The XGC Libraries

C/C++ Libraries for Real-Time Applications

www.xgc.com

The XGC Libraries

C/C++ Libraries for Real-Time Applications

Order Number: XGC-LG-011022

XGC Technology

London
UK
Web: <www.xgc.com>

The XGC Libraries: C/C++ Libraries for Real-Time Applications

by Chris Nettleton

Publication date October 2001

Abstract

This document includes information for users of the XGC software development systems.

Acknowledgements

The text of this manual is based on the relevant ANSI C and real-time POSIX standards, and is customized to conform to the libraries as supplied.

The XGC libraries were developed under European Space Agency contract 11935/NL/JG and are maintained by XGC Technology.

Contents

Tables

Examples

Preface

The XGC Libraries provide a collection of functions that conform to the ANSI C and real-time POSIX standards. They offer both the minimal functionality required to start and run a program on the target computer, as well as functions to support input-output, multi-tasking, interrupts, dynamic memory, string operations, math and diagnostics.

1. About This Manual

This manual is based on the text of the relevant ANSI and POSIX standards.

2. Audience

This manual is written for the experienced programmer who is already familiar with the C programming language and with embedded systems programming in general. We assume some knowledge of the target computer architectures and their limitations.

3. Reader's Comments

We welcome any comments and suggestions you have on this and other XGC manuals.

You can send your comments in the following ways:

- Email: readers-comments@xgc.com

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number is printed on the title page of this book.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of the software that you are using.

Technical support enquiries should be directed to the XGC web site, http://www.xgc.com/.

4. Conventions

This document uses the following typographic conventions:

`%, $`
: A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bash shell.

`#`
: A number sign represents the superuser prompt.

`bash$ vi hello.c`
: Boldface type in interactive examples indicates typed user input.

`file`
: Italic (slanted) type indicates variable values, place-holders, and function argument names.

[|], {|}
: In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.

...
: In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.

cat(1)
: A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Mb/s
: This symbol indicates megabits per second.

MB/s
: This symbol indicates megabytes per second.

Ctrl+**x**
: This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows. In examples, this key combination is printed in bold (for example, **Ctrl**+**C**).

Chapter 1 *Introduction*

XGC is a *conforming free standing* implementation of ANSI C as specified in Section 4 of the ANSI C Standard. This means there is no host environment and the means for starting a program, the effect of program termination and the library facilities are implementation defined.

The standard header files `<float.h>`, `<limits.h>`, `<stdarg.h>` and `<stddef.h>` are supported as defined by ANSI C.

The remaining standard header files `<assert.h>` , `<ctype.h>` , `<errno.h>` , `<locale.h>` , `<math.h>` , `<setjmp.h>`, `<signal.h>`, `<stdio.h>` , `<stdlib.h>` , `<string.h>` and `<time.h>` are also supported, but only to the extent that makes sense for a program running on an embedded target computer. In particular, the functions from `<stdio.h>` that work with files other than the standard files, are not supported.

The XGC run-time system and libraries comprise the following components:

- The ANSI C header files.
- The start file `crt0`, which is also called `art0`.

- The ANSI C library `libc.a`.
- The ANSI C math library `libm.a`.
- The POSIX Threads library `libpthread.a`.
- Several implementation-defined header files and library functions.

The library is written in C, using assembly language where appropriate. Programs written in other languages, such as Ada 95 may also use the libraries by including the appropriate declarations or bindings.

1.1. Program Startup

The entry point is in the run-time system module *crt0*, which initializes the processor and the high level language environment before executing any static constructors and calling the application program main function.

The function *main* can be defined with no parameters, as follows:

Example 1.1. Function main

```
int
main (void)
{
  /* ... */

  return 0;
}
```

Note that the return statement may be omitted, in which case the compiler assumes a return value of zero.

The main function may also be defined with two parameters referred to here as `argc` and `argv`, though any names may be used, as they are local to the function in which they are declared.

Example 1.2. Function main with arguments

```
int
main (int argc, char *argv [])
{
  /* ... */
}
```

The main function is always called with argc = 0. The value of argv is undefined and will not necessarily be a valid address.

1.2. Program Termination

A return from the main program does not call the exit function, but simply returns to the start file `crt0`. In the standard version of `crt0` a return from main is followed by code to execute any static destructors and a jump to the program entry point, where the program will restart.

Note that crt0 may be customized to offer more appropriate behavior for the application.

1.3. Standard Headers

Each library function is declared in a *header* whose contents are made available by the `#include` directive. The header declares a set of related functions plus any necessary types and macros required. All the header files are compatible with C++.

The ANSI C standard headers are:

`<assert.h>`	`<limits.h>`	`<signal.h>`	`<stdlib.h>`
`<ctype.h>`	`<locale.h>`	`<stdarg.h>`	`<string.h>`
`<errno.h>`	`<math.h>`	`<stddef.h>`	`<time.h>`
`<float.h>`	`<setjmp.h>`	`<stdio.h>`	

The real-time POSIX header is `<pthread.h>`.

XGC also supports the following implementation-defined header.

`<report.h>`

1.4. Errors <errno.h>

The header <errno.h> defines several macros all relating to the reporting of error conditions.

The macros required by ANSI C are:

```
EDOM
ERANGE
```

which expand into integral constant expressions with distinct non-zero values, suitable for use in #if pre-processing directives. XGC also includes the following error definition macros:

```
ENOSYS
EIO
EBADF
EINVAL
ENODEV
ENOMEM
EBUSY
```

The variable errno is declared in the library libc.a and may be set and tested at any time.

Note The POSIX Threads library ensures each thread has a private copy of errno.

Note Signal handlers and interrupt handlers do not have a private copy of errno.

1.5. Limits <float.h> and <limits.h>

The headers <float.h> and <limits.h> define several macros that expand to various limits and parameters.

The values in <float.h> are as follows:

Table 1.1. Values in float.h

Macro Name	M1750 Format	IEEE Format
FLT_RADIX	2	2
FLT_ROUNDS	0	0

Macro Name	M1750 Format	IEEE Format
FLT_DIG	6	6
FLT_EPSILON	2.38418595e-7	1.19209290e-07
FLT_MANT_DIG	23	24
FLT_MAX	1.7014116e38	3.40282347e+38
FLT_MAX_10_EXP	38	38
FLT_MAX_EXP	126	128
FLT_MIN	1.46936794e-39	1.17549435e-38
FLT_MIN_10_EXP	-39	-37
FLT_MIN_EXP	-129	-125
DBL_DIG	11	15
DBL_EPSILON	3.637978807092e-12	2.220446049250313e-16
DBL_MANT_DIG	39	53
DBL_MAX	1.701411834602e38	1.797693134862315e+308
DBL_MAX_10_EXP	38	308
DBL_MAX_EXP	126	1024
DBL_MIN	1.469367938528e-39	2.225073858507201e-308
DBL_MIN_10_EXP	-39	-307
DBL_MIN_EXP	-129	-1021

The values in `<limits.h>` are as follows:

Table 1.2. Values in limits.h

Macro Name	16-Bit Targets	32-Bit Targets
CHAR_BIT	16	8
CHAR_MAX	32767	127
CHAR_MIN	-32768	-128
INT_MAX	32767	2147483647
INT_MIN	-32768	-2147483648
LONG_MAX	2147483647	2147483647
LONG_MIN	-2147483648	-2147483648
MB_LEN_MAX	1	1
SCHAR_MAX	32767	127
SCHAR_MIN	-32768	-128

Macro Name	16-Bit Targets	32-Bit Targets
SHRT_MAX	32767	32767
SHRT_MIN	-32768	-32768
UCHAR_MAX	65535	255
UINT_MAX	65535	4294967295
ULONG_MAX	4294967295	4294967295
USHRT_MAX	65535	65535

See the source files for more information.

1.6. Common definitions <stddef.h>

The following types and macros are defined in the standard header `<stddef.h>`. Some are also defined in other headers, as noted in their respective sub clauses.

The types are:

`ptrdiff_t`

which is the signed integral type of the result of subtracting two pointers,

`size_t`

which is the unsigned integral type of the result of the `sizeof` operator: and

`wchar_t`

which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales: the null character shall have the value zero and each member of the basic character set defined in ANSI C 5.2.1 shall have a code value equal equal to its value when used as a lone character in an integer character constant.

The macros are:

`NULL`

which expands to an implementation-defined null pointer constant: and

```
offset_of(type, member-designator)
```

which expands to an integral constant expression that has type `size_t`, the value of which is the offset in bytes, to the structure member (designated by *member-designator*), from the beginning of its structure (designated by *type*). The *member-designator* shall be such that given `static` *type* `t`; then the expression `&(t.`*member-designator*`)` evaluates to an address constant. (If the specified member is a bit field, the behavior is undefined.)

Chapter 2

Diagnostics <assert.h>

The header <assert.h> defines the assert macro and refers to another macro.

NDEBUG

which is not defined by <assert.h>. If NDEBUG is defined as a macro name at the point in the source file where <assert.h> is included, the assert macro is defined simply as

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

The assert macro is implemented as a macro, not as an actual function. If the macro definition is suppressed in order to access an actual function, the behavior is undefined.

