97
`so.write.int32`, 98
`so.write.int64`, 98
`so.write.nl`, 98
`so.write.real32`, 98
`so.write.real64`, 99
`so.write.string`, 99
`so.write.string.nl`, 99
`ss`, 101
`ss.beep`, 108
`ss.clear.eol`, 108
`ss.clear.eos`, 109
`ss.del.line`, 109
`ss.delete.chl`, 109
`ss.delete.chr`, 109
`ss.down`, 109
`ss.goto.xy`, 109
`ss.ins.line`, 110
`ss.insert.char`, 109
`ss.left`, 110
`ss.right`, 110
`ss.scrstream.copy`, 110
`ss.scrstream.fan.out`, 110
`ss.scrstream.from.array`, 111
`ss.scrstream.multiplexor`, 111
`ss.scrstream.sink`, 111
`ss.scrstream.to.array`, 112
`ss.up`, 112
`ss.write.char`, 112
`ss.write.endstream`, 112
`ss.write.hex.int`, 112
`ss.write.hex.int64`, 112
`ss.write.int`, 113
`ss.write.int64`, 113
`ss.write.nl`, 113
`ss.write.real32`, 113
`ss.write.real64`, 113
`ss.write.string`, 114
`ss.write.text.line`, 114
Standard, maths function, 25
`str.shift`, 123
`streamio.lib`, 3
Streams, 69
String handling
 comparison, 115
 editing, 116
 library, 115
 introduction, 3
 searching, 116
`string.pos`, 122
`STRINGTOBOOL`, 130
`STRINGTOHEX`, 130
`STRINGTOHEX16`, 130

STRINGTOHEX32, 130
STRINGTOHEX64, 130
STRINGTOINT, 131
STRINGTOINT16, 131
STRINGTOINT32, 131
STRINGTOINT64, 131
STRINGTOREAL32, 131
STRINGTOREAL64, 132
 Sun, host types, 95
 SunOS, 96
 System call, 93

T

T2 special, maths function, 25
 T4 series, optimized library, 19, 46
 T8 series, maths libraries, 25
 Tabs, in occam source, 146
TAN, 43, 64
TANH, 45, 66
 Target transputer, vi
tbmaths.lib, 3, 19, 46
 Text
 reading, 79
 stream, 81, 86
 writing, 88
 Time, 94
 transputer clock, 162
 Timeout, 137
 channel input, 138
 channel output, 138
TIMER, channels, 162
 Timer, 160
to.lower.case, 123
to.upper.case, 123
 Toolset, documentation, v
 conventions, vi
 Transputer

 targets, vi
 timer, 160
 Trigonometric functions, 26
 TVI920, 108
 Type conversion library, 125
 Type mapping, 155

U

UDPlink, 96
 Ulp, 24
undefined.NaN, 20
UNPACKSN, 13
unstable.NaN, 21
 Upper case, 121, 123
 Usage checking, 169
 arrays, 170
 channels, 170
 disabled, 172

V

Variable
 place at address, 147
 place in memory, 148
VECSPACE, 145, 147
 Vector space, 151, 153, 154
VMS, 96
VSSIZEOF, 14, 17

W

WORKSPACE, 145, 147
 Workspace, 151, 152
WSSIZEOF, 14, 17

X

xlink.lib, 3, 137