inmos®

ANSI C Language and Libraries Reference Manual

INMOS Limited

SGS-THOMSON MICROELECTRONICS

INMOS is a member of the SGS–THOMSON Microelectronics Group

72 TDS 347 01

October 1992

The C compiler implementation was developed from the Perihelion Software "C" Compiler and the Codemist Norcroft "C" Compiler.

INMOS Document Number: 72 TDS 347 01

Contents overview

Contents

Preface

Host versions

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

- IMS D7314 – IBM PC compatible running MS–DOS
- IMS D4314 – Sun 4 systems running SunOS.
- IMS D6314 – VAX systems running VMS.

About this manual

This manual is the *Language and Libraries Reference Manual* to the ANSI C toolset and provides a language reference for the toolset and implementation data.

The manual is divided into two parts: '*Runtime Library*' and '*Language Reference*', plus appendices.

The first section *Runtime Library*:

- introduces the runtime library and summarizes the header files;
- provides a detailed description of each library function, in alphabetical order;
- describes how to modify the runtime startup system by removing segments not required by the user's application. Only very experienced users should attempt this.

The '*Language Reference*' section describes:

- new features in the ANSI standard;
- ANSI C toolset language extensions;
- ANSI C toolset implementation details.

The three appendices cover:

- syntax of language extensions;
- ANSI compliance data;
- further explanation of the cyclic redundancy function provided.

About the toolset documentation set

The documentation set comprises the following volumes:

- *72 TDS 345 01 ANSI C Toolset User Guide*

  Describes the use of the toolset in developing programs for running on the transputer. The manual is divided into two sections; '*Basics*' which describes each of the main stages of the development process and 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. Several of the chapters are generic to other INMOS toolsets.

- *72 TDS 346 01 ANSI C Toolset Reference Manual*

  Provides reference material for each tool in the toolset including command line options, syntax and error messages. Many of the tools in the toolset are generic to other INMOS toolset products i.e. the occam and FORTRAN toolsets and the documentation reflects this. Examples are given in C. The appendices provide details of toolset conventions, transputer types, the assembler, server protocol, ITERM files and bootstrap loaders.

- *72 TDS 347 01 ANSI C Language and Libraries Reference Manual* – (this manual)

- *72 TDS 348 01 ANSI C Optimizing Compiler User Guide*

  Provides reference and user information specific to the ANSI C optimizing compiler. Examples of the type of optimizations available are provided in the appendices. This manual should be read in conjunction with the reference chapter for the standard ANSI C compiler, provided in the *Tools Reference Manual.*

- *72 TDS 354 00 Performance Improvement with the DX314 ANSI C Toolset*

  This document provides advice about how to maximize the performance of the toolset. It brings together information provided in other toolset documents particularly from the *Language and Libraries Reference Manual.* **Note:** details of how to manipulate the software virtual through-routing mechanism are given in the *User Guide.*

- *72 TDS 355 00 ANSI C Toolset Handbook*

  A separately bound reference manual which lists the command line options for each tool and the library functions. It is provided for quick reference and summarizes information provided in more detail in the *Tools Reference Manual* and the *Language and Libraries Reference Manual.*

- *72 TDS 360 00 ANSI C Toolset Master Index*

  A separately bound master index which covers the *User Guide*, *Toolset Reference Manual*, *Language and Libraries Reference Manual*, *Optimizing Compiler User Guide* and the *Performance Improvement* document.

Other documents

Other documents provided with the toolset product include:

- Delivery manual giving installation data, this document is host specific.
- Release notes, common to all host versions of the toolset.

occam and FORTRAN toolsets

At the time of writing the occam and FORTRAN toolset products referred to in this document set are still under development and specific details relating to them are subject to change. Users should consult the documentation provided with the corresponding toolset product for specific information on that product.

Documentation conventions

The following 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.
Italic type	In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.
Braces { }	Used to denote optional items in command syntax.
Brackets [ ]	Used in command syntax to denote optional items on the command line.
Ellipsis . . .	In general terms, used to denote 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.

Runtime Library

1 Introduction and runtime library summary

This chapter introduces the ANSI C runtime library. It describes the library header files that contain the function declarations, explains how to use them, and lists the contents of each file. The chapter ends with a list of runtime errors which may occur.

1.1 Introduction

The ANSI C runtime library is a library of functions which perform common programming operations such as file input/output (I/O) and mathematical transformations. The library supplied with the toolset is a full ANSI standard library with additional support for parallel processing, channel communication, and semaphore handling. Some additional non-ANSI functions are also provided, including `float` versions of the standard mathematical functions, low level file handling functions, and a variety of miscellaneous operations.

A number of *header files* are provided. These contain prototypes for every function in the library, along with useful macros and constants.

Two versions of the ANSI C runtime library are supplied: the full libraries and the reduced libraries.

The full libraries provide access to the host environment via the `iserver`. Thus a file system is available along with other host resources. Communication with the `iserver` is achieved via a pair of host link channels, one coming from the server and one going to the server. Access to these channels is protected by semaphore thus ensuring that communication is not corrupted by concurrent accesses. Such protection cannot be guaranteed if the channels are written to directly.

The reduced library can be thought of as a subset of the full library. It is modified so that routines which require access to the `iserver` in order to carry out their prime function, e.g. file handling routines, are omitted. Other routines which access the `iserver` for secondary reasons, e.g. `exit` when closing files on program termination, are modified so that `iserver` accesses are omitted. The host link channels are not defined for the reduced library. Thus when direct communication with the `iserver` is not required or possible then the reduced library should be used, if the full library is used instead then the behavior of the program is undefined as an `iserver` access may be attempted when no `iserver` is present.

Note: Programs linked with the reduced library must be collected from a configuration binary file, that is, the programs must be *configured*.

1.1.1 Accessing library functions

Library functions must be declared like any other C function, and is simply performed by including the appropriate header file; the correct file to include can be determined from the function synopsis (see chapter 2).

1.1.2 Linking libraries with programs

Function code is incorporated with the program by linking in the appropriate library file.

Several linker indirect files are supplied to aid linking with the C runtime library. Their primary use is to specify the set of C library files required when linking a C program (or a mixed language program which uses C). These linker indirect files and their application are described in detail in section 3.11 of the *ANSI C Toolset User Guide*.

1.1.3 `iserver` protocols

All functions in the library use the communication protocols of the the host file server to perform program I/O. These protocols are invisible to the C applications programmer. `iserver` protocol and its underlying functions are described in appendix D '*iserver protocol*' of the *ANSI C Toolset Reference Manual.*

The library function `server_transaction` provides access to low level `iserver` functions.

1.1.4 Functions which store data in `static`

Certain functions in the Runtime Library store data in the static area. If these functions are called simultaneously by two concurrent processes there may be contention for the same data and return values may be unpredictable.

For example:

`getenv` stores the string associated with an environment variable in the static area. If process '`A`' calls `getenv` for environment variable '`ENVA`', then the string associated with '`ENVA`' is stored in static. Consider now that process '`A`' is descheduled and a second process, '`B`' starts, which then calls `getenv` for '`ENVB`'. Now the string for '`ENVB`' is stored in static, overwriting the string for '`ENVA`'. If process '`A`' now restarts and attempts to use the pointer returned by `getenv` to access '`ENVA`', it will find that it actually reads '`ENVB`'.

Functions which should be used with great care in concurrently executing processes are as follows:

`asctime`	`getenv`	`localtime`	`rand`	`set_abort_action`
`signal`	`stdlib`	`strerror`	`strtok`	`tmpnam`

More information about the the use of these functions can be found under the detailed function descriptions in chapter 2.

The global variable `errno` should also be used with great care in a concurrent environment since there is no protection on its assignment.

1.2 Header files

Header files contain functions declarations, macros, and other definitions grouped together for convenient reference in a program. Header files generally contain declarations of related functions along with definitions of supporting constants and macros. Header files may consist only of macros and constants, for example, `limits.h`.

Header files supplied with the ANSI C toolset are listed in Table 1.1.

Header file	Description
`assert.h*`	Diagnostics.
`bootlink.h`	Boot link channel information.
`channel.h`	Channel handling.
`ctype.h*`	Character handling and manipulation.
`dos.h`	DOS specific operations.
`errno.h*`	Error handling.
`float.h*`	Characteristics of floating types.
`fnload.h`	Dynamic code loading functions.
`host.h`	Host system information.
`hostlink.h`	Host channel information.
`iocntrl.h`	Low level file handling.
`limits.h*`	Language implementation limits.
`locale.h*`	Locale specific data.
`math.h*`	Maths and trig functions.
`mathf.h`	`float` versions of maths and trig functions.
`misc.h`	Miscellaneous functions.
`process.h`	Process startup, handling, and control.
`semaphor.h`	Semaphore handling.
`setjmp.h*`	Non-local jumps.
`signal.h*`	Signal handling.
`stdarg.h*`	Variable argument handling.
`stddef.h*`	Standard definitions.
`stdio.h*`	Standard I/O and file handling.
`stdiored.h`	Reduced library string formatting functions.
`stdlib.h*`	General programming utilities.
`string.h*`	String handling and manipulation.
`time.h*`	System clock date and time.
* ANSI standard files	

Table 1.1 ANSI C toolset header files

The rest of this chapter describes the contents of the header files and is divided into three sections covering the three main groups of files: ANSI standard functions; Concurrency functions; and Other functions.

1.3 ANSI functions

ANSI functions are contained in a series of header files defined in the ANSI standard. They encompass standard function sets such as file I/O, maths and trig functions, character and string handling, error handling, and many other functions in common usage within existing non-ANSI environments.

1.3.1 Diagnostics <assert.h>

The header file `assert.h` contains a single macro definition:

Macro	Description
`assert`	Inserts diagnostics into the program.

The definition of `assert` depends upon the value of the macro `NDEBUG`, which is not itself defined in `assert.h`.

1.3.2 Character handling <ctype.h>

The header file `ctype.h` declares a set of functions for character identification and manipulation.

Function	Description
`isalnum`	Determines whether a character is alphanumeric.
`isalpha`	Determines whether a character is alphabetic.
`iscntrl`	Determines whether a character is a control character.
`isdigit`	Determines whether a character is a decimal digit.
`isgraph`	Determines whether a character is a printable non–space character.
`islower`	Determines whether a character is a lower–case letter.
`isprint`	Determines whether a character is a printable character (including space).
`ispunct`	Determines whether a character is a punctuation character.
`isspace`	Determines whether a character is one which affects spacing.
`isupper`	Determines whether a character is an upper–case letter.
`isxdigit`	Determines whether a character is a hexadecimal digit.
`tolower`	Converts an upper–case letter to its lower–case equivalent.
`toupper`	Converts an lower–case letter to its upper–case equivalent.

1.3.3 Error handling <errno.h>

The header file `errno.h` declares the error variable `errno` and defines codes for the values to which it may be set. The file also contains a number of other error codes, not listed here, which are included for compatibility with earlier INMOS compiler toolsets.

Variable	Description
`errno`	A variable of type `volatile int`. Set to a positive error codes by several library routines.

Macro	Description
EDOM	The argument to a maths function is out of range.
ERANGE	Overflow or underflow in a maths function.
ESIGNUM	Illegal signal number supplied to `signal`.
EIO	Error in low level I/O function used to communicate with the server.
EFILPOS	Error in file positioning functions `ftell`, `fgetpos`, or `fsetpos`.

1.3.4 Floating point constants <float.h>

Macro	Description
FLT_RADIX	Radix of exponent representation.
FLT_ROUNDS	Rounding mode for floating point addition.
FLT_MANT_DIG	Number of digits in a `float` mantissa.
DBL_MANT_DIG	`double` form of `FLT_MANT_DIG`.
LDBL_MANT_DIG	`long double` form of `FLT_MANT_DIG`.
FLT_EPSILON	Minimum number of type float such that 1.0 + x != 1.0
DBL_EPSILON	`double` form of `FLT_EPSILON`
LDBL_EPSILON	`long double` form of `FLT_EPSILON`
FLT_DIG	Number of decimal digits of precision for `float` parameters.
DBL_DIG	`double` form of `FLT_DIG.`
LDBL_DIG	`long double` form of `FLT_DIG.`
FLT_MIN_EXP	Minimum `float` exponent.
DBL_MIN_EXP	`double` form of `FLT_MIN_EXP`
LDBL_MIN_EXT	`long double` form of `FLT_MIN_EXP`
FLT_MIN	Minimum normalized positive number of type float.
DBL_MIN	`double` form of `FLT_MIN`
LDBL_MIN	`long double` form of `FLT_MIN`
FLT_MIN_10_EXP	Minimum negative integer such that 10 raised to that power is a normalized `float` number.
DBL_MIN_10_EXP	`double` form of `FLT_MIN_10_EXP`
LDBL_MIN_10_EXP	`long double` form of `FLT_MIN_10_EXP`
FLT_MAX_EXP	Maximum integer such that `FLT_RADIX` raised to that power minus 1 is a valid `float` number.
DBL_MAX_EXP	`double` form of `FLT_MAX_EXP`
LDBL_MAX_EXP	`long double` form of `FLT_MAX_EXP`

Macro	Description
`FLT_MAX`	Maximum representable number of type `float`
`DBL_MAX`	`double` form of `FLT_MAX`
`LDBL-MAX`	`long double` form of `FLT_MAX`
`FLT_MAX_10_EXP`	Maximum integer such that 10 raised to that power is a valid `float` number.
`DBL_MAX_10_EXP`	`double` form of `FLT_MAX_10_EXP`
`LDBL_MAX_10_EXP`	`long double` form of `FLT_MAX_10_EXP`

1.3.5 Implementation limits `<limits.h>`

`limits.h` defines a number of implementation constants in ANSI C.

Macro	Description
`CHAR_BIT`	The number of bits in a byte.
`SCHAR_MIN`	Minimum value for an object of type `signed char`
`SCHAR_MAX`	Maximum value for an object of type `signed char`
`UCHAR_MAX`	Maximum value for an object of type `unsigned char`
`CHAR_MIN`	Minimum value for an object of type `char`.
`CHAR_MAX`	Maximum value for an object of type `char`.
`SHRT_MIN`	Minimum value for an object of type `short int`.
`SHRT_MAX`	Maximum value for an object of type `short int`.
`USHRT_MAX`	Maximum value for an object of type `unsigned short int`.
`INT_MIN`	Minimum value for an object of type `int`.
`INT_MAX`	Maximum value for an object of type `int`
`UINT_MAX`	Maximum value for an object of type `unsigned int`.
`LONG_MIN`	Minimum value for an object of type `long int`.
`LONG_MAX`	Maximum value for an object of type `long int`.
`ULONG_MAX`	Maximum value for an object of type `unsigned long int`.
`MB_LEN_MAX`	Maximum number of bytes in a multibyte character.

1.3.6 Localization `<locale.h>`

The header file `locale.h` defines two functions, some macros for use by `setlocale`, and a single structure.

Function	Description
`localeconv`	Assigns appropriate values to components in objects of type `struct lconv` for the formatting of numeric quantities, according to the rules of the current locale.
`setlocale`	Sets or interrogates part of the program's locale.

Macro	Description
`LC_ALL`	Names the entire locale (that is, all of the following macros).
`LC_COLLATE`	Used in the string locale functions `strcoll` and `strxfrm`.
`LC_CTYPE`	Used in the character handling functions
`LC_NUMERIC`	Selects the decimal point.
`LC_TIME`	Used in the locale dependent time functions.
`LC_MONETARY`	Affects monetary formatting information returned by the `localeconv` function.

Structure	Description
`lconv`	A structure which describes a complete locale.

INMOS ANSI C supports only the standard "C" locale, which has the following features:

- The execution character set comprises all 256 values 0 to 255. Values 0 to 127 represent the ASCII character set. **Note:** when the compiler command line option '`FC`' is used the execution character set comprises 128 values in the range 0 to 127.
- The collation sequence of the execution character set is the same as for plain ASCII.
- Printing is from left to right.
- The decimal point character is '.'.

No other locales are permitted.

1.3.7 Mathematics library <math.h>

`math.h` declares general maths functions and their associated constants.

Note: the following is true for all functions declared in `math.h`:

On domain errors:	`errno` is set to `EDOM`; 0.0 is returned.
On range errors:	`errno` is set to `ERANGE`; `HUGE_VAL` is returned for overflow errors; `-HUGE_VAL` is returned for underflow errors.

Function	Description
`acos`	Calculates the arc cosine of the argument.
`asin`	Calculates the arc sine of the argument
`atan`	Calculates the arc tangent of the argument.
`atan2`	Calculates the arc tangent of argument 1 divided by argument 2.
`ceil`	Calculates the smallest integer which is not less than the argument.
`cos`	Calculates the cosine of the argument.
`cosh`	Calculates the hyperbolic cosine of the argument.
`exp`	Calculates the exponential of the argument.
`fabs`	Calculates the absolute value of a floating point number.
`floor`	Calculates the largest integer which is not greater than the argument.
`fmod`	Calculates the floating point remainder of argument 1 divided by argument 2.
`frexp`	Separates a floating point number into a mantissa and an integral power of 2.
`ldexp`	Multiplies a floating point number by an integer power of 2.
`log`	Calculates the natural logarithm of the argument.
`log10`	Calculates the base 10 logarithm of the argument.
`modf`	Splits the argument into fractional and integral parts
`pow`	Calculates x to the power y.
`sin`	Calculates the sine of the argument.
`sinh`	Calculates the hyperbolic sine of the argument
`sqrt`	Calculates the square root of the argument.
`tan`	Calculates the tangent of the argument.
`tanh`	Calculates the hyperbolic tangent of the argument.

Macro	Value
`HUGE_VAL`	A constant value returned if overflow or underflow occurs.

1.3.8 Non–local jumps `<setjmp.h>`

The header file setjmp.h declares two functions used to perform non-local gotos, and a single type used by them.

Function	Description
`longjmp`	Performs a non-local jump to a given environment.
`setjmp`	Sets up a non-local jump.

The two functions are used in conjunction to first set a position (`setjmp`), then jump to this position (`longjmp`). When `longjmp` executes, it appears to the user as if the program had just returned from the call to the associated `setjmp`.

Type	Meaning
`jmp_buf`	An array type used to save a calling environment.

1.3.9 Signal handling `<signal.h>`

The header file `signal.h` defines two functions for signal handling, one type, and several constants.

Function	Description
`raise`	Forces a pseudo-exception via the signal handler.
`signal`	Defines the way in which errors and exceptions are handled.

Type	Description
`sig_atomic_t`	Defines an atomic variable. This is a variable whose state is always known, and which cannot be confused by asynchronous interrupts.

Macro	Description
`SIG_DFL`	Uses the default system error/exception handling for the predefined value.
`SIG_IGN`	Ignores the error/exception.
`SIG_ERR`	Returned when the signal handler is invoked in error.
`SIGABRT`	Abort error.
`SIGFPE`	Arithmetic exception.
`SIGILL`	Illegal instruction.
`SIGINT`	Attention request from user.
`SIGSERV`	Bad memory access.
`SIGSTERM`	Termination request.
`SIGIO`	Input/output possible.
`SIGURG`	Urgent condition on I/O channel.
`SIGPIPE`	Write on pipe with no corresponding read.

Macro	Description
`SIGSYS`	Bad argument to system call.
`SIGALRM`	Alarm clock.
`SIGWINCH`	Window changed.
`SIGLOST`	Resource lost.
`SIGUSR1`	User defined signal.
`SIGUSR2`	User defined signal.
`SIGUSR3`	User defined signal.

1.3.10 Variable arguments <`stdarg.h`>

The header file `stdarg.h` contains a three macros and a type definition.

Macro	Description
`va_arg`	Accesses a member of a variable argument list.
`va_end`	Clears up after accessing variable arguments.
`va_start`	Initializes a pointer to a variable number of function arguments in a function definition.

Type	Description
`va_list`	A type used to hold information required by the variable argument functions.

1.3.11 Standard definitions <`stddef.h`>

The header file `stddef.h` defines a number of commonly used data types and macros.

Type	Description
`prtdiff_t`	The signed integral type of the result of subtracting two pointers.
`size_t`	The unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	An integral type whose range of values can represent distinct codes for all members of the largest extended character set amongst the supported locales.

Macro	Description
`NULL`	A null pointer constant which is returned by many library routines.

`offsetof(type, id)`	Expands to an integral constant expression that has type `size_t`. The value is the offset in bytes from the beginning of a structure, designated by `type`, of `id`.
	For example:
	`struct item`
	`{`
	`long int x;`
	`long int y;`
	`};`
	
	`offsetof(struct item, y)` = 4

1.3.12 Standard I/O `<stdio.h>`

The header file `stdio.h` defines the main I/O and file handling functions, three types, and several macros.

Function	Description
`clearerr`	Clears the error and end-of-file indicators for a file stream.
`fclose`	Closes a file stream.
`feof`	Tests the state of the end-of-file indicator.
`ferror`	Tests the state of the file error indicator.
`fflush`	Flushes an output stream.
`fgetc`	Reads a character from a file stream.
`fgetpos`	Gets the position of the read/write file pointer.
`fgets`	Reads a line from a file stream.
`fopen`	Opens a file.
`fprintf`	Writes a formatted string to a file.
`fputc`	Writes a character to a file stream.
`fputs`	Writes a string to a file stream.
`fread`	Reads records from a file.
`freopen`	Closes an open file, and re-opens it in a given mode.
`fscanf`	Reads formatted input from a file stream.
`fseek`	Sets the read/write file pointer to a specified offset in a file stream.
`fsetpos`	Sets the read/write file pointer to a position obtained from `fgetpos`.
`ftell`	Gives the position of the read/write pointer in the file stream.
`fwrite`	Writes records from an array into a file.

Function	Description
getc	Gets a character from a file.
getchar	Reads a character from standard input.
gets	Gets a line from standard input.
perror	Writes an error message to the standard error output.
printf	Writes a formatted string to standard output.
putc	Writes a character to a file stream.
putchar	Writes a character to standard output.
puts	Writes a line to standard output.
remove	Removes access to a file.
rename	Renames a file.
rewind	Sets the file stream's read/write position pointer to the start of the file.
scanf	Reads formatted data from standard input.
setbuf	Controls file buffering.
setvbuf	Defines the way that a file stream is buffered.
sprintf	Writes a formatted string to a string.
sscanf	Reads formatted data from a string.
tmpfile	Creates a temporary file.
tmpnam	Creates a unique filename.
ungetc	Pushes a character back onto a file stream.
vfprintf	Writes a formatted string to a file (alternative form of `fprintf`).
vprintf	Writes a formatted string to standard output (alternative form of `printf`).
vsprintf	Writes a formatted string to a string (alternative form of `sprintf`).

Type	Description
FILE	Defines a type used for recording all the information that the system needs to control a file stream.
fpos_t	Defines a structure able to hold a unique specification of every position within a file.
size_t	The unsigned integral type of the result of the `sizeof` operator.

Macro	Description
NULL	A null pointer constant that is returned by many routines.

The first group of three macros in the following list define integral constants which may be used to control the action of `setvbuf`; the next three macros define integral constants which may be used to control the action of `fseek`, and the remainder in the list are used throughout the I/O library:

Macro	Description
`_IOFBF`	Full I/O buffering required.
`_IOLBF`	Line buffering required.
`_IONBF`	No I/O buffering required.
`SEEK_SET`	Start seek at start of file stream.
`SEEK_CUR`	Start seek at current position in file stream.
`SEEK_END`	Start seek at end of file stream.
`BUFSIZ`	The buffer size used by `setbuf`.
`EOF`	End of file character.
`L_tmpnam`	The size of an array used to hold temporary file names generated by `tmpnam`.
`TMP_MAX`	The maximum number of unique file names generated by `tmpnam`.
`FOPEN_MAX`	The minimum number of files that can be open simultaneously.
`FILENAME_MAX`	Maximum length of filename.

Characteristics of file handling

File handling by works on *streams* and has the following features:

- File naming follows the conventions of the host system.
- Zero length files can exist if they are permitted by the host system.
- The same file can be opened multiple times. However, because there is no support for shared access within `stdio.h` the results may be unpredictable.
- In append mode the file position indicator is initially positioned at the end of the file.
- Spaces written out to a file before the newline character are also read in.
- The last line of a text stream does not require a terminating newline character.
- A write on a text stream does not cause the associated file to be truncated beyond that point.
- No NULL characters are appended to data written to a binary stream.
- The features of file buffering are as follows:
  - In *unbuffered* streams characters appear from the source or destination as soon as possible. Transmission of characters also occurs if input is specifically requested.

- In *line- buffered* streams a block of characters is built up and then sent to the host system when a newline character occurs. Transmission also occurs if input is specifically requested.
- In fully buffered streams a block of characters is sent to the host system when the buffer becomes full.

In all buffering modes characters are also transmitted if the buffer becomes full, or if the stream is explicitly flushed.

1.3.13 Reduced library I/O functions `<stdiored.h>`

The file `stdiored.h` contains declarations of three print formatting functions from `stdio.h`. They are for use in programs linked with the reduced runtime library.

Function	Description
`sprintf`	Writes a formatted string to a string.
`sscanf`	Reads formatted data from a string.
`vsprintf`	Writes a formatted string to a string (alternative form of `sprintf`.

1.3.14 General utilities `<stdlib.h>`

The header file `stdlib.h` contains general programming utilities and associated data types, constants, and macros. Many of the functions are implemented as macros.

Note: the functions `mblen`, `mbtowc`, `mbstowcs`, `wctomb` and `wcstombs` provide a minimal implementation of the ANSI standard.

This is considered sufficient because the current toolset supports only the standard C locale, and therefore any implementation is of limited practical value.

The functions support an implementation of wide characters in which:

```
wchar_t == int
MB_MAX_LEN == 1
```

Function	Description
abort	Causes the program to abort. The abort is equivalent to an abnormal termination of the program.
abs	Calculates the absolute value of an integer.
atexit	Specifies a function to be called when the program ends.
atof	Converts a string of characters to a `double`.
atoi	Converts a string to an `int`.
atol	Converts a string to a `long int`.
bsearch	Searches a sorted array for a given object.
calloc	Allocates memory space for an array of items and initializes the space to zeros.
div	Calculates the quotient and remainder of a division.
exit	Causes normal program termination.
free	Frees an area of memory.
getenv	Obtains the value of an environment variable from the host.
labs	Calculates the absolute value of a `long int`.
ldiv	Calculates the quotient and remainder of a long division.
malloc	Allocates a specified area of memory.
mblen	Determines the number of bytes in a multibyte character.
mbtowc	Converts a multibyte char to a code of type `wchar_t.`
mbstowcs	Converts a sequence of multibyte characters to a sequence of codes of type `wchar_t`
qsort	Sorts an array of objects.
rand	Generates a pseudo-random number.
realloc	Changes the size of an object in memory.
srand	Sets the seed for pseudo-random numbers generated by rand.
strtod	Converts the initial part of a string to a `double` and saves a pointer to the rest of the string.
strtol	Converts the initial part of a string to a `long int` and saves a pointer to the rest of the string.
strtoul	Converts the initial part of a string to an `unsigned long int` and saves a pointer to the rest of the string.
system	Passes a string to the host environment for execution as a host command.
wctomb	Converts a code of type `wchar_t` to a multibyte character.
wcstombs	Opposite of `mbstowcs`. Converts a sequence of codes of type `wchar_t` to a sequence of multibyte characters.

Type	Description
`size_t`	The unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	An integral type whose range of values can represent distinct codes for all members of the largest extended character set amongst the supported locales.
`div_t`	The type returned by `div`.
`ldiv_t`	The type returned by `ldiv`.

Macro	Description
`NULL`	A null pointer constant which is returned by many library routines.
`EXIT_FAILURE`	An integral expression which may be used as an argument to the `exit` function to return unsuccessful termination status to the Host environment.
`EXIT_SUCCESS`	As `EXIT_FAILURE` but for successful termination
`RAND_MAX`	Maximum value returned by `rand` function.
`MB_CUR_MAX`	Maximum number of bytes in a multibyte character.

1.3.15 String handling <`string.h`>

The header file `string.h` declares a number of string handling functions, and one type.

Function	Description
`memchr`	Finds the first occurrence of a character in the first *n* characters of an area of memory.
`memcmp`	Compares the first *n* characters of two areas of memory.
`memcpy`	Copies characters from one area of memory to another (no memory overlap allowed).
`memmove`	Copies characters from one area of memory to another (the areas can overlap).
`memset`	Fills a given area of memory with the same character.
`strcat`	Appends one string onto another.
`strchr`	Finds the first occurrence of a character in a string.
`strcmp`	Compares two strings.
`strcoll`	Compares two strings (transformed according to the program's locale).
`strcpy`	Copies one string to another.
`strcspn`	Counts the number of characters at the start of one string which do not match any of the characters in another string.
`strerror`	Converts an error number into an error message string.
`strlen`	Calculates the length of a string.
`strncat`	Appends one string onto another (up to a maximum number of characters).
`strncmp`	Compares the first *n* characters of two strings.
`strncpy`	Copies one string to another (up to a maximum number of characters).
`strpbrk`	Finds the first character in one string that is present in another string.
`strrchr`	Finds the last occurrence of a given character in a string.
`strspn`	Counts the number of characters at the start of a string which are also in another string.
`strstr`	Finds the first occurrence of one string in another.
`strtok`	Converts a string consisting of delimited tokens into a series of strings with the delimiters removed.
`strxfrm`	Transforms a string according to the locale and copies it into an array (up to a maximum number of characters).

Type	Description
`size_t`	The unsigned integral type of the result of the `sizeof` operator.

Macro	Description
`NULL`	A null pointer constant which is returned by many library routines.

1.3.16 Date and time `<time.h>`

The header file `time.h` declares a number of functions for manipulating time, four types, and some time and date constants.

In all the following functions the local time zone is defined by the host system. Daylight Saving Time is not available.

Function	Description
`asctime`	Converts the values in a *broken-down* time structure to an ASCII string. (See below).
`clock`	Calculates the amount of processor time used.
`ctime`	Converts a calendar time to a string.
`difftime`	Calculates the difference between two calendar times.
`gmtime`	Converts a calendar time to a broken-down time, expressed as coordinated universal time (UTC time). Always returns `NULL`, because UTC time is not available in this implementation.
`localtime`	Converts a calendar time into a broken-down time structure format.
`mktime`	Converts a broken-down structure into a `time_t` value.
`strftime`	Does a formatted conversion of a broken-down time structure to a string.
`time`	Reads the current time.

Type	Description
`size_t`	The unsigned integral type of the result of the `sizeof` operator.
`clock_t`	Used to store times in the form of processor clock ticks per second.
`time_t`	Used to store times in a fixed format.
`struct tm`	A structure representing a broken-down time.

Macro	Description
`NULL`	A null pointer constant which is returned by many library routines.
`CLOCKS_PER_SEC`	The number of processor clock ticks per second (priority sensitive).

Some functions declared in `time.h` act on broken-down times. A broken-down time is represented as a structure as follows:

```
struct tm {
  int tm_sec; /* Secs after min [0,61] */
  int tm_min; /* Mins after hour [0,59] */
  int tm_hour; /* Hours since midnight [0,23] */
  int tm_mday; /* Day of month [1,31] */
  int tm_mon; /* Months since Jan [0,11] */
  int tm_year; /* Years since 1900 */
  int tm_wday; /* Days since Sunday [0,6] */
  int tm_yday; /* Days since Jan 1 [0,365] */
  int tm_isdst; /* Daylight saving flag */
}
```

1.4 Concurrency functions

Concurrency support in the runtime library is separated into three header files: `process.h` which contains functions to set up, run, and control concurrent processes with associated constants; `channel.h` which contains functions for communicating along channels with associated channel constants such as link addresses; and `semaphor.h` which contains the semaphore support functions.

1.4.1 Process control <process.h>

Function	Description
ProcAfter	Delays execution of a process until after a specified time.
ProcAlloc	Allocates stack space and initializes a process.
ProcAllocClean	Cleans up after a process created using ProcAlloc.
ProcAlt	Causes a process to wait for a ready input from a series of channels. Channels are referenced by pointers.
ProcAltList	As ProcAlt but references an array of channel pointers.
ProcGetPriority	Returns the priority of the current process.
ProcInit	Initializes a process.
ProcInitClean	Cleans up after a process created using ProcInit.
ProcJoin	Waits for a list of asynchronous processes to terminate.
ProcJoinList	Waits for a list (passed as an array) of asynchronous processes to terminate.
ProcPar	Starts a number of synchronized processes in parallel.
ProcParam	Alters process parameters.
ProcParList	As ProcPar but takes a list passed as an array of processes.
ProcPriPar	Starts two processes in parallel, the first being executed at high priority and the second at low priority.
ProcReschedule	Reschedules a process, that is, places it on the end of the process queue.
ProcRun	Starts a process at the same priority as the calling process (the *current* priority).
ProcRunHigh	Starts a high priority process.
ProcRunLow	Starts a low priority process.
ProcSkipAlt	Similar to ProcAlt but does not wait if there are no channels are ready.
ProcSkipAltList	As ProcSkipAlt but takes an array of pointers to channels.
ProcStop	Stops a process.
ProcTime	Reads the transputer clock.
ProcTimeAfter	Determines the sequence of two transputer clock times.
ProcTimeMinus	Gives the difference between two transputer clock times.
ProcTimePlus	Gives the result of adding two transputer clock times.
ProcTimerAlt	As ProcAlt but uses a timeout.
ProcTimerAltList	As ProcAltList but uses a timeout.
ProcWait	Delays execution of a process for a specified time.

Type	Description
`Process`	A structure that holds all the information about a concurrent process.

Constant	Description
`PROC_HIGH`	The value returned by `ProcGetPriority` for a high priority process.
`PROC_LOW`	The value returned by `ProcGetPriority` for a low priority process.
`CLOCKS_PER_SEC_HIGH`	Number of processor clock ticks per second for a high priority process.
`CLOCKS_PER_SEC_LOW`	Number of processor clock ticks per second for a low priority process.

1.4.2 Channel communication <channel.h>

Function	Description
`ChanAlloc`	Allocates and initializes a channel.
`ChanIn`	Inputs a message on a channel.
`ChanInChanFail`	As `ChanIn` but incorporates the ability to reset a channel on receipt of a message sent on another channel (such as a link failure condition).
`ChanInChar`	Inputs a byte on a channel.
`ChanInit`	Initializes a channel.
`ChanInInt`	Inputs an integer on a channel.
`ChanInTimeFail`	As `ChanIn` but incorporates a timeout after which the channel is reset if no communication occurs.
`ChanOut`	Outputs a message on a channel.
`ChanOutChanFail`	As `ChanInChanFail` but for output channels.
`ChanOutChar`	Outputs a byte on a channel.
`ChanOutInt`	Outputs an integer on a channel.
`ChanOutTimeFail`	As `ChanInTimeFail` but for output channels.
`ChanReset`	Resets a channel.
`DirectChanIn` †	Input a message on a channel.
`DirectChanInChar` †	Input a byte on a channel.
`DirectChanInInt` †	Input an integer on a channel.
`DirectChanOut` †	Output a message on a channel.
`DirectChanOutChar` †	Output a byte on a channel.
`DirectChanOutInt` †	Output an integer on a channel.
† `Direct...` functions may not be used in all situations that their counterpart `Chan...` functions can. See chapter 2 for detailed descriptions.	

Type	Description
Channel	The channel type.

Constant	Description
NotProcess_p	A special value used in process communication and scheduling. Returned by ChanReset.
LINK0OUT	Link zero output address.
LINK1OUT	Link one output address.
LINK2OUT	Link two output address.
LINK3OUT	Link three output address.
LINK0IN	Link zero input address.
LINK1IN	Link one input address.
LINK2IN	Link two input address.
LINK3IN	Link three input address.
EVENT	Event line address.

1.4.3 Semaphore handling <semaphor.h>

Function	Description
SemInit	Initializes a semaphore.
SemAlloc	Allocates and initializes a semaphore.
SemSignal	Releases a semaphore.
SemWait	Acquires a semaphore.

Type	Description
Semaphore	Defines a semaphore type.

Macro	Description
SEMAPHOREINIT	Initializes a semaphore (same action as SemInit but implemented as a macro).

1.5 Other functions

The header files iocntrl.h, mathf.h, host.h, hostlink.h, bootlink.h, dos.h, fnload.h and misc.h contain some further extensions to the ANSI run-time library. These include UNIX-like I/O primitives; short maths functions; host system utilities, host channel access utilities; DOS specific functions; dynamic code loading functions and miscellaneous functions including debugging support for idebug.

1.5.1 I/O primitives <iocntrl.h>

Function	Description
`close`	Low level file close.
`creat`	Low level file create.
`filesize`	Returns the size of a given file.
`getkey`	Gets the next character from the keyboard. Waits indefinitely for the next key press. Does not echo the character to the screen.
`isatty`	Checks for terminal files.
`lseek`	Low level file seek.
`open`	Low level file open.
`pollkey`	Gets the next character from the keyboard. Returns immediately if no key press is available. Does not echo the character to the screen.
`read`	Low level read-from-file.
`server_transaction`	Allows access to `iserver` functions in a controlled way.
`unlink`	Low level file remove (corresponds to ANSI standard function `remove`).
`write`	Low level write-to-file.

The following macros are defined to control `lseek`:

Macro	Description
`L_SET`	Seek from start of file.
`L_INCR`	Seek from current position.
`L_XTND`	Seek from end of file.

The following macros which define the mode in which a file is opened, are used by `creat` and `open`:

Macro	Description
`O_RDONLY`	Open file in read only mode.
`O_WRONLY`	Open file in write only mode.
`O_RDWR`	Open file for reading and writing.
`O_APPEND`	Open file in append mode.
`O_TRUNC`	File is truncated before writing.
`O_BINARY`	Open file in binary mode.
`O_TEXT`	Open file in text mode.

1.5.2 `float` maths <mathf.h>

The header file `mathf.h` contains declarations of the short maths functions. Short maths functions are identical to ANSI standard functions except that all arguments and results are of type `float` rather than `double`. Errors which generate the error code `HUGE_VAL` (out of range) in the ANSI functions return `HUGE_VAL_F` in the short maths functions.

Note: the following is true for all functions declared in `mathf.h`:

On domain errors:	`errno` is set to `EDOM`; 0.0 is returned.
On range errors:	`errno` is set to `ERANGE`; `HUGE_VAL_F` is returned for overflow errors; `-HUGE_VAL_F` is returned for underflow errors.

Function	Description
`acosf`	Calculates the arc cosine of the `float` argument.
`asinf`	Calculates the arc sine of the `float` argument.
`atanf`	Calculates the arc tangent of the `float` argument.
`atan2f`	Calculates the arc tangent of (*argument 1* divided by *argument 2*) where the numerator and denominator arguments are both `floats`.
`ceilf`	Calculates the smallest integer which is not less than the `float` argument.
`cosf`	Calculates the cosine of the `float` argument.
`coshf`	Calculates the hyperbolic cosine of the `float` argument.
`expf`	Calculates the exponential function of the `float` argument.
`fabsf`	Calculates the absolute value of the `float` argument.
`floorf`	Calculates the largest integer which is not greater than the `float` argument.
`fmodf`	Calculates the floating point remainder of (*argument 1* divided by *argument 2*) where the numerator and denominator arguments are both `floats`.
`frexpf`	Separates a floating point number into a mantissa and integral power of two.
`ldexpf`	Multiplies a floating point number by an integral power of two.
`logf`	Calculates the natural logarithm of the `float` argument.
`log10f`	Calculates the base-10 logarithm of the `float` argument.
`modff`	Splits the `float` argument into fractional and integral parts.
`powf`	Calculates x to the power of y where both x and y are `floats`.
`sinf`	Calculates the sine of the `float` argument.
`sinhf`	Calculates the hyperbolic sine of the `float` argument.
`sqrtf`	Calculates the square root of the `float` argument.
`tanf`	Calculates the tangent of the `float` argument.
`tanhf`	Calculates the hyperbolic tangent of the `float` argument.

1.5.3 Host utilities <host.h>

The header file `host.h` contains one function that returns host system information and a number of host system constants.

Function	Description
`host_info`	Returns information about the host system and transputer board.

Constant	Description
`_IMS_HOST_PC`	Standard PC host.
`_IMS_HOST_NEC`	NEC PC–9801 series host.
`_IMS_HOST_VAX`	VAX host.
`_IMS_HOST_SUN3`	Sun 3 host.
`_IMS_HOST_SUN4`	Sun 4 host.
`_IMS_HOST_SUN386i`	Sun 386i host..
`_IMS_HOST_APOLLO`	APOLLO host.
`_IMS_HOST_IBM370`	IBM 370 host.
`_IMS_OS_DOS`	DOS operating system.
`_IMS_OS_HELIOS`	HELIOS operating system.
`_IMS_OS_VMS`	VMS operating system.
`_IMS_OS_SUNOS`	SunOS operating system.
`_IMS_OS_CMS`	CMS operating system.
`_IMS_BOARD_B004`	IMS B004 PC transputer board.
`_IMS_BOARD_B008`	IMS B008 transputer module (TRAM) Motherboard.
`_IMS_BOARD_B010`	IMS B010 4-TRAM NEC PC Motherboard.
`_IMS_BOARD_B011`	IMS B011 2-TRAM VME board.
`_IMS_BOARD_B014`	IMS B014 8-TRAM VMEbus slave card.
`_IMS_BOARD_DRX11`	INMOS VAX link interface board.
`_IMS_BOARD_QT0`	Caplin QT0 VAX/VMS link interface board.
`_IMS_BOARD_B015`	IMS B015 NEC 9800 PC TRAM motherboard.
`_IMS_BOARD_CAT`	IBM CAT transputer board.
`_IMS_BOARD_B016`	IMS B016 VMEbus master/slave motherboard.
`_IMS_BOARD_UDP_LINK`	IMS UDP Link support product.

1.5.4 Host channel access utilities <hostlink.h>

The header file `hostlink.h` contains two functions that return a pointer to the link channel going to and coming from the host system.

Function	Description
`from_host_link`	Retrieves the channel coming from the host.
`to_host_link`	Retrieves the channel going to the host.

1.5.5 Boot link channel functions <bootlink.h>

This header file contains one function to obtain the channels associated with the boot link.

Function	Description
`get_bootlink_channels`	Obtains the channels associated with the boot link.

1.5.6 MS-DOS system functions <dos.h>

The header file `dos.h` contains a number of functions for performing MS-DOS system operations, plus one type. The file also contains definitions of associated structures, not documented here.

All the MS-DOS specific functions return an error if they are used on operating systems other than MS-DOS.

Function	Description
`alloc86`	Allocates a block of host memory for use with the `to86` and `from86` functions.
`bdos`	Performs a MS-DOS function call interrupt
`free86`	Frees a block of host memory previously allocated with `alloc86`.
`from86`	Copies a block of host memory to transputer memory.
`int86`	Raises a software interrupt. Segment registers are untouched.
`int86x`	As `int86` but also sets the processor segment registers.
`intdos`	As `int86` but specific for a MS-DOS function call.
`intdosx`	As `intdos` but also sets the segment registers.
`segread`	Reads the segment registers.
`to86`	Copies a block of transputer memory to host memory.

Type	Description
`pcpointer`	A type that can be used to hold a standard PC pointer.

1.5.7 Dynamic code loading functions <fnload.h>

The header file `fnload.h` contains functions to support dynamic code loading using `.rsc` files. The functions interact with three 'flavors' of `.rsc` files:

- `.rsc` file
- `.rsc` file stored in ROM or RAM
- `.rsc` file received over a channel

Two functions are provided for each case; one to retrieve information from the file or file image and one to load the code from the file into internal memory.

Function	Description
`get_code_details_from_file`	Retrieves details from a `.rsc` file.
`get_code_details_from_memory`	Retrieves details from the image of a `.rsc` file, held in internal memory.
`get_code_details_from_channel`	Retrieves details from a `.rsc` file that is received over a channel.
`load_code_from_file`	Loads the code of a `.rsc` file into internal memory.
`load_code_from_memory`	Transfers the code of a `.rsc` file image from one section of internal memory to another.
`load_code_from_channel`	Loads the code of a `.rsc` file, received over a channel, into internal memory.

`fnload.h` defines the type `fn_info` which has the following structure definition:

```
struct fn_data
  {
    int target_processor_type;  /* as given in the .rsc file */
    size_t stack_size;          /* in bytes */
    size_t vectorspace_size;    /* in bytes */
    size_t static_size;         /* in bytes */
    size_t entry_point_offset;  /* in bytes */
    size_t code_size;           /* in bytes */
  };
typedef struct fn_data fn_info;
```

`target_processor_type` gives the processor type for which the code in the `.rsc` file is compiled. The processor type is encoded as an integer; a list of possible values is given in section 3.5 of the *ANSI C Toolset Reference Manual.*

1.5.8 Miscellaneous functions `<misc.h>`

The header file `misc.h` declares some additional non-ANSI functions, including three debugging support functions, plus three constants that control the operation of `set_abort_action`. It also contains functions to perform bit manipulation, block moves and CRC calculations.

Function	Description
`BlockMove`	Copies a block of memory.
`BitCnt`	Count the number of bits set.
`BitCntSum`	Count the number of bits set and sum with an integer.
`BitRevNBits`	Reverse the order of the least significant bits of an integer.
`BitRevWord`	Reverse the order of the bits in an integer.
`call_without_gsb`	Calls the function (pointed to) without passing in the global static base (gsb).
`CrcByte`	Calculates CRC of most–significant byte of an integer.
`CrcFromLsb`	Calculates the CRC of a byte sequence starting at the least significant bit.
`CrcFromMsb`	Calculates the CRC of a byte sequence starting at the most significant bit.
`CrcWord`	Calculates CRC of an integer.
`debug_assert`	Stops a process on a specified condition.
`debug_message`	Inserts a debugging message.
`debug_stop`	Stops a process.
`exit_noterminate`	Exits the program without terminating the server. Used for configured programs, otherwise like `exit`.
`exit_repeat`	Program termination with restart. As `exit` but allows the program to be restarted on the processor.
`exit_terminate`	Terminates the server. Used for configured programs, otherwise like `exit`.
`get_param`	Reads `interface` parameters for a configured process.
`halt_processor`	Halts the processor on which it is executed.
`max_stack_usage`	Estimates runtime stack usage in a program.
`set_abort_action`	Sets or queries the action to be taken by `abort`. The possible actions are:exit without clearing files; or halt the transputer.

Function	Description
`get_details_of_free_memory`	Reports the details of memory considered by the configurer to be unused.
`get_details_of_free_stack_space`	Reports the limits of free space on the current stack.
Note: These two functions have been separated out from the main list of functions purely because of the length of their names.	

Macro	Description
ABORT_EXIT	Directs set_abort_action to cause a normal program exit on abort.
ABORT_HALT	Directs set_abort_action to halt the transputer on abort.
ABORT_QUERY	Directs set_abort_action to return the current abort action without resetting it.

1.6 Fatal runtime errors

Errors are generated at severity level *Fatal* by the C runtime system when the program cannot be run. Such errors may occur at startup or during program execution.

The main causes of runtime errors in a program are summarized below.

- Insufficient memory at startup.
- Stack overflow during execution.
- Illegal conditions detected by the library functions **free**, and **realloc** and the concurrency library functions. These errors are described in detail under the function descriptions in chapter 2.

When runtime errors occur the program terminates immediately with an error message. All runtime error messages are prefixed with '**Fatal-C_Library**'.

1.6.1 Runtime error messages

Fatal-C_Library-Bad workspace pointer

This error message is issued when the stack checking code or dynamic code loading functions detect that the current process is running in an illegal stack area. Legal stack areas are the main stack area defined at program startup or parallel process stacks. **Note:** that this error may also mean that global data stored in the static area has been corrupted.

Fatal-C_Library-Out of memory in system startup [*number*]

This error is generated when insufficient static or heap space is available to run the program. *number* can take the following values:

1 – Insufficient memory to accommodate static area.

2 – Insufficient memory to accommodate static area.

3 – Insufficient heap space for the input and output channel arrays.

4 – Insufficient heap space for command line parameters to the pro gram.

5 – Insufficient heap space to set up low level I/O system.

6 – Insufficient heap space to set up ANSI `stdio` level I/O system.

If this error occurs then either the available memory can be increased or the program re-coded in a less memory-intensive way.

Fatal-C_Library-Stack overflow

This message is only generated when stack checking is enabled in the compiler. It indicates stack overflow in the program and may be remedied by increasing the specified stack size. If no stack size has been specified and the default has been assumed by the program then the stack size cannot be increased and the program should be re-coded.

Fatal-C_Library-Error in free (), bad pointer or heap corrupted

This error indicates an invalid pointer passed to `free` or corruption of the heap. No specific recovery is possible and the program should be debugged.

Fatal-C_Library-Error in realloc (), bad pointer or heap corrupted

This error indicates an invalid pointer passed to `realloc` or corruption of the heap. No specific recovery is possible and the program should be debugged.

Fatal-C_Library-Incorrect allocation of process workspace

This error is generated by `ProcInit` if an attempt is made to define a workspace which is nested within the workspace of an existing process or is taken from the main program stack. An example of this would be an attempt to use an automatic array as a process workspace.