2.1. Program Diagnostics

The assert macro

assert

Synopsis

```
#include <assert.h>

void assert (expression);

int expression;
```

Description

The `assert` macro puts diagnostics into programs. When it is executed, if `expression` is false (that is, it compares equal to zero), the `assert` macro writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number–the latter are respectively the values of the preprocessing macros `__FILE__` and `__LINE__`) on the standard error file in an implementation-defined format. It then calls the *abort* function.

Returns

The `assert` macro returns no value.

See Also

The abort function [72]

Implementation Notes

The `assert` macro calls the library function `_assert`.

Chapter 3

Character Handling <ctype.h>

The header `<ctype.h>` declares several functions useful for testing and mapping characters. In all cases the argument is an `int`, the value of which shall be representable as an `unsigned char` or shall be equal to the value of the macro `EOF`. If the argument has any other value, the behavior is undefined.

The behavior of these functions is effected by the current locale[1]. Those functions that have implementation-defined aspects only when not in the "`C`" locale are noted below.

The term *printing character* refers to a member of an implementation-defined set of characters, each of which occupies one printing position on a display device: the term *control character* refers to a member of an implementation-defined set of characters that are not printing characters.

3.1. Implementation Notes

All the character testing functions are defined as macros that test one or more bits in a constant array indexed by the given character, which must be in the range -1 to 255.

[1]The current locale is always the `C` locale.

3.2. Character Testing Functions

The isalnum function

isalnum

Synopsis

#include <ctype.h>

```
int isalnum (c);

int c ;
```

Description

The *isalnum* function tests for any character for which `isalpha` or `isdigit` is true.

The isalpha function

isalpha

Synopsis

#include <ctype.h>

```
int isalpha (c);

int c ;
```

Description

The *isalpha* function tests for any character for which `isupper` or `islower` is true, or any character that is one of an implementation-defined set of characters for which non of `iscntrl`, `isdigit`, `ispunct`, or `isspace` is true. In the `"C"` locale, `isalpha` returns true for only the characters for which `isupper` or `islower` is true.

The iscntrl function

iscntrl

Synopsis

#include <ctype.h>

```
int iscntrl (c);

int c;
```

Description

The *iscntrl* function tests for any control character.

The isdigit function

isdigit

Synopsis

#include <ctype.h>

```
int isdigit (c);

int c;
```

Description

The *isdigit* function tests for any decimal-digit character (as defined in ANSI C refsection 5.2.1).

The isgraph function

isgraph

Synopsis

#include <ctype.h>

```
int isgraph (c);

int c;
```

Description

The *isgraph* function tests for any printing character except space (' ').

The islower function

islower

Synopsis

#include <ctype.h>

```
int islower (c);

int c;
```

Description

The *islower* function tests for any character that is an lowercase letter or is one of an implementation-defined set of characters for which none of `iscntrl`, `isdigit`, `ispunct` or `isspace` is true. In the `"C"` locale, `islower` returns true only for the characters defined as lower case letters (as defined in ANSI C refsection 5.2.1).

The isprint function

isprint

Synopsis

#include <ctype.h>

```
int isprint (c);

int c;
```

Description

The *isprint* function tests for any of the printing characters including space (`' '`).

The ispunct function

ispunct

Synopsis

#include <ctype.h>

```
int ispunct (c);

int c;
```

Description

The *ispunct* function tests for any printing character that is neither a space (`' '`) nor a character for which `isalnum` is true.

The isspace function

isspace

Synopsis

```
#include <ctype.h>

int isspace (c);

int c;
```

Description

The *isspace* function tests for any character that is a standard white space character or is one of an implementation-defined set of characters for which `isalnum` is false. The standard white space characters are the following: space (`' '`), form feed (`'\f'`), new_line (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), and vertical tab (`'\v'`). In the "`C`" locale, `isspace` returns true only for standard white space characters.

The isupper function

isupper

Synopsis

```
#include <ctype.h>

int isupper (c);

int c;
```

Description

The *isupper* function tests for any character that is an uppercase letter or is one of an implementation-defined set of characters for which none of `iscntrl`, `isdigit`, `ispunctor` `isspace` is true. In the `"C"` locale, `isupper` returns true only for the characters defined as upper case letters (as defined in ANSI C refsection 5.2.1).

The isxdigit function

isxdigit

Synopsis

#include <ctype.h>

```
int isxdigit (c);

int c;
```

Description

The *isxdigit* functions tests for any hexadecimal-digit character (as defined in ANSI C refsection 6.1.3.2).

3.3. Character Case Mapping Functions

The tolower function

tolower

Synopsis

#include <ctype.h>

```
int tolower (c);

int c;
```

Description

The *tolower* function converts an upper case letter to the corresponding lower case letter.

Returns

If the argument is a character for which `isupper` is true and there is a corresponding character for which `islower` is true, the *tolower* function returns the corresponding character; otherwise, the argument is returned unchanged.

The toupper function

toupper

Synopsis

#include <ctype.h>

```
int toupper (c);

int c;
```

Description

The *toupper* function converts an lower case letter to the corresponding upper case letter.

Returns

If the argument is a character for which `islower` is true and there is a corresponding character for which `isupper` is true, the *toupper* function returns the corresponding character; otherwise, the argument is returned unchanged.

Chapter 4

Localization <locale.h>

The header `<locale.h>` declares two functions, one type and several macros.

The type is

`struct lconv`

which contains members related to the formatting of numeric values. The structure shall contain at least the following members, in any order. The semantics of the members and their normal ranges is explained in the ANSI C specification refsection 7.4.2.1. In the “C” locale, the members shall have values specified in the comments.

```
struct lconv
{
  char *decimal_point;          /* "." */
  char *thousands_sep;          /* "" */
  char *grouping;               /* "" */
  char *int_curr_symbol;        /* "" */
  char *currency_symbol;        /* "" */
  char *mon_decimal_point;      /* "" */
  char *mon_thousands_sep;      /* "" */
```

```
  char *mon_grouping;           /* "" */
  char *positive_sign;          /* "" */
  char *negative_sign;          /* "" */
  char int_frac_digits;         /* CHAR_MAX */
  char frac_digits;             /* CHAR_MAX */
  char p_cs_precedes;           /* CHAR_MAX */
  char p_sep_by_space;          /* CHAR_MAX */
  char n_cs_precedes;           /* CHAR_MAX */
  char n_sep_by_space;          /* CHAR_MAX */
  char p_sign_posn;             /* CHAR_MAX */
  char n_sign_posn;             /* CHAR_MAX */
};
```

The macros defined are `NULL` and the following:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

The functions `setlocale` and `localeconv` are not supported in this version of the library.

Chapter 5

Mathematics <math.h>

The header `<math.h>` declares several mathematical functions and defines one macro. The functions take `double` arguments and return `double` values.

The macro is defined as

```
HUGE_VAL
```

which expands to a positive `double` expression, not necessarily representable as a `float`.

5.1. Treatment of Error Conditions

The behavior of each of these functions is defined for all representable values of its input arguments. Each function shall execute as if it were a single operation, without generating any externally visible exceptions.

For all functions a *domain error* occurs if the input argument is outside the domain over which the mathematical function is defined. The description of each function lists any required domain errors: an implementation may define additional domain errors, provided that such errors are consistent with the mathematical definition of

the function. On a domain error the function returns an implementation-defined value: the value of the macros EDOM is stored in errno.

Similarly a *range error* occurs if the result of the function cannot be represented as a double value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro HUGE_VAL with the same sign (except for the tan function) as the correct value of the function: the value of the macro ERANGE is stored in errno. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero: whether the integer expression errno acquires the value of the macro ERANGE is implementation-defined.

5.2. Notes

HUGE_VAL is defined as DBL_MAX.
XGC supports all the math functions defined in the ANSI standard. XGC does not support math functions for long double arguments. However because long double is defined to use the same machine representation as double, the ANSI standard functions may be used. Functions generally do not raise exceptions for domain errors. Instead they return a value as if the argument was at the nearest limit of the domain. For example, asin (1.5) will return pi/2. Thus by ignoring the value of errno, and by masking the interrupts for arithmetic overflow, we get a saturated arithmetic behavior. Where applicable, the maximum and RMS error values are given.

5.3. Trigonometric Functions

The acos function

acos

Synopsis

#include <math.h>

```
double acos (x);
```

```
double x ;
```

Description

The *acos* function computes the principal value of the arc cosine of x. A domain error occurs for arguments not in the range [-1, +1].

Returns

The *acos* function returns the arc cosine in the range [0, pi] radians.

Implementation Notes

For arguments < -1.0, acos returns zero and sets EDOM.
For arguments > +1.0, acos returns zero and sets EDOM.

The asin function

asin

Synopsis

#include <math.h>

```
double asin (x);

double x ;
```

Description

The *asin* function computes the principal value of the arc sine of *x*. A domain error occurs for arguments not in the range [-1, +1].

Returns

The *asin* function returns the arc sine in the range [-pi/2, +pi/2] radians.

Implementation Notes

For arguments < -1.0, asin returns -pi/2 and sets EDOM.
For arguments > +1.0, asin returns +pi/2 and sets EDOM.

The atan function

atan

Synopsis

#include <math.h>

```
double atan (x);

double x ;
```

Description

The *atan* function computes the principal value of the arc tangent of *x* .

Returns

The *atan* function returns the arc tangent in the range [-pi/2, +pi/2] radians.

See Also

The atan2 function [24]

The atan2 function

atan2

Synopsis

#include <math.h>

```
double atan2 (y, x);

double y;
double x;
```

Description

The *atan2* function computes the principal value of the arc tangent of y/x, using the signs of both arguments to determine the quadrant of the return value. A domain error may occur if both arguments are zero.

Returns

The *atan2* function returns the arc tangent of y/x, in the range [-pi, +pi] radians.

See Also

The atan function [24]

Implementation Notes

None

The cos function

cos

Synopsis

#include <math.h>

```
double cos (x);

double x ;
```

Description

The *cos* function computes the cosine of x (measured in radians).

Returns

The *cos* function returns the cosine value.

Implementation Notes

The absolute error over the range -2*pi to +2*pi is less than 2 * DBL_EPSILON.

The sin function

sin

Synopsis

#include <math.h>

```
double sin (x);

double x ;
```

Description

The *sin* function computes the sine of x (measured in radians).

Returns

The *sin* function returns the sine value.

Implementation Notes

The absolute error over the range -2*pi to +2*pi is less than 2*DBL_EPSILON.

The tan function

tan

Synopsis

#include <math.h>

```
double tan (x);

double x ;
```

Description

The *tan* function returns the tangent of x (measured in radians).

Returns

The *tan* function returns the tangent value.

Implementation Notes

Over the range -pi/4 to +pi/4, the absolute error is less than 2 * DBL_EPSILON. The absolute error increases considerably as the argument approaches pi/2, or -pi/2.
Where the argument is close to any other odd multiple of pi/2, then floating point overflow may be detected and HUGE_VAL or -HUGE_VAL will be returned. If the corresponding interrupt is unmasked then the signal SIGFPE will be raised.

5.4. Hyperbolic Functions

The cosh function

cosh

Synopsis

#include <math.h>

```
double cosh (x);

double x ;
```

Description

The *cosh* function computes the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

Returns

The *cosh* function returns the hyperbolic cosine value.

The sinh function

sinh

Synopsis

#include <math.h>

```
double sinh (x);

double x ;
```

Description

The *sinh* function computes the hyperbolic sine of x. A range error occurs if the magnitude of x is too large.

Returns

The *sinh* function returns the hyperbolic sine value.

Implementation Notes

None

The tanh function

tanh

Synopsis

#include <math.h>

```
double tanh (x);
double x ;
```

Description

The *tanh* function computes the hyperbolic tangent of x.

Returns

The *tanh* function returns the hyperbolic tangent value.

Implementation Notes

None

5.5. Exponential and Logarithmic Functions

The exp function

exp

Synopsis

#include <math.h>

```
double exp (x);
double x ;
```

Description

The *exp* function computes the exponential function of x. A range error occurs if the magnitude of x is too large.

Returns

The *exp* function returns the exponential value.

The frexp function

frexp

Synopsis

#include <math.h>

```
double frexp (value, exp);

double value;
int *exp;
```

Description

The *frexp* function breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the int object pointed to by exp.

Returns

The *frexp* function returns the value x, such that x is a double with magnitude in the interval [1/2,1] or zero, and value equals x times 2 raised to the power *exp. If value is zero, both parts of the result are zero.

See Also

The ldexp function [29]

The ldexp function

ldexp

Synopsis

#include <math.h>

```
double ldexp (x, exp);

double x;
int exp;
```

Description

The *ldexp* function multiplies a floating point number by an integral power of 2. A range error may occur.

Returns

The *ldexp* function returns the value of x times 2 raised to the power exp.

See Also

The frexp function [29]

Implementation Notes

None

The log function

log

Synopsis

#include <math.h>

```
double log (x);

double x ;
```

Description

The *log* function computes the natural logarithm of x. A domain error occurs is the argument is negative. A range error may occur if the argument is zero.

Returns

The *log* function returns the natural logarithm.

Implementation Notes

If the argument is zero, then `errno` is set to `ERANGE`, and `-HUGE_VAL` is returned.
If the argument < zero, then `errno` is set to `EDOM`, and `-HUGE_VAL` is returned.

The log10 function

log10

Synopsis

#include <math.h>

```
double log10 (x);

double x ;
```

Description

The *log10* function computes the base-ten logarithm of `x`. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

Returns

The *log10* function returns the base-ten logarithm.

Implementation Notes

The *log10* function is computed by log10 (e) * log (x).
If the argument is zero, then `errno` is set to `ERANGE`, and `-HUGE_VAL` is returned.
If the argument < zero, then `errno` is set to `EDOM`, and `-HUGE_VAL` is returned.

The modf function

modf

Synopsis

#include <math.h>

```
double modf (value, iptr);

double value;
double *iptr;
```

Description

The *modf* function breaks the argument `value` into integral and fraction parts, each of which has the same sign as the argument. It stores the integral part as a double in the object pointed to by `iptr`.

Returns

The *modf* function returns the signed fraction part of `value`.

Implementation Notes

None

5.6. Power Functions

The pow function

pow

Synopsis

#include <math.h>

```
double pow (x, y);

double x;
double y;
```

Description

The *pow* function computes `x` raised to the power `y`. A domain error occurs if `x` is negative and `y` is not an integral value. A domain error occurs if the result cannot be represented when `x` is zero and `y` is less than or equal to zero. A range error may occur.

Returns

The *pow* function returns the value of `x` raised to the power `y`.

Implementation Notes

None

The sqrt function

sqrt

Synopsis

#include <math.h>

```
double sqrt (x);

double x ;
```

Description

The *sqrt* function computes the non-negative square root of `x`. A domain error occurs if the argument is negative.

Returns

The *sqrt* function returns the value of the square root.

Implementation Notes

The *sqrt* function returns 0.0 for arguments <= 0.0.

5.7. Nearest Integer, Absolute Value and Remainder Functions

The ceil function

ceil

Synopsis

#include <math.h>

```
double ceil (x);
```

```
double x ;
```

Description

The *ceil* function computes the smallest integral value not less than x.

Returns

The *ceil* function returns the smallest integral value not less than x, expressed as a double.

See Also

The floor function [35]

Implementation Notes

None

The fabs function

fabs

Synopsis

#include <math.h>

```
double fabs (x);
```

```
double x ;
```

Description

The *fabs* function computes the absolute value of a floating-point number x.

Returns

The *fabs* function returns the absolute value of x.

Implementation Notes

The fabs function is built in, and the instructions to compute the absolute value will be generated in line. Also the library contains an fabs function for use where the address of the function is needed.

The floor function

floor

Synopsis

#include <math.h>

```
double floor (x);

double x ;
```

Description

The *floor* function computes the largest integral value not greater than x.

Returns

The *floor* function returns the largest integral value not greater than x, expressed as a double.

See Also

The ceil function [33]

Implementation Notes

None

The fmod function

fmod

Synopsis

#include <math.h>

```
double fmod (x, y);

double x;
double y;
```

Description

The *fmod* function computes the floating point remainder of x/y.

Returns

The *fmod* function returns the value `x-i*y` for some integer `i` such that, if `y` is nonzero, the result has the same sign as `x` and a magnitude less than the magnitude of `y`. If `y` is zero, whether a domain error occurs or the *fmod* function returns zero is implementation-defined.

Implementation Notes

If `y` is zero, then the floating point exception `SIGFPE` will be raised. EDOM is not set.

Chapter 6

Nonlocal Jumps <setjmp.h>

The header `<setjmp.h>` defines the macro setjmp, and declares one function and one type, for bypassing the normal function call and return discipline.

The type declared is

`jmp_buf`

which is an array type suitable for holding the information needed to restore a calling environment.

6.1. Save Calling Environment

The setjmp macro

setjmp

Synopsis

#include <setjmp.h>

```
int setjmp (env);
```

```
jmp_buf env;
```

Description

The `setjmp` macro saves its calling environment in its `jmp_buf` argument for later use by the `longjmp` function.

Returns

If the return is from a direct invocation, the `setjmp` macro returns the value zero. If the return is from a call to `longjmp` function, the `setjmp` macro returns a nonzero value.

Environmental Constraint

An invocation of the `setjmp` macro shall 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 an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement:
- the operand of a unary ! operator with the resulting expression being the entire controlling expression of a selection or iteration statement: or
- the entire expression of an expression statement (possibly cast to `void`).

Implementation Notes

If the program is running with expanded memory, then the address state is also saved in the jmp_buf argument.

6.2. Restore Calling Environment

The longjmp function

longjmp

Synopsis

#include <setjmp.h>

```
int longjmp (env, val);

jmp_buf env;
int val;
```

Description

The *longjmp* function restores the calling environment saved by the most recent invocation of the `setjmp` macro in the same invocation of the program, with the corresponding jmpbuf argument. If there has been no such invocation or if the function containing the invocation of the `setjmp` macro has terminated execution than the the behavior is undefined.