Fatal-C_Library-Nested Pri Pars are illegal

This error is generated by `ProcPriPar` when it is called from a high priority process. Calling `ProcPriPar` from a high priority process is prohibited in this toolset.

Fatal-C_Library-Bad pointer to process clean function

An invalid process structure pointer has been pointed to `ProcInitClean` or `ProcAllocClean`.

Fatal-C_Library-Attempt to start a process which is already running.

An attempt has been made to start a process (using `ProcRun`, `ProcRunLow`. `ProcRunHigh`, `ProcPar`, `ProcParList` or `ProcPriPar`) which has already been started and is still executing.

2 Alphabetical list of functions

This chapter contains detailed reference information for the runtime library functions and their operation.

2.1 Format

Function descriptions are laid out in a standard format. First, the function name is given, highlighted in large type, followed on the same line by a brief summary of its action.

The function name is followed by detailed information about the function under the following headings:

Heading	Information given
Synopsis:	The file to be included and the function declaration.
Arguments:	A list of the function's arguments and their meanings.
Results:	The result(s) returned.
Errors:	The action(s) taken on error.
Description:	A detailed description of the function and hints on usage.
Example:	An example of the function's use, where appropriate.
See also:	A list of related functions, where appropriate.

2.1.1 Reduced library

Where functions are not available in the reduced library, this is indicated in the function description.

2.1.2 Macros

Where functions are implemented as macros, or as both macros and regular C functions, this is also indicated in the detailed description.

For these functions the version used by the compiler depends on the syntax of the calling statement. If the call uses parentheses around the function name (as in `(putchar)(ch)`), the regular function is used; if parentheses are omitted (as in `putchar(ch)`), the macro form is used instead.

2.2 List of functions

abort

Aborts the program.

Synopsis:

```
#include <stdlib.h>
void abort(void);
```

Arguments:

None.

Results:

`abort` does not return.

Errors:

None.

Description:

`abort` causes immediate termination of the program. It does not flush output streams, close open streams, or remove temporary files. `abort` passes `SIGABRT` to the signal handler, to show that the program has terminated abnormally.

The default action is to abort the program without halting the processor. The function can be set to halt the processor by first calling `set_abort_action` with the appropriate argument.

If set to halt, `abort` forces the processor to halt even if the program is not in HALT mode, by explicitly setting the Halt-On-Error and Error flags.

See also:

`exit` `exit_terminate` `exit_noterminate` `halt_processor` `set_abort_action` `signal`

abs

Calculates the absolute value of an integer.

Synopsis:

```
#include <stdlib.h>
int abs(int j);
```

Arguments:

`int j`	An integer.

Results:

Returns the absolute value of `j`.

Errors:

If the result cannot be represented the behavior of `abs` is undefined.

Description:

`abs` calculates the absolute value of the integer `j`.

`abs` is side effect free.

See also:

`labs`

acos

Calculates the arc cosine of the argument.

Synopsis:

```
#include <math.h>
double acos(double x);
```

Arguments:

`double x`	A number in the range [–1..+1].

Results:

Returns the arc cosine of **x** in the range [0..pi] radians and 0.0 on domain errors.

Errors:

A domain error occurs if **x** is not in the range [–1..+1]. In this case **errno** is set to EDOM.

Description:

acos calculates the arc cosine of a number.

See also:

acosf

`acosf`

Calculates the arc cosine of a `float` number.

Synopsis:

```
#include <mathf.h>
float acosf(float x);
```

Arguments:

`float x`	A number in the range [–1..+1].

Results:

Returns the arc cosine of `x` in the range [0..pi] radians and 0.0F on domain errors.

Errors:

A domain error occurs if `x` is not in the range [–1..+1]. In this case `errno` is set to `EDOM`.

Description:

`float` form of `acos`.

See also:

`acos`

alloc86

Allocates a block of host memory. MS-DOS only.

Synopsis:

```
#include <dos.h>
pcpointer alloc86(int n);
```

Arguments:

`int n`	The number of bytes of host memory to be allocated.

Results:

Returns a pointer to the allocated block of host memory.

Errors:

Returns a `NULL` PC pointer if the allocation fails and sets `errno` to the value `EDOS`. Any attempt to use `from86` on systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`alloc86` allocates a block of memory on the MS-DOS host and returns a pointer to it. If the memory cannot be allocated, a `NULL` PC pointer is returned. The allocated memory cannot be accessed directly by the transputer program but only by means of the functions `to86` and `from86`.

Note: Intel 80x86 architecture limits the amount of memory which can be contained in a single segment to 65536 bytes; `alloc86` cannot allocate more than this architectural limit.

See also:

`from86` `to86`

`asctime` Converts a broken-down-time structure to an ASCII string.

(See section 1.3.16 for a definition of broken-down-time).

Synopsis:

```
#include <time.h>
char* asctime(const struct tm *timeptr);
```

Arguments:

`const struct tm *timeptr`	A pointer to the broken-down-time structure to be converted.

Results:

Returns a pointer to the ASCII time string.

Errors:

None.

Description:

`asctime` returns the values in the `timeptr` structure as an ASCII string in the form: Thu Nov 05 18:19:01 1987

The string pointed to may be overwritten by subsequent calls to `asctime`.

Example:

```
/* Displays the current time */

#include  <time.h>
#include  <stdio.h>

int main()
{
   struct tm *now;
   time_t clck;

   time(&clck); /* Get current time in secs */

   now = localtime( &clck);
                /* Convert time to
                   a structure (tm) */
   printf("The time is: %s\n", asctime(now));
}
```

Note: Care should be taken when calling `asctime` in a concurrent environment. Calls to the function by independently executing, unsynchronized processes may corrupt the returned time value.

See also:

`ctime localtime strftime clock difftime mktime time`

asin

Calculates the arc sine of the argument.

Synopsis:

```
#include <math.h>
double asin(double x);
```

Arguments:

`double x`	A number in the range [–1..+1].

Results:

Returns the arc sine of `x` in the range [–pi/2..+pi/2] radians and 0.0 on domain errors.

Errors:

A domain error occurs if `x` is not in the range [–1..+1]. In this case `errno` is set to `EDOM`.

Description:

`asin` calculates the arc sine of a number.

See also:

`asinf`

asinf

Calculates the arc sine of a `float` number.

```
#include <mathf.h>
float asinf(float x);
```

Arguments:

`float x`	A number in the range [–1..+1].

Results:

Returns the arc sine of `x` in the range [–pi/2..+pi/2] radians and 0.0F on domain errors.

Errors:

A domain error occurs if `x` is not in the range [–1..+1]. In this case `errno` is set to `EDOM`.

Description:

`float` form of `asin`.

See also:

`asin`

assert

Inserts diagnostic messages.

Synopsis:

```
#include <assert.h>
void assert(int expression);
```

Arguments:

`int expression`	The condition to be asserted.

Results:

Returns no value.

Errors:

None.

Description:

`assert` is a debugging macro. If it is called with `expression` equal to zero, `assert` terminates the program by calling `abort`. The action of `abort` when called by `assert` depends on the most recent call to `set_abort_action`.

If `expression` is non-zero, no action is taken.

If the function is linked with the full runtime library and the expression evaluates to zero, the following message is written to `stderr`:

`*** assertion failed:` *condition*`,` `file` *filename*`,` `line` *linenumber*

If the function is linked with the reduced runtime library then no message is displayed if the assertion fails.

The definition of the `assert` macro depends upon the definition of the `NDEBUG` macro. If `NDEBUG` is defined before the definition of `assert` then `assert` is defined as:

```
#define assert(ignore) ((void)0)
```

If `assert` is defined first the definition is honored and `NDEBUG` is ignored.

Example:

```
#include  <stdio.h>
#include  <assert.h>

float divide (float a, float b)
{
   assert(b != 0.0F);
   return a/b;
}

int main( void )
{
   float res;

   res = divide(1.0F,2.0F);
   printf("1.0 divided by 2.0 is: %f\n",res);
   res = divide(1.0F,0.0F);
   printf("1.0 divided by 0.0 is: %f\n",res);
}
/*
 *   Output:
 *
 * *** assertion failed: b != 0.0,
 *     file assert.c, line 6
 *
 */
```

See also:

abort debug_assert

atan

Calculates the arc tangent of the argument.

Synopsis:

```
#include <math.h>
double atan(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the arctan of **x** in the range [–pi/2..+pi/2] radians.

Errors:

None.

Description:

`atan` calculates the arc tangent of a number.

See also:

`atanf`

atan2

Calculates the arc tangent of y/x.

Synopsis:

```
#include <math.h>
double atan2(double y, double x);
```

Arguments:

`double y`	The numerator.
`double x`	The denominator.

Results:

Returns the arc tangent of **y/x** in the range [–pi..+pi] radians and 0.0F on domain errors.

Errors:

A domain error occurs if **x** and **y** are zero. In this case **errno** is set to **EDOM**.

Description:

`atan2` calculates the arc tangent of **y/x**.

See also:

`atan2f`

atan2f

Calculates arc tangent of y/x where both are `floats`.

Synopsis:

```
#include <mathf.h>
float atan2f(float y, float x);
```

Arguments:

`float y`	The numerator.
`float x`	The denominator.

Results:

Returns the arc tangent of `y/x` in the range [–pi..+pi] radians and 0.0 on domain errors

Errors:

A domain error occurs if `x` and `y` are zero. In this case `errno` is set to `EDOM`.

Description:

`float` form of `atan2`.

See also:

`atan2`

atanf

Calculates the arc tangent of a `float` number.

Synopsis:

```
#include <mathf.h>
float atanf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the arc tangent of **x** in the range [–pi/2..+pi/2] radians.

Errors:

None.

Description:

`float` form of `atan`.

See also:

`atan`

atexit

Specifies a function to be called when the program ends.

Synopsis:

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Arguments:

`void (*func)(void)`	A pointer to the function to be called.

Results:

Returns zero if `atexit` is successful and non-zero if it is not.

Errors:

None.

Description:

`atexit` records that the function pointed to by `func` is to be called (without arguments) at normal termination of the program.

A maximum of 32 functions can be recorded for execution on exit. They will be called in reverse order of their being recorded (that is, last in, first out).

Note: In the parallel environment `atexit` works on program termination rather than process termination. A maximum of 32 functions can be registered as exit functions per program.

Example:

```
#include  <stdlib.h>
#include  <stdio.h>

void first_exit( void )
{
   printf("First_exit called on exit\n");
}

void second_exit( void )
{
   printf("Second_exit called on exit\n");
}
```

```
int main( void )
{
   atexit(second_exit);
   atexit(first_exit);
   printf("About to exit from program\n");
   return 0;
}

/*
 *   Output:
 *
 *          About to exit from program
 *          First_exit called on exit
 *          Second_exit called on exit
 *
 */
```

See also:

`exit`

`atof`

Converts a string of characters to a `double`.

Synopsis:

```
#include <stdlib.h>
double atof(const char *nptr);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.

Results:

Returns the converted value or zero(0) on error.

Errors:

If the string cannot be converted, `atof` returns 0 (zero). If the conversion would cause overflow or underflow in the `double` value, the behavior is undefined.

Description:

`atof` converts the string pointed to by `nptr` to a double precision floating point number. `atof` expects the string to consist of:

1 Leading white space (optional).

2 A plus or minus sign (optional).

3 A sequence of decimal digits, which may contain a decimal point.

4 An exponent (optional) consisting of an 'E' or 'e' followed by an optional sign and a string of decimal digits.

5 One or more unrecognized characters (including the string terminating character).

`atof` ignores the leading white space, and converts all the recognized characters. If there is no decimal point or exponent part in the string, a decimal point is assumed after the last digit in the string.

The string is invalid if the first non-space character in the string is not one of the following characters: + – 0 1 2 3 4 5 6 7 8 9

Example:

```
#include  <stdio.h>
#include  <stdlib.h>

int main()
{
 char *array;
 double x;


 array = "  -4235.120E-3";
 x = atof(array);
 printf("Float = %f\n", x);


 array = "  -735492.45";
 x = atof(array);
 printf("Float = %e\n", x);
}
/*
Prints Float = -4.235120
       Float = -7.354924e+05
*/
```

See also:

atoi atol strtod

atoi

Converts a string of characters to an `int`.

Synopsis:

```
#include <stdlib.h>
int atoi(const char *nptr);
```

Arguments:

`const char *nptr` A pointer to the string to be converted.

Results:

Returns the converted value or zero(0) on error.

Errors:

If the string cannot be converted, `atoi` returns 0. If the conversion would overflow or underflow, the behavior is undefined.

Description:

`atoi` converts the string pointed to by `nptr` to an integer. `atoi` expects the string to consist of:

1 Leading white space (optional).

2 A plus or minus sign (optional).

3 A sequence of decimal digits.

4 One or more unrecognized characters (including the string terminating character).

`atoi` ignores the leading white space, and converts all the recognized characters.

The string is invalid if the first non-space character in the string is not one of the following characters: + – 0 1 2 3 4 5 6 7 8 9

Example:

```
#include  <stdlib.h>
#include  <stdio.h>

int main( void )
{
   char *array;
   int x;

   array = "  -4235";
   x = atoi(array);
   printf("Integer is: %d\n", x);

   array = "-735492 and some rubbish text";
   x = atoi(array);
   printf("Integer is: %d\n", x);
}

/*
 *   Output:
 *
 *          Integer is: -4235
 *          Integer is: -735492
 *
 */
```

See also:

atof atol strtol

atol

Converts a string of characters to a long integer.

Synopsis:

```
#include <stdlib.h>
long int atol(const char *nptr);
```

Arguments:

`const char *nptr` A pointer to the string to be converted.

Results:

Returns the converted value or zero(0) on error.

Errors:

If the string cannot be converted, `atol` returns 0. If the conversion would overflow or underflow, the behavior is undefined.

Description:

`atol` converts the string pointed to by `nptr` to a long integer. `atol` expects the string to consist of:

1 Leading white space (optional).

2 A plus or minus sign (optional).

3 A sequence of decimal digits.

4 One or more unrecognized characters (including the string terminating character).

`atol` ignores the leading white space, and converts all the recognized characters.

The string is invalid if the first non-space character in the string is not one of the following characters: + – 0 1 2 3 4 5 6 7 8 9

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
 char *array;
 long l;

 array = "  -735492 and trailing text";
 l = atol(array);
 printf("Long = %ld\n", l);
}

/*
Prints "Long = -735492"
*/
```

See also:

atof atoi strtod strtol

bdos

Performs a simple MS-DOS function. MS-DOS only.

Synopsis:

```
#include <dos.h>
int bdos(int dosfn, int dosdx, int dosal);
```

Arguments:

`int dosfn`	Value to assign to the **ah** register.
`int dosdx`	Value to assign to the **dx** register.
`int dosal`	Value to assign to the **al** register.

Results:

Returns the value of the **ax** register or zero(0) on error.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `bdos` on operating systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`bdos` performs an MS-DOS function call interrupt on the host with the **ah** register (specifying the MS-DOS function call number) set to `dosfn`, and with the **dx** and **al** registers set to `dosdx` and `dosal` respectively. It is a shorthand form of `int86` for the very simplest MS-DOS function calls only.

`bdos` is not included in the reduced library.

See also:

`intdos int86`

BitCnt

Count the number of bits set.

Synopsis:

```
#include <misc.h>
int BitCnt(int word);
```

Arguments:

`int word`	The integer whose set bits are to be counted.

Results:

Returns the number of bits set in `word`.

Errors:

None.

Description:

The number of bits set in the integer argument `word` are counted. The count is returned.

Example:

```
int data;
int num_bits_set;

num_bits_set = BitCnt(data);
```

When compiling for transputers which have the *bitcnt* instruction, calls to `BitCnt` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

`BitCnt` is side effect free.

See also:

`BitCntSum`

BitCntSum

Count the number of bits set and sum with an integer.

Synopsis:

```
#include <misc.h>
int BitCntSum(int word, int count_in);
```

Arguments:

`int word`	The integer whose set bits are to be counted.
`int count_in`	The value to be summed with the number of bits set in `word`.

Results:

Returns the sum of `count_in` and the number of bits set in `word`.

Errors:

None.

Description:

The number of bits set in the integer argument `word` are counted and summed with `count_in`. The sum is returned. The sum is performed using modulo arithmetic, so no overflow can occur.

Example:

```
int data[10];
int count;
int i;

/* Sum the number of bits set in 'data' */
count = 0;
for (i = 0; i < 10; i++)
  count = BitCntSum(data[i], count);
```

When compiling for transputers which have the *bitcnt* instruction, calls to `BitCnt-Sum` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

`BitCntSum` is side effect free.

See also:

`BitCnt`

`BitRevNBits` Reverse the order of the least significant bits of an integer.

Synopsis:

```
#include <misc.h>
int BitRevNBits(int numbits, int data);
```

Arguments:

`int numbits`	The number of bits to reverse.
`int data`	The integer whose least significant bits are to be reversed.

Results:

Returns `data` with its `numbits` least significant bits reversed and its other bits zeroed.

Errors:

If `numbits` is negative or `numbits` is greater than the number of bits in a word, then the effect of calling `BitRevNBits` is undefined.

Description:

The order of the `numbits` least significant bits of `data` is reversed. All other bits of `data` are zeroed. This result is returned. For example, on a 32–bit processor:

Data:

31					n					0
b_{31}	b_{30}	b_{29}	...	b_{n+1}	b_n	b_{n-1}	...	b_2	b_1	b_0

Result:

31					n					0
0	0	0	...	0	b_0	b_1	...	b_{n-2}	b_{n-1}	b_n

`BitRevNBits` is side effect free.

Example:

```
int data;
int numbits;
int rev_data;

rev_data = BitRevNBits(numbits, data);
```

When compiling for transputers which have the *bitrevnbits* instruction, calls to `BitRevNBits` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

See also:

`BitRevWord`

BitRevWord

Reverse the order of the bits in an integer.

Synopsis:

```
#include <misc.h>
int BitRevWord(int data);
```

Arguments:

`int data`	The integer whose bits are to be reversed.

Results:

Returns `data` with all bits in reversed order.

Errors:

None.

Description:

The bit pattern in `data` is reversed end–for–end. The result is returned. For example, on a 32–bit processor:

Data:

31						0
b_{31}	b_{30}	b_{29}	.....	b_2	b_1	b_0

Result:

31						0
b_0	b_1	b_2	.....	b_{29}	b_{30}	b_{31}

`BitRevWord` is side effect free.

Example:

```
int data;
int rev_data;

rev_data = BitRevWord(data);
```

When compiling for transputers which have the *bitrevword* instruction, calls to `BitRevWord` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

See also:

`BitRevNBits`

BlockMove

Copy a block of memory

Synopsis:

```
#include <misc.h>
void BlockMove(void *dest, const void *source, size_t n);
```

Arguments:

`void *dest`	A pointer to the destination of the copy.
`const void *source`	A pointer to the source of the copy.
`size_t n`	The number of bytes to be copied.

Results:

Returns no result.

Errors:

The behavior of `BlockMove` is undefined if the source and destination overlap.

Description:

`BlockMove` copies `n` bytes from the area of memory pointed to by `source` to the area of memory pointed to by `dest`. The behavior of `BlockMove` is undefined if the source and destination area overlap.

Example:

```
int source[27];
int dest[500];

BlockMove(dest, source, 27 * sizeof(int));
```

Calls to `BlockMove` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

bsearch

Searches a sorted array for a given object.

Synopsis:

```
#include <stdlib.h>
void *bsearch(const void *key,
              const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *,
                            const void *));
```

Arguments:

`const void *key`	A pointer to the object to be matched.
`const void *base`	A pointer to the start of the array.
`size_t nmemb`	The number of objects in the array.
`size_t size`	The size of the array objects.
`int (*compar) (const void *, const void *)`	A pointer to the comparison function.

Results:

Returns a pointer to the object if found; otherwise `bsearch` returns a `NULL` pointer. If more than one object in the array matches the key, it is not defined which one the return value points to.

Errors:

None.

Description:

`bsearch` searches the array pointed to by `base` for an object which matches the object pointed to by `key`. The array contains `nmemb` objects of `size` bytes.

The objects are compared using the comparison function pointed to by `compar`. The function must return an integer less than, equal to, or greater than zero, depending on whether the first argument to the function is considered to be less than, equal to, or greater than the second argument.

The base array must already be sorted in ascending order (according to the comparison performed by the function pointed to by `compar`).

Example:

```
/*
 * Receives a list of arguments from the
 * terminal, and searches them for the
 * string "findme".
 */

#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>

int compare(const void *arg1, const void *arg2)
{
   return(strncmp(*(char **)arg1, *(char **)arg2,
          strlen(*(char **)arg1)));
}


int main(int argc, char *argv[])
{
   char **result;
   char *key = "findme";

   /* sort the command line arguments according
      to the string compare function 'compare' */

   qsort(argv, argc, sizeof(char *), compare);

   /* Find the argument which starts with
      the string in 'key'  */

   result = (char **)bsearch(&key, argv, (size_t)argc,
                             sizeof(char *), compare);

   if (result != NULL)
     printf("\n'%s' found\n", *result);
   else
     printf("\n'%s' not found\n", key);
}
```

See also:

`qsort`

`call_without_gsb` Calls the pointed to function without passing the `gsb`.

Synopsis:

```
#include <misc.h>
void call_without_gsb( void (*fn_ptr)(void),
                       int number_of_words_for_parameters,
                       ... )
```

Arguments:

`void (*fn_ptr)(void)`	A pointer to the function to be called without a `gsb`.
`int number_of_words_for_parameters`	The number of words that the parameters in the ellipsis occupy.
`...`	The parameters of the function to be called in the correct order for that function.

Results:

None.

Errors:

None.

Description:

`call_without_gsb` calls the specified function without passing a `gsb` as the first (hidden) parameter. `call_without_gsb` requires that the called function uses the same calling convention as the INMOS ANSI C toolset.

The function called must return `void`.

Note: no type checking is done on the parameters to the function to be called – it is up to the user to ensure correctness.

In the header file where it is declared this function has the `IMS_nolink pragma` applied to it, so it cannot be called by a pointer to it, other than by use of itself. This function will not work unless the `IMS_nolink` pragma is applied to it.

`calloc`

Allocates memory space for an array of items and initializes the space to zeros.

Synopsis:

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Arguments:

`size_t nmemb`	The number of items in the array to be allocated.
`size_t size`	The size of the array items.

Results:

Returns a pointer to the allocated space if the allocation is successful; otherwise `calloc` returns a `NULL` pointer. If either argument is zero `calloc` returns a `NULL` pointer.

Errors:

`calloc` returns a `NULL` pointer if there is not enough free space in memory or if either argument is zero.

Description:

`calloc` allocates space in memory for an array containing `nmemb` items, where each item is `size` bytes long. The allocated memory is initialized to zeros.

Programming note: On the T2 family of transputers pointers should always be initialized explicitly, because the `NULL` pointer on these machines is represented by a non-zero bit pattern.

See also:

`free` `malloc` `realloc`

`ceil` Calculates the smallest integer not less than the argument.

Synopsis:

```
#include <math.h>
double ceil(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the smallest integer (expressed as a double) which is not less than `x`.

Errors:

None.

Description:

`ceil` calculates the smallest integer which is not less than `x`.

`ceil` is side effect free.

See also:

`floor` `ceilf`

`ceilf` Calculates the smallest integer not less than the `float` argument.

Synopsis:

```
#include <mathf.h>
float ceilf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the smallest integer (expressed as type `float`) which is not less than `x`.

Errors:

None.

Description:

`float` form of `ceil`.

`ceilf` is side effect free.

See also:

`ceil` `floorf`

`ChanAlloc`

Allocates and initializes a channel.

Synopsis:

```
#include <channel.h>
Channel *ChanAlloc(void);
```

Arguments:

None.

Results:

Returns a pointer to an initialized channel, or `NULL` if the space could not be allocated.

Errors:

Returns `NULL` if space could not be allocated.

Description:

Allocates and initializes a channel. The space is allocated using `malloc`.

Note: All channels must have space reserved for them before they are used. The space can be allocated using `ChanAlloc`; explicitly using `malloc` or by using a static or automatic variable. If `ChanAlloc` is not used the channel should be initialized using `ChanInit`.

The space allocated for a channel by `ChanAlloc` can be freed by passing the channel pointer directly to `free`.

See also:

`ChanReset`

ChanIn

Inputs data on a channel.

Synopsis:

```
#include <channel.h>
void ChanIn(Channel *c, void *cp, int count);
```

Arguments:

`Channel *c`	A pointer to the input channel.
`void *cp`	A pointer to the array where the data will be stored.
`int count`	The number of bytes of data.

Results:

Returns no result.

Errors:

None.

Description:

Inputs `count` bytes of data on the specified channel and stores them in the array pointed to by `cp`. The effect of this routine is undefined if `count` ≤ `0`.

See also:

`ChanOut` `ChanInInt` `ChanInChar` `ChanInChanfail` `ChanInTimeFail`

`ChanInChanFail`

Inputs data on a link channel or aborts.

Synopsis:

```
#include <channel.h>
int ChanInChanFail(Channel *chan, void *cp,
                   int count, Channel *failchan);
```

Arguments:

`Channel *c`	A pointer to the input channel.
`void *cp`	A pointer to an array where the data will be stored.
`int count`	The number of bytes of data.
`Channel *failchan`	A pointer to the channel on which the failure message is received.

Results:

Returns zero (0) if communication completes, one (1) if communication is aborted by a message on the failure channel.

Errors:

None.

Description:

`ChanInChanFail` is used to perform reliable channel communication on a link. The function inputs count bytes of data on the specified channel into the array pointed to by `cp`. It can be aborted by an integer, and only an integer, passed on `failchan`. Typically `failchan` will be a channel from a process which is monitoring the integrity of the link.

Note: this function may not be used on a virtual channel supplied from either the configurer or from the debugger `idebug` in interactive mode. This is described further in section 6.3.2 of the *ANSI C Toolset User Guide*.

See also:

`ChanIn` `ChanInTimeFail`

ChanInChar

Inputs one byte on a channel.

Synopsis:

```
#include <channel.h>
unsigned char ChanInChar(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the input channel.

Results:

Returns the input byte.

Errors:

None.

Description:

Inputs a single byte on a channel.

> **Note:** The prototype of `ChanInChar` has changed from previous releases of the toolset i.e. the D7214, D6214, D5214 and D4214 products, where `ChanInChar` was of type `Char`.

See also:

`ChanOutChar` `ChanIn`

ChanInInt

Inputs an integer on a channel.

Synopsis:

```
#include <channel.h>
int ChanInInt(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the input channel.

Results:

Returns the input integer.

Errors:

None.

Description:

Inputs a single integer on a channel.

See also:

`ChanOutInt ChanIn`

ChanInit

Initializes a channel pointer.

Synopsis:

```
#include <channel.h>
void ChanInit(Channel *chan);
```

Arguments:

`Channel *chan`	A pointer to a channel.

Results:

Returns no result.

Errors:

None.

Description:

Initializes the channel pointed to by `chan` to the value `NotProcess_p`.

`NotProcess_p` is defined in `channel.h`.

Example:

```
#include <channel.h>
#include <stdlib.h>

Channel c1, *c2;

ChanInit(&c1);
c2 = (Channel *)malloc(sizeof(Channel));
ChanInit(c2);
```

See also:

`ChanReset`

ChanInTimeFail

Inputs data on a channel or times out.

Synopsis:

```
#include <channel.h>
int ChanInTimeFail(Channel *chan, void *cp,
                   int count, int time);
```

Arguments:

`Channel *c`	A pointer to the input channel.
`void *cp`	A pointer to an array where the data will be stored.
`int count`	The number of bytes of data.
`int time`	The absolute time after which the communication is aborted if no input occurs.

Results:

Returns zero (0) if the communication is successful, one (1) if timeout occurs before the communication completes.

Errors:

None.

Description:

`ChanInTimeFail` is used to timeout channel communication on a link. It inputs count bytes of data on the specified channel and stores them in the array pointed to by `cp`, or aborts if the transputer clock reaches the specified absolute time. Typically it is used to notify delay on a link so that the communication can be routed elsewhere.

Note: this function may not be used on a virtual channel supplied from either the configurer or from the debugger `idebug` in interactive mode. This is described further in section 6.3.2 of the *ANSI C Toolset User Guide*.

See also:

`ChanIn ChanInChanFail ChanOutTimeFail`

ChanOut

Outputs data on a channel.

Synopsis:

```
#include <channel.h>
void ChanOut(Channel *c, void *cp, int count);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`void *cp`	A pointer to an array containing the output data.
`int count`	The number of bytes of data.

Results:

Returns no result.

Errors:

None.

Description:

Outputs count bytes of data on the channel `c`. The data is taken from the array pointed to by `cp`. The effect of this routine is undefined if `count` ≤ `0`.

See also:

`ChanIn ChanOutInt ChanOutChar`

ChanOutChanFail

Outputs data or aborts on failure.

Synopsis:

```
#include <channel.h>
int ChanOutChanFail(Channel *chan, void *cp,
                    int count, Channel *failchan);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`void *cp`	A pointer to an array containing the output data.
`int count`	The number of bytes of data.
`Channel *failchan`	A pointer to the channel on which the failure message is received.

Results:

Returns zero (0) if communication completes normally, one (1) if communication is aborted by a message on the failure channel.

Errors:

None.

Description:

`ChanOutChanFail` is used to perform reliable channel communication on a link. It outputs `count` bytes of data on the specified channel from the array pointed to by `cp`. The function can be aborted by an integer, and only an integer, passed on the channel `failchan`. Typically `failchan` will be a channel from a process which is monitoring the integrity of the link.

Note: this function may not be used on a virtual channel supplied from either the configurer or from the debugger `idebug` in interactive mode. This is described further in section 6.3.2 of the *ANSI C Toolset User Guide*.

See also:

`ChanOut` `ChanOutTimeFail`

ChanOutChar

Outputs one byte on a channel.

Synopsis:

```
#include <channel.h>
void ChanOutChar(Channel *c, unsigned char ch);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`unsigned char ch`	The byte to be output.

Results:

Returns no result.

Errors:

None.

Description:

Outputs a single byte on a channel.

> **Note:** The prototype of `ChanOutChar` has changed from previous releases of the toolset i.e. the D7214, D6214, D5214 and D4214 products, where `ChanOutChar` was of type `Char`.

See also:

`ChanInChar` `ChanOut`

ChanOutInt

Outputs an integer on a channel.

Synopsis:

```
#include <channel.h>
void ChanOutInt(Channel *c, int n);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`int n`	The integer to be output.

Results:

Returns no result.

Errors:

None.

Description:

Outputs a single integer on a channel.

See also:

`ChanOutInt ChanIn`

ChanOutTimeFail

Outputs data on a channel or times out.

Synopsis:

```
#include <channel.h>
int ChanOutTimeFail(Channel *chan, void *cp,
                    int count, int time);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`void *cp`	A pointer to an array containing the output data.
`int count`	The number of bytes of data.
`int time`	The absolute time after which the communication is aborted if no output occurs.

Results:

Returns zero if the communication is successful, one (1) if timeout occurs before the communication completes.

Errors:

None.

Description:

`ChanOutTimeFail` is used to timeout channel communication on a link. It outputs count bytes of data on the specified channel from the array pointed to by `cp`. The functions aborts if the transputer clock reaches the specified absolute time before the communication takes place. Typically it is used to notify delay on a link so that the communication can be routed elsewhere.

Note: this function may not be used on a virtual channel supplied from either the configurer or from the debugger `idebug` in interactive mode. This is described further in section 6.3.2 of the *ANSI C Toolset User Guide*.

See also:

`ChanOut` `ChanOutChanFail`

ChanReset

Resets a channel.

Synopsis:

```
#include <channel.h>
int ChanReset(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the channel to be reset.

Results:

Returns either `NotProcess_p`, or the transputer process descriptor **Wdesc**.

Errors:

None.

Description:

Resets a channel to the value `NotProcess_p` and returns the transputer process descriptor of the process waiting to communicate on the channel, or `NotProcess_p`. If the value returned is `NotProcess_p`, no process was waiting on the channel, and any communication on that channel had completed successfully.

This function should not be used to reset a soft channel (a channel that connects processes on the same processor), which has not been previously initialized using `ChanInit` or `ChanAlloc`. There is in fact little point using this function on a soft channel, because communication in that case can be assumed to be secure.

`NotProcess_p` is defined in `channel.h`.

Note: this function may not be used on a virtual channel supplied from either the configurer or from the debugger `idebug` in interactive mode. This is described further in section 6.3.2 of the *ANSI C Toolset User Guide*.

See also:

`ChanInit`

clearerr

Clears error and end of file indicators for a file stream.

Synopsis:

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns no value.

Errors:

None.

Description:

`clearerr` clears the error and end of file indicators for a file stream.

See also:

`rewind`

clock

Determines the amount of processor time used.

Synopsis:

```
#include <time.h>
clock_t clock(void);
```

Arguments:

None.

Results:

Returns the time used by the program since it started, or (`clock_t`)–1 on error.

`clock` returns a value at the priority of the calling process.

Errors:

The value (`clock_t`)–1, indicating an error, is returned if any of the following occur:

- the processor time is not available;
- the value cannot be represented;
- the priority of the process calling `clock` is different to that of the main process.

Description:

`clock` returns the processor time used by the program since it started. The era for the `clock` function extends from directly before the user's `main` function is called until program termination.

To obtain the time in seconds the return value should be divided by `CLOCKS_PER_SEC`.

Note: `CLOCKS_PER_SEC` takes the constant value `CLOCKS_PER_SEC_HIGH` or `CLOCKS_PER_SEC_LOW` depending on the priority of the process calling `clock` i.e. high or low respectively.

- `CLOCKS_PER_SEC_HIGH` has the value 1000000
- `CLOCKS_PER_SEC_LOW` has the value 15625

When the priority of the call to `clock` is known `CLOCKS_PER_SEC_HIGH` or `CLOCKS_PER_SEC_LOW` can be used directly.

`CLOCKS_PER_SEC` is defined in the header file `time.h`, the two constants `CLOCKS_PER_SEC_HIGH` and `CLOCKS_PER_SEC_LOW` are defined in the header file `process.h`.

Warning: the type definition of `clock_t` is `unsigned int`, however, on a 16–bit transputer the value of high priority `CLOCKS_PER_SEC` is to big to be held in type `clock_t`.

Thus in the case of a high priority process on a 16–bit transputer, compiling the following expression (which calculates elapsed time in seconds) will result in a type `long` instead of `int`.

```
clock() / CLOCKS_PER_SEC
```

In addition, because the high priority timer on a 16–bit transputer wraps around after the very short interval of 65 ms, the result of the above expression will always be '0' in this case.

`clock` is side effect free.

Note: `clock` should not be used in any C code which is to be imported by occam using `callc.lib`.

See also:

`asctime ctime localtime strftime difftime mktime time`

`close`

Closes a file. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int close(int fd);
```

Arguments:

`int fd`	File descriptor of the file to be closed.

Results:

Returns 0 if successful or –1 on error.

Errors:

If an error occurs `close` sets `errno` to the value `EIO` and returns –1.

Description:

`close` is the lower level function used by `fclose`. It takes a file descriptor as a argument instead of a `FILE` pointer. The file descriptor will usually have been returned by the `open` or `creat` functions.

`close` is not included in the reduced library.

cos

Calculates the cosine of the argument.

Synopsis:

```
#include <math.h>
double cos(double x);
```

Arguments:

`double x`	A number in radians.

Results:

Returns the cosine of `x` in radians.

Errors:

None.

Description:

`cos` calculates the cosine of a number.

See also:

`cosf`

`cosf`

Calculates the cosine of a `float` number.

Synopsis:

```
#include <mathf.h>
float cosf(float x);
```

Arguments:

`float x`	A number in radians.

Results:

Returns the cosine of `x` in radians.

Errors:

None.

Description:

`float` form of `cos`.

See also:

`cos`

`cosh`

Calculates the hyperbolic cosine of the argument.

Synopsis:

```
#include <math.h>
double cosh(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the hyperbolic cosine of `x` or if a range error occurs returns `HUGE_VAL` (with the same sign as the correct value of the function).

Errors:

A range error will occur if `x` is so large that `cosh` would result in an overflow. In this case `cosh` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`cosh` calculates the hyperbolic cosine of a number.

See also:

`coshf`

coshf

Calculates the hyperbolic cosine of a `float` number.

Synopsis:

```
#include <mathf.h>
float coshf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the hyperbolic cosine of `x` or if a range error occurs returns `HUGE_VAL_F` (with the same sign as the correct value of the function).

Errors:

A range error will occur if `x` is so large that `coshf` would result in an overflow. In this case `coshf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `cosh`.

See also:

`cosh`

CrcByte

Calculate CRC of most significant byte of an integer.

Synopsis:

```
#include <misc.h>
int CrcByte(int data, int crc_in, int generator);
```

Arguments:

`int data`	The most significant byte of this integer forms the data for the CRC calculation.
`int crc_in`	Initial value of CRC, or CRC value obtained from previous call.
`int generator`	The CRC generating polynomial.

Results:

Returns the CRC of the most significant byte of `data` combined with `crc_in`.

Errors:

None.

Description:

A full description of all the CRC functions supplied is given in appendix C.

`CrcByte` performs a cyclic redundancy check over the most significant byte of `data` using `crc_in` as the initial CRC value. `generator` is the CRC generating polynomial.

`CrcByte` is side effect free.

Example:

```
int data;
int crc_in;
int crc;
int generator;

crc = CrcByte(data, crc_in, generator);
```

When compiling for transputers which have the *crcbyte* instruction, calls to `CrcByte` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

See also:

`CrcWord CrcFromLsb CrcFromMsb`

CrcFromLsb Calculates the CRC of a byte sequence starting at the least significant bit.

Synopsis:

```
#include <misc.h>
int CrcFromLsb(const char *string, size_t length,
               int generator, int old_crc);
```

Arguments:

`const char *string`	Pointer to the start of the byte sequence for which the CRC is to be calculated.
`size_t length`	Number of bytes in the sequence pointed to by `string`.
`int generator`	The CRC generating polynomial.
`int old_crc`	Initial value of CRC.

Results:

CRC of the given byte sequence, starting at the least significant bit.

Errors:

None.

Description:

A full description of all the CRC functions supplied is given in appendix C.

The `CrcFromLsb` function is provided to accommodate byte sequences in big-endian format. The most significant bit of `string` is taken to be bit 0 of `string[0]`. The generated CRC is given in big-endian format. `generator` and `old_crc` are taken to be in little-endian format.

See also:

`CrcFromMsb` `CrcWord` `CrcByte`

CrcFromMsb Calculates the CRC of a byte sequence starting at the most significant bit.

Synopsis:

```
#include <misc.h>
int CrcFromMsb(const char *string, size_t length,
               int generator, int old_crc);
```

Arguments:

`const char *string`	Pointer to the start of the byte sequence for which the CRC is to be calculated.
`size_t length`	Number of bytes in the byte sequence pointed to by `string`.
`int generator`	The CRC generating polynomial.
`int old_crc`	Initial value of CRC.

Results:

CRC of the given byte sequence, starting at the most significant bit.

Errors:

None.

Description:

A full description of all the CRC functions supplied is given in appendix C.

The `CrcFromMsb` function is intended for byte sequences in normal transputer format (little-endian). The most significant bit of the given byte sequence is taken to be bit-16 or bit-32, depending, on the word size of the processor, of `string[length - 1]`.

`generator`, `old_crc` and the result of `CrcFromMsb` are all also in normal transputer format (little-endian).

See also:

`CrcFromLsb` `CrcWord` `CrcByte`

CrcWord

Calculate CRC of an integer.

Synopsis:

```
#include <misc.h>
int CrcWord(int data, int crc_in, int generator);
```

Arguments:

`int data`	The data for the CRC calculation.
`int crc_in`	Initial value of CRC, or CRC value obtained from previous call.
`int generator`	The CRC generating polynomial.

Results:

Returns the CRC of `data` combined with `crc_in`.

Errors:

None.

Description:

A full description of all the CRC functions supplied is given in appendix C.

`CrcWord` performs a cyclic redundancy check over the single `int data` using `crc_in` which is the CRC value obtained from the previous call (or the initial CRC value). `generator` is the CRC generating polynomial. Can be used iteratively on a sequence of `ints` to obtain a CRC value for the sequence.

`CrcWord` is side effect free.

Example:

```
int data[10];
int i;
int crc;
int generator;

crc = 0;
for (i = 0; i < 10; i++)
  crc = CrcWord(data[i], crc, generator);
```

When compiling for transputers which have the *crcword* instruction, calls to `CrcWord` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

See also:

`CrcByte CrcFromLsb CrcFromMsb`

`creat`

Creates a file for writing. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int creat(char *name, int flag);
```

Arguments:

`char *name`	The name of the file to be created.
`int flag`	A number which specifies the mode in which the file is opened.

Results:

Returns a file descriptor for the file, or –1 on error.

Errors:

If an error occurs `creat` sets `errno` to the value `EIO` and returns –1.

Description:

`creat` creates a file with filename `name` and opens it in 'write' and 'truncate' modes. If the file already exists, and if the host system permits, the file is overwritten.

The value of `flag` determines how the file is opened. It can take two values, as follows:

`O_BINARY`	Open file in binary mode.
`O_TEXT`	Open file as a text file.

The default is to open the file as a text file.

`creat` has the same effect as a call to `open` with the following arguments:

```
open(name, O_WRONLY | O_TRUNC | flag);
```

`creat` is not included in the reduced library.

See also:

`open`

ctime

Converts a calendar time value to a string.

Synopsis:

```
#include <time.h>
char *ctime(const time_t *timer);
```

Arguments:

`const time_t *timer`	A pointer to the calendar time.

Results:

Returns a pointer to a string representation of the time.

Errors:

None.

Description:

`ctime` converts the value pointed to by `timer` to a broken–down time structure, and then writes the contents of the structure into a string in the following form:

Thu Nov 05 18:19:01 1987

(See section 1.3.16 for a definition of broken–down time).

`ctime` is equivalent to the following call to `asctime`:

```
asctime (localtime(timer));
```

Example:

```
/* Displays the current time */
#include  <time.h>
#include  <stdio.h>

int main( void )
{
   time_t now;

   time(&now);
   printf("The time is: %s\n",ctime(&now));
}
```

Note: Care should be taken when calling `ctime` in a concurrent environment. Calls to the function by independently executing unsynchronized processes may corrupt the returned time value.

See also:

`asctime` `localtime` `strftime` `clock` `difftime` `mktime` `time` `gmtime`

debug_assert

Stops process/alerts debugger if condition fails.

Synopsis:

```
#include <misc.h>
void debug_assert(const int exp);
```

Arguments:

`const int exp`	An integer expression for the condition to be asserted.

Results:

Returns no result.

Errors:

None.

Description:

`debug_assert` replaces `assert` for programs that will be debugged in breakpoint mode. If `expression` evaluates FALSE `debug_assert` stops the process and sends process data to the debugger. If `expression` evaluates TRUE no action is taken.

If the program is not being run within the breakpoint debugger and the assertion fails, then the function behaves like `debug_stop`.

See also:

`assert` `debug_message` `debug_stop`

debug_message

Inserts a debugging message.

Synopsis:

```
#include <misc.h>
void debug_message(const char *message);
```

Arguments:

`const char *message`	The text of the message.

Results:

Returns no result.

Errors:

None.

Description:

`debug_message` sends a message to the debugger which is displayed along with normal program output. Only the first 80 characters of the message are displayed.

If the program is not being run within the breakpoint debugger the function has no effect.

See also:

`debug_assert` `debug_stop`

debug_stop

Stops a process and notifies the debugger.

Synopsis:

```
#include <misc.h>
void debug_stop(void);
```

Arguments:

None.

Results:

Returns no result.

Errors:

None.

Description:

`debug_stop` stops the process and sends process data to the debugger.

If the program is not being run within the breakpoint debugger then the function stops the process or processor, depending on the error mode in which the processor is executing.

See also:

`debug_assert` `debug_message` `halt_processor`

difftime

Calculates the difference between two calendar times.

Synopsis:

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Arguments:

`time_t time1`	The first time.
`time_t time0`	The second time.

Results:

Returns the difference, in seconds, between `time1` and `time0`.

Errors:

None.

Description:

`difftime` calculates the difference in time between `time1` and `time0` (`time1 - time0`).

`difftime` is side effect free.

See also:

`asctime ctime localtime strftime clock mktime time gmtime`

DirectChanIn

Inputs data on a channel.

Synopsis:

```
#include <channel.h>
void DirectChanIn(Channel *c, void *cp, int count);
```

Arguments:

`Channel *c`	A pointer to the input channel.
`void *cp`	A pointer to the array where the data will be stored.
`int count`	The number of bytes of data.

Results:

Returns no result.

Errors:

None.

Description:

Inputs `count` bytes of data on the specified channel and stores them in the array pointed to by `cp`. The effect of this routine is undefined if `count` $\leq$ `0`.

This routine is a fast, inline, version of `ChanIn`: input is performed directly, using the transputer's input instruction; therefore this routine can only be used on the following sorts of channel:

- a *soft* channel; i.e. any channel which communicates with a process on the same processor
- a *direct* channel *provided* `idebug` is not being used in interactive mode. A direct channel is a configuration level channel which occurs when no more than two channels (one in each direction) are placed on a single link, between adjacent processors.

  The suggested use is with either soft channels or *edge* channels which communicate outside the network with a device other than the host. **Note:** it can be dangerous to make assumptions about the implementation of direct channels. See section 6.3.1 in the *ANSI C Toolset User Guide* for further guidance.

Calls to `DirectChanIn` are implemented inline, provided that the header file `<channel.h>` has been included in the source.

See also:

`ChanIn` `ChanInChar` `ChanInInt` `ChanInit`
`DirectChanInChar` `DirectChanInInt`

DirectChanInChar

Input one byte on a channel.

Synopsis:

```
#include <channel.h>
unsigned char DirectChanInChar(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the input channel.

Results:

Returns the input byte.

Errors:

None.

Description:

Inputs a single byte on a channel.

This routine is a fast, inline, version of `ChanInChar`: input is performed directly, using the transputer's input instruction; therefore this routine can only be used on the following sorts of channel:

- a *soft* channel; i.e. any channel which communicates with a process on the same processor
- a *direct* channel *provided* `idebug` is not being used in interactive mode. A direct channel is a configuration level channel which occurs when no more than two channels (one in each direction) are placed on a single link, between adjacent processors.

  The suggested use is with either soft channels or *edge* channels which communicate outside the network with a device other than the host. **Note:** it can be dangerous to make assumptions about the implementation of direct channels. See section 6.3.1 in the *ANSI C Toolset User Guide* for further guidance.

Calls to `DirectChanInChar` are implemented inline, provided that the header file `<channel.h>` has been included in the source.

See also:

`ChanInChar` `ChanOutChar`
`DirectChanIn` `DirectChanOutChar`

DirectChanInInt

Inputs an integer on a channel.

Synopsis:

```
#include <channel.h>
int DirectChanInInt(Channel *c);
```

Arguments:

`Channel *c`	A pointer to the input channel.

Results:

Returns the input integer.

Errors:

None.

Description:

Inputs a single integer on a channel.

This routine is a fast, inline, version of **`ChanInInt`**: input is performed directly, using the transputer's input instruction; therefore this routine can only be used on the following sorts of channel:

- a *soft* channel; i.e. any channel which communicates with a process on the same processor
- a *direct* channel *provided* **`idebug`** is not being used in interactive mode. A direct channel is a configuration level channel which occurs when no more than two channels (one in each direction) are placed on a single link, between adjacent processors.

  The suggested use is with either soft channels or *edge* channels which communicate outside the network with a device other than the host. **Note:** it can be dangerous to make assumptions about the implementation of direct channels. See section 6.3.1 in the *ANSI C Toolset User Guide* for further guidance.

Calls to `DirectChanInInt` are implemented inline, provided that the header file **`<channel.h>`** has been included in the source.

See also:

`ChanInInt ChanOutInt`
`DirectChanIn DirectChanOutInt`

DirectChanOut

Outputs data on a channel.

Synopsis:

```
#include <channel.h>
void DirectChanOut(Channel *c, void *cp, int count);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`void *cp`	A pointer to an array containing the output data.
`int count`	The number of bytes of data.

Results:

Returns no result.

Errors:

None.

Description:

Outputs `count` bytes of data on the channel `c`. The data is taken from the array pointed to by `cp`. The effect of this routine is undefined if `count` ≤ `0`.

This routine is a fast, inline, version of `ChanOut`: output is performed directly, using the transputer's output instruction; therefore this routine can only be used on the following sorts of channel:

- a *soft* channel; i.e. any channel which communicates with a process on the same processor
- a *direct* channel *provided* `idebug` is not being used in interactive mode. A direct channel is a configuration level channel which occurs when no more than two channels (one in each direction) are placed on a single link, between adjacent processors.

  The suggested use is with either soft channels or *edge* channels which communicate outside the network with a device other than the host. **Note:** it can be dangerous to make assumptions about the implementation of direct channels. See section 6.3.1 in the *ANSI C Toolset User Guide* for further guidance.

Calls to `DirectChanOut` are implemented inline, provided that the header file `<channel.h>` has been included in the source.

See also:

`ChanOut` `ChanOutInt` `ChanOutChar`
`DirectChanIn` `DirectChanOutInt` `DirectChanOutChar`

DirectChanOutChar

Outputs one byte on a channel.

Synopsis:

```
#include <channel.h>
void DirectChanOutChar(Channel *c, unsigned char ch);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`unsigned char ch`	The byte to be output.

Results:

Returns no result.

Errors:

None.

Description:

Outputs a single byte on a channel.

This routine is a fast, inline, version of `ChanOutChar`: output is performed directly, using the transputer's output instruction; therefore this routine can only be used on the following sorts of channel:

- a *soft* channel; i.e. any channel which communicates with a process on the same processor
- a *direct* channel *provided* `idebug` is not being used in interactive mode. A direct channel is a configuration level channel which occurs when no more than two channels (one in each direction) are placed on a single link, between adjacent processors.

  The suggested use is with either soft channels or *edge* channels which communicate outside the network with a device other than the host. **Note:** it can be dangerous to make assumptions about the implementation of direct channels. See section 6.3.1 in the *ANSI C Toolset User Guide* for further guidance.

Calls to `DirectChanOutChar` are implemented inline, provided that the header file `<channel.h>` has been included in the source.

See also:

`ChanInChar` `ChanOutChar`
`DirectChanInChar` `DirectChanOut`

DirectChanOutInt

Outputs an integer on a channel.

Synopsis:

```
#include <channel.h>
void DirectChanOutInt(Channel *c, int n);
```

Arguments:

`Channel *c`	A pointer to the output channel.
`int n`	The integer to be output.

Results:

Returns no result.

Errors:

None.

Description:

Outputs a single integer on a channel.

This routine is a fast, inline, version of `ChanOutInt`: output is performed directly, using the transputer's output instruction; therefore this routine can only be used on the following sorts of channel:

- a *soft* channel; i.e. any channel which communicates with a process on the same processor
- a *direct* channel *provided* **`idebug`** is not being used in interactive mode. A direct channel is a configuration level channel which occurs when no more than two channels (one in each direction) are placed on a single link, between adjacent processors.

  The suggested use is with either soft channels or *edge* channels which communicate outside the network with a device other than the host. **Note:** it can be dangerous to make assumptions about the implementation of direct channels. See section 6.3.1 in the *ANSI C Toolset User Guide* for further guidance.

Calls to `DirectChanOutInt` are implemented inline, provided that the header file `<channel.h>` has been included in the source.

See also:

`ChanInInt ChanOutInt`
`DirectChanInInt DirectChanOut`

div

Calculates the quotient and remainder of a division.

Synopsis:

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Arguments:

`int numer`	The numerator.
`int denom`	The denominator.

Results:

Returns a structure of type `div_t` which consists of the quotient and remainder. The structure contains:

`int quot`	The quotient.
`int rem`	The remainder.

Errors:

If the result cannot be represented the behavior of `div` is undefined.

Description:

`div` calculates the quotient and remainder formed by dividing the numerator `numer` by the denominator `denom`.

`div` is side effect free.

See also:

`ldiv`

exit

Terminates a program.

Synopsis:

```
#include <stdlib.h>
void exit(int status);
```

Arguments:

`int status`	A value denoting the program termination status.

Results:

`exit` does not return.

Errors:

None.

Description:

`exit` causes normal program termination and passes a termination code back to the calling environment.

`exit` performs the following actions before the returning control to the calling environment:

1 The functions recorded by `atexit` are called in reverse order of their registration.

2 All open output streams are flushed.

3 All open files are closed.

4 All files created by `tmpfile` are removed.

The value of `status` denotes success or failure of the program and determines the value of the termination code passed back to the calling environment. If `status` is zero or equal to `EXIT_SUCCESS` then the program is deemed to have been successful and the value of the termination code passed to the calling environment is `EXIT_SUCCESS`. If `status` is `EXIT_FAILURE` then the program is deemed to have been unsuccessful in some way and the value of the termination code passed back to the calling environment is `EXIT_FAILURE`. If `status` has any other value then the termination code passed back to the calling environment is equal to `status`.

Further actions on program termination are determined by the host environment of the program. There are three cases:

1 A program linked with the full library which has not been dynamically loaded:

The environment of a program linked with the full library is its connection to the server. `exit` causes all such programs, except those using the `PROC.ENTRY` entry point, to terminate the server. The server returns the same termination code as is set up by `exit` except that `EXIT_SUCCESS` and `EXIT_FAILURE` are translated to the equivalent host specific success and failure code.

2 A program linked with the reduced library which has not been dynamically loaded :

Such a program can be considered to have no environment as such. There is no server and so nowhere to pass the termination code to. In this case the termination code is lost.

3 A program which has been dynamically loaded:

The environment of a dynamically loaded program is the program which loaded and invoked it, its parent. It is not the job of a child program to terminate the server, this is a task for the parent, if the parent is of type 1 above. The termination code set up by `exit` is stored in an implementation defined manner (see section 3.6.9).

A summary of the action of `exit`, when not used in a dynamically loaded program is as follows:

C entry point	Terminate server
`C.ENTRYD` (linked with `cstartup.lnk`)	Yes
`C.ENTRYD.RC` (linked with `cstartrd.lnk`)	No
`C.ENTRY` (linked with `cnonconf.lnk`)	Yes
`MAIN.ENTRY` (Type 1 interface) †	Yes
`PROC.ENTRY` (Type 2 interface) †	No
`PROC.ENTRY.RC` (Type 3 interface) †	No
† Entry points used by occam interface code – a method of mixed language programming described in chapter 10 of the *ANSI C Toolset User Guide*.	

For configured programs which are not dynamically loaded and which use the `C.ENTRYD` entry point (i.e. are linked with `cstartup.lnk`), but which do not require to terminate the server, the equivalent function `exit_noterminate` should be used.

Caution : `exit` should not be called from a function which is invoked as a C parallel process. The effect on the program may be unpredictable. This restriction does not apply to a call to `exit` which is meant to terminate the execution of a dynamically loaded program which has been invoked as a parallel process.

Note: that `exit` should not be used by any C code which is to be imported by occam, using `callc.lib`.

Note: The behavior of `exit` has changed from previous releases of the toolset i.e. the D7214, D6214, D5214 and D4214 products, where `exit` did not terminate the server. Using the depreciated startup linker file `startup.lnk`, gives the original behavior.

Example:

```
#include  <stdlib.h>
#include  <stdio.h>

int main( void )
{
   printf("About to do an exit\n");
   exit(EXIT_SUCCESS);
   printf("Not printed\n");
}
```

See also:

`atexit` `exit_repeat` `exit_terminate` `exit_noterminate`

exit_noterminate

Version of `exit` for configured processes.

Synopsis:

```
#include <misc.h>
void exit_noterminate(int status);
```

Arguments:

`int status`	A value to be passed back to the calling environment.

Results:

Returns no result.

Errors:

None.

Description:

`exit_noterminate` is equivalent to `exit`, but designed for use in a configured process when it is not desirable for the default action of terminating the server to occur.

`exit_noterminate` will only override the termination of the server in configured programs linked with the full runtime library. In all other cases it acts like `exit` and `status` is passed back to the calling environment.

The effect of `exit_noterminate` on server termination is summarized as follows:

C entry point	Terminate server
`C.ENTRYD` (linked with `cstartup.lnk`)	No
`C.ENTRYD.RC` (linked with `cstartrd.lnk`)	No
`C.ENTRY` (linked with `cnonconf.lnk`)	Yes
`MAIN.ENTRY` (Type 1 interface) †	Yes
`PROC.ENTRY` (Type 2 interface) †	No
`PROC.ENTRY.RC` (Type 3 interface) †	No
† Entry points used by occam interface code – a method of mixed language programming described in chapter 10 of the *ANSI C Toolset User Guide*.	

Note: if use is made of the predefined constants `EXIT_FAILURE` or `EXIT_SUCCESS` then the header file `stdlib.h` must be included.

Caution: `exit_noterminate` should not be called from a C function that is running *in parallel* with any other function. The effect on the program may be unpredictable. This restriction does not apply to a call to `exit_noterminate` which is meant to terminate the execution of a dynamically loaded program which has been invoked as a parallel process. Calling `exit_noterminate` from a dynamically loaded code is equivalent to calling `exit`.

Note: that `exit_noterminate` should not be used by any C code which is to be imported by occam, using `callc.lib`.

See also:

`exit` `exit_repeat` `exit_terminate`

`exit_repeat` Terminates a program so that it can be restarted.

Caution: *use of this function should be avoided since it will not be supported in future releases of the toolset.*

Synopsis:

```
#include <misc.h>
void exit_repeat(int status);
```

Arguments:

`int status`	A value to be passed back to the calling environment.

Results:

Returns no result.

Errors:

None.

Description:

`exit_repeat` terminates the C program and returns its argument to the calling environment. Unlike `exit`, `exit_repeat` retains the program and allows it to be rerun without re-booting the transputer.

Only programs which consist of a single C program running on a single transputer, and which have been made bootable using the collector '`T`' option, can be repeat invoked. In all other cases `exit_repeat` acts like `exit`.

Caution: `exit_repeat` should not be called from a C function that is running *in parallel* with any other function. The effect on the program may be unpredictable.

The first element of the `argv` array is lost in the process of calling `exit_repeat`. Therefore programs that read the program name from the first element of the array will need to be re-booted.

Note: If use is made of the predefined constants `EXIT_FAILURE` or `EXIT_SUCCESS` then the header file `stdlib.h` must be included.

Note: that `exit_repeat` should not be used by any C code which is to be imported by occam, using `callc.lib`.

See also:

`exit`

exit_terminate

Version of `exit` for configured processes.

Synopsis:

```
#include <misc.h>
void exit_terminate(int status);
```

Arguments:

`int status`	A value to be passed back to the calling environment.

Results:

Returns no result.

Errors:

None.

Description:

`exit_terminate` has exactly the same action as `exit`. It is included for compatibility with earlier issues of the toolset e.g. the D7214, D6214, D5214 and D4214 products and may not be supported in future versions of the toolset.

Caution: `exit_terminate` should not be called from a C function that is running *in parallel* with any other function. The effect on the program may be unpredictable.

Note: that `exit_terminate` should not be used by any C code which is to be imported by occam, using `callc.lib`.

See also:

`exit` `exit_repeat` `exit_noterminate`

exp

Calculates the exponential function of the argument.

Synopsis:

```
#include <math.h>
double exp(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the exponential function of **x** or returns **HUGE_VAL** (with the same sign as the correct value of the function) if a range error occurs.

Errors:

A range error occurs if the result of raising e to the power of **x** would cause overflow. In this case **exp** returns the value **HUGE_VAL** (with the same sign as the correct value of the function) and **errno** is set to **ERANGE**.

Description:

exp calculates the value of the constant e (2.71828...) raised to the power of a number.

See also:

expf

`expf` Calculates the exponential function of a `float` number.

Synopsis:

```
#include <mathf.h>
float expf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the exponential function of `x` returns `HUGE_VAL_F` (with the same sign as the correct value of the function) if a range error occurs.

Errors:

A range error occurs if the result of raising e to the power of `x` would cause overflow. In this case `expf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `exp`.

See also:

`exp`

fabs

Calculates the absolute value of a floating point number.

Synopsis:

```
#include <math.h>
double fabs(double x);
```

Arguments:

`double x` A number.

Results:

Returns the absolute value of the argument.

Errors:

None.

Description:

`fabs` calculates the absolute value of a number.

`fabs` is side effect free.

See also:

`fabsf`

`fabsf`

Calculates the absolute value of a `float` number.

Synopsis:

```
#include <mathf.h>
float fabsf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the absolute value of the argument.

Errors:

None.

Description:

`float` form of `fabs`.

`fabsf` is side effect free.

See also:

`fabs`

fclose

Closes a file stream.

Synopsis:

```
#include <stdio.h>
int fclose(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to the file stream.

Results:

Returns zero if the close was successful and `EOF` if it was not.

Errors:

If an error is detected `fclose` returns `EOF`.

Description:

`fclose` closes the file stream pointed to by `stream`; any associated buffers are flushed. Any buffer which was allocated by the I/O system is de-allocated.

Buffer data which is waiting to be written is sent to the host environment for writing to the file. Buffer data which is waiting to be read is ignored.

`fclose` is called automatically when `exit` is called. `fclose` is not included in the reduced library.

See also:

`fopen`

`feof`

Tests for end of file.

Synopsis:

```
#include <stdio.h>
int feof(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns zero if the end of file indicator for `stream` is clear, non-zero if it is set.

Errors:

None.

Description:

`feof` tests the state of the end of file indicator for the file stream `stream`. It returns zero if the indicator is clear, and non-zero if it is set.

`feof` is not included in the reduced library.

See also:

`ferror`

`ferror`

Tests for a file error.

Synopsis:

```
#include <stdio.h>
int ferror(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns zero if the error indicator for `stream` is clear, and non-zero if it is set.

Errors:

None.

Description:

`ferror` tests the state of the error indicator for the file stream `stream`. It returns zero if the error indicator is clear, and non-zero if it is set.

`ferror` is not included in the reduced library.

See also:

`feof`

fflush

Flushes an output stream.

Synopsis:

```
#include <stdio.h>
int fflush(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to the stream to be flushed.

Results:

Returns `EOF` if a write error occurred, otherwise 0.

Errors:

If a write error occurs, `fflush` returns `EOF`.

Description:

If `stream` points to an output stream, `fflush` causes any outstanding data for the stream to be written to the file. The behavior is undefined for a stream which is neither open for output nor update.

If `stream` is `NULL`, `fflush` flushes all streams that are open for output.

`fflush` is not included in the reduced library.

See also:

`ungetc`

fgetc

Reads a character from a file stream.

Synopsis:

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns the next character from the file stream.

Errors:

If the stream is at the end of the file, the end of file indicator for the stream is set and `fgetc` returns `EOF`. If a read error occurs, the error indicator for the stream is set and `fgetc` returns `EOF`.

Description:

`fgetc` returns the next character from the opened file identified by the file stream pointer `stream`, and advances the read/write position indicator for the file stream.

`fgetc` is not included in the reduced library.

See also:

`fgets` `fputc` `getc` `ungetc`

`fgetpos`

Obtains the value of the file position indicator.

Synopsis:

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`fpos_t *pos`	A pointer to an object where the current value of the file position indicator can be stored.

Results:

Returns zero if the operation was successful. If the operation fails `fgetpos` sets `errno` to `EFILPOS` and returns non-zero.

Errors:

If the operation was unsuccessful, `fgetpos` returns a non-zero value and stores `EFILPOS` in `errno`.

Description:

`fgetpos` stores the value of the file position indicator of the file stream `stream` in the object pointed to by `pos`. This information is in a form usable by the `fsetpos` function.

`fgetpos` is not included in the reduced library.

See also:

`fsetpos`

fgets

Reads a line from a file stream.

Synopsis:

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Arguments:

`char *s`	A pointer to a buffer to receive the string.
`int n`	The size of the array.
`FILE *stream`	A pointer to a file stream.

Results:

Returns `s` if successful or a `NULL` pointer on error.

Errors:

`fgets` returns a `NULL` pointer if a read error occurs and the contents of the array are undefined. If end of file is encountered before a character is read `fgets` returns `NULL` and the contents of the array remain unchanged.

Description:

`fgets` reads a string of a maximum (n–1) characters from the file stream identified by `stream`. `fgets` stops reading when it encounters a newline character or an end of file character. A string terminating character is written into the array after the last character read. The newline character forms part of the string.

`fgets` is not included in the reduced library.

See also:

`fgetc` `fputs` `gets`

`filesize` Determines the size of a file. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
long int filesize(int fd);
```

Arguments:

`int fd`	A file descriptor.

Results:

Returns the size of the file in bytes or -1 on error.

Errors:

If an error occurs `filesize` sets `errno` to the value `EIO`.

Description:

`filesize` takes a file descriptor and returns the size of the file in bytes. If the file is open for writing, `filesize` returns the current size of the file.

`filesize` is not included in the reduced library.

`floor` Calculates the largest integer not greater than the argument.

Synopsis:

```
#include <math.h>
double floor(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the largest integer (expressed as a double) which is not greater than `x`.

Errors:

None.

Description:

`floor` calculates the largest integer which is not greater than `x`.

`floor` is side effect free.

See also:

`ceil floorf`

floorf

`float` form of `floor`.

Synopsis:

```
#include <mathf.h>
float floorf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the largest integer (expressed as a `float`) which is not greater than `x`.

Errors:

None.

Description:

`float` form of `floor`.

`floorf` is side effect free.

See also:

`ceilf` `floor`

fmod

Calculates the floating point remainder of x/y.

Synopsis:

```
#include <math.h>
double fmod(double x, double y);
```

Arguments:

`double x`	The dividend.
`double y`	The divisor.

Results:

Returns (with the same sign as `x`) the floating point remainder of `x/y`. If `y` is zero `errno` obtains the value `EDOM` and `fmod` returns zero.

Errors:

A domain error occurs if `y` is zero, and the function then returns zero. A range error occurs if the result is not representable.

Description:

`fmod` calculates the floating point remainder of `x/y`.

See also:

`fmodf`

fmodf

Calculates the floating point remainder of `x/y`.

Synopsis:

```
#include <mathf.h>
float fmodf(float x, float y);
```

Arguments:

`float x`	The dividend.
`float y`	The divisor.

Results:

Returns (with the same sign as `x`) the floating point remainder of `x/y`. If `y` is zero `errno` obtains the value `EDOM` and `fmodf` returns zero.

Errors:

A domain error occurs if `y` is zero and a range error occurs if the result is not representable.

Description:

`float` form of `fmod`.

See also:

`fmod`

fopen

Opens a file.

Synopsis:

```
#include <stdio.h>
FILE *fopen(const char *filename,
            const char *mode);
```

Arguments:

`const char *filename`	The name of the file to be opened.
`const char *mode`	A string which specifies the mode in which the file is to be opened.

Results:

Returns a file pointer to the stream associated with the newly opened file. `fopen` returns a `NULL` pointer if it cannot open the file.

Errors:

If a file opened for reading does not exist or the open operation fails for any other reason, `fopen` returns a `NULL` pointer.

Description:

`fopen` opens the file named by the string pointed to by `filename`, in the mode specified by the `mode` string.

`fopen` is not included in the reduced library.

The following are valid mode strings:

"r"	Opens a text file for reading.
"w"	Opens a text file for writing. If the file already exists it is truncated to zero length. If the file does not exist, it is created.
"a"	Opens a text file for appending. If the file does not exist, it is created.
"rb"	Opens a binary file for reading.
"wb"	Opens a binary file for writing. If the file already exists it is truncated to zero length. If the file does not exist, it is created.
"ab"	Opens a binary file for appending. If the file does not exist, it is created.
"r+"	Opens a text file for reading and writing.
"w+"	Creates a text file for reading and writing. If the file exists, it is truncated to zero length.
"a+"	Opens a text file for reading, and writing at the end of the file. If the file does not exist, it will be created.
"r+b" or "rb+"	Opens a binary file for reading and writing.
"w+b" or "wb+"	Creates a binary file for reading and writing. If the file exists, it is truncated to zero length.
"a+b" or "ab+"	Opens a binary file for reading and writing at the end of the file. If the file does not exist, it will be created.

File output must not be followed by file input without an intervening call to `fflush` or one of the file positioning functions `fseek`, `fsetpos` and `rewind`. Similarly, input must not be followed by output without an intervening call to one of these functions unless `EOF` is encountered. If a file is opened with a "+" in the mode string (opened for update), the file can be read from and written to without closing and reopening the file. However, you must call `fflush`, `fseek`, `fsetpos` or rewind between read and write operations.

Example:

```
#include <stdio.h>

int main( void )
{
   FILE *stream;

   stream = fopen("data.dat","r");

   if (stream == NULL)
      printf("Can't open data.dat file for
              read\n");
   else
      printf("data.dat opened for read\n");
}
```

See also:

`fclose fflush freopen fseek fsetpos rewind`

fprintf

Writes a formatted string to a file.

Synopsis:

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

Arguments:

`FILE *stream`	A pointer to an output file stream.
`const char *format`	A string of characters specifying the format.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of characters written, or a negative value if an output error occurs.

Errors:

Returns a negative value if an output error occurs.

Description:

`fprintf` writes the string pointed to by `format` to the file stream `stream`. When `fprintf` encounters a percent sign % in the string, it expands the corresponding argument into the format defined by the format tokens after the sign.

`fprintf` is not included in the reduced library.

The format tokens consist of the following items:

1. Flags (optional):

–	causes the output to be left-justified in its field.
+	causes the output to start with a '+' or '–'.
' ' (blank space)	causes the output to start with a space if positive, and a '–' if negative. If the space and + flags appear together, the space flag is ignored.
#	causes: – an octal number to begin with 0. – a hex number to begin with 0x, or 0X for the x or X conversion specifiers. – a floating point number to contain a decimal point in (e, E, f, G, g,).
0	For d,i,o,u,x,X,e,E,f,g,G, conversions (see below), leading zeros are used to pad the field width. If both 0 and – flags both appear, the 0 is ignored. For d,i,o,u,x,X conversions, if a precision is specified the 0 flag is ignored.

2. Minimum width (optional): The width is an integer constant which defines the minimum number of characters displayed. If the integer constant is replaced by an asterisk ('*'), an `int` argument following the format string in the corresponding position supplies the width.

3. Precision (optional):

The precision is specified by a decimal point followed by an integer constant which defines:

- The maximum number of characters to be written in an 's' conversion
- The number of digits to appear after the decimal point in an 'e', 'E' or 'f' conversion
- The maximum number of significant digits for a 'g' or 'G' conversion
- The minimum number of digits to appear in a 'd', 'o', 'u', 'x' or 'X' conversion.

If the integer constant is replaced by an asterisk ('*'), an `int` argument following the format string in the corresponding position supplies the precision. If the integer constant is omitted the value is taken to be zero.

4. Type specifier (optional):

`h`	Specifies that a following 'd', 'i', 'o', 'u', 'x' or 'X' conversion applies to a `short int` or `unsigned short int`, or a following 'n' conversion applies to a pointer to a short `int`.
`l`	Specifies that a following 'd', 'i', 'o', 'u', 'x' or 'X' conversion applies to a long int or unsigned long int, or a following 'n' conversion applies to a pointer to a `long int`.
`L`	Specifies that a following 'e', 'E', 'f', 'g' or 'G' conversion applies to a `long double`.

5. A single conversion character:

`d,i`	The `int` argument is converted to signed decimal format.
`o`	The `int` argument is converted to unsigned octal format.
`u`	The `int` argument is converted to unsigned decimal format.
`x`	The `int` argument is converted to unsigned hexadecimal format, using the letters 'a' to 'f'.
`X`	The `int` argument is converted to unsigned hexadecimal format, using the letters 'A' to 'F'.
`f`	The `double` argument is converted to the decimal format [–] xxx.xxxx. The number of characters after the decimal point is equal to the precision. The default precision is six.
`e,E`	The `double` argument is converted to the decimal format x.xxxxe±xx. The exponent is introduced with the conversion character (`e` or `E`). The number of characters after the decimal point is equal to the precision. The default precision is six.
`g,G`	The `double` argument is converted to an 'f' format if the exponent is less than –4 or greater than the precision. Otherwise 'g' is equivalent to 'e', and 'G' is equivalent to 'E'. Trailing zeros are removed from the result.
`c`	The `int` argument is converted to `unsigned char` and written as a single character.
`s`	Characters are written from the string pointed to by the argument, up to the string terminating character.
`p`	The argument must be a pointer to `void` and is converted to hex. format for printing.
`n`	The number of characters written so far will be put into the integer pointed to by the argument.
`%`	The % character is written.

Example:

```
#include <stdio.h>

int main( void )
{
   int i = 99;
   int count = 0;
   double fp = 1.5e5;
   char *s = "a sequence of characters";
   char nl = '\n';
   FILE *stream;

   if ( (stream = fopen("data.dat", "w")) == NULL)
      printf("Error opening data.dat for write\n");
   else
   {
     count += fprintf(stream,
                      "This is %s%c", s, nl);
     count += fprintf(stream,
                      "%d\n%f\n", i, fp);
     printf("Number of characters written to file
             was: %d\n", count);
   }
}
```

See also:

`fscanf printf`

fputc

Writes a character to a file stream.

Synopsis:

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Arguments:

`int c`	The character to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the character written if successful. If a write error occurs, `fputc` returns `EOF` and sets the error indicator for the stream.

Errors:

`fputc` returns `EOF` if a write error occurs.

Description:

`fputc` converts `c` to an unsigned char, writes it to the output stream pointed to by `stream`, and moves the read/write position for the file stream as appropriate.

`fputc` is not included in the reduced library.

See also:

`fgetc` `putc`

fputs

Writes a string to a file stream.

Synopsis:

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Arguments:

`const char *s`	A pointer to the string to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns non-negative if successful, and `EOF` if unsuccessful.

Errors:

`fputs` returns `EOF` if unsuccessful.

Description:

`fputs` writes the string pointed to by `s` to the file stream `stream`. The write does not include the string terminating character.

`fputs` is not included in the reduced library.

See also:

`fputc`

fread

Reads records from a file.

Synopsis:

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb
             FILE *stream);
```

Arguments:

`void *ptr`	A pointer to a buffer that the records are read into.
`size_t size`	The size of an individual record.
`size_t nmemb`	The maximum number of records to be read.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the number of records read. This may be less than `nmemb` if an error or end of file occurs. `fread` returns zero if `size` or `nmemb` is zero.

Errors:

Returns the current number of records read if error occurs.

Description:

`fread` reads `nmemb` records of length `size` from the file stream `stream` into the array pointed to by `ptr`.

`fread` is not included in the reduced library.

Example:

```
#include <stdio.h>
#include <stdlib.h>

#define NUMEL 10

int main()
{
  int i;
  int numout, numin, buffin[NUMEL], buffout[NUMEL];
  FILE *stream;

  /* write 10 integers to the file data.dat */

  stream = fopen("data.dat", "wb");

  if (stream == NULL)
  {
    printf("Error opening data.dat for writing\n");
    abort();
  }

  for (i = 0; i < NUMEL; i++)
    buffout[i] = i * i;

  numout = fwrite(buffout, sizeof(int), NUMEL, stream);

  fclose(stream);

  printf("Number of integers written = %d\n", numout);

  /* Now read the integers back again */

  stream = fopen("data.dat", "rb");

  if (stream == NULL)
  {
    printf("Error opening data.dat for reading\n");
    abort();
  }

  numin = fread(buffin, sizeof(int), NUMEL, stream);

  fclose(stream);

  printf("Number of integers read = %d\n", numin);

  for (i = 0; i < NUMEL; i++)
    printf("buffin[%d] = %d\n", i, buffin[i]);
}
```

See also:

`feof` `ferror` `fwrite`

`free`

Frees an area of memory.

Synopsis:

```
#include <stdlib.h>
void free(void *ptr);
```

Arguments:

`void *ptr`	A pointer to the area of memory to be freed.

Results:

Returns no result.

Errors:

If `ptr` does not match any of the pointers previously returned by `calloc`, `malloc`, or `realloc`, or if the space has already been freed by a call to `free` or `realloc`, a fatal runtime error occurs and the following message is displayed:

Fatal-C_Library-Error in free(), bad pointer or heap corrupted

Description:

`free` frees the area of memory pointed to by `ptr` if it has been previously allocated by `calloc`, `malloc`, or `realloc`. If `ptr` is a `NULL` pointer, no action occurs.

See also:

`calloc` `malloc` `realloc`

free86 Frees host memory space allocated by `alloc86`. MS-DOS only.

Synopsis:

```
#include <dos.h>
void free86(pcpointer p);
```

Arguments:

`pcpointer p`	A pointer to the host memory block to be freed.

Results:

Returns no result.

Errors:

If an error occurs `free86` sets `errno` to the value `EDOS`. Any attempt to use `free86` on operating systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`free86` returns the block of host memory identified by `p` to MS-DOS for re-use. `p` *must* be a `pcpointer` previously returned by `alloc86`.

`free86` is not included in the reduced library.

See also:

`alloc86`

freopen

Opens a file that may already be open.

Synopsis:

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode,
             FILE *stream);
```

Arguments:

`const char *filename`	The name of the file to be opened.
`const char *mode`	A string which specifies the mode in which the file is to be opened.
`FILE *stream`	A pointer to a file stream.

Results:

Returns `stream`, or a `NULL` pointer if the file cannot be opened.

Errors:

If the open fails `freopen` returns a `NULL` pointer.

Description:

`freopen` attempts to close the file associated with the file stream `stream`. Failure to close the file is ignored, error and end of file indicators for the stream are cleared, and `freopen` then opens the file referenced by `filename` and associates the file with the file stream `stream`.

The file is opened in the mode specified by the string mode. Valid modes are the same as for `fopen`.

`freopen` is not included in the reduced library.

`freopen` is normally used for redirecting the `stdin`, `stdout` and `stderr` streams.

Example:

```
#include <stdio.h>

int main( )
{
   FILE *stream;

   /* assign stdout to a named file */
   printf("This text goes to stdout\n");

   stream = freopen("data.dat", "w", stdout);
   if (stream == NULL)
      printf("Couldn't freopen stdout to
              data.dat\n");
   else
   {
      printf("This text goes to data.dat\n");
      fclose(stream);
   }
}
```

See also:

fopen

`frexp` Separates a floating point number into a fraction and an integral power of 2.

Synopsis:

```
#include <math.h>
double frexp(double value, int *exp);
```

Arguments:

`double value`	The floating point number.
`int *exp`	A pointer to an integer where the exponent is stored.

Results:

Returns the normalized fractional part of value. The fraction is returned in the range [0.5 ... 1) or zero. The exponent is stored in the `int` pointed to by `exp`.

Errors:

A domain error occurs if `value` is NaN or infinity. In this case the input value is returned unchanged and `*exp` is set to 0.

Description:

`frexp` separates the floating point number value into a normalized fraction and an integral power of 2. The exponent is stored in the `int` pointed to by `exp`. The fraction is returned by the function.

If x is the value returned by `frexp` and `y` is the exponent stored in `*exp` then:

```
value = x * 2**y
```

If `value` is zero then both `x` and `y` will be zero.

Example:

```
#include <math.h>
#include <stdio.h>

int main( )
{
   double x;
   double mantissa;
   int exponent;

   x = 3.141;
   mantissa = frexp(x,&exponent);
   printf("x = %f, mantissa = %f, exponent = %d\n",
           x, mantissa, exponent);
}
/*
 *   Output:
 *
 *          x = 3.141000, mantissa = 0.785250,
 *              exponent = 2
 *
 */
```

See also:

`ldexp frexpf`

`frexpf` Separates a floating point number of type `float` into a fraction and an integral power of 2.

Synopsis:

```
#include <mathf.h>
float frexpf(float value, int *exp);
```

Arguments:

`float value`	The floating point number.
`int *exp`	A pointer to the `int` into which the exponent is put.

Results:

Returns the fractional part of value. The normalized fraction is returned in the range [0.5...1) or zero. The exponent is stored in the `int` pointed to by `exp`.

Errors:

A domain error occurs if `value` is NaN or infinity. In this case the input value is returned unchanged and `*exp` is set to 0.

Description:

`float` form of `frexp`.

See also:

`ldexpf frexp`

`from_host_link` Retrieve the channel coming from the host.

Synopsis:

```
#include <hostlink.h>
Channel* from_host_link( void )
```

Arguments:

None.

Results:

Returns a pointer to the channel coming from the host.

Errors:

None.

Description:

`from_host_link` retrieves the channel coming from the host.

Note: that the link over which communication with the host occurs need not necessarily be the same link as the one from which the transputer was booted.

This function is intended for use with dynamic code loading; care should be taken if it is used elsewhere.

`from_host_link` is not in the reduced library.

See also:

`get_bootlink_channels` `to_host_link`

from86

Transfers host memory to the transputer. MS-DOS only.

Synopsis:

```
#include <dos.h>
int from86(int len, pcpointer there, char *here);
```

Arguments:

`int len`	The number of bytes of host memory to be transferred.
`pcpointer there`	A pointer to the host memory block.
`char *here`	A pointer to the receiving block in transputer memory.

Results:

Returns the actual number of bytes transferred.

Errors:

Returns the number of bytes transferred until the error occurred and sets `errno` to the value `EDOS`. Any attempt to use `from86` on systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`from86` transfers `len` bytes of host memory starting at `there` to a corresponding block starting at `here` in transputer memory. The function returns the number of bytes actually transferred. The host memory block used will normally have been previously allocated by a call to `alloc86`.

`from86` is not included in the reduced library.

See also:

`to86 alloc86`

fscanf

Reads formatted input from a file stream.

Synopsis:

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Arguments:

`FILE *stream`	An input file stream.
`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of inputs which have been successfully converted. If an end of file character occurred before any conversions took place, `fscanf` returns `EOF`.

Errors:

If an end of file character occurred before any conversions took place, `fscanf` returns `EOF`. Other failures cause termination of the procedure.

Description:

`fscanf` matches the data read from the input stream `stream` to the specifications set out by the format string. The format string can include white space, ordinary characters, or conversion tokens:

1. White space causes the next series of white space characters read to be ignored.

2. Ordinary characters in the format string cause the characters read to be compared to the corresponding character in the format string. If the characters do not match, conversion is terminated.

3. A conversion token in the format string causes the data sequence read in to be checked to see if it is in the specified format. If it is, it is converted and placed in the appropriate argument following the format string. If the data is not in the correct format, conversion is terminated.

The conversion tokens consist of the following items:

1. Token signifier:

`%` (percent character)

2. Assignment suppressor (optional):

   * (asterisk). This causes the data sequence to be read in but not assigned to an argument. Tokens that use the assignment suppressor should not have a corresponding argument in the argument list.

3. Maximum width (optional):

   The width is a decimal integer constant defining the maximum number of characters to be read.

4. Type specifier (optional):

`h`	Specifies that a following 'd', 'i', 'n', 'o', 'u', or 'x' conversion applies to a `short int` or `unsigned short int`.
`l`	Specifies that a following 'd', 'i', 'n', 'o', 'u' or 'x' conversion applies to a `long int` or `unsigned long int`, and a following 'e', 'f' or 'g' conversion applies to a `double`.
`L`	Specifies that a following 'e', 'f' or 'g' conversion applies to a `long double`.

5. A single conversion character:

`d`	Expects an (optionally signed) decimal integer. Requires a pointer to an integer as the corresponding argument.
`i`	Expects an (optionally signed) integer constant. The integer constant may be a hexadecimal or octal value, provided the correct prefix is supplied. Requires a pointer to an integer as the corresponding argument.
`o`	Expects an (optionally signed) octal integer.
`u`	Expects an (optionally signed) decimal integer. Requires a pointer to an unsigned integer as the corresponding argument.
`x`	Expects an (optionally signed) hex integer (optionally preceded by an 0x or 0X). Requires a pointer to an integer as the corresponding argument.
`e, f, g`	Expects an (optionally signed) floating point character consisting of the following sequence of characters: i A plus or minus sign (optional). ii A sequence of decimal digits, which may contain a decimal point. iii An exponent (optional) consisting of an 'E' or 'e' followed by an optional sign and a string of decimal digits. Requires a pointer to a `double` as the corresponding argument.
`s`	Expects a string. Requires a pointer to an array large enough to hold (size of the string plus a terminating null char) characters as the corresponding argument.

`[` Denotes the start of a scan set.

Expects a non–empty string of characters. Acceptable characters are denoted by the scan set. The corresponding argument should be a pointer to an array large enough to accept the string plus a terminating null character.

The characters between the left bracket '[' and the right bracket ']' make up the scan list.

The scan set is equal to the scan list unless the first character in the scan list is a (^) in which case the scan set is made up of all those characters which do not occur in the scan list.

The right bracket (]) can be included in the scan list if it is the first character in the scan list, i.e. [ ] is in the format string, or if it is the second character in the scan list after the (^), i.e. [^] is in the format string. In these cases the scan list is terminated by the next occurrence of a left bracket (]).

The string is read up until the first character which is not in the scan set e.g.:

format string	meaning
`"%[abc]"`	match a string made up of a, b and c only.
`"%[^abc]"`	match a string made up of any characters except a, b and c.
`"%[]abc]"`	match a string made up of a, b, c and] only.
`"%[^]abc]"`	match a string made up of any characters except a, b, c and].

`p` Expects a hexadecimal string. Requires a pointer to a void pointer as the corresponding argument.

`n` The number of characters received so far will be put into the integer pointed to by the argument. This does not increment the assignment count returned or read from the stream.

`%` Matches the % character.

Any mismatch between the token format and the data received causes an early termination of **`fscanf`**.

`fscanf` is not included in the reduced library.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int i, numout, numin;
  FILE *stream;
  float fp;

  /* create a file of items to read back */

  stream = fopen("data.dat", "w");

  if (stream == NULL)
  {
    printf("Error opening data.dat for writing\n");
    abort();
  }

  numout = fprintf(stream, "%f %d", 3.141, 1024);

  fclose(stream);

  printf("Number of characters written = %d\n", numout);

  /* Now read the items back again */

  stream = fopen("data.dat", "r");

  if (stream == NULL)
  {
    printf("Error opening data.dat for reading\n");
    abort();
  }

  numin = 0;
  numin += fscanf(stream, "%f %d", &fp, &i);

  fclose(stream);

  printf("Number of fields read = %d\n", numin);
  printf("items read were: %f %d\n", fp, i);
}
```

See also:

`fprintf`

fseek

Sets the file position indicator to a specified offset.

Synopsis:

```
#include <stdio.h>
int fseek(FILE *stream, long int offset,
          int whence);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`long int offset`	The distance the file position indicator is moved.
`int whence`	The start position for the seek.

Results:

Returns non-zero on error, otherwise `fseek` returns zero.

Errors:

`fseek` returns non-zero on error.

Description:

`fseek` is used to move the file position indicator of a file to a specified offset within the file stream `stream`. The offset is measured from a position defined by `whence` and can take the following values:

`SEEK_SET` is the start of the file stream.
`SEEK_CUR` is the current position in the file stream.
`SEEK_END` is the end of the file stream.

If the file stream is a text stream the offset should either be zero or `whence` should be set to `SEEK_SET`, and `offset` should be a value returned by a `ftell`.

`fseek` clears the end of file indicator for `stream` and undoes the effects of `ungetc`. The file stream may be both read from and written to after `fseek` has been called, provided the stream has been opened in an appropriate mode.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  FILE *stream;
  int result;

  stream = fopen("data.dat", "wb+");

  if (stream == NULL)
  {
    printf("Error opening data.dat for update\n");
    abort();
  }
  /* write something to the file so we can fseek around it */

  fprintf(stream, "1232456789");

  /* reset to the beginning of the file */
  result = fseek(stream, 0L, SEEK_SET);

  if (result)
  {
    printf("fseek failed\n");
    abort();
  }
  printf("first char in file is %c\n", getc(stream));

  /* reset to the beginning of the file */
  result = fseek(stream, 0L, SEEK_SET);

  /* move to third char in file */
  result = fseek(stream, 2L, SEEK_CUR);

  if (result)
  {
    printf("fseek failed\n");
    abort();
  }
  printf("third char in file is %c\n", getc(stream));

  /* move to last char in file */
  result = fseek(stream, -1L, SEEK_END);

  if (result)
  {
    printf("fseek failed\n");
    abort();
  }
  printf("last char in file is %c\n", getc(stream));
  fclose(stream);
}
```

See also:

`fsetpos`, `ftell`, `ungetc`

`fsetpos` Sets the file position indicator to an `fpos_t` value obtained from `fgetpos`.

Synopsis:

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`const fpos_t *pos`	A pointer to an object containing the new value of the file position indicator.

Results:

Returns zero if the operation was successful, and non-zero on failure.

Errors:

If the operation was unsuccessful, `fsetpos` sets `errno` to `EFILPOS` and returns a non-zero value.

Description:

`fsetpos` sets the file position indicator of the file stream `stream` to the value in `pos`. `pos` shall contain a value previously returned by `fgetpos`.

A successful call to `fsetpos` clears the end of file indicator for the stream and will undo the effects of an `ungetc` operation on the same stream. The file stream may be both read from and written to after `fsetpos` has been called, provided it has been opened in an appropriate mode.

`fsetpos` is not included in the reduced library.

Example:

```
#include <stdio.h>

int main( )
{
   FILE *stream;
   fpos_t filepos;
   int ch;

   stream = fopen("data.dat","w+");
   if (stream == NULL)
     printf("Couldn't open data.dat for read\n");
   else
   {
      fprintf(stream, "123456789");
      rewind(stream);
      ch = getc(stream);
      printf("First char in file is '%c'\n",ch);

      /*
       * Remember: getc() advances file pointer,
       *           so it now points
       * to the second character in the file.
       */

      if (fgetpos(stream,&filepos) != 0)
        printf("Error with fgetpos\n");

      ch = getc(stream);
      printf("Second char in file is '%c'\n",ch);
      ch = getc(stream);
      printf("Third character in file is '%c'\n",ch);

      if (fsetpos(stream,&filepos) !=0)
        printf("Error with fsetpos\n");

      ch = getc(stream);
      printf("Reset file ptr and read 2nd char which is '%c'\n", ch);
      fclose(stream);
    }
}
```

See also:

fgetpos fseek ungetc

`ftell` Returns the position of the file position indicator for a file stream.

Synopsis:

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns the current value of the file position indicator for the file stream `stream`, or –1 on error.

Errors:

`ftell` returns –1 on error and sets `errno` to `EFILPOS`.

Description:

`ftell` returns the current value of the file position indicator for the file stream `stream`. For a binary stream the value is the number of characters from the beginning of the file. For a text stream the value is unspecified but can be used by `fseek` to reposition the file position indicator to its original position at the time of the call to `ftell`.

`ftell` is not included in the reduced library.

See also:

`fseek`

fwrite

Writes records from an array into a file.

Synopsis:

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

Arguments:

`void *ptr`	A pointer to a buffer that the records are read from.
`size_t size`	The size of an individual record.
`size_t nmemb`	The maximum number of records to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the number of records written. This may be less than `nmemb` if a write error occurs.

Errors:

`fwrite` returns zero if `size` or `nmemb` is zero. If an error occurs the number of records read up to the error is returned.

Description:

`fwrite` writes `nmemb` records of length `size` from the array pointed to by `ptr` to the file stream `stream`. If an error occurs, the value of the file position indicator is indeterminate.

`fwrite` is not included in the reduced library.

See `fread` for an example.

See also:

`fread`

get_bootlink_channels

Obtains the channels associated with the boot link.

Synopsis:

```
#include <bootlink.h>
int get_bootlink_channels( Channel** in_ptr,
                           Channel** out_ptr )
```

Arguments:

`Channel** in_ptr`	The address of a variable which will be assigned a pointer to the input channel associated with the boot link.
`Channel** out_ptr`	The address of a variable which will be assigned a pointer to the output channel associated with the boot link.

Results:

Returns zero if the operation was successful and non–zero on failure.

Errors:

If the operation fails, `*in_ptr` and `*out_ptr` are undefined.

Description:

`get_bootlink_channels` retrieves the channels that are associated with the link that the transputer was booted from.

Note: that the link over which communication with the host occurs need not necessarily be the same link as the one from which the transputer was booted.

If used in a boot from ROM case, the obtained addresses will be undefined.

See also:

`from_host_link` `to_host_link`

`get_code_details_from_channel` Retrieves details from a dynamically loadable file that is transmitted over a channel.

Synopsis:

```
#include &lt;fnload.h&gt;
int get_code_details_from_channel(Channel* in_channel,
                                  fn_info* fn_details)
```

Arguments:

`Channel* in_channel`	A pointer to the channel over which the dynamically loadable (`.rsc`) file is received.
`fn_info* fn_details`	The address of a variable which will be assigned the details from the transmitted file.

Results:

Returns zero if the operation was successful and non–zero on failure.

Errors:

If the operation was unsuccessful, `*fn_details` is undefined.

Description:

`get_code_details_from_channel` retrieves details from a dynamically loadable file that is transmitted over a channel. It is assumed, on entry to this function, that the next transmission over the specified channel will be the header of the dynamically loadable (`.rsc`) file. The header data is received as a series of individual byte transmissions.

See also:

`load_code_from_channel`

`get_code_details_from_file` Retrieves details from a dynamically loadable file which is stored on disc.

Synopsis:

```
#include <fnload.h>
int get_code_details_from_file(const char* filename,
                               fn_info* fn_details,
                               size_t* file_hdr_size)
```

Arguments:

`const char* filename`	A string which is the name of the dynamically loadable file.
`fn_info* fn_details`	The address of a variable which will be assigned the details from the `.rsc` file.
`size_t* file_hdr_size`	The address of a variable which will be assigned the number of bytes at the start of the file before the code block.

Results:

Returns zero if the operation was successful and non–zero on failure.

Errors:

If the operation was unsuccessful, `*fn_details` and `*file_hdr_size` are undefined. The operation may fail for various reasons. For example, the given file-name may refer to a file that does not exist or cannot be read.

Description:

`get_code_details_from_file` retrieves details from a dynamically loadable code file. Such files have the default extension `.rsc`.

If `get_code_details_from_file` is used in a program linked with the reduced library it always returns non–zero.

See also:

`load_code_from_file`

`get_code_details_from_memory` Retrieves details from the image of a dynamically loadable file which is stored in internal memory .

Synopsis:

```
#include <fnload.h>
int get_code_details_from_memory(const void* addr_of_file_image,
                                 fn_info* fn_details,
                                 size_t* file_hdr_size,
                                 loaded_fn_ptr* function_pointer)
```

Arguments:

`const void* addr_of_file_image`	The start address of the image of the dynamically loadable (`.rsc`) file in internal memory.
`fn_info* fn_details`	The address of a variable which will be assigned the details from the file image.
`size_t* file_hdr_size`	The address of a variable which will be assigned the number of bytes at the start of the file image before the code block.
`loaded_fn_ptr* function_pointer`	The address of a variable which will be assigned a pointer to the function entry point in the file image.

Results:

Returns zero if the operation was successful and non–zero on failure.

Errors:

If the operation was unsuccessful, `*fn_details`, `*file_hdr_size` and `*function_pointer` are undefined.

Description:

`get_code_details_from_memory` retrieves details from the image of a dynamically loadable (`.rsc`) file which is held in internal memory.

The file contents are assumed to be laid out in increasing memory from the value of `addr_of_file_image`.

If the file image is in ROM and it is known that it does not write to itself then `*function_pointer` can be cast, if necessary, and used immediately to call the code in the file image. If the file image is in ROM and does write to itself then the code in it must first be loaded into RAM before that code can be called.

See also:

`load_code_from_memory`

`get_details_of_free_memory`

Reports the details of memory considered by the configurer to be unused.

Synopsis:

```
#include <misc.h>
int get_details_of_free_memory( void** base_of_free_memory,
                                size_t* size_of_free_memory )
```

Arguments:

`void** base_of_free_memory`	The address of a variable which will be assigned the word aligned address of the start of unused memory.
`size_t* size_of_free_memory`	The address of a variable which will be assigned the amount of unused memory, in words.

Results:

Returns zero if the operation was successful and non–zero on failure.

Errors:

If the operation fails, `*base_of_free_memory` and `*size_of_free_memory` are undefined.

Description:

When configuring one uses a configuration description. The configuration description gives, amongst other things, the amount of memory attached to each processor. The actual memory used on the processor is usually not the full amount as given in the configuration description, and so there is unused memory at the top of memory. It is the base and amount of this unused memory that is reported by this function.

There is no free memory in the non–configured case.

get_details_of_free_stack_space

Reports the limits of free space on current stack.

Synopsis:

```
#include <misc.h>
void get_details_of_free_stack_space(void** stack_limit_ptr,
                                     size_t* remaining_stack_space_ptr)
```

Arguments:

`void** stack_limit_ptr`	The address of a variable which will be assigned the limit of the current stack.
`size_t* remaining_stack_space_ptr`	The address of a variable which will be assigned the approximate number of bytes still unused of the present stack.

Results:

Returns no result.

Errors:

None.

Description:

`get_details_of_free_stack_space` reports the limits of unused space on the current stack.

The value given by `*stack_limit_ptr` is the address of the last word on the stack, not to the first word after the top of the stack.

Just how approximate the value given by `*remaining_stack_space_ptr` is, depends on when one uses the value; it is most accurate immediately after the call to this function when it is slightly smaller than the exact value. This function does not take into account the150 words that `max_stack_usage()` includes in its return value.

Note: `get_details_of_free_stack_space` should not be used by any C code which is to be imported by occam, using `callc.lib`.

See also:

`max_stack_usage`

`get_param` Reads parameters from the configuration level. Applies only to configured processes.

Synopsis:

```
#include <misc.h>
void *get_param(int n);
```

Arguments:

`int n`	The index of the required parameter in the interface list.