Returns

After `longjmp` is completed, program execution continues as if the corresponding invocation of the `setjmp` macro had just returned the value specified by val. The `longjmp` function cannot cause the `setjmp` macro to return the value 0: if val is 0, the `setjmp` macro returns the value 1.

Implementation Notes

In GCC-1750, if `longjmp` is used with expanded memory then the calling environment includes the address state.

Chapter 7

Signal Handling <signal.h>

The header `<signal.h>` declares a type and two functions and defines several macros for handling various signals (conditions that may be reported during program execution).

The type defined is

`sig_atomic_t`

which is the integral type of an object that can be access as an atomic entity, even in the presence of asynchronous interrupts.

The macros defined are:

`SIG_DFL`

`SIG_ERR`

`SIG_IGN`

which expand to constant expressions with distinct values that are type compatible with the second argument to and the return value of the signal function, and whose value compares unequal to the address of any declarable function; and the following, each of which expands to a positive integral constant expression that is the signal number corresponding to the specified condition.

`SIGHUP`
: Hangup (POSIX).

`SIGINT`
: Interrupt (ANSI).

`SIGQUIT`
: Quit (POSIX).

`SIGILL`
: Illegal instruction (ANSI).

`SIGABRT`
: Abort (ANSI).

`SIGTRAP`
: Trace trap (POSIX).

`SIGIOT`
: IOT trap (4.2 BSD).

`SIGEMT`
: EMT trap (4.2 BSD).

`SIGFPE`
: Floating-point exception (ANSI).

`SIGKILL`
: Kill, unblock-able (POSIX).

`SIGBUS`
: Bus error (4.2 BSD).

`SIGSEGV`
: Segmentation violation (ANSI).

`SIGSYS`
: Bad argument to system call (4.2 BSD)

`SIGPIPE`
: Broken pipe (POSIX).

`SIGALRM`
: Alarm clock (POSIX).

`SIGTerm`
: Termination (ANSI).

`SIGUSR1`
User-defined signal 1 (POSIX).

`SIGUSR2`
User-defined signal 2 (POSIX).

`SIGCHLD`
Child status has changed (POSIX).

`SIGCLD`
Same as SIGCHLD (System V).

`SIGPWR`
Power failure restart (System V).

7.1. Specify Signal Handling

The signal function

signal

Synopsis

#include <signal.h>

```
(*signal (int sig, void (*func)(int))) signal
(int);

int;
```

Description

The *signal* function chooses one of three ways in which receipt of the signal number `sig` is to subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal will occur. If the value of `func` is `SIG_IGN`, the signal will be ignored. Otherwise `func` shall point to a function to be called when the signal occurs. Such a function is called a *signal handler*.

When a signal occurs, if `func` points to a function, first the equivalent of `signal(SIG,SIG_DFL);` is executed or an implementation-defined blocking of the signal is performed. (If the value of the signal is `SIG_KILL`, whether the reset to `SIG_DFL` occurs is implementation defined.) Next the equivalent of

`(*func)(sig);` is executed. The function `func` may terminate by executing a `return` statement of by calling the `abort`, `exit` or `longjmp` function. If `func` executes a `return` statement, and the value of `sig` was `SIGFPE` or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler calls any function in the standard library other than the `signal` function itself (with a first argument of the signal number corresponding to the signal that cause the invocation of the handler) or refers to any object with status storage duration other than by assigning to a static storage duration variable of type `volatile sig_atomic_t`. Furthermore, if such a call to the *signal* function results in a `SIG_ERR` return, the value of `errno` is indeterminate.

At program startup, the equivalent of `signal(sig, SIG_IGN);` is executed for some signal selected in an implementation-defined manner, the equivalent of `signal(sig, SIG_DFL);` is executed for all other signals defined by the implementation.

The implementation shall behave as if no library function calls the *signal* function.

Returns

If the request can be honored, the *signal* function returns the value of `func` for the most recent call to `signal` for the specified signal `sig`. Otherwise a value of `SIG_ERR` is returned and a positive value is stored in `errno`.

See Also

The abort function [72]
The exit function [73]

7.2. Send Signal

The raise function

raise

Synopsis

#include <signal.h>

```
int raise (sig);

int sig;
```

Description

The *raise* function sends the signal `sig` to the executing program or partition.

Returns

The *raise* function returns zero if successful, nonzero if unsuccessful.

Chapter 8

Variable Arguments <stdarg.h>

The header `<stdarg.h>` declares a type and defines three macros, for advancing through a list of arguments whose number and types and not known to the called function when it is compiled.

A function may be called with a variable number of arguments of varying types. As described in the ANSI C Standard Section 6.7.1, its parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism, and will be designated paramN in this description.

The type declared is:

`va_list`

which is a type suitable for holding information needed by the macros `va_start`, `va_arg`, and `va_end`. If access to the varying arguments is desired, the called function shall declare an object (referred to as `ap` in this refsection) having type `va_list`. The object `ap` may be passed as an argument to another function: if that function invokes the `va_arg` macro with parameter `ap` the value of `ap` in the calling function is indeterminate and shall be passed to the `va_end` macro prior to any further reference to `ap`.

8.1. Variable Argument List Access Macros

The va_start macro

va_start

Synopsis

#include <stdarg.h>

```
void va_start (ap, paramN);

va_list ap;
paramN;
```

Description

The `va_start` macro shall be invoked before any access to the unnamed arguments.

The `va_start` macro initializes `ap` for subsequent use by `va_arg` and `va_end`.

The parameter *paramN* is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). If the parameter *paramN* is declared with the `register` storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.

Returns

The `va_start` macro returns no value.

See Also

The va_arg macro [48]
The va_end macro [49]

The va_arg macro

va_arg

Synopsis

#include <stdarg.h>

```
type va_arg (ap, type);

va_list ap;
type;
```

Description

The `va_arg` macro expands to an expression that has the type and value of the next argument in the call. The parameter `ap` shall be the same as the `va_list ap` initialized by `va_start`. Each invocation of `va_arg` modifies `ap` so that the values of successive arguments are returned in turn. The parameter *type* is a type name specified such that the type of a pointer to an object that has the specified type can be obtained by simply postfixing a * to *type*. If there is no actual next argument (as promoted according to the default argument promotions), the behavior is undefined.

Returns

The first invocation of the `va_arg` macro after that of the `va_start` macro returns the value of the argument after that specified by *paramN*. Successive invocations return values of the remaining arguments in succession.

See Also

The va_start macro [48]
The va_end macro [49]

The va_end macro

va_end

Synopsis

#include <stdarg.h>

```
void va_end (ap);

va_list ap;
```

Description

The va_end macro facilitates a normal return from the function whose variable argument list was referred to by the expansion of va_start that initialized the va_list ap. The va_end macro may modify ap so that it is no longer usable (without intervening invocation of va_start). If there is no corresponding invocation of the va_start macro, or if the va_end macro is not invoked bore the return, the behavior is undefined.

Returns

The va_end macro returns no value.

See Also

The va_start macro [48]
The va_arg macro [48]

Chapter 9

Input/output <stdio.h>

Support for <stdio.h> is limited to those function that are useful in the embedded environment. In particular, the only files available are the three standard files, stdin, stdout and stderr. Other files are not available, and functions that operate on such files are omitted.

9.1. Formatted Input/Output Functions

The printf function

printf

Synopsis

#include <stdio.h>

```
int printf (format, );

const char *format ;
... ;
```

Description

The *printf* function is supported as specified in ANSI C 7.9.6.3.

Returns

The *printf* function returns the number characters printed.

Implementation Notes

The *printf* function uses an internal buffer of 256 characters. This limits the length of the text printed by one call to `printf` to 255 characters.

The sprintf function

sprintf

Synopsis

#include <stdio.h>

```
int sprintf (s, format, );

char *s;
const char *format;
... ;
```

Description

The *sprintf* function is supported as specified in ANSI C 7.9.6.5.

Returns

The *sprintf* function returns the number of characters printed.

The vprintf function

vprintf

Synopsis

#include <stdio.h>

```
int vprintf (const, format, va_list, arg);
```

```
const;
char *format;
va_list;
arg;
```

Description

The *vprintf* function is supported as specified in ANSI C 7.9.6.8.

Returns

The *vprintf* function returns the number of characters printed.

Implementation Notes

The *vprintf* function uses an internal buffer of 256 characters. This limits the length of the text printed by one call to `vprintf` to 255 characters.

The vsprintf function

vsprintf

Synopsis

#include <stdio.h>

```
int vsprintf (s, const, format, arg);

char *s;
const;
char *format;
va_list arg;
```

Description

The *vsprintf* function is supported as specified in ANSI C 7.9.6.9.

Returns

The *vsprintf* function returns the number of characters printed.

9.2. Character Input/Output Functions

The fgetc function

fgetc

Synopsis

#include <stdio.h>

```
int fgetc (stream);

FILE *stream ;
```

Description

The *fgetc* function obtains the next character (if present) as an *unsigned char* converted to an *int*, from the input stream pointed to by *stream*, and advances the associated file position indicator for the stream (if defined).

Returns

The *fgetc* function returns the next character from the input stream pointed to by *stream*. If the stream is at end-of-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*.

Implementation Notes

The value of *stream* must be *stdin*.

The fgets function

fgets

Synopsis

#include <stdio.h>

```
int fgets (s, n, stream);

char *s;
```

```
int n;
FILE *stream;
```

Description

The *fgets* function reads at most one less than the number of characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns

The *fgets* function returns *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Implementation Notes

The value of *stream* must be *stdin*.

The fputc function

fputc

Synopsis

#include <stdio.h>

```
int fputc (c, stream);

int c ;
FILE *stream ;
```

Description

The *fputc* function is supported as specified in ANSI C 7.9.7.3.

Returns

The *fputc* function returns the character written. If a write error occurs the error indicator is set and *fputc* returns *EOF*.

Implementation Notes

The value of *stream* must be *stdout* or *stderr*.

The fputs function

fputs

Synopsis

#include <stdio.h>

```
int fputs (s, stream);

const char *s;
FILE *stream;
```

Description

The *fputs* function writes the string pointed to by *s* to the stream pointed to by *stream*. The terminating null character is not written.

Returns

The *fputs* function returns *EOF* if a write error occurs: otherwise it returns a non-negative value.

Implementation Notes

The value of *stream* must be *stdout* or *stderr*.

The getc function

getc

Synopsis

#include <stdio.h>

```
int getc (stream);

FILE *stream ;
```

Description

The *getc* function is equivalent to *fgetc*, except that if it is implemented as a macro, it may evaluate *stream* more than once, so the argument should never be an expression with side effects.

Returns

The *getc* function returns the next character from the input stream pointed to by *stream*. If the stream is at end-of-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 *getc* returns *EOF*.

Implementation Notes

The *getc* function

The getchar function

getchar

Synopsis

#include <stdio.h>

```
int getchar ();

void  ;
```

Description

The *getchar* function is equivalent to *getc* with the argument *stdin*.

Returns

The *getchar* function returns the next character from the input stream pointed to by *stdin*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and *getchar* returns *EOF*. If a read error occurs, the error indicator for the stream is set and *getchar* returns *EOF*.

Implementation Notes

The *getchar* function

The gets function

gets

Synopsis

#include <stdio.h>

```
int gets (s);

char *s ;
```

Description

The *gets* function reads character from the input stream pointed to by *stdin*, into the array pointed to by *s*, until an end-of-file is encountered or a new-line character is read. any new-line character is discarded and a null character is written immediately after the last character read into the array.

Returns

The *gets* function returns *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Implementation Notes

The *gets* function

The putc function

putc

Synopsis

#include <stdio.h>

```
int putc (c, stream);

int c ;
FILE *stream ;
```

Description

The *putc* function is supported as specified in ANSI C 7.9.7.8.

Returns

The *putc* function returns the character written. If a write error occurs, the error indicator for the stream is set and *putc* returns *EOF*.

Implementation Notes

The value of *stream* must be *stdout* or *stderr*.

The putchar function

putchar

Synopsis

#include <stdio.h>

```
int putchar (c);

int c ;
```

Description

The *putchar* function is supported as specified in ANSI C 7.9.7.9.

Returns

The *putchar* function returns the character written. If a write error occurs, the error indicator for the stream is set and *putchar* returns *EOF*.

The puts function

puts

Synopsis

#include <stdio.h>

```
int puts (s);
```

```
const char *s ;
```

Description

The *puts* function writes the string pointed to by *s* to the stream pointed to by *stdout* and appends a new-line character to the output. The terminating null character is not written.

Returns

The *puts* function returns *EOF* if a write error occurs: otherwise it returns a non-negative value.

Chapter 10

General Utilities <stdlib.h>

The header `<stdlib.h>` declares four types and several functions of general utility, and defines several macros.

The types declared are `size_t` and `wchar_t` (both described in the ANSI specification refsection 7.1.6),

`div_t`

which is a structure type that is the type of the value returned by the div function, and

`ldiv_t`

which is a structure type that is the type of the value returned by the ldiv function.

The macros defined are `NULL` (described in in the ANSI specification refsection 7.1.6),

`EXIT_FAILURE`

and

EXIT_SUCCESS, which expand to integral expressions that may be used as the argument to the exit function to return unsuccessful or successful termination status respectively, to the host environment,

RAND_MAX which expands to an integral constant expression, the value of which is the maximum value returned by the rand function, and

MB_CUR_MAX which expands to a positive integer expression whose value is the maximum number of bytes in a multi-byte character for the extended character set specified by the current locale (category LC_TYPE), and whose value is never greater than MB_LEN_MAX

10.1. String Conversion Functions

The atof function

atof

Synopsis

#include <stdlib.h>

```
double atof (iptr);

const char *iptr;
```

Description

The *atof* function converts the initial portion of the string pointed to by iptr to double. Except for the behavior on error, it is equivalent to

```
strtod (nptr, (char **)NULL)
```

Returns

The *atof* function returns the converted value as a double length floating point number.

See Also

The strtod function [64]

The atoi function

atoi

Synopsis

#include <stdlib.h>

```
int atoi (nptr);

const char *nptr;
```

Description

The *atoi* function converts the initial portion of the string pointed to by `nptr` to `int` representation. Except for the behavior in error, it is equivalent to

```
(int)strtol (nptr, (char **)NULL, 10)
```

Returns

The *atoi* function returns the converted value.

See Also

The strtol function [65]

The atol function

atol

Synopsis

#include <stdlib.h>

```
long atol (nptr);

const char *nptr;
```

Description

The *atol* function converts the initial portion of the string pointed to by `nptr` to `long int` representation. Except for the behavior in error, it is equivalent to

```
strtol (nptr, (char **)NULL, 10)
```

Returns

The *atol* function returns the converted value.

See Also

The strtol function [65]

The strtod function

strtod

Synopsis

#include <stdlib.h>

```
double strtod (nptr, endptr);

const char *nptr;
char **endptr;
```

Description

The *strtod* function converts the initial portion of the string pointed to by `nptr` to `double` representation. First it decomposes the string into three parts: an initial, possibly empty, sequence of white space characters (as specified by the `isspace` function), a subject sequence resembling a floating point constant: and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to a floating point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then a non-empty sequence of digits optionally containing a decimal-point character, then an optional exponent part as defined in the ANSI specification refsection 6.1.3.1, but no

floating suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is empty or consists of entirely white space, or if the first non-white space character is other than a sign, a digit or a decimal point.

Returns

The *strtod* function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus of minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macros `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and the value of the macro `ERANGE` is store in `errno`.

Implementation Notes

In GCC-1750, the accuracy of the result is generally better than 10 digits. In most cases, 13 decimal digits is sufficient to represent any of the 2**48 values of the M1750 extended precision floating point format. Powers of 10 within the range of the M1750 extended precision format (roughly 1.0e-38 to 1.0e+38) are converted exactly.

The strtol function

strtol

Synopsis

#include <stdlib.h>

```
long strtol (nptr, endptr, base);

const char *nptr;
char **endptr;
int base;
```

Description

The *strtol* function converts the initial portion of the string pointed to by `nptr` to long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white space characters (as specified by the The isspace function The

isspace function [15]), a subject sequence resembling an integer represented in some radix determined by the value of base, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an integer, and returns the result.

Returns

The *strtol* function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and the value of the macro ERANGE is is stored in errno.

The strtoul function

strtoul

Synopsis

#include <stdlib.h>

```
unsigned long strtoul (nptr, endptr, base);

const char *nptr;
char **endptr;
int base;
```

Description

The *strtoul* function converts the initial portion of the string pointed to by nptr to unsigned long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white space characters (as specified by the The isspace function [15], a subject sequence resembling an unsigned integer represented in some radix determined by the value of base, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

Returns

The *strtoul* function returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct

value is outside the range of representable values, `ULONG_MAX` is returned, and the value of the macro `ERANGE` is is stored in `errno`.

10.2. Pseudo-Random Sequence Generation Functions

The rand function

rand

Synopsis

#include <stdlib.h>

```
int rand ();
```

Description

The *rand* function computes a sequence of pseudo-random integers in the range `0` to `RAND_MAX`.

The implementation shall behave as if no library function calls the `rand` function.

Returns

The *rand* function returns a pseudo random integer.

See Also

The srand function [67]

Implementation Notes

The value of `RAND_MAX` is `0x7fff`.

The srand function

srand

Synopsis

#include <stdlib.h>

```
void srand (seed);
```

```
unsigned int seed;
```

Description

The *srand* function uses the argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand`. If `srand` is then called with the same seed value, the sequence of pseudo random numbers shall be repeated. If `rand` is called before any calls to `srand` have been made, the same sequence shall be generated as when `srand` is first called with a seed value of 1.

The implementation shall behave as if no library function calls the `srand` function.

Returns

The *srand* function returns no value.

Example

The following functions define a portable implementation of `rand` and `srand`.

```
unsigned long next = 1;
int rand (void)
{
  next = next * 1103515245L + 12345L;
  return (unsigned int)((next > 16) & 0x7fff;
}

void
srand (unsigned int seed)
{
 next = seed;
}
```

See Also

The rand function [67]

Implementation Notes

16-bit targets use the algorithm in the example above, and the value of RAND_MAX is 32767.