Results:

Returns a pointer to the specified configuration level parameter. If the parameter is a scalar then a pointer to the parameter is returned. If the parameter is a channel or array then the channel or array pointer itself is returned.

Errors:

The function returns `NULL` on error. Possible causes of errors are:

> Using the function when it is not valid, i.e. from a program not configured using `icconf`.
>
> Using a value of `n` less than 1.
>
> Using a value of `n` which is greater than the number of available parameters.

Description:

`get_param` reads parameters from the list specified in the `interface` attribute for a configured process. It can only be used from a program which has been configured using `icconf`. It must *not* be used in a program which uses the special case entry points `MAIN.ENTRY`, `PROC.ENTRY` or `PROC.ENTRY.RC` described in chapter 10 of the accompanying *ANSI C Toolset User Guide*.

`get_param` is used to access the parameters given to a process in the interface list at configuration level. It enables access to the `n`th parameter in the parameter list (n is a non-zero positive integer). If the parameter is a scalar then a pointer to the parameter is returned. If the parameter is a channel or array then the channel or array pointer itself is returned.

`get_param` is side effect free.

The following example shows how a C program can use `get_param` to obtain the value of a variable defined in the interface parameter list of a process defined at configuration level.

C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <misc.h>

int main ()
{
    int *value;

    value = (int *)get_param(3);
    printf("value = %d\n", *value);
    exit_terminate(EXIT_SUCCESS);
}
```

Configuration description:

```
/* Hardware description */
T414(memory = 2M) B403;

connect B403.link[0], host;

/* Software description */
process(stacksize = 20k, heapsize = 20k,
        interface(input in,
                  output out,
                  int value)) test;

test(value = 427);

input from_host;
output to_host;

connect test.in, from_host;
connect test.out, to_host;

/* Network mapping */
use "test1.lku" for test;
place test on B403;

place to_host   on host;
place from_host on host;

place test.in  on B403.link[0];
place test.out on B403.link[0];
```

The C program obtains the value 427 by reading the third interface parameter to the configured process `test` and then displays it.

getc

Gets a character from a file.

Synopsis:

```
#include <stdio.h>
int getc(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

Returns the next character from the file stream or `EOF` on error.

Errors:

If the next character is the end of file character, or a read error occurs, `getc` returns `EOF`.

Description:

`getc` returns the next character from the opened file identified by the file stream pointer.

`getc` is not included in the reduced library.

See also:

`fgetc` `getchar` `putc`

getchar

gets a character from `stdin`

Synopsis:

```
#include <stdio.h>
int getchar(void);
```

Arguments:

None.

Results:

Returns the next character from `stdin` or `EOF` on error.

Errors:

If the next character is the end of file character, or a read error occurs, `getchar` returns `EOF`.

Description:

`getchar` is equivalent to `getc` with the argument `stdin`.

`getchar` is not included in the reduced library.

See also:

`getc fgetc putc putchar`

getenv Returns a pointer to the string associated with a host environment variable.

Synopsis:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Arguments:

`const char *name`	A pointer to the host environment variable name to be matched.

Results:

Returns a pointer to the string associated with the given environment variable. A `NULL` pointer is returned if the environment variable is not defined on the host, or the program is linked with the reduced library.

Errors:

Returns `NULL` if the environment variable is not defined on the host.

Description:

`getenv` returns a pointer to the string associated with the host environment variable *name*. The string must not be modified by the program but can be overwritten by a subsequent call to `getenv`.

If `getenv` is used in a program linked with the reduced library a `NULL` pointer is always returned.

Note: Care should be taken when calling `getenv` in a concurrent environment. Calls to the function by independently executing, unsynchronized processes may corrupt the string pointed to by the returned `char` pointer.

Example:

```
#include <stdlib.h>
#include <stdio.h>

int main( void )
{
   char *envvar;
   envvar = getenv("IBOARDSIZE");
   if (envvar == NULL)
      printf("IBOARDSIZE variable not set\n");
   else
      printf("IBOARDSIZE is : %s\n",envvar);
}
```

getkey

Reads a character from the keyboard.

Synopsis:

```
#include <iocntrl.h>
int getkey(void);
```

Arguments:

None.

Results:

Returns the ASCII value of the character, or –1 on error.

Errors:

Returns –1 if an error occurs.

Description:

`getkey` returns the ASCII value of the next character typed at the keyboard. The routine waits indefinitely for the next keystroke and only returns when a key is available. The effect on any buffered data in the standard input stream is host-defined. The character read is not echoed at the terminal.

`getkey` is not included in the reduced library.

See also:

`pollkey`

gets

Reads a line from from `stdin`

Synopsis:

```
#include <stdio.h>
char *gets(char *s);
```

Arguments:

`char *s`	A pointer to an array where the read characters are stored.

Results:

Returns `s` if successful or a `NULL` pointer on error.

Errors:

`gets` returns a `NULL` pointer if a read error occurs and the contents of the array are undefined. If end of file is encountered before a character is read gets returns `NULL` and the contents of the array remain unchanged.

Description:

`gets` reads characters from `stdin` into the array pointed to by `s`. The read terminates at end of file or when a new–line character is read. The new–line character is discarded and a null character is written after the last character written into the array.

`gets` is not included in the reduced library.

See also:

`fgets puts fputs`

gmtime Converts a calendar time to a broken–down time, expressed as a UTC time.

Synopsis:

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Arguments:

`const time_t *timer`	Calendar time to be converted.

Results:

Returns a pointer to a broken-down time expressed as UTC time, or `NULL` if UTC time is unavailable.

Errors:

Returns `NULL` if UTC time is not available.

Description:

`gmtime` converts a calendar time into a broken–down time (see section 1.3.16), expressed as Universal Time (UTC).

Note: UTC is unavailable in this implementation and `gmtime` *always* returns `NULL`.

See also:

asctime ctime difftime localtime strftime clock mktime time

halt_processor

Halts the processor

Synopsis:

```
#include <misc.h>
void halt_processor(void);
```

Arguments:

None.

Results:

This macro does not return.

Errors:

None.

Description:

`halt_processor` is implemented as a macro. `halt_processor` halts the processor on which it is executed. This is achieved by setting the **HaltOnError** flag and then explicitly setting the **ErrorFlag**.

See also:

`abort` `debug_stop`

host_info

Gets data about the host system.

Synopsis:

```
#include <host.h>
void host_info(int *host, int *os, int *board);
```

Arguments:

`int *host`	A pointer to an int where the host type code will be stored.
`int *os`	A pointer to an int where the operating system type code will be stored.
`int *board`	A pointer to an int where the board type code will be stored.

Results:

Returns no result.

Errors:

If any host attribute is unavailable it is given the value 0.

Description:

`host_info` returns information about the host environment. It stores codes for the host type, host operating system and transputer board in the locations pointed to by `host`, `os`, and `board` respectively.

`host_info` is not included in the reduced library.

The values that `host` can take are defined in the header `host.h` and are as follows:

`_IMS_HOST_PC`

`_IMS_HOST_NEC`

`_IMS_HOST_VAX`

`_IMS_HOST_SUN3`

`_IMS_HOST_IBM370`

`_IMS_HOST_SUN4`

`_IMS_HOST_SUN386i`

`_IMS_HOST_APOLLO`

The values that `os` can take are as follows:

_IMS_OS_DOS

_IMS_OS_HELIOS

_IMS_OS_VMS

_IMS_OS_SUNOS

_IMS_OS_CMS

The values that board can take are as follows:

_IMS_BOARD_B004

_IMS_BOARD_B008

_IMS_BOARD_B010

_IMS_BOARD_B011

_IMS_BOARD_B014

_IMS_BOARD_DRX11

_IMS_BOARD_QT0

_IMS_BOARD_B015

_IMS_BOARD_CAT

_IMS_BOARD_B016

_IMS_BOARD_UDP_LINK

int86

Performs a MS-DOS software interrupt. MS-DOS only.

Synopsis:

```
#include <dos.h>
int int86(int intno, union REGS *inregs,
          union REGS *outregs);
```

Arguments:

`int intno`	The host software interrupt ID.
`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `int86` on operating systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`int86` calls the host software interrupt identified by `intno` with the registers set to `inregs`. Register values after the interrupt are returned in `outregs` and the contents of the **ax** register are returned as the function result.

Segment registers **cs**, **ds**, **ex**, and **ss** are not set.

`int86` is not included in the reduced library.

See also:

`int86x intdos`

int86x Software interrupt with segment register setting. MS-DOS only.

Synopsis:

```
#include <dos.h>
int int86x(int intno, union REGS *inregs,
           union REGS *outregs,
           struct SREGS *segregs);
```

Arguments:

`int intno`	The MS-DOS software interrupt ID.
`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.
`struct SREGS *segregs`	Values to be placed in segment registers.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `int86x` on operating systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`int86x` calls the host software interrupt identified by `intno` with the registers set to `inregs` and the segment registers set to `segregs`. Register values after the interrupt are returned in `outregs` and the contents of the **ax** register are returned as the function result.

`int86x` is useful for MS-DOS calls which take pointers to objects, normally specified by combining a 16-bit register with a segment register. If only some of the segment registers are modified, `segread` should be used to read values from the others. Failure to do so can produce unpredictable results.

See also:

`int86 intdosx`

intdos

Performs an MS-DOS interrupt. MS-DOS only.

Synopsis:

```
#include <dos.h>
int intdos(union REGS *inregs,
           union REGS *outregs);
```

Arguments:

`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `intdos` on operating systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

As `int86` but calls the specific host software interrupt identified by hexadecimal 21 (MS-DOS function call).

See also:

`int86` `intdosx`

intdosx MS-DOS interrupt with segment register setting. MS-DOS only.

Synopsis:

```
#include <dos.h>
int intdosx(union REGS *inregs,
            union REGS *outregs,
            struct SREGS *segregs);
```

Arguments:

`union REGS *inregs`	Values to be placed in processor registers.
`union REGS *outregs`	Register values after the interrupt.
`struct SREGS *segregs`	Values to be placed in segment registers.

Results:

Returns the value of the **ax** register after the interrupt.

Errors:

Returns zero (0) on error and sets `errno` to the value `EDOS`. Any attempt to use `intdosx` on operating systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the server error message:

`[Encountered unknown primary tag (50)]`

Description:

As `intdos` but also sets segment registers.

See also:

`intdos` `int86x`

`isalnum`

Tests whether a character is alphanumeric.

Synopsis:

```
#include <ctype.h>
int isalnum(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is alphanumeric and zero (false) if it is not.

Errors:

None.

Description:

`isalnum` tests whether the character `c` is in one of the following sets of alphabetic and numeric characters:

'a' to 'z' 'A' to 'Z' '0' to '9'

`isalnum` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`isalpha isdigit`

isalpha

Tests whether a character is alphabetic.

Synopsis:

```
#include <ctype.h>
int isalpha(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is alphabetic and zero (false) if it is not.

Errors:

None.

Description:

`isalpha` tests whether `c` is in one of the following sets of alphabetic characters:

'a' to 'z' 'A' to 'Z'

`isalpha` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`isalnum isdigit`

isatty

Tests for a terminal stream.

Synopsis:

```
#include <iocntrl.h>
int isatty(int fd);
```

Arguments:

`int fd`	A file descriptor.

Results:

Returns 1 (true) if the file descriptor refers to a terminal stream, otherwise returns 0 (false).

Errors:

None.

Description:

`isatty` determines whether a given file descriptor refers to one of the default terminal files `stdin`, `stdout`, and `stderr`.

`isatty` is not included in the reduced library.

`iscntrl` Tests whether a character is a control character.

Synopsis:

```
#include <ctype.h>
int iscntrl(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is a control character and zero (false) if it is not.

Errors:

None.

Description:

`iscntrl` determines whether `c` is a control character (ASCII codes 0–31 and 127).

`iscntrl` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

`isdigit`

Tests whether a character is a decimal digit.

Synopsis:

```
#include <ctype.h>
int isdigit(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is a digit and zero (false) if it is not.

Errors:

None.

Description:

`isdigit` tests whether `c` is one of the following decimal digit characters:

'0' '1' '2' '3' '4' '5' '6' '7' '8' '9'

`isdigit` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`isalnum isalpha`

isgraph

Tests whether a character is printable (non-space).

Synopsis:

```
#include <ctype.h>
int isgraph(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is a printable character (other than space) and zero (false) if it is not.

Errors:

None.

Description:

`isgraph` tests whether `c` belongs to the set of printable characters excluding the space character (' '). The space character is considered in this test to be non-printable.

`isgraph` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`iscntrl isprint isspace`

islower

Tests whether a character is a lower-case letter.

Synopsis:

```
#include <ctype.h>
int islower(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is a lower-case letter and zero (false) if it is not.

Errors:

None.

Description:

`islower` tests whether `c` is a character in the set of lower case characters:

'a' to 'z'

`islower` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`isupper`

`isprint` Tests whether a character is printable (includes space).

Synopsis:

```
#include <ctype.h>
int isprint(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is printable and zero (false) if it is not.

Errors:

None.

Description:

`isprint` tests whether `c` is a printable character (ASCII character codes 32–126).

Note: Unlike `isgraph`, `isprint` considers the space character (' ') to be printable.

`isprint` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`isgraph`

`ispunct`

Tests to see if a character is a punctuation character.

Synopsis:

```
#include <ctype.h>
int ispunct(int c);
```

Arguments:

`int c`	The character to be examined.

Results:

Returns non-zero (true) if the character is a punctuation character and zero (false) if it is not.

Errors:

None.

Description:

`ispunct` tests whether `c` is a punctuation character. For the purposes of this test a punctuation is any printable character other than an alphanumeric or space (' ') character.

`ispunct` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`iscntrl` `isgraph` `isprint`

`isspace` Tests to see if a character is one which affects spacing.

Synopsis:

```
#include <ctype.h>
int isspace(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is a space character and zero (false) if it is not.

Errors:

None.

Description:

`isspace` tests whether `c` belongs to the set of characters which produce white space. Characters which generate white space are as follows:

FORM FEED	('\f')
LINE FEED/NEWLINE	('\n')
RETURN	('\r')
SPACE	(' ')
TAB	('\t')
Vertical TAB	('\v')

`isspace` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

`isupper`

Tests whether a character is an upper–case letter.

Synopsis:

```
#include <ctype.h>
int isupper(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non–zero (true) if the character is an upper–case letter and zero (false) if it is not.

Errors:

None.

Description:

`isupper` tests whether `c` is a character in the set of upper–case letters:

'A' to 'Z'

`isupper` is implemented as both a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

See also:

`islower`

isxdigit

Tests to see if a character is a hexadecimal digit.

Synopsis:

```
#include <ctype.h>
int isxdigit(int c);
```

Arguments:

`int c`	The character to be tested.

Results:

Returns non-zero (true) if the character is a hexadecimal digit and zero (false) if it is not.

Errors:

None.

Description:

`isxdigit` tests whether `c` belongs to the set of hexadecimal digits. These are as follows:

'a' 'b' 'c' 'd' 'e' 'f' 'A' 'B' 'C' 'D' 'E' 'F' '0' '1' '2' '3' '4' '5' '6' '7' '8' '9'

`isxdigit` is implemented both as a macro and a function.

Note: the argument must be representable as an `unsigned char` or be equal to `EOF`, otherwise the behavior of the function is undefined.

`labs`

Calculates the absolute value of a long integer.

Synopsis:

```
#include <stdlib.h>
long int labs(long int j);
```

Arguments:

`long int j`	A long integer.

Results:

Returns the absolute value of `j` as a `long int`.

Errors:

If the result cannot be represented the behavior of `labs` is undefined.

Description:

`labs` calculates the absolute value of the `long int j`.

`labs` is side effect free.

See also:

`abs`

`ldexp` Multiplies a floating point number by an integer power of two.

Synopsis:

```
#include <math.h>
double ldexp(double x, int exp);
```

Arguments:

`double x`	The floating point number.
`int exp`	The exponent.

Results:

Returns the value of:

$$x \times (2^{exp})$$

If a range error occurs returns `HUGE_VAL` (with the same sign as the correct value of the function).

Errors:

A range error will occur if the result of `ldexp` would cause overflow or underflow. In this case `ldexp` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`ldexp` calculates the value of :

$$x \times (2^{exp})$$

See also:

`frexp`

ldexpf

Multiplies a `float` number by an integral power of two.

Synopsis:

```
#include <mathf.h>
float ldexpf(float x, int exp);
```

Arguments:

`float x`	The floating point number.
`int exp`	The exponent.

Results:

Returns the value of:

$$x \times (2^{exp})$$

If a range error occurs returns `HUGE_VAL_F` (with the same sign as the correct value of the function).

Errors:

A range error will occur if the result of `ldexpf` would cause overflow or underflow. In this case `ldexpf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `ldexp`.

See also:

`ldexp` `frexp`

`ldiv` Calculates the quotient and remainder of a long division.

Synopsis:

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

Arguments:

`long int numer`	The numerator.
`long int denom`	The denominator.

Results:

Returns a structure of type `ldiv_t` which consists of the quotient and remainder. The structure contains:

`long int quot`	The quotient.
`long int rem`	The remainder.

Errors:

If the result cannot be represented the behavior of `ldiv` is undefined.

Description:

`ldiv` calculates the quotient and remainder formed by dividing the numerator `numer` by the denominator `denom`. All values are of type `long int`.

`ldiv` is side effect free.

See also:

`div`

`load_code_from_channel` Receives the code block of a dynamically loadable file from a channel and copies it into internal memory.

Synopsis:

```
#include <fnload.h>
loaded_fn_ptr load_code_from_channel(Channel* in_channel
                                     const fn_info* fn_details,
                                     void* dest)
```

Arguments:

`Channel* in_channel`	A pointer to the channel over which the code block is received.
`const fn_info* fn_details`	A pointer to the structure containing details of the code in the code block.
`void* dest`	A pointer to the point in internal memory where the code is to be placed.

Results:

Returns a function pointer to the code that has been loaded.

Errors:

None.

Description:

`load_code_from_channel` receives the code block of a dynamically loadable file, transmitted over a channel, and copies it into a designated area of internal memory. It is assumed that there is enough memory available from `dest`, at increasing addresses, for the code to be placed into it. It is also assumed, on entry to the function, that the next transmission over the channel will be the code block of the dynamically loadable (`.rsc`) file. The code block is received as a series of individual byte transmissions.

See also:

`get_code_details_from_channel`

`load_code_from_file` Transfers code from a dynamically loadable file to internal memory.

Synopsis:

```
#include <fnload.h>
loaded_fn_ptr load_code_from_file(const char* filename,
                                  const fn_info* fn_details,
                                  size_t file_hdr_size,
                                  void* dest)
```

Arguments:

`const char* filename`	A string which is the name of the dynamically loadable (`.rsc`) file.
`const fn_info* fn_details`	A pointer to the structure containing details of the code in the code block.
`size_t file_hdr_size`	The number of bytes at the start of the file before the code block.
`void* dest`	A pointer to the point in internal memory where the code is to be placed.

Results:

Returns a function pointer to the code that has been loaded, if the operation was successful and `NULL` on failure.

Errors:

If the operation is unsuccessful `NULL` is returned.

Description:

`load_code_from_file` transfers the code part of a dynamically loadable (`.rsc`) file to a designated area of internal memory. It is assumed that there is enough memory available from `dest`, at increasing addresses, for the code to be placed into it.

If `load_code_from_file` is used in a program linked with the reduced library it always returns `NULL`.

See also:

`get_code_details_from_file`

`load_code_from_memory` Transfers code from a dynamically loadable file from one area of internal memory to another.

Synopsis:

```
#include <fnload.h>
loaded_fn_ptr load_code_from_memory(const void* src,
                                    const fn_info* fn_details,
                                    size_t file_hdr_size,
                                    void* dest)
```

Arguments:

`const void* src`	The start address of the image of the dynamically loadable file, in internal memory.
`const fn_info* fn_details`	A pointer to the structure containing details of the code in the code block.
`size_t file_hdr_size`	The number of bytes at the start of the file before the code block.
`void* dest`	A pointer to the point in internal memory where the code is to be placed.

Results:

Returns a function pointer to the code that has been loaded.

Errors:

None.

Description:

`load_code_from_memory` transfers the code block of a dynamically loadable (`.rsc`) file image stored in one part of internal memory, to another part of internal memory. It is assumed that the file image is stored in increasing memory locations from `src` and that there is enough memory available from `dest`, at increasing addresses, for the code to be placed into it.

See also:

`get_code_details_from_memory`

`localeconv` Gets numeric formatting data for the current locale.

Synopsis:

```
#include <locale.h>
struct lconv *localeconv(void);
```

Arguments:

None.

Results:

Returns a pointer to a structure of type `lconv` which defines components of the current locale.

Errors:

None.

Description:

The components of a `lconv` structure (defined in `locale.h`) are set according to the current locale and a pointer to this structure is returned.

`localeconv` always returns a pointer to the same `lconv` structure. It should not be overwritten by the program but may be altered by subsequent calls to `setlocale` or `localeconv`.

INMOS ANSI C supports only the standard "C" locale.

`localeconv` is side effect free.

See also:

`setlocale`

`localtime`

Converts a calendar time into a broken–down time, expressed as local time.

Synopsis:

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Arguments:

`const time_t *timer`	A pointer to the calendar time.

Results:

Returns a pointer to a broken–down structure, containing the value of the time expressed as a local time.

Errors:

None.

Description:

`localtime` is used to convert a calendar time to a broken–down time expressed as local time.

Note: Care should be taken when calling `localtime` in a concurrent environment. `localtime` always returns a pointer to the same broken–down time structure and so calls to the function by independently executing, unsynchronized processes may corrupt the returned time value.

Example:

```
/* prints the current date and time as a local time */

#include <time.h>
#include <stdio.h>

int main()
{
  time_t current;
  struct tm *bdt;

  /* get the current time as a calendar time */
  time(&current);

  /* convert this to a broken down time expressed as local time */
  bdt = localtime(&current);

  /* Now convert the broken down time to a string and print it out */
  printf("Date and time = %s\n", asctime(bdt));
}
```

See also:

asctime ctime strftime clock difftime mktime time

log

Calculates the natural logarithm of the `double` argument.

Synopsis:

```
#include <math.h>
double log(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the natural log of `x`. If a range error occurs, it returns `HUGE_VAL` (with the same sign as the correct value of the function). If a domain error occurs, it returns zero.

Errors:

A domain error occurs if `x` is negative. In this case `errno` is set to `EDOM`.

A range error occurs if `x` is zero. In this case `log` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`log` calculates the natural (base e) logarithm of a number.

See also:

`log10` `logf`

logf

Calculates the natural logarithm of a `float` number.

Synopsis:

```
#include <mathf.h>
float logf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the natural log of `x`. If a range error occurs, it returns `HUGE_VAL_F` (with the same sign as the correct value of the function). If a domain error occurs, it returns zero.

Errors:

A domain error occurs if `x` is negative. In this case `errno` is set to `EDOM`.

A range error occurs if `x` is zero. In this case `logf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `log`.

See also:

`log` `log10f`

`log10`

Calculates the base-10 logarithm of the `double` argument.

Synopsis:

```
#include <math.h>
double log10(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the base ten log of x. If a range error occurs returns `HUGE_VAL` (with the same sign as the correct value of the function). If a domain error occurs returns zero.

Errors:

A domain error occurs if `x` is negative. In this case `errno` is set to `EDOM`. A range error occurs if `x` is zero. In this case `log10` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`log10` calculates the base 10 logarithm of a number.

See also:

`log` `log10f`

log10f

Calculates the base-10 logarithm of a float number.

Synopsis:

```
#include <mathf.h>
float log10f(float x);
```

Arguments:

float x	A number.

Results:

Returns the base ten log of x. If a range error occurs returns HUGE_VAL_F (with the same sign as the correct value of the function). If a domain error occurs returns zero.

Errors:

A domain error occurs if **x** is negative. In this case errno is set to EDOM. A range error occurs if **x** is zero. In this case log10f returns the value HUGE_VAL_F (with the same sign as the correct value of the function) and errno is set to ERANGE.

Description:

float form of log10.

See also:

log10 logf

longjmp

Performs a non-local jump to the given environment.

Synopsis:

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Arguments:

`jmp_buf env`	An array holding the environment to be restored.
`int val`	The value to be returned by `longjmp`.

Results:

`longjmp` itself does not return; the effect is as if the corresponding call to `setjmp` which stored the environment in `env` had returned the value of `val`. If `val` is zero, `setjmp` returns 1 (this is because `setjmp` is only allowed to return zero the first time it is called).

Errors:

None.

Description:

`longjmp` performs a non-local jump to the environment saved in `env`, by a previous call to `setjmp`. It returns in such a way that, to the program, it appears that the corresponding `setjmp` function has returned the value `val`.

Example:

```
#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>

jmp_buf env1;

int sub_function()
{
  /* .....
     ..... */

    longjmp(env1, 3);
}

int main()
{
 int a;

 switch(a=setjmp(env1))
 {
  case 0:  printf("1st time in top level\n");
           break;
  default: printf("longjmp to top level - code %d\n", a);
           exit( EXIT_SUCCESS );
 }
 sub_function();
}
```

See also:

`setjmp`

lseek

Repositions the current file position. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int lseek(int fd, long int offset, int origin);
```

Arguments:

`int fd`	A file descriptor.
`long int offset`	The offset by which the file position will move.
`int origin`	The start position for the seek.

Results:

Returns the new file position, or –1 on error.

Errors:

If an error occurs `lseek` sets `errno` to the value `EIO`.

Description:

`lseek` moves the current position within the file with file descriptor `fd`. The offset, given by `offset`, is measured from a position specified by `origin`:

`L_SET`	The start of the file.
`L_INCR`	The current position in the file.
`L_XTND`	The end of the file.

`lseek` is not included in the reduced library.

malloc

Allocates an area of memory.

Synopsis:

```
#include <stdlib.h>
void *malloc(size_t size);
```

Arguments:

`size_t size`	The size of the space to be allocated in bytes.

Results:

Returns a pointer to the allocated space if the allocation was successful. Otherwise a `NULL` pointer is returned. If size is zero `malloc` returns a `NULL` pointer.

Errors:

If there is not enough free space a `NULL` pointer is returned.

Description:

`malloc` allocates an area of memory of `size` bytes and returns a pointer to it. The contents of the allocated space are undefined.

Example:

```
/* Allocate 500 bytes pointed to by array1 */

char *array1;

array1 = (char *)malloc(500);
```

See also:

`calloc` `free` `realloc`

max_stack_usage

Report runtime stack usage.

Synopsis:

```
#include <misc.h>
long max_stack_usage(void);
```

Arguments:

None.

Results:

Returns the number of bytes of stack space used by the program or zero if stack checking is disabled.

Errors:

If stack checking is not enabled in the compiler the function returns zero.

Description:

`max_stack_usage` returns an approximation of the amount of stack used by the C main program up to the point at which `max_stack_usage` was called. A leeway of 150 words is included in the returned value to account for library usage, in which there is no stack checking.

Stack usage is measured on the main stack only, i.e. the stack in which the C main program is executing at program startup. The value does not include any stack used by a parallel process. `max_stack_usage` cannot be used from within a parallel process to obtain the stack usage of that process alone, it will always return the stack usage of the main stack.

Note: This function can only be used when stack checking is enabled. If stack checking is disabled the function returns 0 (zero).

`max_stack_usage` is side effect free.

See also:

`get_details_of_free_stack_space`

`mblen` Determines the number of bytes in a multibyte character.

Synopsis:

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

Arguments:

`const char *s`	Pointer to the multibyte character.
`size_t n`	The maximum number of bytes to be read.

Results:

If `s` is not a `NULL` pointer `mblen` returns the number of bytes that are contained in the multibyte character pointed to by `s`, as long as the next `n` or fewer bytes form a valid multibyte character.

If `s` points to a null character `mblen` returns zero, or –1 if `s` does not point to a valid multibyte character.

`mblen` is side effect free.

Errors:

If the specified sequence does not correspond to a valid multibyte character `mblen` returns –1.

Description:

`mblen` evaluates the number of bytes in a multibyte character. The number of bytes read is limited by `n`. In the current implementation the maximum length of a character is 1 byte.

mbstowcs

Converts multibyte sequence to `wchar_t` sequence.

Synopsis:

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

Arguments:

`wchar_t *pwc`	Pointer to the start of the array that receives the converted codes.
`const char *s`	Pointer to start of the array of multibyte characters to be converted.
`size_t n`	The maximum number of codes stored in `pwcs`.

Results:

`mbstowcs` returns the number of array elements modified, not including any terminating zero codes or returns `(size_t)`−1 if an invalid multibyte character is encountered.

Errors:

If an invalid multibyte character is encountered `mbstowcs` returns `(size_t)`−1.

Description:

`mbstowcs` converts a sequence of multibyte characters into a sequence of codes. It acts like the `mbtowc` function but takes as input an array of characters and returns an array of codes.

Not more than `n` codes are written into `pwcs`. If the initial and receiving objects overlap, the behavior is undefined.

No multibyte characters that follow a null character are examined or converted.

See also:

`mbtowc` `wcstombs`

mbtowc

Converts multibyte character to type `wchar_t`.

Synopsis:

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

Arguments:

`wchar_t *pwc`	Pointer to the storage location for the converted character.
`const char *s`	Pointer to the multibyte character to be converted.
`size_t n`	The maximum number of bytes to be read.

Results:

If `s` is not a `NULL` pointer, `mbtowc` either returns zero (if `s` points to a null character) or returns the number of bytes that are contained in the converted multibyte character, as long as the next `n` or fewer bytes form a valid multibyte character.

If `s` is a `NULL` pointer, `mbtowc` returns zero. `mbtowc` returns –1 on error.

The value returned cannot be greater than `n` or the value of `MB_CUR_MAX`.

Errors:

`mbtowc` returns –1 if the next `n` or fewer bytes do not form a valid multibyte character.

Description:

`mbtowc` converts a multibyte character to a wide character code and stores the result in the object pointed to by `pwc`. In the current implementation the maximum length of a character is 1 byte.

See also:

`mbstowcs`

memchr

Finds first occurrence of a character in an area of memory.

Synopsis:

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Arguments:

`const void *s`	A pointer to the area of memory to be searched.
`int c`	The character to be searched for.
`size_t n`	The size in bytes of the area of memory to be searched.

Results:

If the character is found, `memchr` returns a pointer to the matched character. It returns a `NULL` pointer if the character `c` is not in the first `n` characters of the area of memory.

Errors:

None.

Description:

`memchr` finds the first occurrence of `c` in the first `n` characters of the area of memory pointed to by `s`. `c` is converted to an `unsigned char` before the search begins.

`memchr` is side effect free.

Example:

```
char buffer[100];
char *pointer_to_p;

/*
  Find the first occurrence of 'p'
  in the buffer
*/

pointer_to_p = (char *)memchr(buffer, 'p', 100);
```

See also:

`strchr`

memcmp

Compares characters in two areas of memory.

Synopsis:

```
#include <string.h>
int memcmp(const void *s1, const void *s2,
           size_t n);
```

Arguments:

`const void *s1`	A pointer to one of the areas of memory to be compared.
`const void *s2`	A pointer to the other area of memory to be compared.
`size_t n`	The number of characters to be compared.

Results:

Returns the following:

> A negative integer if the first byte in `s1` which differs from the corresponding byte in `s2` is numerically less than the corresponding byte in `s2`.
>
> A zero value if the two areas of memory are numerically the same.
>
> A positive integer if the first byte in `s1` which differs from the corresponding byte in `s2` is numerically greater than the corresponding byte in `s2`.

Errors:

None.

Description:

`memcmp` compares the first n characters of the areas of memory pointed to by `s1` and `s2`.

The comparison is of the numerical values of the ASCII characters.

`memcmp` is side effect free.

See also:

`strcmp`

memcpy

Copies characters from one area of memory to another (no memory overlap allowed).

Synopsis:

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Arguments:

`void *s1`	A pointer to the destination of the copy.
`const void *s2`	A pointer to the source of the copy.
`size_t n`	The number of characters to be copied.

Results:

Returns the unchanged value of `s1`.

Errors:

The behavior of `memcpy` is undefined if the source and destination overlap.

Description:

`memcpy` copies `n` characters from the area of memory pointed to by `s2` (the source) to the area of memory pointed to by `s1` (the destination). The behavior of `memcpy` is undefined if the source and target areas overlap.

Calls to `memcpy` are implemented inline provided that:

1. The header file `<string.h>` has been included in the source.
2. Either the return result is not required or the argument corresponding to the formal argument `s1` is a simple expression.

```
char source[200];
char destination[200];

memcpy(destination, source, 200);
```

See also:

`memmove`

memmove

Copies characters from one area of memory to another.

Synopsis:

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Arguments:

`void *s1`	A pointer to the destination of the copy.
`const void *s2`	A pointer to the source of the copy.
`size_t n`	The number of characters to be copied.

Results:

Returns the unchanged value of `s1`.

Errors:

None.

Description:

`memmove` copies `n` characters from the area of memory pointed to by `s2` (the source) to the area of memory pointed to by `s1` (the destination). `n` characters from `S2` are first copied to a temporary area from where they are copied to `S1`. Thus the copy is defined if the areas of memory overlap.

See also:

`memcpy`

memset

Fills a given area of memory with the same character.

Synopsis:

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Arguments:

`void *s`	A pointer to the area of memory to be filled.
`int c`	The character to be used for filling.
`size_t n`	The number of characters in the area of memory be filled.

Results:

Returns the unchanged value of `s`.

Errors:

None.

Description:

`memset` fills the first `n` characters of the area of memory pointed to by `s` with the value of the character `c`. `c` is converted to an `unsigned char` before it is written into S.

Example:

```
/*
  Zero the first hundred bytes of a buffer
*/

char buffer[200];

memset(buffer,'\0', 100);
```

`mktime` Converts a broken–down time into a calendar time.

Synopsis:

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Arguments:

`struct tm *timeptr`	A pointer to a structure containing a broken–down time.

Results:

Returns the calendar time equivalent of the broken–down time passed in.

Errors:

If the broken–down time pointed to by `timeptr` cannot be represented as a calendar time, `mktime` returns –1, cast to `time_t`.

Description:

`mktime` converts the broken–down time given in the broken-down-time structure pointed to by `timeptr` into a calendar time of type `time_t`. The values of the structure components `tm_wday` and `tm_yday` are ignored. Other components are not restricted to the ranges specified in section 1.3.16. On completion all elements of the broken–down time structure are set to correct values within the ranges specified. The calendar time value `time_t` represented by the broken–down time structure is returned.

Example:

```
#include <time.h>
#include <stdio.h>

int main( )
{
  /* define a broken-down-time structure. Note that day of month is
     out of range */

  struct tm broken_down_time = {
                                0,  /* seconds                  */
                                0,  /* minutes                  */
                                11, /* hours                    */
                                34, /* day of month             */
                                0,  /* month of year            */
                                93, /* year                     */
                                0,  /* day of week (IGNORED)    */
                                0,  /* day of year (IGNORED)    */
                                0   /* daylight saving flag     */
                               };
  time_t cal_time;

  cal_time = mktime(&broken_down_time);
  printf("Time is %s\n", asctime(&broken_down_time));
  printf("Weekday is %d\n", broken_down_time.tm_wday);
}
```

See also:

`asctime` `ctime` `localtime` `clock` `difftime` `time`

modf

Splits a `double` number into fractional and integral parts.

Synopsis:

```
#include <math.h>
double modf(double value, double *intptr);
```

Arguments:

`double value`	The number to be split.
`double *intptr`	A pointer to the recipient of the integral part.

Results:

Returns the fractional part of `value` (the integral part is stored as a `double` in `*intptr`).

Errors:

If the input value cannot be represented `modf` returns it unchanged and sets `*intptr` to zero.

Description:

`modf` splits `value` into a fractional and integral part. Each part has the same sign as `value`. The integral part is stored as a `double` in `*intptr` and the fractional part is returned by `modf`.

See also:

`modff`

modff

Splits the `float` argument into fractional and integral parts.

Synopsis:

```
#include <mathf.h>
float modff(float value, float *intptr);
```

Arguments:

`float value`	The number to be split.
`float *intptr`	A pointer to the recipient of the integral part.

Results:

Returns the fractional part of `value` (the integral part is stored as a `float` in `*intptr`).

Errors:

If the input value cannot be represented `modff` returns it unchanged and sets `*intptr` to zero.

Description:

`float` form of `modf`.

See also:

`modf`

Move2D

Two–dimensional block move.

Synopsis:

```
#include <misc.h>
void Move2D(const void *src, void *dst, int width,
            int nrows, int srcwidth, int dstwidth);
```

Arguments:

`const void *src`	Source address for the block move.
`void *dst`	Destination address for the block move.
`int width`	The width in bytes of each row to be copied.
`int nrows`	The number of rows to be copied.
`int srcwidth`	The stride of the source array in bytes.
`int dstwidth`	The stride of the destination array in bytes.

Results:

None.

Errors:

The effect of the block move is undefined if either `width` or `nrows` is negative.

The effect of the block move is undefined if the source and destination blocks overlap.

The block move only makes sense if `srcwidth` and `dstwidth` are greater or equal to `width`.

Description:

Move2D copies the whole of the block of `nrows` rows each of `width` bytes from `src` to `dst`. Each row of `src` is of width `srcwidth` bytes; and each row of `dst` is of width `dstwidth` bytes. If either `width` or `nrows` are zero, the 2 dimensional move has no effect.

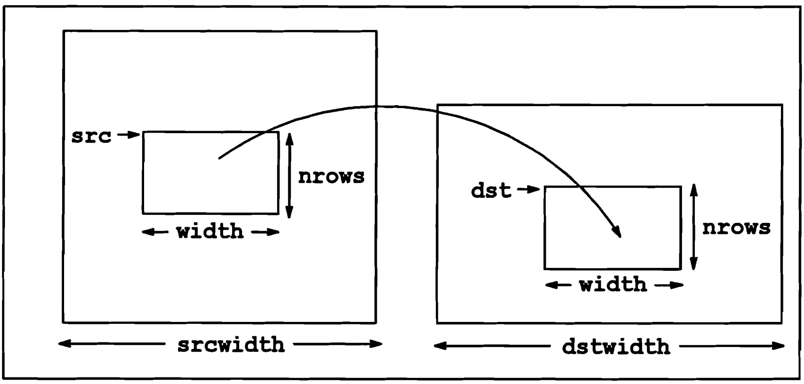

Figure 2.1 Two dimensional block move

When compiling for transputers which have the *move2dinit* and *move2dall* instructions, calls to `Move2D` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

Example:

```
#define SRCWIDTH 30
#define DSTWIDTH 50
char *src[20][SRCWIDTH];
char *dst[40][DSTWIDTH];
int width, nrows;

Move2D(src, dst, width, nrows, SRCWIDTH, DSTWIDTH);
```

See also:

`Move2DNonZero` `Move2DZero`

Move2DNonZero   Two–dimensional block move of non–zero bytes.

Synopsis:

```
#include <misc.h>
void Move2DNonZero(const void *src, void *dst, int width,
                   int nrows, int srcwidth, int dstwidth);
```

Arguments:

`const void *src`	Source address for the block move.
`void *dst`	Destination address for the block move.
`int width`	The width in bytes of each row to be copied.
`int nrows`	The number of rows to be copied.
`int srcwidth`	The stride of the source array in bytes.
`int dstwidth`	The stride of the destination array in bytes.

Results:

None.

Errors:

The effect of the block move is undefined if either `width` or `nrows` is negative.

The effect of the block move is undefined if the source and destination blocks overlap.

The block move only makes sense if `srcwidth` and `dstwidth` are greater or equal to `width`.

Description:

`Move2DNonZero` copies all non–zero bytes of the block of `nrows` rows each of `width` bytes from `src` to `dst`, leaving the bytes in the destination corresponding to the zero bytes in the source, unchanged. This can be used to overlay a non–rectangular picture onto another picture. Each row of `src` is of width `srcwidth` bytes; and each row of `dst` is of width `dstwidth` bytes.

If either `width` or `nrows` are zero, the 2 dimensional move has no effect.

Figure 2.1 (see `Move2D`) illustrates how a two dimensional block move is performed.

When compiling for transputers which have the *move2dinit* and *move2dnonzero* instructions, calls to `Move2DNonZero` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

Example:

```
#define SRCWIDTH 30
#define DSTWIDTH 50
char *src[20][SRCWIDTH];
char *dst[40][DSTWIDTH];
int width, nrows;

Move2DNonZero(src, dst, width, nrows, SRCWIDTH, DSTWIDTH);
```

See also:

Move2D Move2DZero

Move2DZero

Two–dimensional block move of zero bytes.

Synopsis:

```
#include <misc.h>
void Move2DZero(const void *src, void *dst, int width,
                int nrows, int srcwidth, int dstwidth);
```

Arguments:

`const void *src`	Source address for the block move.
`void *dst`	Destination address for the block move.
`int width`	The width in bytes of each row to be copied.
`int nrows`	The number of rows to be copied.
`int srcwidth`	The stride of the source array in bytes.
`int dstwidth`	The stride of the destination array in bytes.

Results:

None.

Errors:

The effect of the block move is undefined if either `width` or `nrows` is negative.

The effect of the block move is undefined if the source and destination blocks overlap.

The block move only makes sense if `srcwidth` and `dstwidth` are greater or equal to `width`.

Description:

`Move2DZero` copies all zero bytes of the block of `nrows` rows each of `width` bytes from `src` to `dst`, leaving the bytes in the destination corresponding to the non–zero bytes in the source, unchanged. This can be used to mask out a non–rectangular shape from a picture. Each row of `src` is of width `srcwidth` bytes; and each row of `dst` is of width `dstwidth` bytes.

If either `width` or `nrows` are zero, the 2 dimensional move has no effect.

Figure 2.1 (see `Move2D`) illustrates how a two dimensional block move is performed.

When compiling for transputers which have the *move2dinit* and *move2dzero* instructions, calls to `Move2DZero` are implemented inline, provided that the header file `<misc.h>` has been included in the source.

Example:

```
#define SRCWIDTH 30
#define DSTWIDTH 50
char *src[20][SRCWIDTH];
char *dst[40][DSTWIDTH];
int width, nrows;

Move2DZero(src, dst, width, nrows, SRCWIDTH, DSTWIDTH);
```

See also:

Move2DNonZero Move2D

open

Opens a file stream. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int open(char *name, int flags);
```

Arguments:

`char *name`	The name of the file to be opened.
`int flags`	Bit values which specify the mode in which the file is to be opened.

Results:

Returns a file descriptor for the file opened or -1 on error.

Errors:

If an error occurs `errno` is set to `EIO`.

Description:

`open` opens the low level file `name` in a mode specified by `flags`. `open` is the low level file function used by `fopen`.

`open` is not included in the reduced library.

The `flags` argument is a combination of bit values joined using the 'bitwise or' (|) operator. The bit values that can be specified are as follows:

Read/write Modes:

Flag	Meaning
`O_RDONLY`	Read only mode (priority 3).
`O_WRONLY`	Write only mode (priority 2).
`O_RDWR`	Read/write mode (priority 1).

File creation modes:

Flag	Meaning
`O_APPEND`	Characters appended to file (priority 1).
`O_TRUNC`	File truncated before writing (priority 2).

File Types:

Flag	Meaning
`O_BINARY`	File opened in binary mode (priority 2).
`O_TEXT`	File opened as a text file. (priority 1).

The **flags** argument should combine values from each of the three sections above. For example, to open a binary file for writing in append mode the call would be as follows:

```
open(filename, O_BINARY | O_WRONLY | O_APPEND);
```

To avoid conflicts between the various combinations of modes, the flag values are assigned priority levels and are decoded accordingly. Priority increases with increasing number. For example, if both `O_WRONLY` (priority 2) and `O_RDONLY` (priority 3) are specified in the same call `O_WRONLY` is ignored.

Priority levels also imply a default setting for open, namely: Read only/Text mode (`O_RDONLY | O_TEXT`). (File create modes have no significance on a read only file).

If a file which already exists is opened using `O_TRUNC` (open for writing in truncate mode), and if the host system permits it, the file will be overwritten.

See also:

creat

perror

Writes an error message to standard error.

Synopsis:

```
#include <stdio.h>
void perror(const char *s);
```

Arguments:

const char *s	A pointer to an error message string.

Results:

No value is returned.

Errors:

None.

Description:

If s is not NULL and does not point to a null character, perror writes the string s to the standard error output, followed by a colon, space, and the error message represented by the value in errno. Otherwise only the error message for errno is written. The entire message is followed by a newline.

Message strings are the same as those returned by strerror given the argument errno.

perror is not included in the reduced library.

See also:

strerror

pollkey

Gets a character from the keyboard.

Synopsis:

```
#include <iocntrl.h>
int pollkey(void);
```

Arguments:

None.

Results:

`pollkey` returns the ASCII value of a key pressed on the keyboard. It immediately returns with –1 if no keystroke is available.

Errors:

None.

Description:

`pollkey` gets a single character from the keyboard. If no keystroke is available the routine returns immediately with –1. The effect on any buffered data in the standard input stream is host-defined. The character read from the keyboard is not echoed at the terminal.

`pollkey` is not included in the reduced library.

See also:

`getkey`

pow

Calculates x to the power y.

Synopsis:

```
#include <math.h>
double pow(double x, double y);
```

Arguments:

`double x`	A number.
`double y`	The exponent.

Results:

Returns the value of `x` raised to the power `y`. If a range error occurs returns `HUGE_VAL` (with the same sign as the correct value of the function). If a domain error occurs it returns zero (0.0).

Errors:

A domain error will occur in the following situations:

1.	x == 0	AND	y <= 0
2.	x < 0	AND	y is not an integer

In these cases `errno` is set to `EDOM`.

A range error will occur if the result of `pow` is too large to fit in a double. In this case `pow` returns the value `HUGE_VAL` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`pow` calculates the value of `x` raised to the power `y`.

See also:

`powf`

powf

Calculates x to the power of y where both x and y are floats.

Synopsis:

```
#include <mathf.h>
float powf(float x, float y);
```

Arguments:

`float x`	A number.
`float y`	The exponent.

Results:

Returns the value of `x` raised to the power `y`. If a range error occurs returns `HUGE_VAL_F` (with the same sign as the correct value of the function). If a domain error occurs it returns zero (0.0F).

Errors:

A domain error will occur in the following situations:

1.	x == 0	AND	y <= 0
2.	x < 0	AND	y is not an integer

In these cases `errno` is set to `EDOM`.

A range error will occur if the result of `powf` is too large to fit in a double. In this case `powf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `pow`.

See also:

`pow`

printf

Writes a formatted string to standard output.

Synopsis:

```
#include <stdio.h>
int printf(const char *format, ...);
```

Arguments:

`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of characters written, or a negative value if an output error occurred.

Errors:

`printf` returns a negative value if an output error occurred.

Description:

`printf` writes the string pointed to by `format` to standard output. When `printf` encounters a percent sign % in the format string, it expands the equivalent argument into the format defined by the format tokens after the %. The meaning of the format string is as described for `fprintf`.

`printf` is not included in the reduced library.

See also:

`fprintf`

ProcAfter

Blocks a process until a specified transputer clock time.

Synopsis:

```
#include <process.h>
void ProcAfter(int time);
```

Arguments:

`int time`	The transputer clock time at which the process will restart.

Results:

Returns no result.

Errors:

None.

Description:

Delays execution of the current process until a specified transputer clock time. The process will begin executing some time after the clock corresponding to the current process priority reaches the value given by the input argument.

See also:

`ProcWait`

ProcAlloc

Allocates the space for and sets up a parallel process.

Synopsis:

```
#include <process.h>
Process *ProcAlloc(void (*func)(),
                   int wsize, int param_words, ...);
```

Arguments:

`void (*func)()`	A pointer to the function which will be executed as a parallel process.
`int wsize`	The size in bytes of the stack space required by the process.
`int param_words`	The number of words taken up by the arguments to `func` (less the initial process pointer).
`...`	A list of arguments which are to be passed to `func`.

Results:

Returns a pointer to a process structure which is subsequently used to refer to the process, or a `NULL` pointer if `ProcAlloc` was unable to set up the process correctly.

Errors:

Returns a `NULL` pointer if an error occurs.

Description:

`ProcAlloc` sets up a function as a parallel process and returns a pointer which is subsequently used to refer to the process.

`func` is a pointer to a function which is to be executed as a parallel process. The function pointed to by `func` must be defined in the correct manner for a C parallel process, i.e. it must accept one fixed argument and zero or more non–fixed arguments. The fixed argument is the first argument and is a process pointer. See section 5.5 of the *ANSI C Toolset User Guide.*

`wsize` is the size of the stack space required by the program and is specified as a number of bytes. If `wsize` is given the value 0 the default stack sizes of 4K on 32 bit machines and 1K on 16 bit machines are used. It is important that enough space is allocated for the stack for the process. If insufficient space is provided, the results are undefined. The runtime library needs 150 words (600 bytes for 32 bit, 300 bytes for 16 bit machines), this must be allowed for, as well as the stack requirement of the user functions (e.g. `max_stack_usage`).

`param_words` is the number of words taken up by the non–fixed arguments to the function pointed to by `func`. `ProcAlloc` expects the single fixed argument and so this need not be included in the `param_words` value. If all the non–fixed arguments are word sized then `param_words` can be considered to be the number of non–fixed arguments. If some arguments are not word sized then care should be taken to ensure that `param_words` equals the number of words occupied by the non–fixed arguments. In particular be sure to round up aggregate types to the nearest word and be careful when using argument types which will be subject to the C default argument promotions (see section 4.2.3). Because `ProcAlloc` accepts the non–fixed arguments via a variable argument list (denoted by the '`...`' in the argument list) the C default argument promotions are used on any arguments passed as part of the variable argument list, e.g. all float arguments are automatically promoted to double when passed to `ProcAlloc`. To overcome these difficulties it may be easier to pass pointers to arguments which are larger than a word or are subject to default argument promotions. Pointers are always word sized.

When the process is started it begins executing as if `func` were called with arguments equivalent to the non–fixed arguments set up in the call to `ProcAlloc`.

`ProcAlloc` uses `malloc` to allocate memory space for use by the process. All calls to `ProcAlloc` should be followed by a check for successful allocation. The behavior of running an unitialized process is not defined.

Example:

```
/* to set up fred as a concurrent process with default workspace */

#include <process.h>
#include <stdlib.h>

void fred(Process *p, int a, int b, int c)
{
  /* p is the fixed parameter */
  /* a, b and c are the non-fixed parameters */

  /* code for fred */
}

/* code fragment */

Process *p;

p = ProcAlloc(fred,  /* function to be used as a parallel process */
              0,     /* use the default stack space size          */
              3,     /* number of words taken up by non-fixed
                        parameters to fred. a, b and c are all 1
                        word long */
              1,     /* value of a when fred starts executing     */
              2,     /* value of b when fred starts executing     */
              3);    /* value of c when fred starts executing     */

if (p == NULL)
  abort();
```

See also:

`ProcInit` `ProcAllocClean`

ProcAllocClean

Cleans up after a process setup using `ProcAlloc`.

Synopsis:

```
#include <process.h>
void ProcAllocClean(Process *p);
```

Arguments:

`Process *p`	A pointer to a process structure.

Results:

None.

Errors:

If an invalid pointer is passed to `ProcAllocClean` a fatal runtime error occurs and the following message is displayed:

Fatal-C_Library-Bad pointer to process clean function

and the processor is halted. If the reduced library is used no message is displayed.

Description:

`ProcAllocClean` is used to clean up after a process when it is known to have terminated. The process is denoted by the process pointer passed in as the argument, which must have been initially set up using `ProcAlloc`. It will *not* work correctly for processes set up using `ProcInit` and if used in such a case may produce undefined behavior.

`ProcAllocClean` removes the process structure pointed to by its argument from the list of initialized processes and frees any heap space used for the process structure and workspace.

Caution: `ProcAllocClean` should only be used in the following situations:

1 with synchronous processes, i.e. those started using `ProcPar` or `ProcParList`, and it can be safely used only after the call to `ProcPar` or `ProcParList` has returned;

2 with asynchronous processes which are synchronized using `ProcJoin` or `ProcJoinList`, and it can only be safely used after the call to `ProcJoin` or `ProcJoinList` returns.

Any other use of this function may give rise to undefined behavior.

See also:

`ProcAlloc` `ProcInitClean`

`ProcAlt`

Waits for input on one of a number of channels.

Synopsis:

```
#include <process.h>
int ProcAlt(Channel *c1, ...);
```

Arguments:

`Channel *c1`	The first in a `NULL` terminated list of pointers to channels.
`...`	The remainder of the list.

Results:

Returns an index into the argument list for the ready channel.

Errors:

None.

Description:

`ProcAlt` blocks the calling process until one of the channel arguments is ready to input. The index returned for the ready channel is an integer which indicates the position of the channel in the argument list. The index numbers begin at zero for the first argument. `ProcAlt` only returns when a channel is ready to input. It does not perform the input operation, which must be done by the code following the call to `ProcAlt`.

Example:

```
/* select from channels c1, c2, c3 */

#include <process.h>

Channel *c1, *c2, *c3;
int i;

/* allocate all channels */

i = ProcAlt(c1, c2, c3, NULL);
switch(i)
{
  case 0: /* c1 selected */
          /* consume input from c1 */
          break;
  case 1: /* c2 selected */
          /* consume input from c2 */
          break;
  case 2: /* c3 selected */
          /* consume input from c3 */
          break;
}
```

See also:

`ProcAltList`

ProcAltList

Waits for input on one of a list of channels.

Synopsis:

```
#include <process.h>
int ProcAltList(Channel **clist);
```

Arguments:

`Channel **clist`	An array of pointers to channels terminated by `NULL`.

Results:

Returns an index into the `clist` array for the ready channel, or –1 if the first element in the array is `NULL` (the array is empty).

Errors:

Returns –1 if `clist` is empty.

Description:

As `ProcAlt` but takes an array of pointers to channels. Returns –1 if the `clist` array is empty.

See also:

`ProcAlt`

ProcGetPriority

Returns the priority of the calling process.

Synopsis:

```
#include <process.h>
int ProcGetPriority(void);
```

Arguments:

None.

Results:

Returns zero (0) i.e. `PROC_HIGH` for a high priority process and one (1) i.e. `PROC_LOW` for a low priority process.

Errors:

None.

Description:

Determines the priority level (high or low) of the process from which it is called. The macros `PROC_HIGH` and `PROC_LOW` are defined for use with this function.

Calls to `ProcGetPriority` are implemented inline provided that the header file `<process.h>` has been included in the source.

`ProcGetPriority` is side effect free.

See also:

`ProcReschedule`

ProcInit

Sets up a parallel process.

Synopsis:

```
#include <process.h>
int ProcInit(Process *p, void (*func)(), int *ws,
             int wsize, int param_words, ...);
```

Arguments:

`Process *p`	A pointer to a process structure which can subsequently be used to refer to the process.
`void (*func)()`	A pointer to the function which will be executed as a parallel process.
`int *ws`	A pointer to an area of memory to be used as the stack.
`int wsize`	The size in bytes of the memory area pointed to by `ws`.
`int param_words`	The number of words taken up by the arguments to `func`, (less the initial process pointer).
`...`	A list of arguments which are to be passed to `func`.

Results:

Returns zero (0) if successful, non-zero otherwise.

Errors:

If insufficient stack space has been allocated to accommodate the arguments to the function then `ProcInit` returns a non-zero value.

If the stack space pointed to by `ws` is nested within the stack space of an existing process then a fatal runtime error occurs. The fatal runtime error causes the processor to halt. If the full library is used then the following message is also output:

Fatal-C_Library-Incorrect allocation of process workspace

Description:

`ProcInit` sets up a function as a parallel process.

`p` is a pointer to a process structure which is initialized by `ProcInit`. When `ProcInit` returns, `p` is subsequently used to refer to the process. `func` is a pointer to a function which is to be executed as a parallel process. The function pointed to by `func` must be defined in the correct manner for a C parallel process, i.e. it must accept one fixed argument and zero or more non-fixed arguments. The fixed argument is the first argument and is a process pointer. See section 5.5 of the *ANSI C Toolset User Guide.*

`ws` is a pointer to the memory region which is to be used as the stack space for the parallel process. This memory region can reside anywhere within the address space of the transputer as long as it is not nested within the stack space of an existing process or main program. Thus an automatic array may not be used as stack space for a process. Usually stack space will be allocated using `malloc`, `calloc` or `realloc` or will have been declared as a static array. Failure to allocate this memory region properly will cause `ProcInit` to fail with a fatal error.

`wsize` is the size of the memory region pointed by `ws` in bytes.

`param_words` is the number of words taken up by the non-fixed arguments to the function pointed to by `func`. `ProcInit` expects the single fixed argument and so this need not be included in the `param_words` value. If all the non-fixed arguments are word sized then `param_words` can be considered to be the number of non-fixed arguments. If some arguments are not word sized then care should be taken to ensure that `param_words` equals the number of words occupied by the non-fixed arguments. In particular be sure to round up aggregate types to the nearest word and be careful when using argument types which will be subject to the C default argument promotions (see section 4.2.3). Because `ProcInit` accepts the non-fixed arguments via a variable argument list (denoted by the '`...`' in the argument list) the C default argument promotions are used on any arguments passed as part of the variable argument list, e.g. all float arguments are automatically promoted to double when passed to `ProcInit`. To overcome these difficulties it may be easier to pass pointers to arguments which are larger than a word or are subject to default argument promotions. Pointers are always word sized.

When the process is started it begins executing as if `func` were called with arguments equivalent to the non-fixed arguments set up in the call to `ProcInit`.

`ProcInit` allows more control of the memory allocated for use by the parallel process. If such control is not required then the user is recommended to use `ProcAlloc` instead.

Example:

```
/* to set up fred as a concurrent process with 4K of stack space */

#include <process.h>
#include <stdlib.h>

#define SIZE 4096

void fred(Process *p, int a, int b, int c)
{
  /* p is the fixed parameter */
  /* a, b and c are the non-fixed parameters */

  /* code for fred */
}

/* code fragment */

Process *p;
int *ws;
int result;
```

```
/* Allocate the process structure */

p = (Process *)malloc(sizeof(Process));
if (p == NULL)
  abort();

/* Allocate the stack space */

ws = (int *)malloc(SIZE);
if (ws == NULL)
  abort();

result = ProcInit(p,     /* pointer to a process structure which is
                            subsequently used as a handle to refer to
                            the process. */
                  fred,  /* function to be used as a parallel process */
                  ws,    /* pointer to stack space for the process    */
                  SIZE,  /* size in bytes of stack space allocated    */
                  3,     /* number of words taken up by non-fixed
                            parameters to fred. a, b and c are all 1
                            word long                                 */
                  1,     /* value of a when fred starts executing     */
                  2,     /* value of b when fred starts executing     */
                  3);    /* value of c when fred starts executing     */

if (result != 0)
  abort();
```

See also:

`ProcAlloc ProcInitClean`

`ProcInitClean` Cleans up after a process set up using `ProcInit`.

Synopsis:

```
#include <process.h>
void ProcInitClean(Process *p);
```

Arguments:

`Process *p`	A pointer to a process structure.

Results:

None.

Errors:

If an invalid pointer is passed to `ProcInitClean` a fatal runtime error occurs and the following message is displayed:

Fatal-C_Library-Bad pointer to process clean function

and the processor is halted. If the reduced library is used no message is displayed.

Description:

`ProcInitClean` is used to clean up after a process when it is known to have terminated. The process is denoted by the process pointer passed in as the argument, which must have been initially set up using `ProcInit`. It will *not* work correctly for processes set up using `ProcAlloc` and if used in such a case may produce undefined results.

`ProcInitClean` removes the process structure pointed to by its argument from the list of initialized processes. After `ProcInitClean` has been called, any area of heap allocated for the process structure and workspace may be safely freed, or if another memory region was used for the workspace, it may be reused.

If the workspace is freed or reused before a call to `ProcInitClean` then the behavior is undefined. **Note:** that `ProcInitClean` does not itself free *workspace* taken from the heap; this must be performed by the programmer, using the function `free`.

Caution: `ProcInitClean` should only be used in the following situations:

1 with synchronous processes, i.e. those started using `ProcPar` or `ProcParList`, and it can be safely used only after the call to `ProcPar` or `ProcParList` has returned;

2 with asynchronous processes which are synchronized using `ProcJoin` or `ProcJoinList`, and it can only be safely used after the call to `ProcJoin` or `ProcJoinList` returns.

Any other use of this function may give rise to undefined behavior.

See also:

`ProcInit` `ProcAllocClean`

`ProcJoin` Waits for a number of asynchronous processes to terminate.

Synopsis:

```
#include <process.h>
int ProcJoin(Process *p1, ...);
```

Arguments:

`Process *p1`	The first in a list of pointers to process structures.
`...`	The remainder of the list, terminated by `NULL`.

Results:

Returns 0 for success and –1 for error.

Errors:

Returns the error result –1 if an empty argument list is received.

Description:

`ProcJoin` takes as its arguments a `NULL` terminated list of process pointers. The function will not return until all the processes, denoted by the process pointers passed in as arguments, have completed (or if there was an error).

The pointers are either returned from `ProcAlloc` or initialized by a call to `ProcInit`.

`ProcJoin` is only for use with asynchronous processes started using `ProcRun`, `ProcRunHigh` and `ProcRunLow`. An attempt to use `ProcJoin` with synchronous processes (those started using `ProcPar`, `ProcParList` or `ProcPriPar`) will give undefined results.

A process which makes a call to `ProcStop` should not be used with `ProcJoin`. `ProcStop` will stop the process thereby preventing it from terminating normally, thus `ProcJoin` will be unable to detect the termination of the process.

See also:

`ProcJoinList` `ProcStop`

ProcJoinList Waits for a number of asynchronous processes to terminate.

Synopsis:

```
#include <process.h>
int ProcJoinList(Process **p);
```

Arguments:

`Process **p`	An array of pointers to process structures terminated by `NULL`.

Results:

Returns 0 for success and –1 for error.

Errors:

Returns the error result –1 if an empty array is passed in.

Description:

As `ProcJoin` but takes a `NULL` terminated array of process pointers as its argument.

The pointers are either returned from `ProcAlloc` or initialized by a call to `ProcInit`.

`ProcJoinList` is only for use with asynchronous processes started using `ProcRun`, `ProcRunHigh` and `ProcRunLow`. An attempt to use `ProcJoinList` with synchronous processes (those started using `ProcPar`, `ProcParList` or `ProcPriPar`) will give undefined results.

A Process which makes a call to `ProcStop` should not be used with `ProcJoinList`. `ProcStop` will stop the process thereby preventing it from terminating normally, thus `ProcJoinList` will be unable to detect the termination of the process.

See also:

`ProcJoin ProcStop`

ProcPar

Starts a group of processes in parallel.

Synopsis:

```
#include <process.h>
void ProcPar(Process *p1, ...);
```

Arguments:

`Process *p1`	The first in a list of pointers to process structures.
`...`	The remainder of the list. Terminated by `NULL`.

Results:

Returns no result.

Errors:

If `ProcPar` detects that a process which it is about to start is already running then the following fatal runtime error is issued:

Fatal–C_Library–Attempt to start a process which is already running

Thus it is illegal to attempt to run a process in parallel with itself.

Description:

`ProcPar` takes a `NULL` terminated list of pointers to processes and starts the corresponding processes in parallel with each other at the priority of the calling process. `ProcPar` will not return until all of the processes associated, with pointers passed as arguments to it, have terminated. The process pointers are either returned from `ProcAlloc` or initialized by `ProcInit`.

A process started using `ProcPar` is called a 'synchronous process'.

Example:

```
/* start the four processes denoted by process
   pointers p1, p2, p2, p4 in parallel. */

#include <process.h>

Process *p1, *p2, *p3, *p4;

/* Set up and allocate processes */

ProcPar(p1, p2, p3, p4, NULL);
```

See also:

`ProcParList ProcStop`

ProcParam

Changes process arguments.

Synopsis:

```
#include <process.h>
void ProcParam(Process *p, ...);
```

Arguments:

`Process *p`	A pointer to a process structure.
`...`	A list of arguments which are passed to the function associated with `p`.

Results:

Returns no result.

Errors:

None.

Description:

`ProcParam` can be used to change the non-fixed arguments (see `ProcAlloc` or `ProcInit` for a definition of 'non-fixed arguments') of the function associated with `p`. See also section 5.5 of the *ANSI C Toolset User Guide.*

`p` is a pointer to a process structure which was previously returned from a call to `ProcAlloc` or set up using a call to `ProcInit`.

The number of words of arguments should be the same as that specified in the original call to `ProcAlloc` or `ProcInit` which set up `p`. If too many words of arguments are given, the extra words are ignored. If too few words of arguments are given then the unspecified words are undefined.

`ProcParam` must be used before the process begins execution. If it used while the process is running then the results are undefined.

See also:

`ProcAlloc ProcInit ProcAllocClean`

`ProcParList`

Starts a group of parallel processes.

Synopsis:

```
#include <process.h>
void ProcParList(Process **plist);
```

Arguments:

`Process **plist`	A array of pointers to processes terminated by `NULL`.

Results:

Returns no result.

Errors:

If `ProcParList` detects that a process which it is about to start is already running then the following fatal runtime error is issued:

Fatal–C_Library–Attempt to start a process which is already running

Thus it is illegal to attempt to run a process in parallel with itself.

Description:

As `ProcPar` but takes an array of pointers to processes. The pointers are either returned directly from `ProcAlloc` or are pointers to processes initialized by `ProcInit`.

A process started using `ProcParList` is called a 'synchronous process'.

See also:

`ProcPar`

`ProcPriPar`

Starts a pair of processes at high and low priority.

Synopsis:

```
#include <process.h>
void ProcPriPar(Process *phigh, Process *plow)
```

Arguments:

`Process *phigh`	A pointer to the high priority process.
`Process *plow`	A pointer to the low priority process.

Results:

Returns no result.

Errors:

Any attempt to call `ProcPriPar` from a high priority process generates a runtime fatal error and the following message is displayed:

Fatal-C_Library-Nested Pri Pars are illegal

If `ProcPriPar` detects that a process which it is about to start is already running then the following fatal runtime error is issued:

Fatal–C_Library–Attempt to start a process which is already running

Thus it is illegal to attempt to run a process in parallel with itself.

Description:

Starts two processes in parallel, the first at high priority and the second at low priority. Process pointers will have been returned directly from `ProcAlloc`, or are pointers to processes initialized by `ProcInit`.

`ProcPriPar` cannot be called from a high priority process.

A process started using `ProcPriPar` is called a 'synchronous process'.

See also:

`ProcPar ProcStop`

ProcReschedule

Reschedules a process.

Synopsis:

```
#include <process.h>
void ProcReschedule(void);
```

Arguments:

None.

Results:

Returns no result.

Errors:

None.

Description:

Causes the calling process to be rescheduled, that is, placed at the end of the active process queue.

Calls to `ProcReschedule` are implemented inline provided that the header file `<process.h>` has been included in the source.

See also:

`ProcGetPriority`

ProcRun

Starts a process at the current priority.

Synopsis:

```
#include <process.h>
void ProcRun(Process *p);
```

Arguments:

`Process *p`	A pointer to a process.

Results:

Returns no result.

Errors:

If `ProcRun` detects that a process which it is about to start is already running then the following fatal runtime error is issued:

Fatal–C_Library–Attempt to start a process which is already running

Thus it is illegal to attempt to run a process in parallel with itself.

Description:

Executes a process in parallel with the calling process and at the same priority. The two processes run independently and any interaction between them must be specifically set up using channel communication routines. The process pointer is returned directly from `ProcAlloc` or is a pointer to a process initialized by `ProcInit`.

`ProcRun` returns immediately after starting the process. Thus a process started using `ProcRun` is called an 'asynchronous process'.

Care should be taken that asynchronous processes do not attempt to communicate with the server when it has been terminated by the main program. The `ProcJoin` function can be used to guard against this. For more details see section 5.5.5 in the *ANSI C Toolset User Guide*.

See also:

`ProcRunHigh` `ProcRunLow` `ProcPar` `ProcParList` `ProcPriPar` `ProcStop` `ProcJoin` `ProcJoinList`

ProcRunHigh

Starts a high priority process.

Synopsis:

```
#include <process.h>
void ProcRunHigh(Process *p);
```

Arguments:

`Process *p`	A pointer to a process.

Results:

Returns no result.

Errors:

If `ProcRunHigh` detects that a process which it is about to start is already running then the following fatal runtime error is issued:

Fatal–C_Library–Attempt to start a process which is already running

Thus it is illegal to attempt to run a process in parallel with itself.

Description:

As `ProcRun` but starts the process at high priority. The process pointer will have been returned directly from `ProcAlloc`, or will be a pointer to a process initialized by `ProcInit`.

As with `ProcRun` care should be taken that processes started with this function terminate before the main program.

A process started using `ProcRunHigh` is called an 'asynchronous process'.

See also:

`ProcRun` `ProcRunLow` `ProcPar` `ProcParList` `ProcPriPar` `ProcStop`

ProcRunLow

Starts a low priority process.

Synopsis:

```
#include <process.h>
void ProcRunLow(Process *p);
```

Arguments:

`Process *p`	A pointer to a process.

Results:

Returns no result.

Errors:

If `ProcRunLow` detects that a process which it is about to start is already running then the following fatal runtime error is issued:

Fatal–C_Library–Attempt to start a process which is already running

Thus it is illegal to attempt to run a process in parallel with itself.

Description:

As `ProcRun` but starts the process at low priority. The process pointer will have been returned directly from `ProcAlloc`, or will be a pointer to a process initialized by `ProcInit`.

As with `ProcRun` care should be taken that processes started with this function terminate before the main program.

A process started using `ProcRunLow` is called an 'asynchronous process'.

See also:

`ProcRunHigh` `ProcRun` `ProcPar` `ProcParList` `ProcPriPar` `ProcStop`

ProcSkipAlt

Checks specified channels for ready input.

Synopsis:

```
#include <process.h>
int ProcSkipAlt(Channel *c1, ...);
```

Arguments:

`Channel *c1`	The first in a list of pointers to channels.
`...`	The remainder of the list. Terminated by `NULL`.

Results:

Returns an index into the argument list for the channel ready to input, or –1 if no channel is ready.

Errors:

None.

Description:

As `ProcAlt` but does not wait for a ready channel. If no channel is ready `ProcSkipAlt` returns immediately with the value –1.

Example:

```
/* select from channels c1, c2, c3 */

#include <process.h>

Channel *c1, *c2, *c3;
int i;

/* set up channels */

i = ProcSkipAlt(c1, c2, c3, NULL);
switch(i)
{
  case -1: /* no channel ready */
           break;
  case 0:  /* c1 selected */
           /* consume input from c1 */
           break;
  case 1:  /* c2 selected */
           /* consume input from c2 */
           break;
  case 2:  /* c3 selected */
           /* consume input from c3 */
           break;
}
```

See also:

`ProcAlt` `ProcSkipAltList`

ProcSkipAltList

Checks a list of channels for ready input.

Synopsis:

```
#include <process.h>
int ProcSkipAltList(Channel **clist);
```

Arguments:

`Channel **clist`	An array of pointers to channels terminated by `NULL`.

Results:

As `ProcSkipAlt`.

Errors:

None.

Description:

As `ProcSkipAlt` but takes a list of pointers to channels.

See also:

`ProcSkipAlt`

ProcStop

De-schedules a process.

Synopsis:

```
#include <process.h>
void ProcStop(void);
```

Arguments:

None.

Results:

Returns no result.

Errors:

None.

Description:

`ProcStop` causes the current process to be stopped. The process stops executing immediately and is removed from the transputer scheduling lists. Thus it cannot be restarted again.

`ProcStop` should not be used in a synchronous process (started using `ProcPar`, `ProcParList` or `ProcPriPar`) or in any asynchronous process (started using `ProcRun`, `ProcRunHigh` or `ProcRunLow`) which is the subject of a call to `ProcJoin` or `ProcJoinList`. This is because `ProcStop` prevents normal termination of a process.

Thus if a process which is associated with a call to one of `ProcPar`, `ProcParList`, `ProcPriPar`, `ProcJoin` or `ProcJoinList` makes a call to `ProcStop` then these functions are unable to terminate because they rely on all their associated processes terminating normally.

`ProcStop` may also be used to stop processes, declared at configuration level i.e. in the configuration description file. This is achieved by calling `ProcStop` from the main thread of execution of a C program.

See also:

`ProcJoin` `ProcJoinList` `ProcPar` `ProcParList` `ProcPriPar` `ProcRun` `ProcRunHigh` `ProcRunLow`

ProcTime

Determines the transputer clock time.

Synopsis:

```
#include <process.h>
int ProcTime(void);
```

Arguments:

None.

Results:

Returns the value of the transputer clock.

Errors:

None.

Description:

Determines the transputer clock time. The value of the high priority clock is returned for high priority processes and the value of the low priority clock is returned for low priority processes. Values returned by this function can be used by `ProcTimeAfter`, `ProcTimePlus`, and `ProcTimeMinus`.

Calls to `ProcTime` are implemented inline provided that the header file `<process.h>` has been included in the source.

`ProcTime` is side effect free.

See also:

`ProcTimeAfter` `ProcTimePlus` `ProcTimeMinus`

`ProcTimeAfter` Determines the relationship between clock values.

Synopsis:

```
#include <process.h>
int ProcTimeAfter(const int time1, const int time2);
```

Arguments:

`const int time1`	A transputer clock value returned by `ProcTime`.
`const int time2`	A transputer clock value returned by `ProcTime`.

Results:

Returns 1 if `time1` is after `time2`, otherwise 0.

Errors:

None.

Description:

Determines the relationship between two transputer clock values. Remember that the transputer clock is cyclic.

This is equivalent to:

```
(ProcTimeMinus(time1, time2) > 0)
```

`ProcTimeAfter` is side effect free.

See also:

`ProcTime` `ProcTimePlus` `ProcTimeMinus`

ProcTimeMinus

Subtracts two transputer clock values.

Synopsis:

```
#include <process.h>
int ProcTimeMinus(const int time1, const int time2);
```

Arguments:

`const int time1`	A transputer clock value returned by `ProcTime`.
`const int time2`	A transputer clock value returned by `ProcTime`.

Results:

Returns the result of subtracting `time2` from `time1`.

Errors:

None.

Description:

Subtracts one clock value from another using modulo arithmetic. No overflow checking takes place and the clock values are cyclic.

`ProcTimeMinus` is side effect free.

See also:

`ProcTime` `ProcTimeAfter` `ProcTimeMinus`

`ProcTimePlus`

Adds two transputer clock values.

Synopsis:

```
#include <process.h>
int ProcTimePlus(const int time1, const int time2);
```

Arguments:

`const int time1`	A transputer clock value returned by `ProcTime`.
`const int time2`	A transputer clock value returned by `ProcTime`.

Results:

Returns the result of adding `time1` to `time2`.

Errors:

None.

Description:

Adds one clock value to another using modulo arithmetic. No overflow checking takes place and the values are cyclic.

`ProcTimePlus` is side effect free.

See also:

`ProcTime` `ProcTimeAfter` `ProcTimeMinus`

ProcTimerAlt

Checks input channels with time out.

Synopsis:

```
#include <process.h>
int ProcTimerAlt(int time, Channel *c1, ...);
```

Arguments:

`int time`	An absolute transputer clock time, after which the function aborts if no communication occurs.
`Channel *c1`	The first in a list of pointers to channels.
`...`	The remainder of the list. The list must be terminated by `NULL`.

Results:

Returns an index to the argument list, or –1 if the routine times out.

Errors:

None.

Description:

As `ProcAlt` but controlled by a timeout. If the transputer clock value associated with the current priority exceeds `time` before any communication occurs, the routine terminates and returns the value –1.

Example:

```
/* select from channels c1, c2, c3 */

#include <process.h>

Channel *c1, *c2, *c3;
int i;

/* set up channels */

i = ProcTimerAlt(ProcTimePlus(ProcTime(), 50000), c1, c2, c3, NULL);
switch(i)
{
  case -1: /* timed out */
           break;
  case 0:  /* c1 selected */
           /* consume input from c1 */;
           break;
  case 1:  /* c2 selected */
           /* consume input from c2 */
           break;
  case 2:  /* c3 selected */
           /* consume input from c3 */
           break;
}
```

See also:

`ProcAlt` `ProcTimerAltList`

`ProcTimerAltList` Checks a list of channels for input with time out.

Synopsis:

```
#include <process.h>
int ProcTimerAltList(int time, Channel **clist)
```

Arguments:

`int time`	The absolute transputer clock time after which the function aborts if no communication occurs.
`Channel **clist`	An array of pointers to channels terminated by `NULL`.

Results:

Returns an index into the clist array for the ready channel, or –1 if either the routine times out or the first element in the array is `NULL` (an empty array).

Errors:

None.

Description:

As `ProcTimerAlt`, but takes an array of pointers to channels.

See also:

`ProcTimerAlt`

ProcWait

Suspends a process for a specified time.

Synopsis:

```
#include <process.h>
void ProcWait(int time);
```

Arguments:

`int time`	The time delay measured in transputer clock ticks.

Results:

Returns no result.

Errors:

None.

Description:

Suspends execution of a process for a specified period of time. After the period expires, the process is rescheduled. The delay is measured at the current clock priority.

See also:

`ProcAfter`

`putc`

Writes a character to a file stream.

Synopsis:

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Arguments:

`int c`	The character to be written.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the character written if the write is successful, or `EOF` if a write error occurs.

Errors:

`putc` returns `EOF` if a write error occurs.

Description:

`putc` converts `c` to an unsigned char, writes it to the output stream pointed to by `stream`, and advances the read/write position indicator for the file stream.

`putc` is not included in the reduced library.

See also:

`fputc`

putchar

Writes a character to standard output.

Synopsis:

```
#include <stdio.h>
int putchar(int c);
```

Arguments:

`int c`	The character to be written.

Results:

Returns the character written if successful. If a write error occurs, `putchar` returns `EOF`.

Errors:

`putchar` returns `EOF` if a write error occurs.

Description:

`putchar` converts `c` to an unsigned char, writes it to the standard output stream, and advances the read/write position indicator for that file stream.

`putchar` is not included in the reduced library.

See also:

`fputc` `getchar` `putc`

puts

Writes a line to standard output.

Synopsis:

```
#include <stdio.h>
int puts(const char *s);
```

Arguments:

`const char *s`	A pointer to the string to be written.

Results:

Returns non-negative if successful, `EOF` if unsuccessful.

Errors:

`puts` returns `EOF` if unsuccessful.

Description:

`puts` writes the string pointed to by `s` to the standard output file stream, followed by a newline character. The write does not include the string terminating character.

`puts` is not included in the reduced library.

See also:

`fputs` `getchar` gets `putchar`

qsort

Sorts an array of objects.

Synopsis:

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Arguments:

`void *base`	A pointer to the start of the array to be sorted.
`size_t nmemb`	The number of objects in the array.
`size_t size`	The size of the array objects.
`int (*compar)(const void *, const void *)`	A pointer to the comparison function.

Results:

Returns no value.

Errors:

None.

Description:

`qsort` sorts objects in the array pointed to by `base` into ascending order, according to comparisons performed by the function pointed to by `compar`. The array contains `nmemb` objects of `size` bytes. The comparison function must return an integer less than, equal to, or greater than zero, depending on whether the first argument to the function is considered to be less than, equal to, or greater than the second argument. If two elements compare equal their order in the sorted array is undefined.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int sort_compare(const void *arg1,
                 const void *arg2)
{
  return (int) (*((int *)arg1) - *((int *)arg2));
{

int main()
{
  int i[10] = {1, 4, 6, 5, 2, 7, 9, 3, 8, 0};
  int j;

  qsort(i, 10, sizeof(int), sort_compare);
  for (j = 0; j < 10; ++j)
    printf("%d\n", i[j]);
}
```

See also:

`bsearch`

raise

Forces a pseudo–exception via a signal handler.

Synopsis:

```
#include <signal.h>
int raise(int sig);
```

Arguments:

`int sig`	A signal number, as defined in `signal.h`.

Results:

Returns zero (0) if successful, non-zero if unsuccessful.

Errors:

If `raise` is called with an unrecognized signal number, it returns a non-zero value.

Description:

`raise` is used to send a signal to the running program. It causes the function associated with signal number `sig` to be called. Functions are associated with signal numbers using the `signal` function.

Signals which can be raised are listed under the signal handling setup function `signal`.

See also:

`signal`

`rand`

Generates a pseudo-random number.

Synopsis:

```
#include <stdlib.h>
int rand(void);
```

Arguments:

None.

Results:

Returns a positive pseudo-random integer.

Errors:

None.

Description:

`rand` generates a pseudo-random integer in the range 0 to `RAND_MAX`.

See also:

`srand`

read

Reads bytes from a file. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int read(int fd, char *buf, int n);
```

Arguments:

`int fd`	A file descriptor.
`char *buf`	A pointer to a buffer where the bytes will be stored.
`int n`	The maximum number of bytes that `read` will attempt to obtain.

Results:

Returns the number of bytes read or –1 on error.

Errors:

If an error occurs `read` sets `errno` to the value `EIO`.

Description:

`read` attempts to read `n` bytes from the file described by `fd` into the buffer pointed to by `buf`. It returns the number of bytes actually read. `read` may return a value less than `n` if an end of file occurred or if the file is a terminal file, e.g. standard input, if an end–of–line is encountered. `n` may be zero or negative but in these cases no input will occur.

`read` is not included in the reduced library.

See also:

`write`

realloc

Changes the size of an object previously allocated using malloc, calloc or realloc.

Synopsis:

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Arguments:

void *ptr	A pointer to the area of memory.
size_t size	The new size of the area of memory.

Results:

Returns a pointer to the allocated space. If it was not possible to allocate size bytes, or if the size requested is zero and the pointer argument is NULL, realloc returns a NULL pointer.

Errors:

If it is not possible to allocate size bytes, realloc returns a NULL pointer. If ptr does not point to an area of memory which was previously allocated by calloc, malloc, or realloc and which has not been deallocated by a call to free or realloc, a fatal runtime error occurs and the following message is generated:

Fatal-C_Library-Error in realloc(), bad pointer or heap corrupted

Description:

realloc allocates an area of memory of size size, and copies the previously allocated area of memory pointed to by ptr into the newly allocated area. If the previous area is larger than the new area, the overflow will be lost.

If ptr is NULL, realloc behaves like a call to malloc.

If size is zero and ptr is not a NULL pointer, the object pointed to by ptr is freed. If ptr is invalid a runtime error from free may be generated.

See also:

calloc free malloc

remove

Removes a file.

Synopsis:

```
#include <stdio.h>
int remove(const char *filename);
```

Arguments:

`const char *filename`	A pointer to the filename string.

Results:

Returns zero (0) if successful and non-zero if unsuccessful.

Errors:

If the remove operation was unsuccessful, `remove` returns a non-zero value.

Description:

`remove` deletes the file identified by the string pointer `filename`. If the file is open it will be deleted only if this is permitted by the host system.

`remove` is not included in the reduced library.

See also:

`rename`

rename

Renames a file.

Synopsis:

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

Arguments:

`const char *old`	A pointer to the old filename.
`const char *new`	A pointer to the new filename.

Results:

Returns zero if `rename` was successful and non-zero if it was not.

Errors:

If the rename was unsuccessful, `rename` returns a non-zero value.

Description:

`rename` changes the name of the file from `old` string to `new` string. If a file with the new name already exists the existing file will only be overwritten if this is permitted by the host operating system.

`rename` is not included in the reduced library.

See also:

`remove`

rewind

Sets the file position indicator to the start of a file stream.

Synopsis:

```
#include <stdio.h>
void rewind(FILE *stream);
```

Arguments:

`FILE *stream`	A pointer to a file stream.

Results:

No value is returned.

Errors:

None.

Description:

`rewind` sets the file position indicator of the file `stream` stream to the start of the file. The error indicators for the stream are cleared.

`rewind` is not included in the reduced library.

Example:

```
#include <stdio.h>

int main( )
{
   FILE *stream;

   stream = fopen("data.dat","w+");

   if (stream == NULL)
      printf("Couldn't open data.dat for write.\n");
   else
   {
      fprintf(stream, "01234");
      rewind(stream);
      printf("First character in data.dat is: '%c'\n", getc(stream));
   }
}

/*
 * Output:
 *       First character in data.dat is '0'
 */
```

See also:

`fsetpos`

`scanf` Reads formatted data from standard input.

Synopsis:

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Arguments:

`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of inputs which have been successfully converted. If an end of file character occurred before any conversions took place, `scanf` returns `EOF`.

Errors:

If an end of file character occurred before any conversions took place, `scanf` returns `EOF`. Other failures cause termination of the procedure.

Description:

`scanf` matches the data read from the standard input to the specifications set out by the format string, `format`. See `fscanf` for a description of the format string.

`scanf` is not included in the reduced library.

See also:

`fscanf`

`segread` Reads host processor segment registers. MS-DOS only.

Synopsis:

```
#include <dos.h>
void segread(struct SREGS *segregs);
```

Arguments:

`struct SREGS *segregs`	The read-in values of the segment registers.

Results:

Returns no result.

Errors:

Any error sets `errno` to the value `EDOS`. Any attempt to use `segread` on operating systems other than MS-DOS also sets `errno`. Failure of the function may also generate the server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`segread` reads the current values of the host 80*x* 86 processor's segment registers into `segregs`.

`segread` is not included in the reduced library.

See also:

`intdos intdosx`

`SemAlloc`

Allocates and initializes a semaphore.

Synopsis:

```
#include <semaphor.h>
Semaphore *SemAlloc(int value);
```

Arguments:

`int value`	The initial value of the semaphore.

Results:

Returns a pointer to an initialized semaphore or `NULL` on error.

Errors:

If space cannot be allocated `SemAlloc` returns a `NULL` pointer.

Description:

Allocates space for a semaphore and returns a pointer to it. The semaphore is set to the `value` argument.

The space reserved for the semaphore by `SemAlloc` may subsequently be freed by passing the returned semaphore pointer to `free`.

See also:

`SemInit`

`SemInit`

Initializes an existing semaphore.

Synopsis:

```
#include <semaphor.h>
void SemInit(Semaphore *sem, int value);
```

Arguments:

`Semaphore *sem`	A pointer to a semaphore.
`int value`	The initial value of the semaphore.

Results:

Returns no result.

Errors:

None.

Description:

`SemInit` initializes the semaphore pointed to by `sem` and assigns to it the initial value `value`.

See also:

`SemAlloc`

SemSignal

Releases a semaphore.

Synopsis:

```
#include <semaphor.h>
void SemSignal(Semaphore *sem);
```

Arguments:

`Semaphore *sem`	A pointer to a semaphore.

Results:

Returns no result.

Errors:

None.

Description:

Releases the semaphore pointed to by `sem` and runs the next process on the semaphore's queue. If no processes are waiting on the queue the semaphore value is incremented.

See also:

`SemWait`

SemWait

Acquires a semaphore.

Synopsis:

```
#include <semaphor.h>
void SemWait(Semaphore *sem);
```

Arguments:

`Semaphore *sem`	A pointer to a semaphore.

Results:

Returns no result.

Errors:

None.

Description:

Blocks the current process if the semaphore is already set to zero (acquired), otherwise acquires the semaphore, decrements its value, and continues the process. Blocked processes are added to a queue associated with the semaphore and do not continue until the semaphore is released by a call to `SemSignal` by another process.

See also:

`SemSignal`

server_transaction

Calls any `iserver` function.

Synopsis:

```
#include <iocntrl.h>
int server_transaction(char *message, int length,
                       char *reply);
```

Arguments:

`char *message`	The server packet to be sent.
`int length`	The length of the server packet.
`char *reply`	A pointer to an array where the reply packet is to be stored.

Results:

Returns the length in bytes of the server reply packet, or –1 if an error occurs.

Errors:

possible causes of error are:

> `length` being less than the minimum server packet length of 6 bytes.
>
> `length` being greater than 510.
>
> `length` being an odd number.

Description:

The runtime library provides functions which access a defined subset of ISERVER functions. Some server functions are therefore not directly accessible by C function calls.

`server_transaction` allows controlled access to any ISERVER function from a C program. It allows the full functionality of the supplied ISERVER to be used from C and supports the calling of user-defined functions and alternative servers. A list of callable functions supplied with the standard toolset ISERVER can be found in appendix D '*ISERVER protocol*' of the accompanying *ANSI C Toolset Reference Manual.*

`server_transaction` sends the packet pointed to by `message`, of length `length`, to the server. The server reply is stored in the array pointed to by `reply`.

For those familiar with occam, `server_transaction` performs the equivalent of the following occam output and input statements:

```
ToServer   ! length::message
FromServer ? replylen::reply
```

where: `ToServer` and `FromServer` are the server channels.

`length` and `replylen` are the packet lengths and `message` and `reply` are the data packets themselves.

`replylen` is the value returned by the function if no error occurs.

`server_transaction` provides low level access to the server in a secure manner. The user constructed packet is forwarded to the server, and the reply sent, via *protected* channels.

Note: There is no protection against the message and reply pointers being the same, in which case the original message packet is overwritten.

The following example uses `server_transaction` to obtain the transputer board size by calling the `Getenv` server function.

The structure of the packet to request the boardsize environment variable is given below. Numbers along the top row are Byte numbers.

0	1	2	3	4	5	6	7	8	9	10	11	12
32	10	00	I	B	O	A	R	D	S	I	Z	E

Byte 0 is the tag of the `Getenv` function. Bytes 1 and 2 make up a 16 bit number which represents the length of the string `IBOARDSIZE`. The string follows from byte 3 onwards.

The reply packet is similar except that byte 0 is the result byte and the string contains the value of the environment variable.

Example:

```
#include <misc.h>
#include <stdio.h>

int main()
{
  /* 512 byte buffers */
  char message[512], reply[512];
  /* The env variable of interest */
  char *name = "IBOARDSIZE";
  int length, i;

  /* set up packet to send */
  message[0] = 32;                /* getenv tag */
  /* length of env variable name */
  message[1] = strlen(name);
  message[2] = 0;
  strcpy(&message[3], name);
  /* calculate total length of packet */
  length = 3 + strlen(name);
  /* make sure length is an even number */
  length = (length + 1) & ~1;
  /* perform the transaction */
  length = server_transaction(message, length, reply);
  /* process reply */
  if (length == -1)
    printf("error in server transaction\n");
  else
  {
    /* print out result byte */
    printf("result = %d\n", reply[0]);
    /* print out length of env variable value */
    printf("length of result string = %d\n",reply[1]);
    /* terminate the result string */
    reply[(int)reply[1] + 3] = '\0';
    /* print out the result string */
    printf("string = [%s]\n", &reply[3]);
  }
}
```

set_abort_action

Sets/queries action taken by abort.

Synopsis:

```
#include <misc.h>
int set_abort_action(int mode);
```

Arguments:

`int mode`	The mode to be set.

Results:

Returns the previous termination mode (the mode in operation before `set_abort_action` was called).

Errors:

None.

Description:

Sets, or queries, the mode of termination for `abort`. `mode` can have any of the following values:

`ABORT_EXIT`	Causes a call to `abort` to exit the program without halting the transputer.
`ABORT_HALT`	Causes a subsequent call to `abort` to halt the transputer.
`ABORT_QUERY`	Returns the current abort mode. Leaves the mode un changed.

If `ABORT_HALT` is used `abort` first enables `HALT` mode by setting the `Halt-On-Error` flag and then sets the processor Error flag. When the transputer halts, a message similar to the following message is displayed by the server:

```
Error: Transputer error flag has been set.
```

Note: Care should be taken when calling `set_abort_action` in a concurrent environment. Calls to the function by independently executing, unsynchronized processes may change the abort action. `set_abort_action` should normally be called at the start of the program to set the action of `abort` for the entire program.

See also:

`abort`

setbuf

Controls file buffering.

Synopsis:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`char *buf`	A pointer to an array of size BUFSIZ or `NULL`.

Results:

Returns no value.

Errors:

None.

Description:

`setbuf` may be called after the file associated with `stream` has been opened, but before it has been read from or written to. `setbuf` causes `stream` to be fully buffered in the array `buf`. It is equivalent to a call to `setvbuf` with the values `_IOFBF` for `mode` and `BUFSIZ` for `size`. If `buf` is a `NULL` pointer, the stream will not be buffered.

`setbuf` is not included in the reduced library.

See also:

`setvbuf`

setjmp

Sets up a non-local jump.

Synopsis:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Arguments:

`jmp_buf env`	An array into which a copy of the calling environment is put.

Results:

When first called, `setjmp` stores the calling environment in `env` and returns zero. After a subsequent call to `longjmp` it returns a value set by `longjmp`, which is always non-zero.

Errors:

The `setjmp` function should only appear in one of the following contexts:

- The entire controlling expression of a selection or iteration statement.
- One operand of a relational or equality operator with the other operand being an integral constant expression. The resultant expression controls a selection or iteration statement.
- The operand of a unary ! operator. The resultant expression controls a selection or an iteration statement.
- The complete expression of an expression statement.

Description:

`setjmp` is used to set up a non-local goto by saving the calling environment in `env`. This environment is used by the `longjmp` function.

When first called, `setjmp` stores the calling environment in `env` and returns zero. A subsequent call to `longjmp` using `env` will cause execution to continue as if the call to `setjmp` had just returned with the value given in the call to `longjmp`. This value will always be non-zero, if `longjmp` is called with a value of 0 then the corresponding `setjmp` returns 1.

See also:

`longjmp`

setlocale

Sets or interrogates part of the program's locale.

Synopsis:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

Arguments:

`int category`	A specification of which part of the locale is to be set or interrogated.
`const char *locale`	A pointer to the string which selects the environment of the locale.

Results:

Returns "C" if `locale` is `NULL`, if `*locale` is `NULL`, or if `*locale` is "C". Otherwise returns `NULL`.

Errors:

Returns `NULL` if the arguments are invalid.

Description:

`setlocale` sets or interrogates part of the program's locale according to the values of `category` (the part to be set) and `locale` (a pointer to a string describing the environment to which it is to be set).

`category` can take the following values:

1	`LC-ALL`	All categories.
2	`LC_COLLATE`	Affects `strcoll` and `strxfrm`.
3	`LC_CTYPE`	Affects character handling
4	`LC_NUMERIC`	Affects the format of the decimal point (e.g., '.' ',', etc).
5	`LC_TIME`	Affects the `strftime` function.
6	`LC_MONETARY`	Affects monetary formatting information. If

`locale` is a null string, `setlocale` returns the current locale for the given category. In the current implementation the only acceptable locale is "C".

See also:

`localeconv`

setvbuf

Defines the way that a file stream is buffered.

Synopsis:

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode,
            size_t size);
```

Arguments:

`FILE *stream`	A pointer to a file stream.
`char *buf`	A pointer to a file buffer.
`int mode`	The way the file stream is to be buffered.
`size_t size`	The size of the file buffer.

Results:

`setvbuf` returns zero if successful, and non-zero if the operation fails.

Errors:

If `mode` or `size` is invalid, or `stream` cannot be buffered, `setvbuf` returns a non-zero value.

Description:

`setvbuf` may be called after the file associated with `stream` has been opened, but before it has been read from or written to. `setvbuf` causes `stream` to be buffered in the format specified by mode. Valid formats are:

`_IOFBF`	Fully buffered I/O
`_IOLBF`	Line buffered output
`_IONBF`	Unbuffered I/O

The buffer used is of `size` bytes. If `buf` is not a `NULL` pointer, it is used as the buffer, otherwise an internally allocated array is used.

`setvbuf` is not included in the reduced library.

See also:

`setbuf`

`signal` Defines the way that errors and exceptions are handled.

Synopsis:

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

Arguments:

`int sig`	A signal number (a predefined value, describing an error/exception type).
`void (*func)(int)`	A signal handler function which is invoked when signal `sig` is raised.

Results:

If the signal number is recognized a pointer to the function previously associated with the signal number `sig` is returned, otherwise `SIG_ERR` is returned.

Errors:

If the predefined error/exception value is not recognized by `signal`, `signal` returns `SIG_ERR` and sets `errno` to the value `ESIGNUM`.

Description:

`signal` specifies the functions to be called on reception of particular, predetermined signal values.

`func` can be any user-defined function which takes a single `int` parameter and returns `void`, or one of the following two predefined functions which are implemented as macros in the `signal.h` header file:

`SIG_DFL`	Uses the default system error/exception handling for the pre-defined value.
`SIG_IGN`	Ignores the error/exception.

The functions will then be called in response to a `raise` or other invocation of the signal handler, using a signal number as a parameter.

When a signal is raised the default signal handling is reset by a call of the form `signal (sig, SIG_DFL)` and then the signal handler function is called. If `sig` takes the value `SIGILL` then the default resetting still occurs.

The available signal numbers are as follows:

1	SIGABRT	Abort error
2	SIGFPE	Arithmetic exception
3	SIGILL	Illegal instruction
4	SIGINT	Attention request from user
5	SIGSEGV	Bad memory access
6	SIGSTERM	Termination request
8	SIGIO	Input/output possible
9	SIGURG	Urgent condition on I/O channel
10	SIGPIPE	Write on pipe with no corresponding read
11	SIGSYS	Bad argument to system call
12	SIGALRM	Alarm clock
13	SIGWINCH	Window changed
14	SIGLOST	Resource lost
15	SIGUSR1	User defined signal
16	SIGUSR2	User defined signal
17	SIGUSR3	User defined signal

The default handling and handling at program startup for all signals except **SIGABRT** and **SIGTERM** is no action. For **SIGABRT** the handling depends on **set_abort_action**, and for **SIGTERM** the program is terminated via a call to **exit** with the parameter **EXIT_FAILURE**.

Example:

```
/*
 * To arrange that an interrupt by the user
 * should not go through the default exception
 * handling system, call
 *
 *      signal( SIGILL, SIG_IGN )
 *
 * If the signal is then raised in a
 * later part of the program:
 *
 *      raise( SIGILL )
 *
 * the signal will be ignored.
 */
```

Note: Care should be taken when using **signal** in a concurrent environment. Although simultaneous access to the function is controlled through a semaphore, the registration of a function with the *same* signal number, for example by independent parallel processes overrides the previous value.

See also:

raise

`sin`

Calculates the sine of the argument.

Synopsis:

```
#include <math.h>
double sin(double x);
```

Arguments:

`double x`	A number in radians.

Results:

Returns the sine of **x** in radians.

Errors:

None.

Description:

`sin` calculates the sine of a number (given in radians).

sinf

Calculates the sine of a `float` number.

Synopsis:

```
#include <mathf.h>
float sinf(float x);
```

Arguments:

`float x`	A number in radians.

Results:

Returns the sine of **x** in radians.

Errors:

None.

Description:

`float` form of `sin`.

See also:

`sin`

sinh

Calculates the hyperbolic sine of the argument.

Synopsis:

```
#include <math.h>
double sinh(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the hyperbolic sine of **x** or if a range error occurs returns **HUGE_VAL** (with the same sign as the correct value of the function).