10.3. Memory Management Functions

The order and contiguity of storage allocated by `calloc`, `malloc`, and `realloc` functions is not specified.

The calloc function

calloc

Synopsis

#include <stdlib.h>

```
void *calloc (nmemb, size);

size_t nmemb;
size_t size;
```

Description

The *calloc* function allocates space for an array of `nmemb` objects, each of whose size is `size`. The space is initialized to all bits zero.

Returns

The *calloc* function returns either a null pointer or a pointer to the allocated space.

See Also

The free function [69]
The malloc function [70]
The realloc function [71]

Implementation Notes

The pointer returned is always 8-byte aligned so that objects of type `double` may be assigned.

The free function

free

Synopsis

#include <stdlib.h>

```
void free (ptr);

void *ptr;
```

Description

The free function causes the space pointed to by ptr to be deallocated that is, made available for further allocation.

Returns

The free function returns no value.

See Also

The calloc function [69]
The malloc function [70]
The realloc function [71]

Implementation Notes

None

The malloc function

malloc

Synopsis

#include <stdlib.h>

```
void *malloc (size);

size_t size;
```

Description

The *malloc* function allocates space for an object whose size is specified by `size` and whose value is indeterminate

Returns

The *malloc* function returns either a null pointer o a pointer to the allocated space.

See Also

The calloc function [69]
The free function [69]
The realloc function [71]

Implementation Notes

The pointer returned is always 8-byte aligned so that objects of type `double` may be assigned.

The realloc function

realloc

Synopsis

#include <stdlib.h>

```
void *realloc (ptr, size);

void *ptr;
size_t size;
```

Description

The *realloc* function allocates space for an object whose size is specified by `size` and whose value is indeterminate

Returns

The *realloc* function returns either a null pointer or a pointer to the allocated space.

See Also

The calloc function [69]
The free function [69]
The malloc function [70]

Implementation Notes

The pointer returned is always 8-byte aligned so that objects of type `double` may be assigned.

10.4. Communication with the Environment

The abort function

abort

Synopsis

#include <stdlib.h>

```
void abort (void);
```

```
void;
```

Description

The *abort* function causes abnormal termination to occur unless the signal `SIGABRT` is being caught and the signal handler does not return. Whether open output streams are flushed or open streams are closed or temporary files removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call `raise(SIGABRT)`.

Returns

The *abort* function cannot return to its caller.

Implementation Notes

The abort function calls `raise(SIGABRT)` and does not return.

See Also

The raise function [44]

The atexit function

atexit

Synopsis

#include <stdlib.h>

```
void atexit ();

void (*func)(void) ;
```

Description

The *atexit* function registers the function pointed to by `func`, to be called without arguments at normal program termination.

Implementation Limits

The implementation shall support the registration of at least 32 functions.

Returns

The *atexit* function returns zero if the registration succeeds, nonzero if it fails.

Implementation Notes

The XGC library supports 32 registrations.

The exit function

exit

Synopsis

#include <stdlib.h>

```
void exit (status);

int status;
```

Description

The *exit* function causes normal program termination to occur. If more than one call to the *exit* function is executed by a program, the behavior is undefined.

First, all functions registered by the *atexit* function are called, in the reverse order of their registration.

Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the *tmpfile* function are removed.

Finally, control is returned to the host environment. If the value of `status` is zero or `EXIT_SUCCESS`, an implementation-defined form of the status *successful termination* is returned. If the value of status is `EXIT_FAILURE`, and implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Returns

The *exit* function cannot return to its caller.

Implementation Notes

The host environment consists of the file crt0, which initializes the stack then calls the main program. On return from the main program, the environment restarts and runs the main program again. The XGC library does not support buffered streams or temporary files.

The system function

system

Synopsis

#include <stdlib.h>

```
int system (string);

const char *string;
```

Description

The *system* function passes the string pointed to by `string` to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer may be used for string to inquire whether a command processor exists.

Returns

If the argument is a null pointer, the *system* function returns nonzero only if a command processor is available. If the argument is not a null pointer, the system function returns an implementation-defined value.

Implementation Notes

There is no *command processor*.
If the argument is a null pointer, then zero is returned to indicate that a command processor is not available.
If the argument is not a null pointer, the `system` function returns -1, and `errno` is set to `ENOSYS`.

10.5. Searching and Sorting Utilities

The bsearch function

bsearch

Synopsis

#include <stdlib.h>

```
void *bsearch (key, base, nmemb, size,
(*compar) (const void *, const void *));

const void *key;
const void *base;
size_t nmemb;
size_t size;
int (*compar) (const void *, const void *) ;
```

Description

The *bsearch* function searches an array of `nmemb` objects, the initial element of which is pointed to by `base`, for an element than matches the object pointed to by `key`. The size of each element in the array is specified by `size`.

The comparison function pointed to by `compar` is called with two arguments that point to the key object and an array element in that order. The function shall return an integer less than, equal to, or greater than zero if the `key` object is considered respectively to be less than, equal to or greater than the array element. The array shall consist of: all the objects that compare less than, all the elements that compare equal to, and all the elements that compare greater than the `key` object, in that order.

Returns

The *bsearch* function returns a pointer to a matching element of the array, or a null pointer if no match is found. If two elements compare as equal, which element is matched is unspecified.

Implementation Notes

For the M175- target, the bsearch function cannot be used with expanded memory since it is not possible to pass the address of the compare function using the 1750's 16-bit address format.
A custom binary search function is likely to be smaller and faster than the library function. See *Sedgewick, R., Algorithms in C, ISBN 0-201-51425-7, pp 198*

10.6. Integer Arithmetic Functions

The abs function

abs

Synopsis

#include <stdlib.h>

```
int abs (j);

int j;
```

Description

The *abs* function computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.

Returns

The *abs* function returns the absolute value.

Implementation Notes

The abs function is both built-in and supplied as a library function. Usually the built-in function is used, and the appropriate instructions will be generated. If the address of the abs function is taken, then the address is the address of the library function.
On the M1750, in the special case where `j` is `-32768`, fixed point overflow will be detected by the 1750 and if the corresponding interrupt is enabled the signal `SIGFIXED_OVERFLOW` will be raised. The standard behavior may be modified in the run-time system file `crt0.s`.

The div function

div

Synopsis

#include <stdlib.h>

```
div_t div (numer, denom);

int numer;
int denom;
```

Description

The *div* function computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined: otherwise `quot * denom + rem` shall equal `numer`.

Returns

The *div* function returns a structure of type `div_t` comprising both the quotient and the remainder. The structure shall contain the following members, in either order:

```
int quot; /* quotient */
int rem;  /* remainder */
```

Implementation Notes

If `denom` is zero, or if `numer` is `INT_MIN` and `denom` is -1, then fixed point overflow is detected. If the corresponding interrupt is enabled, then `SIGFIXED_OVERFLOW` is raised.

The labs function

labs

Synopsis

#include <stdlib.h>

```
long labs (long);

long;
```

Description

The *labs* function computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.

Returns

The *labs* function returns the absolute value.

Implementation Notes

The labs function is both built-in and supplied as a library function. Usually the built-in function is used, and in this case the 1750 instruction DABS will be generated. If the address of the abs function is taken, then the address is the address of the library function.

In the special case where `j` is `LONG_MIN`, fixed point overflow will occur. If the corresponding interrupt is enabled the signal `SIGFIXED_OVERFLOW` will be raised.
The standard behavior may be modified in the run-time system file `crt0.s`.

The ldiv function

ldiv

Synopsis

#include <stdlib.h>

```
ldiv_t ldiv (numer, denom);

long numer;
long denom;
```

Description

The *ldiv* function computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is nearest to the algebraic quotient. If the result cannot be represented, the behavior is undefined: otherwise `quot * denom + rem` shall equal `numer`.

Returns

The *ldiv* function returns a structure of type `ldiv_t` comprising both the quotient and the remainder. The structure shall contain the following members, in either order:

```
long int quot; /* quotient */
long int rem;  /* remainder */
```

Implementation Notes

If `denom` is zero, or if `numer` is `LONG_MIN` and `denom` is -1, then fixed point overflow is detected. If the corresponding interrupt is enabled, then `SIGFIXED_OVERFLOW` is raised.

10.7. Multi-byte Character Functions

The mblen function

mblen

Synopsis

#include <stdlib.h>

```
int mblen (s, n);

const char *s;
size_t n;
```

Description

If s is not a null pointer, the *mblen* function determines the number of bytes contained in the multi-byte character pointed to by s. Except that the shift state of the mbtowc function is not affected, it is equivalent to

```
mbtowc((wchar_t *)0, s, n);
```

The implementation shall behave as if no library function calls the mblen function.

Returns

If s is a null pointer, the *mblen* function returns a non-zero or zero value, if multi-byte encodings, respectively, do or do not have state-dependent encodings. If s is not a null pointer, the *mblen* function either returns 0 (if s points to the null character) or returns the number of bytes that are contained in the multi-byte character (if the next n or fewer bytes form a multi-byte character), or returns -1 (if they do not form a valid multi-byte character).