Errors:

A range error will occur if **x** is so large that **sinh** would result in an overflow. In this case **sinh** returns the value **HUGE_VAL** (with the same sign as the correct value of the function) and **errno** is set to **ERANGE**.

Description:

sinh calculates the hyperbolic sine of a number.

`sinhf`

Calculates the hyperbolic sine of a `float` number.

Synopsis:

```
#include <mathf.h>
float sinhf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the hyperbolic sine of `x` or if a range error occurs returns `HUGE_VAL_F` (with the same sign as the correct value of the function).

Errors:

A range error will occur if `x` is so large that `sinhf` would result in an overflow. In this case `sinhf` returns the value `HUGE_VAL_F` (with the same sign as the correct value of the function) and `errno` is set to `ERANGE`.

Description:

`float` form of `sinh`.

See also:

`sinh`

`sprintf`

Writes a formatted string to another string.

Synopsis:

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Arguments:

`char *s`	A string that the output is written to.
`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of characters written, excluding the string terminating character.

Errors:

None.

Description:

`sprintf` writes the string pointed to by `format` to `s`. When `sprintf` encounters a percent sign (%) in the format string, it expands the equivalent argument into the format defined by the tokens after the %.

For the interpretation of the format string see the description of `fprintf`.

Each token acts on the equivalent argument, that is, the third token relates to the third argument after the format string. There must be a single argument for each token. If the token or its equivalent argument is invalid, the behavior is undefined.

To use `sprintf` in the reduced library include the header file `stdiored.h`.

See also:

`fprintf`

`sqrt`

Calculates the square root of the argument.

Synopsis:

```
#include <math.h>
double sqrt(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the non-negative square root of `x` or zero (0.0) on domain error.

Errors:

A domain error will occur if `x` is negative. In this case `errno` is set to EDOM.

Description:

`sqrt` calculates the square root of a number.

sqrtf

Calculates the square root of the `float` argument.

Synopsis:

```
#include <mathf.h>
float sqrtf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the non-negative square root of **x** or zero (0.0F) on domain error.

Errors:

A domain error will occur if **x** is negative. In this case **errno** is set to **EDOM**.

Description:

`float` form of `sqrt`.

See also:

`sqrt`

srand

Sets the seed for pseudo-random numbers generated by `rand`.

Synopsis:

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Arguments:

`unsigned int seed`	The new seed to be used by `rand`.

Results:

No value is returned.

Errors:

None.

Description:

`srand` causes `rand` to be seeded with the value `seed`. Subsequent calls to `rand` will start a new sequence of pseudo-random numbers. If `srand` is called again with the same value of `seed` the random number sequence will be repeated.

If `rand` is called before any calls to `srand` have been made the effect will be the same as if `srand` had been called with a `seed` value of 1.

Care should be taken in parallel environments where concurrent calls to `srand` will reseed all calls to `rand`, not just those in the calling process.

See also:

`rand`

sscanf

Reads formatted data from a string.

Synopsis:

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

Arguments:

`const char *s`	The string the data is read from.
`const char *format`	A format string.
`...`	Subsequent arguments to the format string.

Results:

Returns the number of inputs which have been successfully converted. If a string terminating character occurred before any conversions took place, `sscanf` returns `EOF`.

Errors:

If a string terminating character occurred before any conversions took place, `sscanf` returns `EOF`. Other failures cause termination of the procedure.

Description:

`sscanf` matches the data read from the string `s` to the specifications set out by the format string. See `fscanf` for a description of the format string.

Each token acts on the equivalent argument, that is, the third token relates to the third argument after the format string. There must be a single conversion sequence received for each token. If the token is invalid, the behavior is undefined. Any mismatch between the token format and the data received causes an early termination of `sscanf`.

To use `sscanf` in the reduced library include the header file `stdiored.h`.

See also:

`fscanf`

strcat

Appends one string to another.

Synopsis:

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Arguments:

`char *s1`	A pointer to the string to be extended.
`const char *s2`	A pointer to the string to be appended.

Results:

Returns the unchanged value of `s1`.

Errors:

None.

Description:

`strcat` appends the string pointed to by `s2` (including the null terminating character) onto the end of the string pointed to by `s1`. The first character of `s2` overwrites the null terminating character of `s1`.

The string pointed to be `s1` must be large enough to accept the extra characters from `s2`.

See also:

`strncat`

strchr

Finds the first occurrence of a character in a string.

Synopsis:

```
#include <string.h>
char *strchr(const char *s, int c);
```

Arguments:

`const char *s`	A pointer to the string to be searched.
`int c`	The character to be searched for.

Results:

If the character is found, `strchr` returns a pointer to the matched character. It returns a `NULL` pointer if the character `c` is not in the string.

Errors:

None.

Description:

`strchr` finds the first occurrence of `c` in the string pointed to by `s`. The search includes the null terminating character. `c` is converted to a `char` before the search begins.

`strchr` is side effect free.

Example:

```
char string[12] = "fdakjrejnij";
char *n_pointer;

n_pointer = strchr(string, 'n');
```

See also:

`memchr` `strpbrk` `strrchr`

`strcmp`

Compares two strings.

Synopsis:

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to one of the strings to be compared.
`const char *s2`	A pointer to the other string to be compared.

Results:

Returns the following :

A negative integer if the `s1` string is numerically less than the `s2` string.

A zero value if the two strings are numerically the same.

A positive integer if the `s1` string is numerically greater than the `s2` string.

Errors:

None.

Description:

`strcmp` compares the two strings pointed to by `s1` and `s2`. The comparison is of the numerical values of the ASCII characters.

`strcmp` is side effect free.

See also:

`memcmp` `strcoll` `strncmp`

`strcoll` Compares two strings (transformed according to the program's locale).

Synopsis:

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to one of the strings to be compared.
`const char *s2`	A pointer to the other string to be compared.

Results:

Returns the following :

> A negative integer if the `s1` string is numerically less than the `s2` string.
>
> A zero value if the two strings are numerically the same.
>
> A positive integer if the `s1` string is numerically greater than the `s2` string.

Errors:

None.

Description:

`strcoll` compares the two strings pointed to by `s1` and `s2`. Before comparison takes place the two strings are transformed according to the `LC_COLLATE` category of the program's locale. Since the only permissible locale in the current implementation is "C", `strcoll` is equivalent to `strcmp`.

The string comparison is of the characters' numerical ASCII codes.

`strcoll` is side effect free.

See also:

`memcmp` `strcmp` `strncmp`

strcpy

Copies a string into an array.

Synopsis:

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

Arguments:

`char *s1`	A pointer to the array used as the copy destination.
`const char *s2`	A pointer to the string used as the copy source.

Results:

Returns the unchanged value of `s1`.

Errors:

The behavior of `strcpy` is undefined if the source and destination overlap.

Description:

`strcpy` copies the source string (pointed to by `s2`) into the destination string (pointed to by `s1`). The copy includes the null terminating character. The behavior of `strcpy` is undefined if the source and destination overlap.

A call to `strcpy` will be transformed into a call to `memcpy` provided that:

1 The header file `<string.h>` has been included in the source.

2 The actual argument corresponding to the formal argument `s2` is a string literal.

This call to `memcpy` may subsequently be compiled inline.

See also:

`memcpy` `strncpy`

strcspn Counts the number of characters at the start of a string which do not match any of the characters in another string.

Synopsis:

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be measured.
`const char *s2`	A pointer to the string containing the characters to be checked.

Results:

Returns the length of the unmatched segment.

Errors:

None.

Description:

`strcspn` counts the number of characters at the start of the string pointed to by `s1` which are not in the string pointed to by `s2`. As soon as `strcspn` finds a character present in both strings it stops and returns the number of characters counted.

The null terminating character is not considered to be part of the `s2` string.

`strcspn` is side effect free.

Example:

```
#include <stdio.h>
#include <string.h>

/* Print string up to any numeric characters. */

int main( )
{
   char *dec_string = "1234567890";
   char *given_string = "Hello there 123hello";
   size_t no_chars;

   no_chars = strcspn(given_string, dec_string);
   given_string[no_chars] = '\0';
   puts(given_string);
   /* prints "Hello there" */
}
```

See also:

strspn strtok

strerror

Maps an error number to an error message string.

Synopsis:

```
#include <string.h>
char *strerror(int errnum);
```

Arguments:

`int errnum`	The error number to be converted.

Results:

Returns a pointer to the error message string.

Errors:

None.

Description:

`strerror` generates one of the following error messages according to the value of `errnum`:

Value of `errnum`	Message
`EDOM`	**EDOM - function argument out of range**
`ERANGE`	**ERANGE - function result not representable**
`ESIGNUM`	**ESIGNUM - illegal signal number to signal()**
`EIO`	**EIO - error in low level server I/O**
`EFILPOS`	**EFILPOS - error in file positioning functions**
0	**No error (`errno` = 0)**

If `errnum` is not one of the above values the following error is generated:

Error code <errno> <`errnum`> has no associated message

where: <`errnum`> is the value passed to `strerror`.

Note: Care should be taken when calling `strerror` in a concurrent environment. Calls to the function by independently executing, unsynchronized processes may corrupt the returned error string.

See also:

`perror`

`strftime` Does a formatted conversion of a broken–down time to a string.

Synopsis:

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
                const char *format,
                const struct tm *timeptr);
```

Arguments:

`char *s`	A pointer to the string where the formatted string is written.
`size_t maxsize`	The maximum number of characters to be written into the string.
`const char *format`	A pointer to the format string.
`const struct tm *timeptr`	A pointer to a broken–down time.

Results:

If the number of characters written is less than `maxsize`, `strftime` returns the number of characters written (not including the null terminating character). Otherwise `strftime` returns zero (0).

Errors:

If the number of characters to be written exceeds `maxsize`, `strftime` returns zero, and the contents of the string pointed to by s are undefined.

Description:

`strftime` is used to convert the values in a broken–down time structure according to the demands of a format string, and to write the resulting string to a string. The format string consists of ordinary characters and tokens. Normal characters are written directly to `s`, and tokens are expanded. Tokens are single characters, preceded by the percent character '`%`'.

Token	Meaning	Range
%a	Abbreviated day	(Mon – Sun).
%A	Full day	(Monday – Sunday).
%b	Abbreviated month	(Jan – Dec).
%B	Full month	(January – December).
%c	Date and time	(e.g. Sun Jul 23 11:27:32 1989).
%d	Day of the month as a decimal number.	01 – 31
%H	Hours using twenty-four hour clock.	00 – 23
%	Hours using twelve hour clock.	01 – 12
%j	Day of the year.	001 – 366
%m	Month as a decimal number.	01 – 12
%M	Minutes.	00 – 59
%p	AM or PM.	
%S	Seconds.	00 – 61
%U	Week number, counting Sunday as first day of week one.	00 – 53.
%w	Day of week, counting from Sunday.	0 – 6
%W	Week number, counting Monday as first day	00 – 53.
%x	Date in default format.	(e.g. Sun Jul 23 1989).
%X	Time in default format.	(e.g. 11:27:32).
%y	Year without century.	00 – 99
%Y	Year with century.	e.g. 1989
%Z	Time zone if one exists.	–
%%	'%'.	–

Example:

```
#include <stdio.h>
#include <time.h>

/* Display the day in different ways */

int main( void )
{
   char day_line[300];
   struct tm *bdt;
   time_t current;

   time( &current );
   bdt = localtime( &current );
   strftime (day_line, 300,
            "Different days are %a, %A, %j, %d, %w",
            bdt);
   printf(day_line);
}
```

See also:

`asctime` `ctime` `localtime` `clock` `difftime` `mktime` `time`

strlen

Calculates the length of a string.

Synopsis:

```
#include <string.h>
size_t strlen(const char *s);
```

Arguments:

`const char *s`	A pointer to the string to be measured.

Results:

Returns the length of the string (excluding the null terminating character).

Errors:

None.

Description:

`strlen` counts the number of characters in the string up to, but not including, the null terminating character.

`strlen` is side effect free.

Example:

```
char *string = "String to be measured";
size_t result;

result = strlen(string);


/*
  Gives a result of 21
*/
```

`strncat` Appends one string onto another (up to a maximum number of characters).

Synopsis:

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

Arguments:

`char *s1`	A pointer to the string to be extended.
`const char *s2`	A pointer to the string to be appended.
`size_t n`	The maximum number of characters to be appended.

Results:

Returns the unchanged value of `s1`.

Errors:

None.

Description:

`strncat` copies a maximum of `n` characters from the string pointed to by `s2` onto the end of the string pointed to by `s1`. The first character of `s2` overwrites the null terminating character of `s1`. A null terminating character is appended to the end of the result.

The string pointed to be `s1` must be large enough to accept the extra characters from `s2`.

See also:

`strcat`

strncmp

Compares the first n characters of two strings.

Synopsis:

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Arguments:

`const char *s1`	A pointer to one of the strings to be compared.
`const char *s2`	A pointer to the other string to be compared.
`size_t n`	The maximum number of characters to be compared.

Results:

Returns the following :

A negative integer if the `s1` string is numerically less than the `s2` string.

A zero value if the two strings are numerically the same.

A positive integer if the `s1` string is numerically greater than the `s2` string.

Errors:

None.

Description:

`strncmp` compares up to the first n characters of the strings pointed to by `s1` and `s2`. The comparison is of the numerical values of the ASCII characters.

`strncmp` is side effect free.

Example:

```
/*
  Compares two strings
*/

char string1[50], string2[50];
int result;

strcpy(string1, "Text");
strcpy(string2, "Textual difference");
result = strncmp(string1, string2, 4);

/*
strncmp returns 0
*/
```

See also:

`memcmp` `strcmp` `strcoll` `strncmp`

`strncpy` Copies a string into an array (to a maximum number of characters).

Synopsis:

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Arguments:

`char *s1`	A pointer to the array used as the copy destination.
`const char *s2`	A pointer to the string used as the copy source.
`size_t n`	The maximum number of characters to be copied.

Results:

Returns the unchanged value of `s1`.

Errors:

The behavior of `strncpy` is undefined if the source and destination overlap.

Description:

`strncpy` copies up to n characters from the source string (pointed to by `s2`) into the destination array (pointed to by `s1`). The behavior of `strncpy` is undefined if the source and destination overlap.

If the source string is less than `n` characters long, the extra spaces in the destination array will be filled with null characters.

See also:

`strcpy`

strpbrk

Finds the first character in one string present in another string.

Synopsis:

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be searched.
`const char *s2`	A pointer to the string containing the characters to be searched for.

Results:

Returns a pointer to the first character found in both strings. If none of the characters in the `s2` string occur in the `s1` string, `strpbrk` returns a `NULL` pointer.

Errors:

None.

Description:

`strpbrk` finds the first character in the string pointed to by `s1` which is also contained within the string pointed to by `s2`.

`strpbrk` is side effect free.

Example:

```
/* Return a pointer to the first occurrence of
                              'r', 'c', or 'm', */

#include <stdio.h>
#include <string.h>

int main()
{
  char *string = "The Inmos C Compiler";
  char *result;

  result = strpbrk(string, "rcm");
  printf("%s\n", result);
}

/* result = "mos C Compiler" */
```

See also:

`strchr` `strrchr`

strrchr

Finds the last occurrence of a given character in a string.

Synopsis:

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Arguments:

`const char *s`	A pointer to the string to be searched.
`int c`	The character to be searched for.

Results:

Returns a pointer to the last occurrence of the character.

Errors:

Returns `NULL` if `c` does not occur in the string.

Description:

`strchr` finds the last occurrence of `c` in the string pointed to by `s`. The search includes the null terminating character. `c` is converted to a `char` before the search begins.

`strrchr` is side effect free.

Example:

```
/*  Finds the last time that '9' occurs in a string */

#include <stdio.h>
#include <string.h>

int main()
{
   char *string = "9 times 9 = 81";
   char *result;

   result = strrchr(string, '9');
   printf("%s\n", result );
   /* result = "9 = 81" */
}
```

See also:

`strpbrk strchr`

`strspn` Counts the number of characters at the start of a string which are also in another string.

Synopsis:

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be measured.
`const char *s2`	A pointer to the string containing the characters to be searched for.

Results:

Returns the length of the matched segment.

Errors:

None.

Description:

`strspn` counts the characters at the start of the string pointed to by `s1` which are also present in the string pointed to by `s2`. As soon as `strspn` finds a character in the first string which is not present in the second string, it stops and returns the number of characters counted.

`strspn` is side effect free.

Example:

```
#include <string.h>
#include <stdio.h>

int main( void )
{
   char *string = "cracking";
   size_t result;

   result = strspn(string, "arc");
   printf("%d\n", result );
   /* 4 in this case */
}
```

See also:

`strcspn` `strtok`

strstr

Finds the first occurrence of one string in another.

Synopsis:

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Arguments:

`const char *s1`	A pointer to the string to be searched.
`const char *s2`	A pointer to the string to be searched for.

Results:

Returns a pointer to the string in `s1`, if found. If `s2` points to a string of zero length, the function returns `s1`. If the `s2` string does not occur within the `s1` string the function returns `NULL`.

Errors:

None.

Description:

`strstr` finds the first occurrence of the `s2` string (excluding the null terminating character) in the `s1` string.

`strstr` is side effect free.

Example:

```
#include <string.h>
#include <stdio.h>

int main()

{
   char *string1 = "string to be searched";
   char *string2 = "sea";

   printf("%s\n", strstr(string1, string2));
 }

/* Displays "searched" */
```

See also:

`strpbrk` `strspn`

strtod Converts the initial part of a string to a double and saves a pointer to the rest of the string.

Synopsis:

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.
`char **endptr`	A pointer to the object which is to receive a pointer to the rest of the string.

Results:

Returns the converted value if the conversion is successful. If no conversion is possible or underflow occurs, `strtod` returns zero. `HUGE_VAL` is returned if overflow occurs.

Errors:

If the result would cause overflow, `errno` is set to `ERANGE` and the value `HUGE_VAL` is returned. If the result would cause underflow, `errno` is set to `ERANGE` and zero is returned.

Description:

`strtod` converts the initial part of the string pointed to by `nptr` to a number represented as a `double`. `strtod` expects the string to consist of the following sequence:

1. Leading white space (optional).
2. A plus or minus sign (optional).
3. A sequence of decimal digits, which may contain a decimal point.
4. An exponent (optional) consisting of an 'E' or 'e' followed by an optional sign and a string of decimal digits.
5. One or more unrecognized characters (including the null string terminating character).

`strtod` ignores the leading white space, and converts all the recognized characters. If there is no decimal point or exponent part in the string, a decimal point is assumed after the last digit in the string.

The string is invalid if the first non-space character in the string is not one of the following characters:

+ – . 0 1 2 3 4 5 6 7 8 9

If `endptr` is not `NULL`, and the conversion took place, a pointer to the unrecognized part of the string is stored in the object pointed to by `endptr`. If conversion did not take place, the location is set to the value of `nptr`.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
 char *array = "97824.3E+4Goodbye";
 char *number_end;
 double x;

 x = strtod(array, &number_end);
 printf("strtod gives %f\n", x);
 printf("Number ended at %s\n", number_end);
}

/*
Prints:
  strtod gives 978243000.000000
  Number ended at Goodbye
*/
```

See also:

`atof atoi atol strtol`

strtok

Converts a delimited string into a series of string tokens.

Synopsis:

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Arguments:

`char *s1`	A pointer to the string to be broken up or a `NULL` pointer.
`const char *s2`	A pointer to the delimiter string.

Results:

Returns a pointer to the first character of a token. A `NULL` pointer is returned if no token is found.

Errors:

None.

Description:

`strtok` is used to break up the string pointed to by `s1` into separate strings. The input string is assumed to consist of a series of tokens separated from one another by one of the characters in the delimiter string pointed to by `s2`.

When `strtok` is first called, each character in the string pointed to by `s1` is checked to see if it is also present in the delimiting string pointed to by `s2`. `strtok` recognizes the first character which is not in the delimiter string as the start of the first token. If no such character is found it is assumed that there are no tokens in `s1`, and `strtok` returns a `NULL` pointer.

Having found the start of a token, the `strtok` function searches for the end of the token, represented by a character present in the delimiting string. If such a character is found, it is overwritten with the null terminating character and `strtok` saves a pointer to the following character for use in a subsequent call. If no such character is found the token extends to the end of the string. `strtok` returns a pointer to the first character of the token.

The next token from the string is extracted by calling `strtok` with a `NULL` pointer as the first argument. This causes `strtok` to use the pointer saved during the previous execution.

Note: Care should be taken when calling `strtok` in a concurrent environment. Calls to the function by independently executing, unsynchronized processes change the pointer saved internally by `strtok` in an unpredictable way and may produce unexpected results.

Example:

```
#include <stdio.h>
#include <string.h>

int main()
{
   char *string = "String^of things,to,,be^split";
   char *token;

   token = strtok(string, "^ ,");
   while (token != NULL)
   {
      printf("Token found = %s\n", token);
      token = strtok(NULL, "^ ,");
   }
}
   /*
    *  Gives the output:
    *      Token found = String
    *      Token found = of
    *      Token found = things
    *      Token found = to
    *      Token found = be
    *      Token found = split
    */
```

`strtol`

Converts the initial part of a string to a `long int` and saves a pointer to the rest of the string.

Synopsis:

```
#include <stdlib.h>
long int strtol(const char *nptr,
                char **endptr, int base);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.
`char **endptr`	A pointer to the object which is to receive a pointer to the rest of the string.
`int base`	The radix representation of the integer string to be converted.

Results:

Returns the converted value if the conversion is successful. If no conversion is possible, `strtol` returns zero. If the result would cause overflow the value `LONG_MAX` or `LONG_MIN` is returned (depending on the sign of the result).

Errors:

If the result would cause overflow the value `LONG_MAX` or `LONG_MIN` is returned (depending on the sign of the result), and `errno` is set to `ERANGE`.

Description:

`strtol` converts the initial part of the string pointed to by `nptr` to a long integer. `strtol` expects the string to consist of the following:

1. Leading white space (optional).
2. A plus or minus sign (optional).
3. An octal '0' or hexadecimal '0x' or '0X' prefix (optional).
4. A sequence of digits within the range of the appropriate base. The letters 'a' to 'z', and 'A' to 'Z' may be used to represent the values 10 to 35. For example, if base is set to 18, the characters for the values 0 to 17 ('0' to '9' and 'a' to 'h' or 'A' to 'H') are permitted.
5. One or more unrecognized characters (including the null string terminating character).

`strtol` ignores leading blanks, and converts all recognized characters. The string is invalid if the first non-space character in the string is not a sign, an octal or hexadecimal prefix, or one of the permitted characters.

If `endptr` is not `NULL`, and the conversion took place, a pointer to the rest of the string is stored in the location pointed to by `endptr`. If no conversion was possible, and `endptr` is not `NULL`, the value of `nptr` is stored in that location.

Example:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   char *array = "12345abcGoodbye";
   char *number_end;
   int base;
   long l;

   for(base = 2; base < 12; base += 3)
   {
      l = strtol(array, &number_end, base);
      printf("base = %d, strtol gives %ld\n",
              base, l);
      printf("Number ended at %s\n\n", number_end);
   }
}


/* Prints  base = 2, strtol gives 1
 *         Number ended at 2345abcGoodbye
 *
 *
 *         base = 5, strtol gives 194
 *         Number ended at 5abcGoodbye
 *
 *
 *         base = 8, strtol gives 5349
 *         Number ended at abcGoodbye
 *
 *
 *         base = 11, strtol gives 194875
 *         Number ended at bcGoodbye
 */
```

See also:

`atoi` `atol` `strtod` `strtoul`

strtoul

Converts the initial part of a string to an `unsigned long int` and saves a pointer to the rest of the string.

Synopsis:

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr,
                          char **endptr, int base);
```

Arguments:

`const char *nptr`	A pointer to the string to be converted.
`char **endptr`	A pointer to the location which is to receive a pointer to the rest of the string.
`int base`	The radix representation of the integer string to be converted.

Results:

Returns the converted value if the conversion is successful. If no conversion is possible, `strtoul` returns zero. If the result would cause overflow the value `ULONG_MAX` is returned.

Errors:

If the result would cause overflow the value `ULONG_MAX` is returned and `errno` is set to `ERANGE`.

Description:

`strtoul` converts the initial part of the string pointed to by `nptr` to an `unsigned long int`. `strtoul` expects the string to consist of the following:

1. Leading white space (optional).
2. An octal '0' or hexadecimal '0x' or '0X' prefix (optional).
3. A sequence of digits within the range of the appropriate base. The letters 'a' to 'z', and 'A' to 'Z' may be used to represent the values 10 to 35. For example, if base is set to 18, the characters for the values 0 to 17 ('0' to '9' and 'a' to 'h' or 'A' to 'H') are permitted.
4. One or more unrecognized characters (including the null string terminating character).

`strtoul` ignores the leading white space, and converts all the recognized characters. The string is invalid if the first non-space character in the string is not an octal or hexadecimal prefix, or one of the permitted characters (signs are not permitted). If `endptr` is not `NULL`, and the conversion took place, a pointer to the rest of the string is stored in the location pointed to by `endptr`. If no conversion was possible, and `endptr` is not `NULL`, the value of `nptr` is stored in that location.

See also:

`atoi` `atol` `strtod` `strtol`

strxfrm Transforms a string according to the locale and copies it into an array (up to a maximum number of characters).

Synopsis:

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

Arguments:

`char *s1`	A pointer to the array used as the copy destination.
`const char *s2`	A pointer to the string used as the copy source.
`size_t n`	The maximum number of characters to be copied.

Results:

`strxfrm` returns the length of the transformed string.

Errors:

None.

Description:

`strxfrm` copies up to `n` characters from the source string (pointed to by `s2`) into the destination array (pointed to by `s1`), after transforming the source string according to the program's locale. Since the only permissible locale is "C", `strxfrm` is equivalent to `strncpy`. The behavior of `strxfrm` is undefined if the source and destination overlap.

If `n` is zero then `s1` may be a `NULL` pointer, in which case `strxfrm` returns the number of characters in the transformed string.

If the source string is less than `n` characters long, the extra spaces in the destination array will be filled with null characters.

Because "C" is the only locale supported by this implementation, the behavior of `strxfrm` resembles that of a less efficient `strncpy`.

See also:

`strncpy`

system

Passes a command to host operating system for execution.

Synopsis:

```
#include <stdlib.h>
int system(const char *string);
```

Arguments:

`const char *string`	A pointer to the string to be passed to the host.

Results:

If `string` is a `NULL` pointer, `system` returns a non–zero value if a command processor exists or zero otherwise. If `string` is not a `NULL` pointer `system` returns the return value of the command which is host-defined.

Errors:

None.

Description:

`system` passes the string pointed to by `string` to the host environment to be executed by a command processor. `string` can be any command defined on the host system, but should not be a command which causes the transputer to be rebooted as this would overwrite the program executing the call.

If `string` is a `NULL` pointer the call to `system` is an enquiry as to whether there is a command processor.

The mode of execution of the command is defined by the host system.

Use of `system` in the reduced library always returns 0 as there is no command processor available in this case.

Note: Issuing a command that boots a program onto the transputer running the current program causes the program to fail by overwriting the memory.

tan

Calculates the tangent of the argument.

Synopsis:

```
#include <math.h>
double tan(double x);
```

Arguments:

`double x`	A number in radians.

Results:

Returns the tangent of `x` in radians.

Errors:

None.

Description:

`tan` calculates the tangent of a number (given in radians).

See also:

`tanf`

tanf

Calculates the tangent of a `float` number.

Synopsis:

```
#include <mathf.h>
float tanf(float x);
```

Arguments:

`float x`	A number in radians.

Results:

Returns the tangent of **x**.

Errors:

None.

Description:

`float` form of `tan`.

See also:

`tan`

tanh

Calculates the hyperbolic tangent of the argument.

Synopsis:

```
#include <math.h>
double tanh(double x);
```

Arguments:

`double x`	A number.

Results:

Returns the hyperbolic tangent of `x`.

Errors:

None.

Description:

`tanh` calculates the hyperbolic tangent of a number.

See also:

`tanhf`

`tanhf` Calculates the hyperbolic tangent of a `float` number.

Synopsis:

```
#include <mathf.h>
float tanhf(float x);
```

Arguments:

`float x`	A number.

Results:

Returns the hyperbolic tangent of `x`.

Errors:

None.

Description:

`float` form of `tanh`.

See also:

`tanh`.

`time`

Reads the current time.

Synopsis:

```
#include <time.h>
time_t time(time_t *timer);
```

Arguments:

`time_t *timer`	A pointer to an object where the current time can be stored.

Results:

Returns the value of the current time. If the current time is not available, `time` returns –1, cast to `time_t`.

Errors:

`time` returns (`time_t`)–1, if the current time is not available.

Description:

`time` returns the closest possible approximation to the current time, and loads it into the location pointed to by `timer`, unless `timer` is `NULL`.

`time` always returns –1 in the reduced library since there is no access to the current time in this case.

See also:

`asctime` `ctime` `localtime` `strftime` `clock` `difftime` `mktime`

`tmpfile`

Creates a temporary binary file.

Synopsis:

```
#include <stdio.h>
FILE *tmpfile(void);
```

Arguments:

None.

Results:

Returns a pointer to the newly created file stream, or a `NULL` pointer if the file could not be created.

Errors:

Returns a `NULL` pointer if the file cannot be created.

Description:

`tmpfile` attempts to create a temporary binary file in the *current* directory. If the file is successfully created it is opened for update, that is, in mode "**wb+**". The file will automatically be removed when the program terminates or the temporary file is explicitly closed.

`tmpfile` is not included in the reduced library.

See also:

`tmpnam`

tmpnam

Creates a unique filename.

Synopsis:

```
#include <stdio.h>
char *tmpnam(char *s);
```

Arguments:

`char *s`	A pointer to the destination string for the filename.

Results:

If `s` is a `NULL` pointer, `tmpnam` returns a pointer to an internal object containing the new filename. Otherwise the new filename is put in the string pointed to by `s`, and `tmpnam` returns the unchanged value `s`. In this case `s` must point to an array of at least `L_tmpnam` characters.

Errors:

The effect of calling `tmpnam` more than `TMP_MAX` times is undefined.

Description:

`tmpnam` creates a unique filename (that is, one which does not match any existing filename) in the *current* directory. A different string is created each time `tmpnam` is called. `tmpnam` may be called up to `TMP_MAX` times.

Note: Care should be taken when calling `tmpnam` in a concurrent environment. Calls to the function by independently executing, unsynchronized processes may corrupt the returned file pointer.

`tmpnam` is not included in the reduced library.

See also:

`tmpfile`

to_host_link

Retrieve the channel going to the host.

Synopsis:

```
#include <hostlink.h>
Channel* to_host_link( void )
```

Arguments:

None.

Results:

Returns a pointer to the channel going to the host.

Errors:

None.

Description:

`to_host_link` retrieves the channel going to the host.

Note: that the link over which communication with the host occurs need not necessarily be the same link as the one from which the transputer was booted.

This function is intended for use with dynamic code loading; care should be taken if it is used elsewhere.

`to_host_link` is not in the reduced library.

See also:

`from_host_link` `get_bootlink_channels`

to86

Transfers transputer memory to the host. MS-DOS only.

Synopsis:

```
#include <dos.h>
int to86(int len, char *here, pcpointer there);
```

Arguments:

`int len`	The number of bytes of transputer memory to be transferred.
`char *here`	A pointer to the transputer memory block.
`pcpointer there`	A pointer to the host memory block.

Results:

Returns the actual number of bytes transferred.

Errors:

Returns the number of bytes transferred until the error occurred and sets `errno` to the value `EDOS`. Any attempt to use `to86` on operating systems other than MS-DOS also sets `errno` to `EDOS`. Failure of the function may also generate the following server error message:

```
[Encountered unknown primary tag (50)]
```

Description:

`to86` transfers `len` bytes of transputer memory starting at `here` to a corresponding block starting at `there` in host memory. The function returns the number of bytes actually transferred. The host memory block used will normally have been previously allocated by a call to `alloc86`.

`to86` is not included in the reduced library.

See also:

`from86` `alloc86`

`tolower` Converts upper-case letter to its lower-case equivalent.

Synopsis:

```
#include <ctype.h>
int tolower(int c);
```

Arguments:

`int c`	The character to be converted.

Results:

Returns the lower-case equivalent of the given character. If the given character is not an upper-case letter it is returned unchanged.

Errors:

None.

Description:

`tolower` converts the character `c` to its lower-case equivalent. If `c` is not an upper-case letter it is not converted. Valid upper-case letters are ASCII characters in the range 'A' to 'Z'.

`tolower` is side effect free.

See also:

`toupper`

`toupper` Converts lower-case letter to its upper-case equivalent.

Synopsis:

```
#include <ctype.h>
int toupper(int c);
```

Arguments:

`int c`	The character to be converted.

Results:

Returns the upper-case equivalent of the given character. If the given character is not a lower-case letter it is returned unchanged.

Errors:

None.

Description:

`toupper` converts the character `c` to its upper-case equivalent. If `c` is not a lower-case letter, it is not converted. Valid lower-case letters are ASCII characters in the range 'a' to 'z'.

`toupper` is side effect free.

See also:

`tolower`

ungetc

Pushes a character back onto a file stream.

Synopsis:

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Arguments:

`int c`	The character to be pushed back.
`FILE *stream`	A pointer to a file stream.

Results:

Returns the pushed back character if successful, or `EOF` if unsuccessful.

Errors:

Returns `EOF` if unsuccessful.

Description:

`ungetc` converts `c` to an unsigned char and pushes it back onto the input stream pointed to by `stream`. The next use of any of the `getc` family of functions will return `c` unless a repositioning function has been called in between (`fflush`, `fseek`, `rewind` or `fsetpos`).

If `ungetc` is called more than once on the same stream without the file stream being read in the meantime, the operation will fail.

`ungetc` is not included in the reduced library.

Example:

```
#include <stdio.h>
#include <ctype.h>

/*
 * Function to read an integer.
 * Leaves the next character to be read
 * as the one immediately after the number.
 */

int get_number()
{
   int dec = 0;
   int ch;

   while(isdigit(ch = getc(stdin)))
      dec = dec * 10 + ch - '0';
   ungetc(ch,stdin);
   return(dec);
}
```

See also:

`fflush` `getc`

unlink

Deletes a file.

Synopsis:

```
#include <iocntrl.h>
int unlink(char *name);
```

Arguments:

`char *name`	The name of the file to be deleted.

Results:

Returns 0 if successful or –1 on error.

Errors:

If an error occurs `unlink` sets `errno` to the value `EIO`.

Description:

`unlink` deletes the file by removing the filename from the host file system. It is equivalent to the ANSI library function `remove`.

`unlink` is not included in the reduced library.

See also:

`remove`

`va_arg` Accesses a variable number of arguments in a function definition.

Synopsis:

```
#include <stdarg.h>
type va_arg(va_list ap, type );
```

Arguments:

`va_list ap`	A pointer to a variable argument list.
type	Any C type.

Results:

`va_arg` returns the value of the next argument in the variable argument list which is assumed to have type *type*.

Errors:

If the type specified in `va_arg` disagrees with the type of the next argument in the argument list the effects are undefined.

If there is no next argument in the list, or the next argument is a `register` variable, an array type, or a function, the behavior is undefined. If the next argument is of a type incompatible with the variable type after default promotions (see section 4.2.3), the following compile time error is generated:

Serious–icc–<filename>(linenumber) – illegal type used with va_arg

Description:

Each invocation of `va_arg` extracts a single argument value from a variable length argument list. `va_arg` must have been initialized by a previous call to `va_start`. The final use of `va_arg` should be followed by a call to `va_end` to ensure a clean termination.

`va_arg` can only be used when there is at least one fixed argument in the variable length argument list.

`va_arg` is implemented as a macro.

Example:

```
#include <stdio.h>
#include <stdarg.h>

/*
 * Sends the number of strings defined in
   number_of_strings,
 * and given in the parameter list,
   to standard output.
 */

void var_string_print( int number_of_strings, ...)
{
   va_list ap;

   va_start(ap, number_of_strings);
   while (number_of_strings-- > 0)
      puts(va_arg(ap, char *));
   va_end(ap);
}

int main()
{
   var_string_print( 2, "Hello", "World" );

   /*
    * Displays:
    *           Hello
    *           World
    */
}
```

See also:

`va_end` `va_start` `vfprintf` `vprintf` `vsprintf`

`va_end`

Cleans up after accessing variable arguments.

Synopsis:

```
#include <stdarg.h>
void va_end(va_list ap);
```

Arguments:

`va_list ap`	A pointer to a variable argument list.

Results:

No value is returned.

Errors:

None.

Description:

`va_end` tidies up after the use of `va_start` and `va_arg`. If it is not used, abnormal function return may occur.

`va_end` can only be used when there is at least one fixed argument in the variable length argument list.

`va_end` is implemented as a macro.

See also:

`va_arg` `va_start`

`va_start` Initializes a pointer to a variable number of function arguments in a function definition.

Synopsis:

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Arguments:

`va_list ap`	A pointer to a variable argument list.
`parmN`	The name of the last fixed argument in the function definition.

Results:

No value is returned.

Errors:

If `parmN` is declared as storage class `register`, as a function or array, or as a type that is incompatible with the type of the variable after argument promotion, the behavior is undefined.

Description:

`va_start` is used in conjunction with `va_arg` and `va_end`. It initializes `ap` for further use by `va_arg`. `va_start` can only be used when there is at least one fixed argument in the variable length argument list.

`va_start` is implemented as a macro.

See also:

`va_arg` `va_end`

vfprintf

An alternative form of `fprintf`. Which accepts a variable argument list in `va_list` form.

Synopsis:

```
#include <stdio.h>
int vfprintf(FILE *stream, const char *format,
             va_list arg);
```

Arguments:

`FILE *stream`	An output file stream.
`const char *format`	A format string.
`va_list arg`	A pointer to a variable argument list, initialized by `va_start`.

Results:

Returns the number of characters written, or a negative value if an output error occurs.

Errors:

Returns a negative value if an output error occurs.

Description:

`vfprintf` is a form of `fprintf` in which the variable arguments are replaced by a pointer to a variable argument list. `vfprintf` should be preceded by a call to `va_start`, and followed by a call to `va_end`.

`vfprintf` is not included in the reduced library.

See `fprintf` for a description of the format string.

Example:

```
#include <stdio.h>
#include <stdarg.h>

void write_file(FILE *stream, char *format, ... )
{
   va_list apo;

   va_start(apo,format);
   fputs("WRITE FILE TEXT ", stream);
   vfprintf(stream, format, apo);
   va_end(apo);
}

int main()
{
 FILE *stream;
 int a = 10;
 char *b = "string";

 stream = fopen("newfile","w");
 if (stream == NULL)
   printf("Error opening file\n");
 else
 {
   write_file(stream, "%d, %s", a, b);
   fclose(stream);
 }
}

/* writes the string "WRITE_FILE TEXT 10, string"
   to the file newfile */
```

See also:

fprintf va_arg va_end va_start vprintf vsprintf

vprintf

An alternative form of `printf`. Which accepts a variable argument list in the form of a `va_list`.

Synopsis:

```
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Arguments:

`const char *format`	A format string
`va_list arg`	A pointer to a variable argument list, initialized by `va_start`.

Results:

Returns the number of characters written, or a negative value if an output error occurred.

Errors:

`vprintf` returns a negative value if an output error occurs.

Description:

`vprintf` is a form of `printf` in which the variable arguments are replaced by a pointer to a variable argument list. `vprintf` should be preceded by a call to `va_start`, and followed by a call to `va_end`.

`vprintf` is not included in the reduced library.

See `fprintf` for a description of the format string.

See also:

`printf` `va_arg` `va_start` `va_end` `vfprintf` `vsprintf`

`vsprintf`

An alternative form of `sprintf`. Which accepts a variable argument list in the form of a `va_list`.

Synopsis:

```
#include <stdio.h>
int vsprintf(char *s, const char *format,
             va_list arg);
```

Arguments:

`const char *s`	The string to which the formatted string is written.
`const char *format`	A format string.
`va_list arg`	A pointer to a variable argument list, initialized by `va_start`.

Results:

Returns the number of characters written.

Errors:

None.

Description:

`vsprintf` is a form of `sprintf` in which the variable arguments are replaced by a pointer to a variable argument list. `vsprintf` should be preceded by a call to `va_start`, and followed by a call to `va_end`.

To use `vsprintf` in the reduced library include the header file `stdiored.h`.

See `fprintf` for a description of the format string.

See also:

`sprintf` `vfprintf` `va_arg` `va_end` `va_start`

wcstombs

Converts wchar_t sequence to multibyte sequence.

Synopsis:

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

Arguments:

char *s	Pointer to the start of the array where the results will be stored.
const wchar_t *pwcs	Pointer to the start of the wide character sequence to be converted.
size_t n	The maximum number of bytes to be stored.

Results:

wcstombs returns the number of bytes modified, not including any terminating zero codes or –1 on error.

Errors:

If an invalid code is encountered wcstombs returns (size_t)-1.

Description:

wcstombs converts a sequence of wide-character codes into a sequence of multibyte characters. It acts like the wctomb function but takes as input an array of codes and returns an array of characters.

Not more than n bytes are written into s. If the initial and receiving objects overlap, the behavior is undefined.

Storage of a null character terminates the function.

wctomb

Converts type wchar_t to multibyte character.

Synopsis:

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

Arguments:

char *s	Pointer to the array object that will receive the multibyte character.
wchar_t wchar	Code of wide character to be converted.

Results:

If s is not a NULL pointer, wctomb returns the number of bytes in the multibyte character corresponding to wchar.

If s is a NULL pointer, wctomb returns zero. wctomb returns –1 on error.

The value returned cannot be greater than n or the value of MB_CUR_MAX.

Errors:

If wchar does not correspond to a valid multibyte character wctomb returns –1.

Description:

wctomb converts a wide-character code to a multibyte character to and stores the result in the array pointed to by s. At most MB_CUR_MAX characters are stored.

write

Writes bytes to a file. File handling primitive.

Synopsis:

```
#include <iocntrl.h>
int write(int fd, char *buf, int n);
```

Arguments:

`int fd`	A file descriptor.
`char *buf`	A pointer to a buffer from which the bytes are obtained.
`int n`	The maximum number of bytes that `write` will attempt to output.

Results:

Returns the number of bytes written or –1 on error.

Errors:

If an error occurs `write` sets `errno` to the value `EIO`.

Description:

`write` writes `n` bytes from the buffer pointed to by `buf` to the file specified by `fd`. If `n` is zero or negative no output occurs.

`write` is not included in the reduced library.

See also:

`read`

3 Modifying the runtime startup system

This chapter describes a version of the C runtime startup code, supplied in source form, which may be modified by users. It enables the runtime startup code to be tailored for a particular application, removing procedures which are not required and thereby reducing the runtime overhead. The supplied source code is fully commented and should be read in conjunction with this document. **Note:** the supplied source is only applicable to this release of the toolset (Dx314) and cannot be guaranteed to work with future releases.

Only users who are knowledgeable about the implementation of ANSI C and are familiar with the construction of C runtime systems in general, should attempt to modify this code. It is intended as a means of tuning system performance and is aimed at experienced users.

This chapter covers the following topics:

- A description of the runtime startup code and how it is built.
- Recompiling and linking modified runtime source code.
- An example of a modified runtime system together with the procedure to build it.

The degree to which the supplied startup code is modified is at the user's discretion and it is their responsibility to ensure that any procedures removed are truly redundant to the application. A single library entry or whole sections of the startup code may be removed e.g. the code to set up heap or stack checking or to initialize the input/output (I/O) system.

3.1 Introduction

The runtime system supplied as source code and which is described here, is designed for use in configured systems only. (A separate startup system is provided without source code for non–configured programs). The configuration system considers the C system entered via the runtime startup as a `process`. Thus, within this chapter the current invocation of a C main program is referred to as the 'current process'.

The source which is shipped is the same as that used to create the runtime startup system for configured systems, which is supplied as part of the standard library. The code produces the `C.ENTRYD` and `C.ENTRYD.RC` entry points used via `cstartup.lnk` or `cstartrd.lnk` for linking modules prior to configuration.

3.2 Overview of system

The code as supplied can be compiled in two ways: one for the full library; and one for the reduced library. The reduced version is a subset of the full system, having no host server I/O support.

The runtime startup code consists of two stages using the routines in the files: `centryd1.c` and `centryd2.c`. The first routine is called by the configuration system. This in turn calls the second routine which then calls `main()`. See figure 3.1.

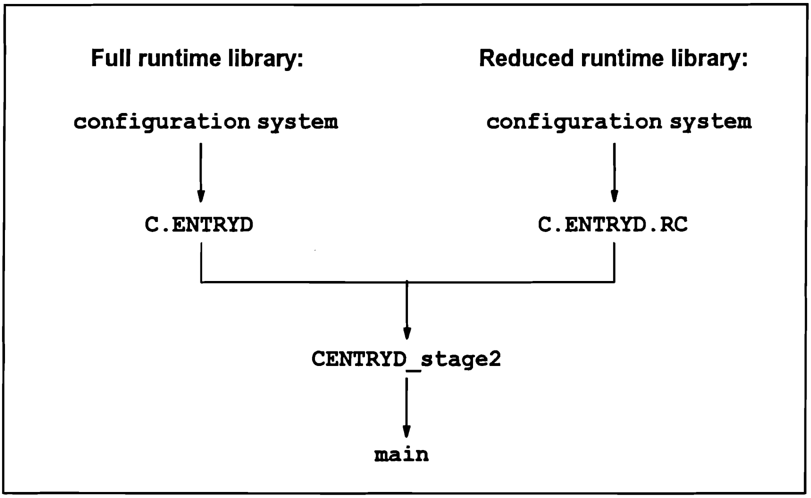

Figure 3.1 Runtime startup system calling sequence

`C.ENTRYD` and `C.ENTRYD.RC` are the entry points to stage 1 of the startup code for the full and reduced systems respectively. `CENTRYD_stage2` is the common entry point to stage 2 of the startup system and is used for both versions.

Both `centryd1.c` and `centryd2.c` use pre–processor conditional compilation directives which enable full and reduced versions of the runtime startup code to be generated from a common source. The symbol '`REDUCED`' may be defined at compile time, in order to build the reduced version of the library, see section 3.9.

If the full library is used and communication with the server is required then the first two configuration parameters to the process must be channels. The first being the channel from the server; the second being the channel to the server.

The actions performed by the supplied runtime startup code are shown in figure 3.2.

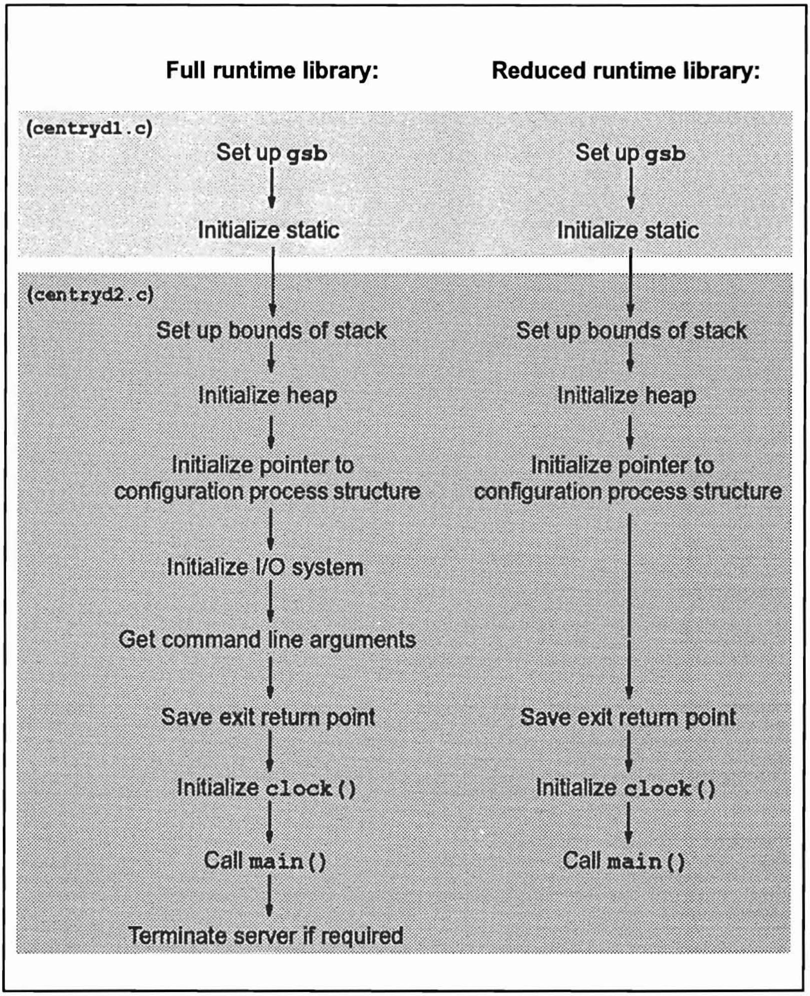

Figure 3.2 Actions performed by runtime startup code

3.3 The gsb and use of the `IMS_nolink` pragma

All C functions find the static area by means of a hidden first parameter, the global static base (`gsb`), which is the address of the base of the static area. This parameter is passed implicitly to all C functions at the front of the parameter list. User parameters follow the `gsb` directly.

When a function calls another function it passes the `gsb` that it received (as its hidden first parameter) as the first parameter to the called function. So, the C compiler automatically adds the `gsb` to the front of a parameter list when making a call. Similarly the called function has code added by the compiler which picks up the `gsb`. This parameter is therefore, completely invisible to the user.

The passing of the `gsb` can be disabled by declaring the function to be called as a `nolink` function using the `IMS_nolink` pragma. A function can also be instructed not to expect a `gsb` by declaring the function as `nolink` in the file in which it is defined.

3.4 Interface to runtime startup code

The runtime system is selected by the user via the linker indirect file specified at link time i.e. `cstartup.lnk` or `cstartrd.lnk`. Configuration data is then passed to the runtime startup code during configuration.

The full runtime system has the following interface:

```
void CENTRYD(struct Conf_Process *pdata);
```

This is the prototype for the full system. The reduced system has the same format but a different name i.e. `CENTRYD_RC`. **Note:** the name is translated to an occam style name including a dot e.g.