See Also

The mbtowc function [81]

Implementation Notes

In GCC-1750 all characters occupy one byte (which is 16 bits) and state-dependent encodings are not supported.
If `s` is a null pointer or if `s` points to the null character then `mblen` will return zero. Otherwise if `s` points to a valid multi-byte character (n >= 1), then `mblen` will return 1. Otherwise `mblen` will return -1.

The mbtowc function

mbtowc

Synopsis

#include <stdlib.h>

```
int mbtowc (pwc, s, n);

wchar_t *pwc;
const char *s;
size_t n;
```

Description

If `s` is not a null pointer, the `mbtowc` function determines the number of bytes in the multi-byte character pointer to by `s`. It then determines the code for the value of type `wchar_t` that corresponds to the multi-byte character. I the multi-byte character is valid and `pwc` is not a null pointer then `mbtowc` stores the code in the object pointer to by `pwc`. At most `n` bytes of the array pointed to by `s` will be examined.

The implementation shall behave as if no library function calls th `mbtowc` function.

Returns

If `s` is a null pointer, `mbtowc` returns a non-zero or zero value, if multi-byte character encodings respectively do or do no have state dependent encodings. If `s` is not a null pointer the *mbtowc* function either returns zero (if `s` points to the null character), or returns the number of bytes that are contained in the converted multi-byte character (if the next `n` or fewer bytes for a valid multi-byte character), or returns -1 (if they do not form a valid multi-byte character).

In no case will the value returned be greater than n or the value of the MB_CUR_MAX macro.

See Also

The wctomb function [82]

Implementation Notes

None

The wctomb function

wctomb

Synopsis

#include <stdlib.h>

```
int wctomb (s, wchar);

char *s;
wchar_t wchar;
```

Description

The *wctomb* function determines the number of bytes needed to represent the multi-byte character corresponding to the code whose value is wchar (including any change in shift state). It store the multi-byte character representation in the array pointed to by s (if s is not a null pointer). At most MB_CUR_MAX characters are stored. If the value of wchar is zero, th wctomb function is left in the initial shift state.

The implementation shall behave as if no library function calls th wctomb function.

Returns

If s is a null pointer, wctomb returns a non-zero or zero value, if multi-byte character encodings respectively do or do no have state dependent encodings. If s is not a null pointer the *wctomb* function either returns zero (if s points to the null character), or returns the number of bytes that are contained in the converted multi-byte character (if the next n or fewer bytes for a valid multi-byte

character), or returns -1 (if the do not form a valid multi-byte character).

In no case will the value returned be greater than `n` or the value of the MB_CUR_MAX macro.

See Also

The mbtowc function [81]

Implementation Notes

None

10.8. Multi-byte String Functions

The mbstowcs function

mbstowcs

Synopsis

#include <stdlib.h>

```
size_t mbstowcs (pwcs, s, n);

wchar_t *pwcs;
const char *s;
size_t n;
```

Description

The *mbstowcs* function converts a sequence of multi-byte characters that begin the initial shift state from the array pointed to by `s` into a sequence of corresponding codes and stores not more than `n` codes into the array pointed to by `pwcs`. No multi-byte characters that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multi-byte character is converted as if by a call to the `mbtowc` function, except the shift state of the *mbtowc* function is not affected.

No more `n` elements will be modified in the array pointed to by pwcs. If copying takes place between objects that overlap, the behavior is undefined.

Returns

If an invalid multi-byte character is encountered, the `mbstowcs` function returns `(size_t) -1`. Otherwise the `mbstowcs` function returns the number of array elements modified, not including a terminating zero code, if any.

See Also

The mbtowc function [81].

Implementation Notes

The mbstowcs function behaves like strncpy.

The wcstombs function

wcstombs

Synopsis

#include <stdlib.h>

```
size_t wcstombs (s, pwcs, n);

char *s;
const wchar_t *pwcs;
size_t n;
```

Description

The *wcstombs* function converts a sequence of codes that correspond to multi-byte characters from the array pointed to by `pwcs` into a sequence of multi-byte characters that begins in the initial shift state and stores these multi-byte characters into the array pointed to by `s`, stopping if a multi-byte character would exceed the limit of `n` total bytes or if a null character is stored. Each code is converted as if by a call to the *wctomb* function, except the shift state of the *wctomb* function is not affected.

No more `n` bytes will be modified in the array pointed to by `s`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

If a code is encountered that does not correspond to a valid multi-byte character, the *wcstombs* function returns (size_t) -1. Otherwise the *wcstombs* function returns the number of bytes modified, not including a terminating null character, if any.

See Also

The mbstowcs function [83]

Implementation Notes

The wcstombs function behaves like strncpy.

Chapter 11

String Handling <string.h>

11.1. String Function Conventions

The header `<string.h>` declares one type and several functions, and defines one macro useful for manipulating arrays of character type and other objects treated as arrays of character type. The type is `size_t` and the macro is `NULL` (both described in ANSI C 7.1.6). Various methods are used for determining the lengths of the arrays, but in all cases a `char *` or `void *` argument points to the initial (lowest addressed) character of the array. If an array is accessed beyond the end of an object, the behavior is undefined.

11.2. Copying Functions

The memcpy function

memcpy

Synopsis

#include <string.h>

```
void *memcpy (s1, s2, n);

void *s1;
void *s2;
size_t n;
```

Description

The *memcpy* function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The *memcpy* function returns the value of `s1`.

See Also

The memmove function [88]

Implementation Notes

The *memcpy* function is implemented in line using the `MOV` instruction.

The memmove function

memmove

Synopsis

#include <string.h>

```
void *memmove (s1, s2, n);

void *s1;
void *s2;
size_t n;
```

Description

The *memmove* function copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` characters from the object pointed to by `s2` are first copied into a temporary array of `n` characters that does not overlap the

objects pointed to by `s1` and `s2`, and then the `n` characters from the temporary array are copied into the object pointed to by `s1`.

Returns

The *memmove* function returns the value of `s1`.

See Also

The memcpy function [87]

Implementation Notes

None

The strcpy function

strcpy

Synopsis

#include <string.h>

```
void *strcpy (s1, s2);

char *s1;
char *s2;
```

Description

The *strcpy* function copies the string pointed to by `s2` (including the terminating null character) into the array pointed to by `s1`. If copying takes place between objects that overlap then the behavior is undefined.

Returns

The *strcpy* function returns the value of `s1`.

Implementation Notes

None

The strncpy function

strncpy

Synopsis

#include <string.h>

```
void *strncpy (s1, s2, n);

char *s1;
char *s2;
size_t n;
```

Description

The *strncpy* function copies not more than `n` characters (characters that follow a null character are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The *strncpy* function returns the value of `s1`.

Implementation Notes

None

11.3. Concatenation Functions

The strcat function

strcat

Synopsis

#include <string.h>

```
void *strcat (s1, s2);

void *s1;
const char *s2;
```

Description

The *strcat* function appends a copy of the string pointed to by `s2` (including the terminating null character) to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The *strcat* function returns the value of `s1`.

Implementation Notes

None

The strncat function

strncat

Synopsis

#include <string.h>

```
void *strncat (s2, s2, n);

void *s2;
const char *s2;
size_t n;
```

Description

The *strncat* function appends not more than `n` characters (a null character and characters that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial character of `s2` overwrites the null character at the end of `s1`. A terminating null character is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The *strncat* function returns the value of `s1`.

Implementation Notes

None

11.4. Comparison Functions

The memcmp function

memcmp

Synopsis

#include <string.h>

```
void *memcmp (s1, s2, n);

void *s1;
void *s2;
size_t n;
```

Description

The *memcmp* function compares the first `n` characters of the object pointed to by `s1` to the first `n` characters of the object pointed to by `s2`.

Returns

The *memcmp* function returns an integer greater than, equal to, or less than zero, accordingly as the object pointed to by `s1` is greater than, equal to, or less than the object pointed to by `s2`.

See Also

The strcmp function [93]
The strncmp function [93]

Implementation Notes

None

The strcmp function

strcmp

Synopsis

#include <string.h>

```
void *strcmp (s1, s2);

char *s1;
char *s2;
```

Description

The *strcmp* function compares the string pointed to by `s1` to the string pointed to by `s2`.

Returns

The *strcmp* function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`.

See Also

The strncmp function [93]
The memcmp function [92]

The strncmp function

strncmp

Synopsis

#include <string.h>

```
void *strncmp (s1, s2, n);

char *s1;
char *s2;
size_t n;
```

Description

The *strncmp* function compares not more than `n` characters (characters that follow a null character are not compared) from the array pointed to by `s1` to the array pointed to by `s2`.

Returns

The *strncmp* function returns an integer greater than, equal to, or less than zero, accordingly as the possibly null-terminated array pointed to by `s1` is greater than, equal to, or less than the possibly null-terminated array pointed to by `s2`.

See Also

The strcmp function [93]

Implementation Notes

None

The strcoll function

strcoll

Synopsis

#include <string.h>

```
void *strcoll (s1, s2);

char *s1;
char *s2;
```

Description

The *strcoll* function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

Returns

The *strcoll* function returns an integer greater than, equal to, or less than zero, accordingly as the string pointed to by `s1` is greater

than, equal to, or less than the string pointed to by s2, when they are both interpreted as appropriate to the current locale.