```
CENTRYD    becomes C.ENTRYD
CENTRYD_RC becomes C.ENTRYD.RC
```

In addition the configurer expects an occam style descriptor; the C compiler pragma `IMS_descriptor` is used for this purpose.

The descriptor defines the workspace and vector space requirements of the runtime startup code. The vector space requirement is zero as C does not use vector space. A workspace requirement of 5 words is defined. This is in keeping with occam which automatically specifies the workspace for each routine it compiles. Five words is a somewhat arbitrary amount to specify but is derived from the following:

- 3 words to cover the transputers below workspace requirement. 3 is a conservative estimate as only 2 words are required for the current range of transputers.
- 1 word to cover the amount by which `C.ENTRYD` and `C.ENTRYD.RC` adjust the workspace.
- 1 word of leeway.

This amount of workspace is generally not required as the startup code could just as easily reside in the user specified stack space. However, if the workspace requirement of C.ENTRYD is specified as zero and the user makes a mistake and specifies a stack space that is extremely small, e.g. 1 or 2 words, then there would not be enough room to accommodate even the below workspace requirements of the call to C.ENTRYD. The allocation of the 5 words of workspace ensures that the transputer can at least set up its process chains correctly.

Since the function is called as if it were occam, a gsb is not passed, so the function is declared as nolink before it is defined. Thus it will not expect a hidden gsb parameter.

The single parameter passed in to the function is a pointer to the configuration process structure for the current process. This structure contains the following information used by the runtime startup code:

- Address of the start of the static area.
- Size of the static area in bytes.
- Address of the start of the heap area.
- Size of the heap area in bytes.
- Address of the origin of the stack area.
- Size of the stack area in bytes.
- The configuration parameter data. Used in the startup code for the full runtime system to obtain the channels from and to the server. It is also used if the user makes a call to get_param().

The above details are set up by the configurer according to the information supplied by the user in a configuration description (.cfs) file.

The internal details of the structure are not important to this description and cannot be guaranteed to stay the same in future. Accesses to the relevant parts of the structure can be found in the source code.

3.5 Details of stage 1 of the runtime startup code

Stage 1 of the runtime startup code is responsible for initializing the static area and calling the second stage of the runtime startup in such a way that the hidden static base parameter, the gsb, is set up.

Stage 1 of the runtime startup code can be found in the source file centryd1.c.

3.5.1 Initialize static

The first job of stage 1 is to initialize the static area by calling the routine initialise_static. Before this is done no accesses to static data or external variables may be made.

Stage 1 of the runtime startup is declared as **nolink** (see section 3.4) and therefore a valid **gsb** is not obtained. Furthermore **initialise_static** cannot be called as if it were a normal C function (because it would expect a **gsb**). In order for **initialise_static** to work correctly it must be passed a **gsb** explicitly.

To achieve this **initialise_static** is declared as **nolink** to the stage 1 runtime startup and the address of the base of static is passed as an extra parameter at the start of the parameter list. The definition of **initialise_static** in **istatic.c**, (see section 3.9) is not declared as **nolink** and so it picks up the passed first parameter as if it were the hidden **gsb**.

Apart from the **gsb**, **initialise_static** takes a pointer to the base of the static area plus two size values. The first is the static size required, the second is the amount of space available. In the supplied source these two sizes are the same.

In some cases it is possible that **initialise_static** could be called to initialize an area of memory which is larger or smaller than the required size, e.g. setting up a static area using the **init.static** routine from the occam library **callc.lib**.

If the area is too small then the routine returns the value 1. This error does not occur in the source (as supplied), and is therefore not checked for. If any modifications are made which would mean that the required static size is different to the size of static area provided then the return value of **initialise_static** should be checked. If an error is detected the only safe course of action is to halt the processor e.g.

```
if (initialise_static(...))
  halt_processor();
```

This is because no static has been set up and so no error messages can be printed, neither can any library function like **abort** be called as they depend on static data. More details about static initialization can be found in section 3.8.

3.5.2 Call stage 2 startup code and set up gsb

Having set up the static area, the second stage of the runtime startup is called. It is important to ensure that the correct value of **gsb** is propagated through the program. This is achieved by declaring the call to stage 2 as **nolink** while declaring its definition as normal (the same as for **initialise_static**) and passing the address of the static area explicitly as the first parameter.

Stage 2 picks this up as if it were the hidden **gsb** and subsequently passes it as a hidden first parameter to any functions it calls, including **main**. These functions in turn pass the **gsb** on in any calls they make and so on. In this way the correct value of **gsb** is propagated through the program.

If no static data is required by the process then **main** can be called directly from stage 1 thereby omitting stage 2. Details are given at the end of the source file **centryd1.c**.

3.6 Details of stage 2 of the runtime startup code

Stage 2 of the runtime startup code is responsible for setting up global data required by the runtime system. The sequence of operations performed by this code is described in the following sections. Stage 2 of the runtime startup code can be found in the source file `centryd2.c`.

3.6.1 Set up bounds of stack

The first task of stage 2 is to define the boundaries of the stack for the main thread of execution, i.e. the stack that the program is running within, when the `main` function begins executing. The bounds of the stack are defined by setting up two global variables as follows:

`_IMS_stack_base`	A pointer to the origin of the stack.
`_IMS_stack_limit`	A pointer to the bottom of the memory area set aside for use as the stack. This represents the maximum extent to which the stack can grow.

The implementation of the following facilities uses the two global variables to determine whether a pointer points into the main thread stack:

- Stack checking.
- Parallel process initialization routines.
- The `get_details_of_free_stack_space` function.
- The `max_stack_usage` function.

The two global variables must be set up if any of the above facilities are used.

3.6.2 Initialize heap

The next task of stage 2 is to initialize the heap. This is achieved by setting up four global variables. True heap initialization will not take place until the first use of a heap allocation function. The variables are as follows:

`_IMS_heap_init_implicit`	A boolean flag used to determine whether heap initialization occurs implicitly on the first use of a memory allocation function or whether an explicit initialization call is required. In this runtime system implicit heap initialization is used so this variable must always be set to TRUE.
`_IMS_heap_start`	A pointer to the base of the memory area to be used as the heap.
`_IMS_heap_size`	The size of the memory area to be used as the heap. This size is given in bytes.
`_IMS_sbrk_alloc_request`	The size of the block of memory that `sbrk` adds to the space available for use by the heap allocation routines. This size is given in bytes.

`sbrk` is a low level routine which returns a block of memory for use by the heap allocation routines: `calloc`, `malloc` and `realloc`. These blocks of memory are contiguously allocated from the heap area, defined by the variables `_IMS_heap_start` and `_IMS_heap_size`. The size of these blocks of memory is given by `_IMS_sbrk_alloc_request`. The default sizes for the blocks are 4K on a 32 bit processor and 1K on a 16 bit processor. The minimum size for `_IMS_sbrk_alloc_request` is 16 bytes on a 32 bit processor and 8 bytes on a 16 bit processor. A value smaller than this does not allow enough space for the memory allocation functions to maintain information on the state of the heap.

If no heap is required then all these initializations can be omitted.

Note: that the runtime system depends on the presence of a heap for its implementation of I/O. Thus removing the heap precludes the use of the full library. The heap may only be removed if the reduced library is to be used.

3.6.3 Initialize pointer to configuration process structure

The next item to be initialized is a global variable which points to the configuration process structure which was passed to `C.ENTRYD` (or `C.ENTRYD.RC`).

`_IMS_PData`	A pointer to the configuration process structure for this process.

This global variable is used by the following functions:

- `get_param`
- `get_bootlink_channels`
- `get_details_of_free_memory`

These functions obtain information via the configuration process structure. In particular `get_param` needs `_IMS_PData` so that it can find the data block containing the parameters set up at the configuration level.

Note: that `_IMS_PData` must be set up if the I/O system is to be used because the I/O system obtains the server channel via `get_param`.

3.6.4 Initialize I/O system

Now the I/O system can be set up. This is not done in the reduced case.

The first job in setting up the I/O system is to establish a link to a server. In a configured system using the full library the first two configuration parameters must be the server channels. `get_param` is used to obtain these channels and then the function `set_host_link` is called which stores the channels for use by the runtime system.

The function `io_and_hostinfo_init` is now called. This allocates the space required by the I/O system and initializes file system data. It also obtains information from the server about which host system is being used. Setting up the I/O system requires a heap to have been initialized.

3.6.5 Get command line arguments

The next job is to obtain the command line arguments `argc` and `argv`. This is not done in the reduced case. The arguments are obtained by calling the function `GetArgsMyself`. Server communication must have been established before this call.

3.6.6 Save exit return point

A call to `setjmp` is the next action. This records the position to `longjmp` to when `exit` is called. The return position is stored in the following global variable:

`_IMS_startenv`	The position to `longjmp` to when `exit` is called.

3.6.7 Initialize `clock`

The final action before calling `main` is to store the current process time and current process priority. These values are used by the `clock` function when calculating elapsed processor time and are stored in the following variables:

`_IMS_StartTime`	The value of the processor clock just before the call to `main`.
`_IMS_clock_priority`	The priority at which the startup code is running.

The priority is required because `clock` is defined to work only at the priority at which the C program was started. If `clock` is not required, these initializations may be omitted.

3.6.8 Call `main`

The runtime system is now set up and `main` is called. The call is different in the full and reduced cases. The reduced case does not have true values of `argc` and `argv` and so these are set up in a way that satisfies the ANSI standard.

`main` is called as an argument to `exit`. Thus returning from `main` with a value behaves the same as calling `exit` with that value.

The call to `exit` can be omitted if required. **Note:** that if the call to `exit` is used, then the call to `setjmp` must also remain, otherwise `exit` will not know where to `longjmp` to.

3.6.9 Terminate server if required

The final action of the startup code is to determine whether to terminate the server. This depends on how the program (once `main` has returned) was exited. The default action is to terminate the server. This can be overridden by calling `exit_noterminate`.

The global variable `_IMS_entry_term_mode` is used to determine how the program exited. It is set up by the exit functions. Bit 2 of `_IMS_entry_term_mode` is set if the server is to be terminated.

If the server is to be terminated the value returned by `main` or passed as the argument to an exit function must be returned to the calling environment. This value is stored in the global variable `_IMS_retval`. To terminate the server the function `terminate_server` is called with the return value as its argument.

Special action is taken in the case of the two values `EXIT_SUCCESS` and `EXIT_FAILURE`. These are word length values; the server expects 32 bit values for these special status values and so these are converted before the call to `terminate_server`.

`_IMS_entry_term_mode`	Used to determine whether the server should be terminated. If bit 2 is set then the server is terminated
`_IMS_retval`	The value to be passed to the server when it terminates. Either returned from `main` or the argument to an exit function.

3.7 Interface to main

The INMOS interface to `main` is as follows:

```
#include <channel.h>

int main(int argc, char *argv[], char *envp,
         Channel *in[], int inlen,
         Channel *out[], int outlen);
```

In this version of the runtime startup only `argc` and `argv` are of interest. The rest of the arguments are included for compatibility with previous systems. They are set up as follows:

Argument	Value
`envp`	`NULL`
`in`	`NULL`
`inlen`	0
`out`	`NULL`
`outlen`	0

3.8 Static initialization

The function `initialise_static` performs static initialization in two stages. The first stage is to clear the entire static area to all zeros. Thus all static data without explicit initializers is set to zero. The next stage initializes all non-zero static data.

Each object file which defines static or external data has included within it a static initialization routine. This routine initializes the parts of the static area associated with the object file. During linking the linker creates a chain of all the static initialization routines called the "static initialization chain". The second stage of static initialization walks this chain calling each routine in turn.

Each entry on the chain consists of a header and a routine. The header is used to link the chain together, it contains the byte offset to the next entry in the chain or zero if the entry is the last on the chain. The start of the chain is found using a word patched by the linker. This word contains the byte offset to the first entry in the chain. The function `get_init_chain_start`, (defined in `getinit.s`, see section 3.9) returns a pointer to this word. Figure 3.3 illustrates the layout of a static initialization chain in memory.

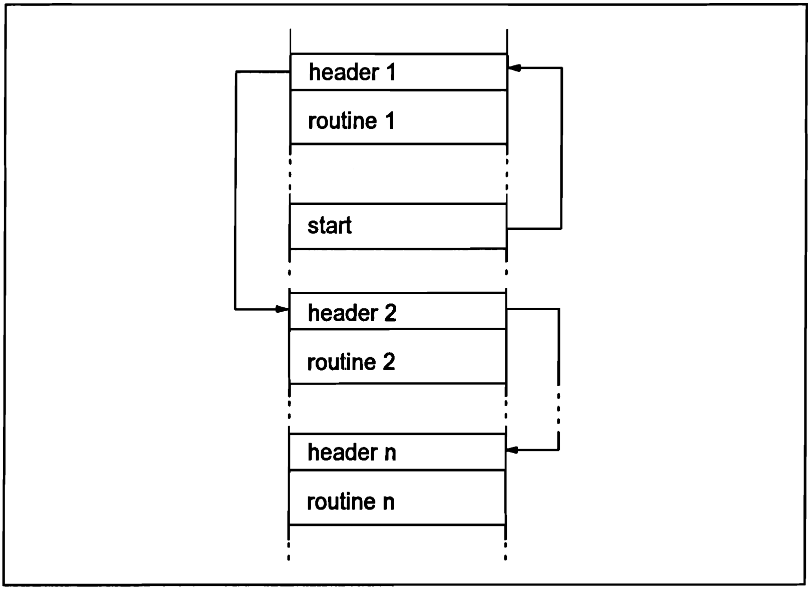

Figure 3.3 Static initialization chain

In figure 3.3 `start` contains the offset to `header 1`, which contains the offset to `header 2`, and so on to `header n` which contains the value 0 to denote the end of chain.

Having obtained the address of the header, incrementing it by one word yields the address of the routine. The routine has the prototype:

```
void routine(void);
```

and can be called via a function pointer.

3.9 Source files supplied and rebuilding

This section provides a summary of the source files supplied and describes how to rebuild the runtime code once it has been modified.

The following source files are supplied:

`centryd1.c`	The stage 1 runtime startup code. This is the entry point called by the configuration system.
`centryd2.c`	The stage 2 runtime startup. This is called by stage 1 and is responsible for initializing global data for the runtime system.
`uglobal.h`	A header file declaring all the global variables which are initialized by the stage 2 runtime startup code.
`startup.h`	A header file defining all the support functions called by the runtime startup code.
`config.h`	A header file defining the structure passed to the runtime system by the configuration system.
`istatic.c`	The `initialise_static` function. This is responsible for initializing the static area and is called from the stage 1 runtime startup code.
`getinit.s`	The `get_init_chain_start` function. This returns a pointer to the head of the static initialization chain. It is called by `initialise_static`.

In order to generate a runtime system which is suitable for use with all possible processor types it is usual to compile the above for the T2, TA and T8 processor classes.

Note: the compilation should be performed using the standard `icc` compiler. The optimizing `icc` compiler does not include some of the support required, i.e. it does not support 16–bit transputers or debug information.

The following example shows how to compile the above source files for the T8 transputer class:

UNIX based toolsets:

```
icc centryd1.c -t8
icc centryd2.c -t8
icc istatic.c -t8

icc getinit.s -pp -t8 > getinit.pps
icc getinit.pps -t8 -as
```

MS–DOS based toolsets:

```
icc centryd1.c /t8
icc centryd2.c /t8
icc istatic.c /t8

icc getinit.s /pp /t8 > getinit.pps
icc getinit.pps /t8 /as
```

VMS based toolsets:

```
icc centryd1.c /t8
icc centryd2.c /t8
icc istatic.c /t8

DEFINE SYS$OUTPUT temp.pps
icc getinit.s /pp /t8
DEASSIGN SYS$OUTPUT
icc temp.pps /t8 /as /o getinit.tco
```

Note: that stack checking must *NOT* be enabled for any of these files. The stack checking code is not set up properly until after the startup code has executed and would fail if used before. Pragmas in the source files ensure that stack checking is not enabled.

Note: how `getinit.s` is built in two stages. The first stage uses the C preprocessor, the second uses the assembler. (The `icc` '`PP`' option sends output to `stdout` by default; this is redirected to a named file, ready for input to the assembler).

If the reduced versions are required use the command line option '`d`' to add the symbol '`REDUCED`' to the command line of all invocations of `icc`, except those which use the '`as`' option.

This procedure may be repeated for classes T2 and TA as appropriate.

The object files produced from the above should be added to the linker command line along with all other object files. ***They should NOT be made into a library.*** (If they are in a library then the linker cannot be guaranteed to link in the modified version of the startup code in preference to that which exists in the standard library).

An example of how to recompile and link the runtime source is given in section 3.11.

3.10 Notes

This section lists some final considerations:

- The final size of the bootable obtained depends on the bootstrap scheme used. See the documentation for the configurer `icconf` and the collector `icollect` for details of this. (Chapters 2 and 3 of the *ANSI C Toolset Reference Manual*, respectively).
- The runtime startup code includes a 16 byte `information%module`. This is a special TCOFF module, used by other tools e.g. the debugger, to find the address of the `main` routine. The `information%module` also contains TCOFF comments giving the version number of the library. If required, the `information%module` can be omitted by removing the following line from either `cstartup.lnk` or `cstartrd.lnk`, before linking:

  ```
  #reference information%module
  ```

3.11 Example

In the following example a copy of `centryd2.c` is modified to omit the code to obtain the command line arguments and initialize the clock function. The modified entry point is to be built for the full runtime library.

```
/* @(#)centryd2.c      1.18 10/1/92 */
/* Copyright (C) INMOS Ltd, 1992 */

/*****************************************************************************
 *                                                                           *
 * This file contains the second stage of the C runtime startup code for    *
 * use in configured systems. This source is used to build the entry        *
 * points:                                                                   *
 *                                                                           *
 *   C.ENTRYD    : linked using cstartup.lnk                                 *
 *   C.ENTRYD.RC : linked using cstartrd.lnk                                 *
 *                                                                           *
 * The reduced version is obtained by defining the REDUCED preprocessor     *
 * symbol.                                                                   *
 *                                                                           *
 * By modifying this code it is possible to greatly reduce the size of      *
 * runtime overhead which is added by the standard C entry points.          *
 *                                                                           *
 * Note that this code relies on the presence of a static area. If no       *
 * static area is required then main() can be called directly from the     *
 * first stage and this stage may be omitted. See the file centryd1.c for   *
 * more information.                                                         *
 *                                                                           *
 * FULL STAGE 2 :    a) Set up bounds of stack.                              *
 *                   b) Set up heap.                                         *
 *                   c) Set up pointer to configuration process structure.  *
 *                   d) Set up I/O system and host system type.             *
 *                   e) Get command line args.                               *
 *                   f) Save return point for exit.                          *
 *                   g) Set up clock.                                        *
 *                   h) Call main.                                           *
 *                   i) Terminate server if required                         *
 *                                                                           *
 * REDUCED STAGE 2 : a) Set up bounds of stack.                              *
 *                   b) Set up heap.                                         *
 *                   c) Set up pointer to configuration process structure.  *
 *                   d) Save return point for exit.                          *
 *                   e) Set up clock.                                        *
 *                   f) Call main.                                           *
 *                                                                           *
 * Note that the order in which the above tasks are done is significant.    *
 * Changing the order may cause the system to fail.                          *
 *                                                                           *
 *****************************************************************************/

/*****************************************************************************
 *                                                                           *
 * Make sure stack checking is disabled when this file is compiled. Stack   *
 * checking must not be enabled in the start up code because global data    *
 * required by the stack checking code is not set up yet.                   *
 *                                                                           *
 *****************************************************************************/

#pragma IMS_off(stack_checking)

/*****************************************************************************
 *                                                                           *
 * Include files.                                                            *
 *                                                                           *
```

```
*******************************************************************************/

#include <setjmp.h>      /* for setjmp                          */
#include <channel.h>     /* for Channel                         */
#include <stddef.h>      /* for NULL                            */
#include <stdlib.h>      /* for exit                            */
#include <process.h>     /* for ProcTime and ProcGetPriority */
#include "uglobal.h"     /* for globals                         */
#include "startup.h"     /* for startup internal functions      */
#include <misc.h>        /* for get_param                       */
#include "config.h"      /* for Conf_Process                    */

/******************************************************************************
 *                                                                            *
 * Declare main using the INMOS standard argument list.                       *
 *                                                                            *
 ******************************************************************************/

extern int main(int argc, char **argv, char **envp,
                Channel *in[], int inlen,
                Channel *out[], int outlen);

/******************************************************************************
 *                                                                            *
 * Define the second stage routine. The name is translated to avoid invading *
 * the user's name space.                                                     *
 *                                                                            *
 ******************************************************************************/

#pragma IMS_translate(CENTRYD_stage2, "CENTRYD_stage2%c")

void CENTRYD_stage2(struct Conf_Process *pdata)
{
  /*========================================================================*
   * This is where the argc and argv variables that are passed to main were *
   * defined. They are removed because we are not providing this facility   *
   * in the modified code.                                                  *
   *========================================================================*/


  /*************************************************************************
   *                                                                       *
   * Set up the bounds of the stack for the main thread of execution. These *
   * globals are used by the following:                                    *
   *   1. Stack checking.                                                  *
   *   2. The get_details_of_free_stack_space function.                    *
   *   3. Parallel processes (use of ProcAlloc and ProcInit).              *
   *   4. The max_stack_usage function.                                    *
   * If any of these features are used then the following initialisations *
   * may not be omitted.                                                   *
   *                                                                       *
   *    _IMS_stack_limit        : The maximum extent to which the stack can *
   *                              grow. Note that the stack is a falling   *
   *                              stack.                                   *
   *    _IMS_stack_base         : Pointer to base of stack.                *
   *                                                                       *
   *************************************************************************/

  _IMS_stack_limit = (int *)((unsigned int)pdata->StackAddr -
                             pdata->StackSize);
  _IMS_stack_base  = (int *)(pdata->StackAddr);
```

```
/****************************************************************************
 *                                                                          *
 * Set up the heap. If no heap is required then these initialisations can *
 * be omitted.                                                              *
 * Note that a heap must be set up if the full library is being used.      *
 *    _IMS_heap_start         : A pointer to the base of the heap.          *
 *    _IMS_heap_init_implicit : A boolean which is set to TRUE if the heap *
 *                              is initialised implicitly on the first call*
 *                              of a memory allocation function. This must *
 *                              always be set to TRUE otherwise the heap   *
 *                              allocation functions will fail.            *
 *    _IMS_heap_size          : The size of the heap memory region in      *
 *                              bytes.                                     *
 *    _IMS_sbrk_alloc_request : The size of block which sbrk adds to the   *
 *                              memory space available to malloc.          *
 ****************************************************************************/

  _IMS_heap_start         = (int *)(pdata->HeapAddr);
  _IMS_heap_size          = pdata->HeapSize;
  _IMS_heap_init_implicit = TRUE;
  _IMS_sbrk_alloc_request = SBRK_REQUEST;


  /****************************************************************************
   * Set up the global variable which is used by some functions to obtain   *
   * a pointer to the configuration process structure set up by the         *
   * configurer.                                                             *
   * The following functions make use of this global:                       *
   *    1. get_param                                                         *
   *    2. get_bootlink_channels                                             *
   *    3. get_details_of_free_memory                                        *
   * If none of these functions are used then this initialisation may be    *
   * omitted.                                                                *
   * Note that get_param is used below, so that if the initialisation of    *
   * _IMS_PData is omitted then make sure that the call to get_param below  *
   * is not required, and hence omitted.                                     *
   ****************************************************************************/

  _IMS_PData = pdata;

#ifndef REDUCED

  /****************************************************************************
   * Set up the host link information. The runtime system assumes that the  *
   * first two configuration parameters are channels fromserver and         *
   * toserver respectively. This is not required in a reduced system.       *
   ****************************************************************************/

  {
    Channel *in, *out;

    in  = (Channel *)get_param(1);
    out = (Channel *)get_param(2);
    set_host_link(in, out);
  }

  /****************************************************************************
   * Set up the I/O system and obtain the host type. The I/O system         *
   * consists of three layers and all three are set up by this call.        *
   * The host information is required so that the I/O system can determine *
   * the type of the host file system. Note that this means that the        *
   * host_info function is only available as long as the following is       *
   * called. The host link information must have been set up before the I/O *
   * system is initialised. This is not required in a reduced system.       *
   * A heap must have been set up in order for this call to succeed.        *
   ****************************************************************************/
```

```
  io_and_hostinfo_init();

  /*=====================================================================*
   * This is where the call to obtain the command line arguments was made.  *
   *=====================================================================*/

#endif /* REDUCED */

  /**************************************************************************
   * Call setjmp to mark the return position for a call to exit. The setjmp *
   * is only required as long as a call to exit() is subsequently used.     *
   **************************************************************************/

  if (setjmp(_IMS_startenv) == 0)
  {
    /*==================================================================*
     * This is where the code to initialise the clock function used to be.  *
     * In this example we do not require the clock function and so we have  *
     * deleted the lines which did the initialisation.                       *
     *==================================================================*/

    /************************************************************************
     * Call main. We call main as an argument to exit. Thus returning from  *
     * main is like a call to exit. The call to exit ensures that ANSI      *
     * behaviour on closing open files etc. is followed. Note that the      *
     * reduced case also sets up argc and argv as required by ANSI.         *
     * If ANSI behaviour is not important then a minimal call to main which *
     * still returns the result of main to the environment is as follows:   *
     *    _IMS_retval = main(0, NULL, NULL, NULL, 0, NULL, 0);              *
     * Since only those systems which terminate the server can return a     *
     * value to the calling environment then we only need to store to       *
     * _IMS_retval if we subsequently call terminate_server.                *
     ************************************************************************/

    /*==================================================================*
     * We force the use of the call to main from the reduced version of    *
     * this file since this sets up some dummy values for argv and argc.   *
     *==================================================================*/

    {
      char *argv[2] = { "", NULL };

      exit(main(1, argv, NULL, NULL, 0, NULL, 0));
    }

  }
#ifndef REDUCED

  /*************************************************************************
   * main has returned, we must now decide whether to terminate the server. *
   * Not required for the reduced case.                                     *
   * We terminate the server only if exit_terminate was called.             *
   * The global variable _IMS_entry_term_mode can be used to decide whether *
   * exit, exit_repeat, exit_terminate or exit_noterminate was called to    *
   * exit the program. exit_repeat and exit_terminate act like exit in      *
   * the configured case so we only worry about whether exit_noterminate is *
   * called. If exit_noterminate is called the bit 2 of _IMS_entry_term_mode*
   * is clear. If this level of control is not required the test or the call*
   * to exit_terminate or both can be omitted.                              *
   * The return value of the program is stored in _IMS_retval by exit. We   *
   * must convert the special values for EXIT_SUCCESS and EXIT_FAILURE to   *
   * their iserver counterparts sps.success and sps.failure. Note that we   *
   * need a long value to contain the server status which is a 32 bit value *
   * on all processors.                                                     *
   *************************************************************************/
```

```
  if ((_IMS_entry_term_mode & TERM_BIT) != 0)
  {
    long int status = (long int)_IMS_retval;
    if (status == EXIT_SUCCESS)
      status = SPS_SUCCESS;
    else if (status == EXIT_FAILURE)
      status = SPS_FAILURE;
    terminate_server(status);
  }

#endif /* REDUCED */
}
```

3.11.1 Building the modified runtime system

The new version of `centryd2.c` must be compiled so that it can be used as part of the startup code. For this example a version is required which works with the full library, on 32 bit transputers which do not have floating point units. The compilation command is as follows:

UNIX based toolsets:	MS–DOS/VMS based toolsets:
`icc centryd2.c -ta`	`icc centryd2.c /ta`

This produces the object file `centryd2.tco`. This object file is linked along with the rest of the object files and libraries which are required to build the program.

For example:

To link in the new version of `centryd2.tco` for a program comprising one file: `main.tco`, targeted at a T425 transputer, use the following command:

UNIX based toolsets:

```
ilink main.tco centryd2.tco -f cstartup.lnk -t5
```

MS–DOS/VMS based toolsets:

```
ilink main.tco centryd2.tco /f cstartup.lnk /t5
```

This creates `main.lku` which consists of a C `main` called via startup code which includes the new version of `centryd2.tco`.

`main.lku` can now be used as part of a configured system.

Language Reference

72 TDS 347 01

October 1992

4 New features in ANSI C

This chapter describes the new features added by the ANSI standard to the C language.

This chapter is not intended to be a reference to ANSI standard C but rather a summary of differences from the previous widely-known definition of the language. For a formal description of the language the reader is referred to the ANSI reference documents and to '*C: A Reference Manual*' by Harbison and Steel.

Kernighan and Ritchie's original description of the language as defined in their book '*The C programming language*' (First edition 1978), is referred to in this chapter as 'K & R C'. Details of these publications can be found in the bibliography to the rear of this manual.

This chapter is divided into two sections:

4.1 A summary of the new features added by ANSI to the original definition of the language.

4.2 Detailed descriptions of the new features.

4.1 Summary of new features in the ANSI standard

The following tables list the new features in the ANSI standard. The tables list the main areas of change and briefly describe how they differ from the original implementation of the language.

Area of change	ANSI standard
Function declarations	Parameter lists in function declarations can include type specifiers with or without identifiers. The new `void` type can be used and the list may end with an ellipsis '...' to indicate a variable number of parameters.

Type specifiers	1. New types:
	`enum`
	`void`
	2. New type qualifiers:
	`const`
	`volatile`
	3. New type specifiers:
	`signed`

Area of change	ANSI standard
	Where specified alone, `signed`, `const`, and `volatile` imply the appropriately qualified `int` type.

	3. New types:
	`unsigned char`
	`unsigned long`
	`signed char`

Identifiers	The first 31 characters of internal names are significant.
Keywords	1. Keyword `entry` is no longer valid.

	2. New keywords:
	`const`
	`enum`
	`signed`
	`void`
	`volatile`

Constants	Integer constants can use the suffix `U` to denote an unsigned integer constant.
	Floating point constants can use the suffixes `F` (for `float`) and `L` (for `long double`).
Operators	New unary operator '+' added to complement '–'.
Character types	Character constants are of type `int` and are sign extended in type conversions.
	New character escape codes: `\"` `\?` `\x` `\a` `\v`
	Signedness of `char` types is implementation defined.
Hardware characteristics	The type `short` is at least 16 bits long and the type `long` at least 32 bits long.

Compiler control lines	New preprocessor directives:
	`#elif`
	`#error`
	`#pragma`

	Some preprocessor macros are also defined.
Structures and unions	Structures and unions can be:
	Assigned to other structures and unions. Passed by value to functions. Returned by functions.
Initialization	Unions can be initialized.
Trigraphs	Character trigraphs are introduced to support the ISO 646 invariant character set.

Table 4.1 New features in ANSI C

4.2 Details of new features

4.2.1 Function declarations

A new form of function declaration is available which allows types to be specified for parameters in the function's parameter list. Declarations can omit parameter identifiers and give only the type specifiers.

It is also possible to specify a variable number of parameters by terminating the parameter list with an ellipsis '. . .'. For example:

```
void add_numbers(int *sum, int a, int b);
        /* Declaration with identifiers */

void add_numbers(int *, int, int);
        /* Declaration without identifiers */

void add_many_numbers(int *sum, int n, ...);
        /* Declaration with variable parameters */
```

A function with no parameters can be specified by specifying the keyword `void` as the only parameter in the parameter list. For example:

```
int hello(void);
```

A function declarator using a parameter type list defines a prototype for that function.

4.2.2 Function prototypes

Function prototypes are a new way of declaring functions. They make programs easier to read and function call errors easier to find.

When using function prototypes:

1. Functions must be explicitly declared before any call is made.
2. Multiple declarations of the same function must agree exactly.
3. Function declarations must use the parameter type list form.
4. When calling a function, the number and types of the parameters must agree with the specification in the declaration.
5. Arguments to functions are converted to the types specified in the declaration.

4.2.3 Functions without prototypes

Non–prototyped functions as described in K & R C are still permitted in ANSI C.

Arguments to non–prototyped functions have the following default argument promotions:

- an argument of type `char`, `short int`, `int` bit-field, or enumeration type are converted to type `int` (signed `int`, if this will correctly represent the argument, unsigned `int` otherwise).

- an argument of type `float` is converted to type `double`.

4.2.4 Declarations

Type qualifiers can be used in pointer declarations. This is particularly useful for creating constant pointers, pointers to constants and pointers to volatiles. For example:

```
const int *ptr_to_constant;
       /* Declares a pointer to a constant int  */

int *const constant_ptr;
       /* Declares a constant pointer to an int */

volatile int *ptr_to_volatile;
       /* Declares a pointer to a volatile int  */
```

4.2.5 Types, type qualifiers and type specifiers

This section describes the ANSI standard syntax for types, type qualifiers and type specifiers.

The following have been added:

Type qualifiers – `const` and `volatile`.

Type specifiers – `enum`, `signed` and `void`.

`const` defines a constant object which cannot be changed in the program. `const` can be used alone or with other type specifiers `struct`, `union`, `enum` or with the type qualifier `volatile`. Used alone it implies `const int`. For example:

```
const int month = 10;

month = 11; /* Not allowed */
month++;    /* Not allowed */
```

`const` can be used within pointer declarations to declare variable pointers to constant values, or constant pointers to variable values.

`enum` is used to create enumerated types. An enumerated type defines a sequence of integer values for groups of logical names. The sequence of values begins at 0 and increments by one unless specific values are assigned. For example:

```
/* Define an enumerated type for the days of the week */
enum days {monday, tuesday, wednesday, thursday,
                              friday, saturday, sunday};
enum days today; /* Declare today as a variableof type days */
today = friday;
if (today == sunday)
.
.
.
```

The default value of a constant can be overridden by assigning a specific integer value. If a member of the list is not assigned a value explicitly, it takes on the value of (previous constant + 1). For example:

```
enum poets {corso, burroughs, ginsberg = 9, cummings};
/* corso = 0, burroughs = 1, cummings = 10 */
```

`signed` complements the existing type specifier `unsigned`. It may be used alone, where it implies `signed int`, or to qualify the following types: `int`, `short int`, `long int`, `char`.

`void` is mainly used to declare functions which do not return a value. For example:

```
void add_numbers();
main()
{
int *answer;
add_numbers(answer,23,42);
}
.
.
.
void add_numbers(sum, b, c)
int *sum;
int b,c;
{
  sum = b + c;
}
```

Another use for `void` is in a cast expression where a returned value is discarded. For example:

```
/* Ignore the return value of fputc */
(void) fputc(ch,stream);
```

`volatile` identifies an object as modifiable outside the control of the implementation. For example, the object may refer to a memory mapped port which is used by a modem. `volatile` can be used to protect objects from unpredictable compiler optimizations.

`volatile` can be used alone or with other type specifiers and qualifiers. Used alone `volatile` implies `volatile int`.

An object can be both `volatile` and `const` in which case it can not be modified by the program but could be modified by an external process (for example, a real time clock). For example:

```
volatile int port_one;
const volatile int clock;
```

4.2.6 Constants

This section summarizes the changes to the syntax for integer, floating point, string and character constants.

The suffix `U` can follow integer constants to indicate type `unsigned`. `U` can be used in conjunction with the existing `L` suffix and the order is not significant. For example:

```
42u   1096U   1001u   2048UL
```

The suffix `F` can follow floating point constants to indicate type `float` and the suffix `L` to indicate type `long double`. For example:

```
3.1F    4.2L
```

The type `long float` is no longer allowed.

Adjacent string constants are concatenated into a single string terminated by a null character ('`\0`').

The following new character escape codes are defined:

Code	Description
`\?`	Gives the question mark character. This should be used where a question mark could be mistaken for part of a trigraph.
`\"`	Gives the double quote character.
`\a`	Rings the bell (equivalent to CTRL-G).
`\v`	Gives a vertical tab.
`\x`*n*	Gives the character represented by *n*, where *n* is the ASCII code of the character represented in hexadecimal. For example, `\x2B` gives the character +.

4.2.7 Preprocessor extensions

This section describes the predefined preprocessor directives and macros.

Compiler directives

Directive	Description
`#elif`	Abbreviation of `#else #if`.

`#error`	Generates a compiler error message containing optional text.
`#pragma`	Causes an implementation-defined effect. In ANSI C this directive is used to select a particular combination of compiler options or to override options given on the command line.

Predefined macros:

Macro	Description
`__DATE__`	The current date, in the form: `Mmm dd yyyy`
`__FILE__`	The name of the current source file, expressed as a string literal.
`__LINE__`	The line number of the current line in the source file, expressed as a decimal constant.
`__STDC__`	A non-zero value if the implementation conforms to ANSI C.
`__TIME__`	The current time, in the form: `hh:mm:ss`.

4.2.8 Structures and unions

In ANSI C structures and unions can be assigned to other structures or unions, passed by value to functions, and returned by functions. Unions can be initialized.

When a structure is given as an argument to a function a copy of the structure is created for use within the function. For example:

```
struct record
{
  char firstname[30];
  int  age;
};

void print_name(struct record person);

struct record test(struct record first,
                   struct record second);

main()
{
  struct record ph;
  struct record rl;

  ph.firstname = "Phil";
  ph.age  = 27;

  /* Assigning a structure to a structure */
  current_person = ph;

  /* Passing a structure as an argument to a
                                    function */
  print_name(current_person);

  /* Returning a structure from a
                          function */
  winner = test(ph, rl);
}
```

Unions can be initialized. The initialization is performed according to the type of its first component and the expression used to perform the initialization must evaluate to the correct type.

For example:

```
union alltypes {
  double bigfloat;
  int digit;
  char letter;
} initalltypes = 3.1;

union complex {
  struct {int a; char b;} s;
  double bigfloat;
} initcomplex = {42, 'x' };
```

4.2.9 Trigraphs

Trigraphs are added to enable C programs to be written using only the ISO 646 invariant code set. ISO 646 is a subset of 7-bit ASCII which contains only those characters present on all keyboards.

Trigraphs and the characters that they represent are listed in the following table.

Trigraph	Character represented
??=	#
??(	[
??)	]
??/	\
??'	^
??<	{
??>	}
??!	|
??-	~

All other trigraph-like sequences are treated as literal strings. For example, the sequence `??+` is not a trigraph and is treated as the literal sequence that it represents.

Trigraphs are converted to the equivalent character before lexical analysis takes place.

Trigraph escape codes

The character escape code `\?` has been added to allow the printing of trigraph strings. The trigraph string should be preceded by the escape character. For example:

```
static char texta[] = "This is a backslash: ??/";
static char textb[] = "This is not a trigraph \??/";
```

5 Language extensions

This chapter summarizes the INMOS extensions to the C language. It describes the concurrency features, compiler pragmas, and lists the predefinitions, all of which are described in detail elsewhere in this book, It also describes the `__asm` statement that supports the insertion of transputer code into C programs.

The INMOS implementation of ANSI C provides the following language extensions beyond the ANSI standard:

- Concurrency support.
- Pragmas.
- Additional predefined macros.
- Assembly language support.

5.1 Concurrency support

Concurrency support is provided by a set of library functions with associated predefined data types and data structures. The library functions are declared in three standard C header files along with all related constants and macros.

Functions are provided for creating and manipulating processes (`process.h`), for synchronizing processes and exchanging data down channels (`channel.h`), and for creating and manipulating semaphores (`semaphor.h`).

Full details of how to create parallel programs using the ANSI C concurrency extensions can be found in chapter 5 '*Parallel processing*' of the accompanying *Toolset User Guide.*

5.2 Pragmas

A series of special compiler operations are implemented as options to the `#pragma` directive. The options available are listed below. Details of the pragmas, their syntax and options can be found in section 1.4.11 in the accompanying *Toolset Reference Manual.*

Pragma	Description
IMS_codepatchsize	Specifies the size of a reserved code patch.
IMS_descriptor	Creates a TCOFF descriptor for C functions.
IMS_linkage	Adds tags for segment ordering.
IMS_nolink	Enables functions to be compiled without a static link parameter. Used when calling occam code from C, and C functions from occam.
IMS_nosideeffects	Marks a function as side effect free. This pragma is implemented for the optimizing C compiler but is ignored by the standard C compiler.
IMS_modpatchsize	Specifies the number of bytes reserved for a module number patch.
IMS_on	Enables specific compiler checks. Checks to be enabled are specified as arguments to the pragma.
IMS_off	Disables specific compiler checks. Takes the same set of check arguments as IMS_on.
IMS_translate	Translates all references to one name into another name. Used to create aliases for external routines which contain prohibited characters.

5.3 Predefined macros

The following predefined macros are provided in the ANSI C toolset in addition to the standard definitions required by the ANSI standard.

Constant	Meaning/value
__CC_NORCROFT	Indicates a compiler derived from the Norcroft C compiler. Set to the decimal constant one (1).
_ICC	Indicates the ANSI C compiler icc. Set to the decimal constant one (1).
_PTYPE	Indicates the target processor type. Takes the following values: '2' – T212 '3' – T225 '4' – T414 '5' – T425/T426/T400 '8' – T800 '9' – T801/T805 'A' – Class TA 'B' – Class TB
_ERRORMODE	A decimal constant indicating the execution error mode. Takes the following values: 1 – HALT 2 – STOP 3 – UNIVERSAL **Note:** all compiled object code generated by icc is in UNIVERSAL mode.
__SIGNED_CHAR__	A decimal constant indicating the signedness of the plain char type. It is only defined if the icc 'FC' command line option is used. When defined it takes the value '1'.

5.4 Assembly language support

The insertion of transputer code into C programs is performed using the `__asm` statement. Sequences of transputer instructions specified in this way are assembled in line by the compiler.

The rest of this section assumes some familiarity with the transputer instruction set. For a list of transputer instructions see appendix A '*Transputer instruction set*' in the accompanying *Toolset User Guide*.

A more detailed description of the instruction set including information about architecture and design can be found in '*Transputer instruction set: a compiler writer's guide*'.

The full syntax of the `__asm` statement is given in section A.3.

5.4.1 Directives and operations

`__asm` statements can contain any number of primary or secondary transputer operations, optionally preceded by a `size` qualifier, or transputer pseudo-operations. Any transputer instruction can be prefixed with a label.

In the transputer instruction set primary operations are *direct* functions, *prefixing* functions, or the special indirect function *opr*. Primary operations are always followed by an operand which can be any constant or constant expression. If additional `pfix` and `nfix` instructions are required to encode large values the assembler automatically generates the required bytes.

Secondary operations are any transputer *operation*, that is, any instruction selected using the *opr* function.

Pseudo-operations are instructions to the assembler, built up from sequences of instructions. Like macros, they expand into one or more transputer instructions, depending on their context and parameters.

Pseudo-operations that are supported by `__asm` are listed in table 5.1.

Pseudo-operation	Description
`ld` *expression*	Loads a value into the **Areg**.
`st` *lvalue*	Stores the value from the **Areg**.
`ldab` *expression, expression*	Loads values into the **Areg** and **Breg**. The left hand expression is placed in **Areg**.
`stab` *lvalue, lvalue*	Stores values from the **Areg** and **Breg**. The leftmost element receives **Areg**.
`ldabc` *expression, expression, expression*	Loads values into **Areg, Breg** and **Creg**. The leftmost expression is placed in **Areg**.
`stabc` *lvalue, lvalue , lvalue*	Stores values from the **Areg**, **Breg**, and **Creg**. The leftmost element receives **Areg**.
`[size` *constant*`] j` *label*	Jump
`[size` *constant*`] cj` *label*	Conditional jump
`[size` *constant*`] call` *label*	Call
`[size` *constant*`] ldlabeldiff` *label – label*	Loads the difference between the addresses of two labels into **Areg**.
`byte` *constant {, constant }*	This instruction takes as an argument a list of constant values. Only the lower 8 bits of the constant values are generated i.e. if the constant is too large to fit in a byte, only the least significant bits will be generated. The assembler copies the literal bytes into the instruction stream.
`word` *constant {, constant }*	Generates constants of the target-machine word length. This instruction takes as an argument a list of constant values. If the constant is too large to fit in a target-machine word, only the lower bits will be generated.
`align`	This instruction takes no operands. It generates padding bytes (prefix 0) until the current code address is on a word boundary.

Table 5.1 Pseudo-operations

*lvalue*s can be any valid modifyable C *lvalue*, and labels can be any valid C label.

The `ldlabeldiff` operation loads the difference between the addresses of two labels into **Areg**.

5.4.2 `size` option on `__asm` statement

The `size` option on `__asm` statements that incorporate transputer operations, direct, prefixing and certain pseudo-instructions, forces the instruction to occupy a set number of bytes. If the instruction is shorter than this, it is padded out with trailing prefix 0 instructions. If the instruction cannot fit in the specified number of bytes, a compiler error is reported. The `size` option allows instructions to be built with the same size and is intended to assist the creation of jump tables.

5.4.3 Labels

Labels can be placed on `__asm` statements or on any line of transputer code. Labels placed inside and outside the `__asm` statement are handled identically. C statements are permitted to goto a label set inside an `__asm` statement and vice versa.

5.4.4 Notes on transputer code programming

1 Floating-point (fp) registers cannot be loaded directly; they must be loaded or stored by first loading a pointer to the register into an integer register and then using the appropriate floating-point load or store instruction.

2 The operands to the load pseudo-ops must be small enough to fit in a register and the operands to the store pseudo-ops must be word-sized modifiable *lvalues*.

5.4.5 Useful built-in variables

Special recognition of the following variables is built into the compiler.

`extern volatile const void *_lsb`	Pointer to the base of a file's static area.
`extern volatile const void *_params`	Pointer to the base of the current function's parameter block.

Given access to a function's parameter block and using the calling conventions described in section 6.16, it is possible to determine the function's return address, the global static pointer, and the calling function's workspace as in the following example:

```
void p(int a, int b)
{
  typedef struct paramblock
     { void *return_address;
       void *gsb;
       int regparam1, regparam2;
     } paramblock;

  extern volatile const void *_params;

  paramblock *pp = _params;
  /* return address is:     pp->return_address
     global static base is: pp->gsb
     caller's wptr is:       (void *)(pp + 1); */
}
```

5.4.6 Transputer code examples

This section contains listings of programs fragments that illustrate common uses of embedded instruction code.

Setting the transputer error flag

```
void set_error_flag(void)
{
  __asm { seterr; }
}
```

Loading constants using literal operands

```
#define answer 42
const int c
__asm {
   ldc   17;          /* decimal */
   ldc   0xff;        /* hex */
   ldc   0377;        /* octal */
   ldc   answer;      /* defined by macro */
   ldc   sizeof(c);   /* constant expression */
   ldc   10+7;        /* ditto  */
}
```

Labels and jumps

```
void p(void)
{
  int a, b, c;
  /* The following code performs
     if (b > c) a = b; else a = c; */
   __asm{
         ld   b;
         ld   c;
         gt;
         cj   label1;
         ld   b;
         st   a;
         j    done;
label1:
         ld   c;
         st   a;
done:    ;
  }
}
```

Jump tables

```
#include <stdio.h>
#define JUMP_SIZE 3
void p(int i)
{
  __asm{ ld          i;
            /* load the index */
        adc         -1;
            /* subtract base subscript */
        ldc          JUMP_SIZE;
            /* scale by size of table entry */
        prod;
        ldlabeldiff table - here;
            /* load pointer to start of table */
        ldpi;
here:
        bsub;
            /* add the offset */
        gcall;
            /* jump to ith. entry */

table:
        size JUMP_SIZE j lab1;
        size JUMP_SIZE j lab2;
        size JUMP_SIZE j lab3;
        size JUMP_SIZE j lab4;
      }
  lab1: printf("i = 1"); return;
  lab2: printf("i = 2"); return;
  lab3: printf("i = 3"); return;
  lab4: printf("i = 4"); return;
}
```

Loading floating point registers

```
void p(void)
{
  float a, b, c;
  /* The following code performs
         a = b - c;              */
   __asm{
         ld     &b;
         fpldnlsn;
         ld     &c;
         fpldnlsn;
         fpsub;
         ld     &a;
         fpstnlsn;
      }
}
```

Using align/word to return an element of a table

```
int p(int i)
{
  /* The following code returns the ith
     element of the table defined below */

  int res;
  __asm{
      ld           i;
      ldlabeldiff table - here;
      ldpi;
here:
      wsub;
      ldnl         0;
      st           res;
      j            done;
      /* Make sure table is word aligned
        for ldnl to work correctly */
      align;
table:
        word        1, 1, 2, 3, 5, 8, 13, 21, 34;
      }
done:
  return res;
}
```

Inserting raw machine code

The following code inserts the actual machine code (in hex) for the *ret* instruction.

```
void ret_hex(void)

{
  __asm { byte 0x22, 0xF0; }
}
```

6 Implementation details

This appendix describes the implementation of the language in areas where the ANSI standard is flexible or allows alternative solutions.

Note: the document '*Performance improvement with the INMOS Dx314 ANSI C Toolset*' which accompanies the toolset, considers performance aspects and suggests ways in which C programs may be improved.

6.1 Data type representation

6.1.1 Scalar types

C scalar type representations on 32 and 16 bit transputers are described in the following table.

`unsigned char`	32 and 16	Represented in a word in which the lower eight bits are significant, the upper bits are zero.
`signed char`	32 and 16	Represented in a word in which the lower eight bits are significant, bit 7 is the sign-bit, the upper bits are zero.
`char`	32 and 16	The representation of char differs depending on whether the compiler `FC` option is used to make plain `chars` signed. When `FC` is used `char` has the same representation as `signed char`; without `FC` the representation is the same as `unsigned char`.
`unsigned short`	32	Represented in a word in which the lower 16 bits are significant, the upper bits are zero.
	16	Represented in a word in which all 16 bits are significant.
`signed short`	32	Represented in a word in which the lower 16 bits are significant, bit 15 is the sign bit, the upper bits are zero.
	16	Represented in a word in which all 16 bits are significant, bit 15 is the sign bit.

unsigned int	32	Represented in a word in which all 32 bits are significant.
	16	Represented in a word in which all 16 bits are significant.
signed int	32	Represented in a word in which all 32 bits are significant, bit 31 is the sign bit.
	16	Represented in a word in which all 16 bits are significant, bit 15 is the sign bit.
unsigned long	32	Represented in a word in which all 32 bits are significant
	16	Represented in two words in which all 32 bits are significant,the lower addressed word contains the least significant bits.
signed long	32	Represented in a word in which all 32 bits are significant, bit 31 is the sign bit
	16	Represented in two words in which all 32 bits are significant, bit 15 of the upper word is the sign bit. The lower addressed word contains the least significant bit.
float	32	Represented in a word, in IEEE single-precision format.
	16	Represented in two words, in IEEE single-precision format.
double	32	Represented in two words, in IEEE double-precision format.
	16	Represented in four words, in IEEE double-precision format.
enumeration	32	Represented in a word in which all 32 bits are significant.
	16	Represented in a word in which all 16 bits are significant.

All signed integer types are represented in twos-complement form and all unsigned integer types in binary form.

All floating point types are represented in a form defined by the ANSI/IEEE standard 754-1985.

6.1.2 Arrays

Each element of an array of `char` occupies 8 bits and each element of an array of `short` occupies 16 bits.

Elements of arrays of any other type are represented as the element would be represented if it was not in an array. An array is padded at the high-end address to the next word boundary: the padding has no defined value.

6.1.3 Structures

Structure fields are allocated starting from the lowest address. Fields of type `char` are allocated on a byte boundary, and are represented in 8 bits.

On 32-bit machines only, fields of type `short` are allocated on an even-address boundary, and are represented in 16 bits. Thus, adjacent `char` or `short` fields may be packed into the same word.

Adjacent bit-fields are packed into the same word if possible: the first bit-field is placed in the least significant bits of the word. If there is not enough room left after a previous bit-field, a bit-field will be placed in the least significant bits of the next word. Fields of any other type are represented as they would be if the field was not in a structure. A structure is padded at the high-end address to the next word boundary: the padding has no defined value.

The compiler uses the following rules when laying out the fields within a structure:

- C requires that structure fields are laid out in memory in the same order that they are in the source code.
- `char`s may have any alignment.
- `short`s are aligned on an even boundary.
- word–sized or larger objects are aligned on a word boundary.
- structures, unions and arrays are aligned on a word boundary.

`char` and `short` fields will be packed into the same word where possible, without breaking any of the above rules.

Example 1 (structuring on a 32–bit processor):

`struct d {`	
`char hid[8];`	The first byte of `hid` is on a word boundary (as the first byte of structure is on a word boundary), it occupies 8 bytes (2 whole words).
`unsigned short inuse;`	This occupies the lower two bytes of the following word.
`char flags1;`	This is packed into byte 2 of the same word as `inuse`.
`char flags2;`	This is packed into the upper byte of the same word as `inuse` and `flags1`.
`unsigned long tkey;`	This occupies the following word.
`unsigned short tfil;`	This occupies the lower two bytes of the following word.
`long npos;`	This has to be allocated on the next word boundary, so two bytes are left unused.
`unsigned short kmod;`	This occupies the lower two bytes of the following word.
`unsigned short kbhz;`	This is packed into the upper two bytes of the same word as `kmod` — 16–bit objects are placed at even addresses (rule 2), not word–addresses.
`unsigned short rmod;`	This occupies the lower two bytes of the following word.
`} structure;`	Two bytes are left unused.

This can be represented graphically as follows:

```
          Byte 0             1            2            3

Word
0              hid[0]        hid[1]       hid[2]       hid[3]
1              hid[4]        hid[5]       hid[6]       hid[7]
2              <--- inuse    --->         flags1       flags2
3              <------------- tkey ----------------------->
4              <---  tfil    --->         <--- unused  --->
5              <------------- npos ----------------------->
6              <---  kmod    --->         <---  kbhz   --->
7              <---  rmod    --->         <--- unused  --->
```

Example 2 (structuring on a 32–bit processor):

If the structure fields are reordered, by moving `tfil` so that it is no longer word aligned, then a more efficient packing could be obtained:

```
struct d {
  char hid[8];
  unsigned short inuse;
  char flags1;
  char flags2;
  unsigned long tkey;
  long npos;
  unsigned short kmod;
  unsigned short kbhz;
  unsigned short rmod;
  unsigned short tfil;
} structure;
```

this would give the following:

```
         Byte  0           1           2           3

Word
0              hid[0]      hid[1]      hid[2]      hid[3]
1              hid[4]      hid[5]      hid[6]      hid[7]
2              <--  inuse  -->         flags1      flags2
3              <-------------  tkey  ------------------->
4              <-------------  npos  ------------------->
5              <--  kmod   -->         <--  kbhz   -->
6              <--  rmod   -->         <--  tfil   -->
```

Note: the INMOS C compiler will generate more efficient code to load a `short` if it is word-aligned, so this new packing means that more code will be needed to access `tfil`, as it is no longer word-aligned. (Again, this is very dependant upon the way the INMOS ANSI C compiler currently handles structures.)

A general rule for obtaining the smallest structure size possible, is to order the fields in increasing order of size.

6.1.4 Unions

Each field of a union is represented as it would be if it was not in a union. A union is padded at the high-end address to the next word boundary: the padding has no defined value.

6.2 Type conversions

6.2.1 Integers

The result of converting an unsigned integer, **u**, to a signed integer, **s**, of equal length, if the value cannot be represented, is calculated as follows:

If max.s is the largest number that can be represented in the signed type then:

$$result = u - 2(max.s + 1)$$

An integer is converted to a shorter signed integer, by first converting it to an unsigned integer of the same length as the shorter signed integer (by taking the nonnegative remainder on division by the number one greater than the largest unsigned number that can be represented in the type with smaller size), and then converting to the corresponding signed integer, as described above.

6.2.2 Floating point

When converting an integral number to a floating-point number that cannot exactly represent the original value, the IEEE 754 'Round to Nearest' rounding mode is used.

When converting a floating-point number to a narrower floating-point number, the IEEE 754 'Round to Nearest' rounding mode is used.

When converting a floating–point number to an integral number, the IEEE 754 'Round to Zero' rounding mode is used; this is mandated by the ANSI standard.

6.3 Compiler diagnostics

Diagnostics are generated at severity levels: *Information; Warning*; *Error*, *Serious*; and *Fatal*. All compiler messages are generated in standard toolset format (see section A.7 of the *ANSI C Toolset Reference Manual*).

6.4 Environment

6.4.1 Arguments to `main`

The interface to main is as follows:

```
#include <channel.h>

int main(int argc, char *argv[], char *envp[],
         Channel *in[], int inlen,
         Channel *out[], int outlen);
```

where: `int argc` is the number of arguments passed to the program from the environment, including the program name.

`char *argv[]` is an array of pointers to the passed arguments.

`char *envp[]` is an array of pointers for the `getenv` function. In this implementation it is set to NULL.

`Channel *in[]` is an array of input channels.

`int inlen` is the number of elements in the input channel array.

`Channel *out[]` is an array of output channels.

`int outlen` is the number of elements in the output channel array.

If there is no server interface, then the number of arguments, `argc`, is set to one, and `argv` points to an array of two elements; `argv[0]` is a pointer to the null string (`""`); and `argv[1]` is `NULL`.

The value of `envp` is always `NULL` in order to retain compatibility with previous releases of the toolset e.g. the D711, D611 and D511 products.

The in and out arrays are set up differently depending on which linker startup file is used:

Configured case:

When the program is configured, either the linker startup file `cstartup.lnk` is used to harness the full runtime system, or `cstartrd.lnk` is used to harness the reduced runtime system. In either case the passing of the `in` and `out` arrays to `main()` is not supported. The values of these parameters are as follows:

`in` is set to `NULL`

`inlen` is set to `0`

`out` is set to `NULL`

`outlen` is set to `0`

Unconfigured case

In this case it is assumed that the program has been collected by `icollect` and linked with the full runtime system, by using the linker startup file `cnonconf.lnk`. The unconfigured case supports the passing of input and output channels from the configuration level to the `in` and `out` arrays in the `main()` parameter list. This is compatible with the previous release of the toolset i.e. the D7214, D6214, D5214 and D4214 products. The values of these parameters are as follows:

`in[0]` is set to `NULL`

`in[1]` FromServer channel

`out[0]` is set to `NULL`

`out[1]` ToServer channel

Note: this case may be unsupported in future releases.

6.4.2 Interactive devices

`stdin`, `stdout` and `stderr` are treated as if they are connected to an interactive device.

6.5 Identifiers

The first 250 characters in an identifier are significant.

Case distinctions are significant in an identifier with external linkage.

6.6 Source and execution character sets

The source character set comprises those characters explicitly specified in the Standard, together with all other printable ASCII characters. The execution character set comprises all 256 values 0 to 255. Values 0 to 127 represent the ASCII character set. **Note:** when the compiler command line option '`FC`' is used the execution character set comprises 128 values in the range 0 to 127.

There are eight bits in a character in the execution character set.

Each member of the source character set is a member of the ASCII character set and maps to the same member of the ASCII character set in the execution character set.

All characters and wide characters are represented in the basic execution characters set. The escape sequences not represented in the basic execution character set are the octal integer and hexadecimal integer escape sequences, whose values are defined by the Standard.

Shift states for encoding multibyte characters

There is only one shift state, which is the initial shift state as specified in the Standard. Multibyte characters do not alter the shift state.

Integer character constants

The value of an integer character constant that contains more than one character is given by:

$$\sum_{i}(\texttt{value of ith character << (8 * i)})$$

Wide character constants which contain more than one multibyte character are disallowed.

Locale used to convert multibyte characters

The only locale supported to convert multibyte characters into corresponding wide characters (codes) for a wide character constant is the 'C' locale.

Plain chars

By default a "plain" char has the same range of values as `unsigned char`. However, if the compiler command line option `FC` is used, a "plain" char has the same range of values as a `signed char`.

6.7 Integer operations

Bitwise operations on signed integers

Signed integers are represented in twos complement form. The bitwise operations operate on this twos complement representation.

Sign of the remainder on integer division

The remainder on integer division takes the same sign as the divisor.

Right shifts on negative–valued signed integral types

Signed integers are represented in twos complement form. The default behavior of the compiler is as follows:

> The right-shift operates on this twos complement form; zero bits are shifted in at the left-hand side; thus a negative-valued signed integer, if right-shifted more than zero places, will become positive.

It is possible, using the '`FS`' command line option, to change this behavior to the following:

> The right-shift operates on this twos complement form; the sign–bit is duplicated at the left–hand side; thus a negative-valued signed integer, will remain negative.

6.8 Registers

The compiler attempts to place register variables at shorter offsets from the workspace pointer.

6.9 Enumeration types

The values of enumeration types are represented as integers.

6.10 Bit fields

A "plain" `int` bit-field is treated as an `unsigned int` bit-field.

Bit-fields are allocated low-order to high-order within an integer (i.e. the first field textually is placed in lower bits in the integer).

A bit-field cannot straddle a word boundary.

6.11 `volatile` qualifier

An access to an object that has volatile-qualified type is a 'read' from the memory location containing the object (if the object's value is required), or a 'write' to the memory location containing the object (if the object is assigned to).

If the volatile object is an array, then the access will be only to the appropriate element of the array.

If the volatile object is a structure and only a field of the structure is required, then the access will be only to the appropriate field. If the object is not an array element or structure field, then the object occupies a whole number of words, and all the words will be accessed. Otherwise, if the array element or structure field is shorter than a word, then only the appropriate bytes will be accessed.

If the object is a bit-field, then in the case of read access, the entire word containing the bit-field will be read; and in the case of write access, the entire word containing the bit-field will be first read, and then written.

Note: If the object is an array element or structure field of type short on a 32-bit transputer, or if the object is larger than two words, then the transputer block move instruction is used for the access. On some transputers, if a block move instruction is interrupted, when it resumes it may reread the same word of memory which was read immediately before the interrupt. This may cause problems with some peripheral devices.

6.12 Declarators

There is no restriction upon the number of declarators that may modify an arithmetic, structure, or union type.

6.13 Switch statement

There is no restriction upon the number of case values in a switch statement.

6.14 Preprocessing directives

Constants controlling conditional inclusion

The value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such a character constant may NOT have a negative value.

Date and time defaults

When date of translation is not available, `__DATE__` expands to

`"Jan 1 1900"`

When time of translation is not available, `__TIME__` expands to

`"00:00:00"`

6.15 Static data layout

The static data area comprises a local static area for each object file (or more specifically, each object file which uses static data) together with a module table. Figure 6.1 illustrates this.

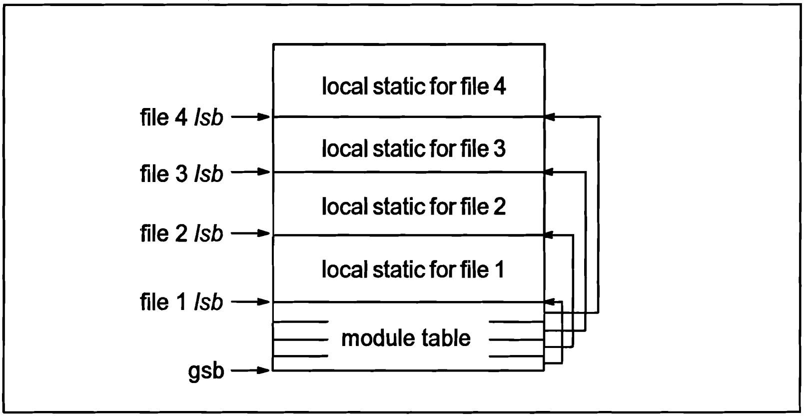

Figure 6.1 Static data layout

The module table contains an entry for every file with a local static area, which consists of a word containing a pointer to that file's local static area.

The base of the module table is called the global static base, or *gsb*.

6.15.1 Local static data layout

Usually, static data objects defined in a file are allocated space in that file's local static area. However, under certain conditions, a static data object may be placed in the text section (i.e. the section which contains the code) for that file, see section 6.15.2.

Local static data is allocated in the local static area in the same order as it appears in the source code.

The global static base (*gsb*), is passed as a hidden first parameter to every routine.

To access a piece of static data, the compiler loads the *gsb*, then does an indirect load to pick up the entry in the module table for the current file. This gives a pointer to the local static area (the local static base, or *lsb*). If the static datum required is in the local static area, it may be accessed using the *lsb*; but if it is in another file's static area, then another level of indirection is required.

If a function makes frequent access to the local static area, then the *lsb* is cached into a temporary in local workspace before the first of its uses (usually, this is on entry to the function).

6.15.2 Constant static objects

If a static data object can be guaranteed to be non–modifiable, then the C compiler is sometimes able to allocate it in the text section (i.e. the section which contains the code) for the file in which it is defined. The object must be non–modifiable, as the text section must be ROMable.

This can be useful as it can reduce the amount of memory required for that object: if the object is placed in the static data are then it must be initialized at program start–up and the value of the initializer is held in the text section. By allocating the object directly in the text section, no initializer is necessary. **Note:** that this will not reduce the size of the text section (and hence the size of the bootable file), but it will reduce the size of the static data area.

The exact conditions which must be satisfied for the object to be placed in the text section are:

- The static data object must be declared as `const`.
- The static data object must not be declared as `volatile`.
- The static data object must have an initializer.
- The initializer must contain no pointers except NULL pointers (absolute pointer values cannot be put into the text section as they are only known at run–time).
- The static data object must not be externally visible (references to external objects have to know whether the object they are referencing is in the text section or the data section).

This can be useful if a program contains a very large table of constants or constant data; for example:

```
static const char data[] = { 1, 27, 34, 52, ...
                             ...
                            ..., 5, 4, 0 }
```

will be allocated in the text section.

Note: that the conditions above require that the constant static data object must not be visible in any other files. This can be worked around by defining a pointer to the constant static object and making the pointer externally visible. For the above, we can define:

```
extern const char *datap = &data[0];
```

and then other files may access `data` indirectly through `datap`.

If it is required to ensure that a data object is *not* allocated in the text section, for instance if ROM space is limited, then it should not be declared as a `const`.

6.16 Calling conventions

6.16.1 Parameter Passing

There are two methods of parameter passing, depending upon whether or not the function involved has a type which includes a prototype.

For a function call, if the function has a type that includes a prototype, then each actual parameter is converted to the type of the corresponding formal parameter, otherwise the default argument promotions are performed on each actual parameter.

- an argument of type `char`, `short int`, `int` bit-field, or enumeration type is converted to type `int`. (Signed `int` if this will correctly represent the argument, unsigned `int` otherwise.)
- an argument of type `float` is converted to type `double`.
- arguments of all other types are unmodified.

For a function definition, if the function type does not include a prototype, then *callee narrowing* is performed upon each formal parameter: this converts it from its promoted type (as obtained by the default argument promotions) to its declared type. If the function type does include a prototype, then no type conversion is performed.

The default argument promotions are performed upon arguments forming part of a variable parameter list.

6.16.2 Calling Sequence

A pointer to the static area is normally passed as an extra parameter to every function. This parameter is called the *global static base* (gsb) and contains the address of the module table, which is at the base of the whole static area for the program.

The compiler pragma `IMS_nolink` (*f*) directs the compiler to compile the function *f* without a gsb parameter. Any direct calls to *f* within the scope of this pragma will not have a gsb included in the argument list. If the function is defined within the scope of the pragma, then it will be compiled without a gsb formal parameter (the compiler will flag a serious error if the function definition requires a gsb, for example, if it accesses static data). This pragma is provided to ease the calling of occam from C and vice versa.

The declared parameters are found in order immediately after the gsb. The type of each parameter is determined using the rules described in section 8.1 above. Each parameter occupies an integral number of words. Parameters are represented in memory exactly the same as if they had been declared as automatic, see section 6.1.

The first three words of parameters are loaded onto the integer register stack (**Areg** will contain the gsb), and are written into memory by the *call* instruction.

Parameters may be modified by the called routine. Thus after the call, they cannot be guaranteed to contain the same value as was passed in.

On entry to a function the contents of both the cpu evaluation stack and the fpu evaluation stack (if it exists) are undefined and the workspace pointer addresses the workspace containing the return address and parameters:

word offset		
		(high addresses)
`n+1`		(last parameter passed in)
	⋮	
`4`		(first `parm` stored by caller)
`3`		(**Creg** as saved by call instruction)
`2`	`parm 1` starts	(**Breg** as saved by call instruction)
`1`	static link	(**Areg** as saved by call instruction)
`0`	`Iret`	(return address if call used)
		← `Wptr`
`-1`	free	
		(top of stack)

The return value from a function is sent back in the **Areg** where possible. If the result is a scalar occupying less than a word, the value returned in **Areg** will be the value of the scalar widened to the number of bits per word.

If the return value will not fit in a register, then the caller will supply another parameter as the second actual parameter (when the user's parameters will begin in position three). This will be a pointer to an area large enough to receive the return value. This will be the case for functions returning structures which are larger than a word and for functions returning `double` values when not executing on a floating point transputer (e.g. T800), or returning `float` or `long` values on a 16 bit transputer (e.g. T225).

For transputers with an on-chip floating-point unit, floating values will be returned in **FAreg**, whether they are float or double. However, for the 32 bit, non-floating point, processors (e.g. T400), float values will be returned as unconverted bit patterns in the **Areg**; and double values returned in an area pointed to by the result pointer parameter. For 16-bit transputers, floating values are always returned via an extra parameter pointing to the return area. Structures and unions that occupy a word (and contain no fields shorter than a word) are returned in **Areg**. All other structures and unions are returned in an area pointed to by a result pointer parameter.

6.16.3 Rules for aliasing between formal parameters

The following rules cover assumptions made by the INMOS C compiler with regard to aliasing between function parameters.

1 The compiler may not assume that there are no aliases between formal parameters.

2 Where a function result is returned by assignment through a result pointer in the function body, the compiler may not assume that there are no aliases of the object referred to by the result pointer parameter.

Hence the compiler must ensure that all accesses to variables which could be potentially aliased by the result pointer have already occurred before the assignment through the result pointer.

6.17 Runtime library

The null pointer constant to which the macro NULL expands to is `(void *)0`.

Appendices

A Syntax of language extensions

This appendix defines the language extensions in the ANSI C toolset.

A.1 Notation

Syntax definitions are presented in a modified Backus-Naur Form (BNF). Briefly:

1. Terminal strings of the language – those not built up by rules of the language – are printed in teletype font e.g. `void`.
2. Each phrase definition is built up using a double colon and an equals sign to separate the two sides.
3. Alternatives are separated by vertical bars ('|').
4. Optional sequences are enclosed in square brackets ('[' and ']').
5. Items which may be repeated zero or more times appear in braces ('{' and '}').

A.2 #pragma directive

```
control-line   ::=   #pragma pragma

pragma         ::=   IMS_on ( parameter { , parameter } )
                   | IMS_off ( parameter { , parameter } )
                   | IMS_linkage ( ["name"] )
                   | IMS_nolink ( functionname )
                   | IMS_modpatchsize( n )
                   | IMS_codepatchsize( n )
                   | IMS_translate ( name, "newname " )
                   | IMS_nosideeffects ( functionname )
                   | IMS_descriptor ( functionname, language, \
                                      workspace, vectorspace, \
                                      "string" )

parameter      ::=   channel_pointers  | cp
                   | inline_ops        | il
                   | inline_string_ops | is
                   | printf_checking   | pc
```

```
| scanf_checking    | sc
| stack_checking    | sc
| warn_bad_target   | wt
| warn_deprecated   | wd
| warn_implicit     | wi
```

A.3 __asm statement

statement	::=	*asm-statement*
asm-statement	::=	`__asm` { { *asm-directive* } }
asm-directive	::=	[`size` *constant*] *primary-op constant* ;
	|	[`size` *constant*] *secondary-op* ;
	|	*pseudo-op* ;
	|	*identifier*: *asm-directive*
	|	;
primary-op	::=	*any primary instruction* (in lower case)
secondary-op	::=	*any secondary instruction* (in lower case)
pseudo-op	::=	`ld` *expression*
	|	`st` *lvalue*
	|	`ldab` *expression* , *expression*
	|	`stab` *lvalue* , *lvalue*
	|	`ldabc` *expression* , *expression* , *expression*
	|	`stabc` *lvalue* , *lvalue* , *lvalue*
	|	[`size` *constant*] `j` *label*
	|	[`size` *constant*] `cj` *label*
	|	[`size` *constant*] `call` *label*
	|	[`size` *constant*] `ldlabeldiff` *label* - *label*
	|	`byte` *constant* { , *constant* }
	|	`word` *constant* { , *constant* }
	|	`align`
lvalue	::=	*expression*
constant	::=	*constant-expression*
label	::=	*identifier*
expression	::=	as defined in X3.159–1989 ANSI standard for C
constant-expression	::=	as defined in X3.159–1989 ANSI standard for C
identifier	::=	as defined in X3.159–1989 ANSI standard for C

primary instructions and *secondary instructions* are listed in appendix A of the *ANSI C Toolset User Guide*.

B ANSI standard compliance data

This appendix lists details of the INMOS implementation of C in areas of the language where formal documentation is required by the ANSI standard. The information is provided for compliance with the standard and to provide a convenient reference point for programmers wishing to port the toolset to other hosts.

The formal ANSI requirement in each area is given followed by a reference to the appropriate section in the standards document. This is followed by a description of the INMOS implementation in that area.

Where the information required is provided in other areas of this book or the ANSI C Toolset documentation a reference is given to the appropriate section.

B.1 Translation

- **How a diagnostic is identified (§ 2.1.1.3)**

  Diagnostics are displayed to `stderr` (UNIX and VMS) or `stdout` (MS-DOS) in a standard format. The display format is described in section A.7 of the *ANSI C Toolset Reference Manual.*

B.2 Environment

- **The semantics of the arguments to main (§ 2.1.2.2.1)**

  The prototype of C `main` is as follows:

```
#include <channel.h>

int main (int argc, char *argv[], char *envp[],
          Channel *in[], int inlen,
          Channel *out[], int outlen);
```

where: `argc` is the number of arguments passed to the program from the environment, including the program name.

`*argv[]` is an array of pointers to the passed arguments.

`*envp[]` is an array of pointers for the `getenv` library function – implemented in ANSI C as NULL.

`Channel *in[]` is an array of input channels.

`int inlen` is the number of elements in the array.

`Channel *out[]` is an array of output channels.

`int outlen` is the number of elements in the array.

An extension for configured programs allows extra parameters to be passed by defining them as `interface` parameters within the configuration description. These configuration level parameters can be accessed by the C program using the runtime library function `get_param`.

- **What constitutes an interactive device (§ 2.1.2.3)**

  `stdin`, `stdout` and `stderr` are treated as if they are connected to an interactive device.

B.3 Identifiers

- **The number of significant initial characters (beyond 31) in an identifier without external linkage (§ 3.1.2).**

  The first 250 characters in the identifier are significant.

- **The number of significant initial characters (beyond 6) in an identifier with external linkage (§ 3.1.2).**

  The first 250 characters in the identifier are significant.

- **Whether case distinctions are significant in an identifier with external linkage (§ 3.1.2).**

  Case distinctions are significant in an identifier with external linkage.

B.4 Characters

- **The members of the source and execution character sets, except as explicitly specified in the Standard (§ 2.2.1).**

  The source character set comprises those characters explicitly specified in the Standard, together with all other printable ASCII characters. The execution character set comprises all 256 values 0 – 255. Values 0 – 127 represent the ASCII character set.

- **The shift states used for the encoding of multibyte characters (§ 2.2.1.2).**

  There is only one shift state, which is the initial shift state as specified in the Standard. Multibyte characters do not alter the shift state.

- **The number of bits in a character in the execution character set (§ 2.2.4.2.1).**

  There are eight bits in a character in the execution character set.

- **The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (§ 3.1.3.4).**

  Each member of the source character set is a member of the ASCII character set. It maps to the same member of the ASCII character set in the execution character set.

- **The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (§ 3.1.3.4).**

  All characters and wide characters are represented in the basic execution character set. The escape sequences not represented in the basic execution character set are the octal integer and hexadecimal integer escape sequences, whose values are defined by the Standard.

- **The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (§ 3.1.3.4).**

  See section 6.6.

- **The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (§ 3.1.3.4).**

  The only locale supported is the 'C' locale.

- **Whether a "plain" char has the same range of values as signed char or unsigned char (§ 3.2.1.1).**

  By default, a "plain" char has the same range of values as unsigned char. However, if the compiler command line option '`FC`' is used, a "plain" char has the same range of values as a signed char.

B.5 Integers

- **The representations and sets of values of the various types of integers (§ 3.1.2.5).**

  For all data-type representations see section 6.1.1 in this manual.

- **The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (§ 3.2.1.2).**

  See section 6.2.1.

- **The results of bitwise operations on signed integers (§ 3.3).**

  Signed integers are represented in twos complement form. The bitwise operations operate on this twos complement representation.

- **The sign of the remainder on integer division (§ 3.3.5).**

  The remainder on integer division takes the same sign as the divisor.

- **The result of a right shift of a negative-valued signed integral type (§ 3.3.7).**

  Signed integers are represented in twos complement form. The right-shift operates on this twos complement form.

  By default, zero bits are shifted in at the left-hand side; thus a negative-valued signed integer, if right-shifted more than zero places, will become positive.

  However, if the compiler command line option '`FS`' is used, the sign bit is duplicated at the left–hand side; thus a negative signed integer, if right–shifted more than zero places, will remain negative.

B.6 Floating point

- **The representations and sets of values of the various types of floating-point numbers (§ 3.1.2.5).**

  For all data-type representations see section 6.1.1 in this manual.

- **The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (§ 3.2.1.3).**

  When converting an integral number to a floating-point number, the IEEE 754 'Round to Nearest' rounding mode is used.

- **The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (§ 3.2.1.4).**

  When converting a floating-point number to a narrower floating-point number, the IEEE 754 'Round to Nearest' rounding mode is used.

B.7 Arrays and pointers

- **The type of integer required to hold the maximum size of an array, that is, the type of the sizeof operator, `size_t` (§ 3.3.3.4, § 4.1.1).**

The type of the sizeof operator, `size_t`, is unsigned int.

- **The result of casting a pointer to an integer or vice versa (§ 3.3.4).**

  When a pointer is cast to an integer or vice versa, and the number of bits in the integer is the same as the number of bits in the pointer, the bit representation remains unchanged.

  When an integer is cast to a pointer, and the number of bits in the integer is different from the number of bits in the pointer, the integer is first cast to type `int`, and the result of this cast is then cast to the pointer type.

  Note: A NULL pointer on a 32-bit transputer has the representation all bits zero, so that casting an integer variable of value zero to a pointer will result in a NULL pointer. However, a NULL pointer on a 16-bit transputer DOES NOT have the representation all bits zero, so that it is incorrect to assume that an integer *variable* of value zero, when cast to a pointer will result in a NULL pointer. (the ANSI standard guarantees that an integer *constant* of value zero, when cast to a pointer, will result in a NULL pointer.)

  On a 32–bit transputer, the value of the NULL pointer constant is `0`; on a 16–bit transputer, the value of the NULL pointer constant is `0x8000`.

- **The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (§ 3.3.6, § 4.1.1).**

- `int`. **Note:** that this means that it is not possible to declare an array of `char`-sized objects which is larger than half of the integer range, and take the difference of a pointer to the end and a pointer to the start. This is particularly important on a 16-bit processor, i.e. `ptrdiff_t` will not correctly represent the difference between the two ends of an array of `char`-sized objects larger than 32767 bytes.

- There is no problem with arrays of elements which are larger than `char`.

B.8 Registers

- **The extent to which objects can actually be placed in registers by use of the `register` storage-class specifier (§ 3.5.1).**

  The `register` storage class specifier is used to allocate objects at a lower offset in workspace. Objects cannot be placed in registers.

B.9 Structures, unions, enumerations, and bit–fields

- **A member of a union object is accessed using a member of a different type (§ 3.3.2.3).**

  For the implementation of unions see section 6.1.4 in this manual.

- **The padding and alignment of members of structures (§ 3.5.2.1). This should present no problem unless binary data written by one implementation are read by another.**

  For the implementation of structures see section 6.1.3 in this manual.

- **Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field (§ 3.5.2.1).**

  A "plain" int bit-field is treated as an unsigned int bit-field.

- **The order of allocation of bit-fields within an int (§ 3.5.2.1).**

  Bit-fields are allocated low-order to high-order within an int (i.e. the first field textually is placed in lower bits in the int).

- **Whether a bit-field can straddle a storage-unit boundary (§ 3.5.2.1).**

  A bit-field cannot straddle a word boundary.

- **The integer type chosen to represent the values of an enumeration type (§ 3.5.2.2).**

  The values of enumeration types are represented as `int`s.

B.10 Qualifiers

- **What constitutes an access to an object that has volatile-qualified type (§ 3.5.3).**

  An access to an object that has volatile-qualified type is a 'read' from the memory location containing the object (if the object's value is required), or a 'write' to the memory location containing the object (if the object is assigned to). If the volatile object is an array, then the access will be only to the appropriate element of the array. If the volatile object is a structure and only a field of the structure is required, then the access will be only to the appropriate field. If the object is not an array element or structure field, then the object occupies a whole number of words, and all the words will be accessed. Otherwise, if the array element or structure field is shorter than a word, then only the appropriate bytes will be accessed.

  If the object is a bit-field, then in the case of read access, the entire word containing the bit-field will be read; and in the case of write access, the entire word containing the bit-field will be first read, and then written.

  Note that if the object is an array element or structure field of type `short` on a 32-bit transputer, or if the object is larger than two words, then the transputer block move instruction is used for the access. On some transputers, if a block move instruction is interrupted, when it resumes it may reread the same word of memory which was read immediately before the interrupt. This may cause problems with some peripheral devices.

B.11 Declarators

- **The maximum number of declarators that may modify an arithmetic, structure, or union type (§ 3.5.4).**

  There is no restriction upon the number of declarators that may modify an arithmetic, structure, or union type.

B.12 Statements

- **The maximum number of case values in a switch statement (§ 3.6.4.2).**

  There is no restriction upon the number of case values in a switch statement.

B.13 Preprocessing directives

- **Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (§ 3.8.1).**

  The value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. Such a character constant may NOT have a negative value.

- **The method for locating includable source files (§ 3.8.2).**

  See section 1.4.9 in the *ANSI C Toolset Reference Manual.*

- **The support of quoted names for includable source files (§ 3.8.2).**

  See section 1.4.9 in the *ANSI C Toolset Reference Manual.*

- **The mapping of source file character sequences (§ 3.8.2).**

  See section 1.4.9 in the *ANSI C Toolset Reference Manual.*

- **The nesting limit for #include directives (§ 3.8.2).**

  There is no nesting limit for `#include` directives.

- **The behavior on each recognized `#pragma` directive (§ 3.8.6).**

  See section 1.4.11 in the *ANSI C Toolset Reference Manual.*

- **The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (§ 3.8.8).**

  When date of translation is not available, `__DATE__` expands to:

  ```
  "Jan 1 1900"
  ```

  When time of translation is not available, `__TIME__` expands to:

  ```
  "00:00:00"
  ```

B.14 Library functions

- **The null pointer constant to which the macro `NULL` expands (§ 4.1.5)** `(void *)0`

- **The diagnostic printed by and the termination behavior of the `assert` function (§ 4.2)**

  `*** assertion failed:` *condition*`,` `file` *file*`,` `line` *line*

  `assert` terminates by calling `abort`. The action of `abort` depends upon the use of the `set_abort_action` function. See the specification of `abort` in chapter 2.

- **The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint` and `isupper` functions (§ 4.3.1)**

  `isalnum` : '0'-'9' 'A'-'Z' 'a'-'z'
  `isalpha` : 'A'-'Z' 'a'-'z'
  `iscntrl` : character codes 0-31 and 127
  `islower` : 'a'-'z'
  `isprint` : character codes 32-126
  `isupper` : 'A'-'Z'

- **The values returned by the mathematics functions on domain errors (§ 4.5.1)**

  All mathematics functions return the value `0.0` on domain errors.

- **Whether the mathematics functions set the integer expression `errno` to the value of the macro `ERANGE` on underflow errors. (§ 4.5.1)**

  The maths functions do set `errno` to `ERANGE` on underflow errors.

- **Whether a domain error occurs or zero is returned when the `fmod` function has a second argument of zero. (§ 4.5.6.4)**

If the second argument to `fmod` is zero then a domain error occurs and the function returns zero.

- **The set of signals for the `signal` function (§ 4.7.1.1)** SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM, SIGIO, SIGURG, SIGPIPE, SIGSYS, SIGALRM, SIGWINCH, SIGLOST, SIGUSR1, SIGUSR2, SIGUSR3.

- **The semantics for each signal recognized by the `signal` function (§ 4.7.1.1)**

SIGABRT	Abnormal termination, such as initiated by the `abort` function.
SIGFPE	Erroneous arithmetic operation, such as zero divide or an operation resulting in overflow.
SIGILL	Detection of an invalid function image, such as an illegal instruction.
SIGINT	Receipt of an interactive attention signal.
SIGSEGV	Invalid access to storage.
SIGTERM	Termination request sent to the program.
SIGIO	Input/output possible.
SIGURG	Urgent condition on IO channel.
SIGPIPE	Write on pipe with no-one to read.
SIGSYS	Bad argument to system call.
SIGALRM	Alarm clock.
SIGWINCH	Window changed.
SIGLOST	Resource lost.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.
SIGUSR3	User-defined signal 3.

- **The default handling and the handling at program startup for each signal recognized by the `signal` function. (§ 4.7.1.1)**

The handling at program startup is identical to the default handling, which is as follows:

SIGABRT	The action of SIGABRT depends upon the `set_abort_action` function. See the specification of `abort` in chapter 2.
SIGFPE	No action.
SIGILL	No action.
SIGINT	No action.

`SIGSEGV`	No action.
`SIGTERM`	Terminate the program via a call of `exit` with the parameter `EXIT_FAILURE`.
`SIGIO`	No action.
`SIGURG`	No action.
`SIGPIPE`	No action.
`SIGSYS`	No action.
`SIGALRM`	No action.
`SIGWINCH`	No action.
`SIGLOST`	No action.
`SIGUSR1`	No action.
`SIGUSR2`	No action.
`SIGUSR3`	No action.

- **If the equivalent of `signal(sig, SIG_DFL);` is not executed prior to the call of a signal handler, the blocking of the signal that is performed (§ 4.7.1.1)**

  The equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler.

- **Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the signal function (§ 4.7.1.1)**

  The default handling is reset if the `SIGILL` signal is received.

- **Whether the last line of a text stream requires a terminating newline character. (§ 4.9.2)**

  The last line of a text stream does not require a terminating newline character.

- **Whether space characters that are written out to a text stream immediately before a newline character appear when read in. (§ 4.9.2)**

  Space characters written out to a text stream immediately before a newline character do appear when read in.

- **The number of null characters that may be appended to data written to a binary stream. (§ 4.9.2)**

  No null characters are appended to data written to a binary stream.

- **Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file. (§ 4.9.3)**

  The file position indicator of an append mode stream is initially positioned at the end of the file.

- **Whether a write on a text stream causes the associated file to be truncated beyond that point. (§ 4.9.3)**

  A write on a text stream will not cause the associated file to be truncated beyond that point.

- **The characteristics of file buffering. (§ 4.9.3)**

  When a stream is unbuffered characters appear from the source or destination as soon as possible.

  When a stream is line buffered characters are transmitted to and from the host environment as a block when a newline character is encountered.

  When a stream is fully buffered characters are transmitted to and from the host environment as a block when a buffer is filled.

  In all buffering modes characters are transmitted when the buffer is full and when input is requested on an unbuffered or line buffered stream, or when the stream is explicitly flushed.

  See also section 1.3.12.

- **Whether a zero length file actually exists (§ 4.9.3)**

  The library can support a zero length file if it is permitted on the host environment.

- **The rules for composing valid file names. (§ 4.9.3)**

  The rules for composing valid file names are the same as those found on the host system.

- **Whether the same file can be opened multiple times. (§ 4.9.3)**

  Although the system will allow a file to be opened multiple times the *icc* `stdio` library has no support for shared access to a single file and so unexpected results may occur if this is attempted.

- **The effect of the `remove` function on an open file. (§ 4.9.4.1)**

  The `remove` function will delete an open file only if this is permitted on the host system.

- **The effect if a file with the new name exists prior to the call to the `rename` function. (§ 4.9.4.2)**

  The rename will cause an existing file with the new name to be overwritten only if this is permitted on the host system.

- **The output for `%p` conversion in the `fprintf` function. (§ 4.9.6.1)**

  The output for the `%p` function is a hexadecimal number.

- **The input for the `%p` conversion in the `fscanf` function. (§ 4.9.6.2)**

  The input for the `%p` conversion is a hexadecimal number.

- **The interpretation of a - character that is neither the first nor the last character in the scanlist for `%[` conversion in the `fscanf` function. (§ 4.9.6.2)**

  A - character is treated in the same manner as all other characters no matter where it appears in the scan set.

- **The value to which the macro `errno`is set by the `fgetpos` or `ftell` function on failure. (§ 4.9.9.1, § 4.9.9.4)**

  `errno` is set to the value `EFILPOS` by the `ftell` or `fgetpos` function on failure.

- **The messages generated by the `perror` function. (§ 4.9.10.4)**

Value of `errno`	**Message**
0 (zero)	**No error (errno = 0)**
`EDOM`	**EDOM - function argument out of range**
`ERANGE`	**ERANGE - function result not representable**
`ESIGNUM`	**ESIGNUM - illegal signal number to signal()**
`EIO`	**EIO - error in low level server I/O**
`EFILPOS`	**EFILPOS - error in file positioning functions**
default	**Error code (errno)** ***errno*** **has no associated message**

- **The behavior of the `calloc`, `malloc`, or `realloc` function if the size requested is zero. (§ 4.10.3)**

  If the size requested is zero in `calloc` or `malloc` then no action is taken and and the functions return `NULL`.

  If the size requested is zero in `realloc` and the pointer parameter is `NULL` then no action is taken and the function returns `NULL`. The case where size is zero and the pointer is not a `NULL` pointer is defined by the ANSI standard.

- **The behavior of the `abort` function with regard to open and temporary files. (§ 4.10.4.1)**

  The `abort` function will cause termination without closing open files or removing temporary files. Note that the behavior of `abort` may be altered by `set_abort_action` (see specification of the function in chapter 2) but whichever behavior is selected, open files will not be closed, and temporary files will not be removed.

- **The status returned by the `exit` function if the value of the argument is other than zero, `EXIT_SUCCESS`, or `EXIT_FAILURE`. (§ 4.10.4.3)**

  The status returned by the `exit` function in this case is the numerical value of the argument.

- **The set of environment names and the method for altering the environment list used by the `getenv` function.(§ 4.10.4.4)**

  The set of environment names is defined by the host system.

  The method of altering the environment list on a given system is particular to the server executing on that system. (Or, more accurately, particular to the compiler with which the server was compiled).

- **The contents and mode of execution of the string by the `system` function. (§ 4.10.4.5)**

  The string shall contain any of the commands which can be supported by the host operating system. Care should be taken so that no commands are issued which would cause the transputer to be booted, thereby overwriting the program which executed the system call. The mode of execution is defined by the host system.

- **The contents of the error message strings returned by the `strerror` function. (§ 4.11.6.2)**

  These are identical to the messages printed by the `perror` function. See above.

- **The local time zone and Daylight Saving Time. (§ 4.12.1)**

  The local time zone is defined by the host system. Daylight Saving Time information is unavailable.

- **The era for the `clock` function. (§ 4.12.2.1)**

  The era for the `clock` function extends from directly before the users main function is called until program termination.

B.15 Locale–specific behavior

- **The content of the execution character set, in addition to the required members. (§ 2.2.1)**

  The execution character set comprises all 256 values 0 – 255. Values 0 – 127 represent the ASCII character set.

- **The direction of printing. (§ 2.2.2)**

  Printing is from left to right.

- **The decimal–point character. (§ 4.1.1)**

  The decimal point is '`.`'.

- **The implementation defined aspects of character testing and case mapping functions (§ 4.3)**

  The only locale supported is "C" and so there are no implementation defined aspects of character testing or case mapping functions.

- **The collation sequence of the execution character set. (§ 4.11.4.4)**

  Only the C locale is supported and so the collation sequence of the execution character set is the same as for plain ASCII.

- **The formats for time and date (§ 4.12.3.5)**

  All the day and month names are in English.

date and time format:	`Thu Nov 9 15:42:39 1989`
date format:	`Thu Nov 9, 1989`
time format:	`15:42:39`

C CRC Résumé

This appendix provides a résumé of the CRC functions supplied with the toolset. Brief descriptions of each function are also given in chapter 2.

C.1 Summary of functions

The following CRC functions are provided:

```
int CrcWord (int data,
             int crc_in,
             int generator); – Calculates the CRC of an integer.
```

```
int CrcByte (int data,
             int crc_in,
             int generator); – Calculates the CRC of the most
                               significant byte of an integer.
```

```
int CrcFromLsb (const char *string,
                size_t length,
                int generator,
                int old_crc); – Calculates the CRC of a byte
                                sequence starting at the least
                                significant bit.
```

```
int CrcFromMsb (const char *string,
                size_t length,
                int generator,
                int old_crc); – Calculates the CRC of a byte
                                sequence starting at the most
                                significant bit.
```

C.2 Cyclic redundancy polynomials

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 $P(x) = x^7 + x^5 + x^2$.

`CrcWord` and `CrcByte` calculate the remainder of the modulo 2 polynomial division:

$$(x^n H(x) + F(x))/G(x)$$

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 `crc_in`

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

(`crc_in` 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.)

`CrcFromMsb` and `CrcFromLsb` calculate 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.

`CrcFromMsb` and `CrcFromLsb` calculate the remainder of the modulo 2 polynomial division:

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

where: $F(x)$ corresponds to `string[]`

$G(x)$ corresponds to `generator`

$H(x)$ corresponds to `old_crc`

k is the number of bits in `string[]`

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

(`old_crc` 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.).

C.2.1 Format of result

When representing $G(x)$ in the word `generator`, note that there is an implied bit set to 1 before the msb 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 implicit. 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 and for:

`CrcWord`, the least significant 16 bits of `crc_in` 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.

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

`CrcFromMsb`, the least significant 16 bits of `old_crc` form the initial CRC value; the calculated CRC is the most significant 16 bits of the result from `CrcFromMsb`.

`CrcFromLsb`, the least significant 16 bits of `old_crc` form the initial CRC value; the calculated CRC is the least significant 16 bits of the result from `CrcFromLsb`.

C.3 Notes on the use of the CRC functions

1 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:

$$(x^k H(x) + F(x))/G(x)$$

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` which shift the numerator by x^n.

2 The `CrcFromMsb` function is intended for byte sequences in normal transputer format (little-endian). The most significant bit of the given string is taken to be bit-16 or bit-32, depending, that is, on the word size of the processor, of `string[length - 1]`. `generator`, `old_crc` and the result of `CrcFromMsb` are all also in normal transputer format (little-endian).

3 The `CrcFromLsb` function is provided to accommodate byte sequences in big-endian format. The most significant bit of `string` is taken to be bit 0 of `string[0]`. The generated CRC is given in big-endian format. `generator` and `old_crc` are taken to be in little-endian format.

C.4 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 sequence from `(string +4)` to `(string+(4+message_length))`, where there are `message_length` bytes in the message. Both the transmitter and receiver use the same 32-bit generating polynomial and `old_crc` 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 `string` and the message is sent. The receiver can either:

> give the received `string` (which is `(message_length + 4)` bytes long) to `CrcFromMsb` and expect a result of zero,
>
> or
>
> give the received `(string + 4)` to `CrcFromMsb` and check that the result is equal to the `int` contained in the first four bytes of the received `string`.

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.

Index

Symbols

D

E

G

H

L