See Also

The strcmp function [93]

Implementation Notes

The XGC library supports the "C" locale only.

The strxfrm function

strxfrm

Synopsis

#include <string.h>

```
size_t strxfrm (s1, s2, n);

char *s1;
char *s2;
size_t n;
```

Description

The *strxfrm* function transforms the string pointed to by s2 and places the resulting string into the array pointed to by s1. The transformation is such that if the strcmpfunction is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the *strcoll* function applied to the same two original strings. No more than n characters are placed into the resulting array pointed to by s1, including the terminating null character. If n is zero, s1 is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

Returns

The *strxfrm* function returns the length of the transformed string (not including the terminating null character). If the value returned is n or more, then the contents of the array pointed to be s1 are indeterminate.

Implementation Notes

In the XGC library, the strxfrm function copies the characters with no transformation.

11.5. Search Functions

The memchr function

memchr

Synopsis

#include <string.h>

```
void *memchr (s, c, n);

const void *s;
int c;
size_t n;
```

Description

The *memchr* function locates the first occurrence of `c` (converted to an `unsigned char`) in the initial `n` characters (each interpreted as an `unsigned char`) of the object pointed to by `s`.

Returns

The *memchr* function returns a pointer to the located character, or a null pointer if the character does not occur in the object.

Implementation Notes

None

The strchr function

strchr

Synopsis

#include <string.h>

```
char *strchr (s, c);

char *s;
int c;
```

Description

The *strchr* function locates the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null character is considered to be part of the string.

Returns

The *strchr* function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

Implementation Notes

None

The strcspn function

strcspn

Synopsis

#include <string.h>

```
size_t strcspn (s1, s2);

char *s1;
char *s2;
```

Description

The *strcspn* function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters *not* from the string pointed to by `s2`.

Returns

The *strcspn* function returns the length of the segment.

See Also

The strspn function [99]

Implementation Notes

None

The strpbrk function

strpbrk

Synopsis

#include <string.h>

```
size_t strpbrk (s1, s2);

char *s1;
char *s2;
```

Description

The *strpbrk* function locates the first occurrence in the string pointed to by `s1` of any character from the string pointed to by `s2`.

Returns

The *strpbrk* function returns a pointer to the character, or a null pointer if no character from `s2` occurs in `s1`.

Implementation Notes

None

The strrchr function

strrchr

Synopsis

#include <string.h>

```
char *strrchr (s, c);

char *s;
int c;
```

Description

The *strrchr* function locates the last occurrence of `c` (converted to char) in the string pointed to by `s`. The terminating null character is considered to be part of the string.

Returns

The *strrchr* function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

Implementation Notes

None

The strspn function

strspn

Synopsis

#include <string.h>

```
size_t strspn (s1, s2);

const char *s1;
const char *s2;
```

Description

The *strspn* function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2`.

Returns

The *strspn* function returns the length of the segment.

See Also

The strcspn function [97]

Implementation Notes

None

The strstr function

strstr

Synopsis

#include <string.h>

```
char *strstr (s1, s2);

char *s1;
const char *s2;
```

Description

The *strstr* function locates the first occurrence in the string pointed to by `s1` of the sequence of characters (excluding the terminating null character) in the string pointed to by `s2`.

Returns

The *strstr* function returns a pointer to the located string, or a null pointer if the string is not found. If `s2` points to a string with zero length, the function returns `s1`.

Implementation Notes

None

The strtok function

strtok

Synopsis

#include <string.h>

```
char *strtok (s1, s2);

char *s1;
const char *s2;
```

Description

A sequence of calls to the *strtok* function breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a character from the string pointed to by `s2`. The first call in the sequence has `s1` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `s2` may be different from call to call.

Returns

The *strtok* function returns a pointer to the first character of a token, or a null pointer if there is no token.

Implementation Notes

None

11.6. Miscellaneous Functions

The memset function

memset

Synopsis

#include <string.h>

```
void *memset (s, c, n);

void *s;
int c;
size_t n;
```

Description

The *memset* function copies the value of `c` (converted to an unsigned char) into each of the first `n` characters of the object pointed to by `s`.

Returns

The *memset* function returns the value of `s`.

Implementation Notes

None

The strerror function

strerror

Synopsis

#include <string.h>

```
char *strerror (errnum);

int errnum;
```

Description

The *strerror* function maps the error number in `errnum` to an error message string.

The implementation shall behave as if no library function calls the `strerror` function.

Returns

The *strerror* function returns a pointer to the string, the contents of which are implementation defined.

Implementation Notes

None

The strlen function

strlen

Synopsis

#include <string.h>

```
size_t strlen (s);

char *s;
```

Description

The *strlen* function computes the length of the string pointed to by `s`.

Returns

The *strlen* function returns the number of characters that precede the terminating null character.

Chapter 12

Date and Time <time.h>

The header `<time.h>` defines two macros, declares four types and several functions for manipulating time. Many functions deal with a *calendar time* that represents the current date (according to the Gregorian calendar) and time. Some functions deal with *local time*, which is the calendar time expressed for some specific time zone, and with *Daylight Saving Time*, which is a temporary change to the algorithm for determining local time. The local time zone and Daylight Saving Time are implementation-defined.

The macros defined are `NULL` (described in ANSI C 7.1.6); and `CLOCKS_PER_SEC` which is the number per second of the value returned by the `clock` function.

The types declared are `size_t` (described in ANSI C 7.1.6), `clock_t` and `time_t`, which are arithmetic types capable of representing times, and `struct tm` which holds components of a calendar time, called the *broken-down time*.

The structure `struct tm` is defined as follows:

```
struct tm
  {
```

```
  int tm_sec;    /* seconds after the minute [0,61] */
  int tm_min;    /* minutes after the hour [0,59]   */
  int tm_hour;   /* hour of the day [0,23]          */
  int tm_mday;   /* day of the month [1,31]         */
  int tm_mon;    /* month of the year [0,11]        */
  int tm_year;   /* years since 1900                */
  int tm_wday;   /* days since Sunday [0,6]         */
  int tm_yday;   /* day of the year [0,365]         */
  int tm_isdst;  /* Daylight Saving Time flag       */
};
```

The member `tm_isdst` is:

positive if Daylight Saving Time is in effect
zero if Daylight Saving Time is not in effect
negative if the information is not available

12.1. Time Manipulation Functions

The clock function

clock

Synopsis

#include <time.h>

```
clock_t clock (void);
```

```
void;
```

Description

The *clock* function determines the processor time used.

Returns

The *clock* function returns the implementation's best approximation to the processor time used by the program since the beginning of an implementation-defined era related only to the program invocation. To determine the time in seconds, the value returned by the `clock` function should be divided by the value of the macro `CLOCKS_PER_SEC`. If the processor time used is not available or its

value cannot be represented, then the function returns the value `(clock_t)-1`.

Implementation Notes

The value of `CLOCKS_PER_SEC` is 100 by default.
The value of the function `clock` is zero immediately after the run-time system is started or restarted.
The processor time to wall time ratio is 1:1 while the processor is running.

The difftime function

difftime

Synopsis

#include <time.h>

```
double difftime (time1, time0);

time_t time1;
time_t time0;
```

Description

The *difftime* function computes the difference between two calendar times: `time1 - time0`.

Returns

The *difftime* function returns the difference expressed in seconds as a `double`.

The mktime function

mktime

Synopsis

#include <time.h>

```
time_t mktime (timeptr);

struct tm *timeptr;
```

Description

The *mktime* function converts the broken-down time, expressed as local time, in the structure pointed to by `timeptr` into a calendar time with the same encoding as that of the values returned by the *time* function. The original values of the `tm_wday` and `tm_yday` components of the structure are ignored, and the original values of other components are not restricted to the ranges indicated above. On successful completion, the values of the `tm_wday` and `tm_yday` components of the structure are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of `tm_day` is not set until `tm_mon` and `tm_year` are determined.

Returns

The *mktime* function returns the specified calendar time encoded as a value of the type `time_t`. If the calendar time cannot be represented, the function returns the value `(time_t)-1`.

The time function

time

Synopsis

#include <time.h>

```
time_t time (timer);

time_t *timer;
```

Description

The *time* function determines the current calendar time. The encoding of the value is unspecified.

Returns

The *time* function returns the implementation's best approximation to the current calendar time. The value `(time_t)-1` is returned if the calendar time is not available. If `timer` is not a null pointer, then the return value is also assigned to the object it points to.

Implementation Notes

The current calendar time is held in the run-time system with a resolution of 1 second and a range of 2**32 seconds, or 136 years. Calendar time is reset to zero each time the run-time system is restarted, where time zero is 00:00:00 on Thursday January 1, 1970.

12.2. Time Conversion Functions

The asctime function

asctime

Synopsis

#include <time.h>

```
char *asctime (timeptr);

const struct tm *timeptr;
```

Description

The *asctime* function converts the broken-down time in the structure pointed to by `timeptr` into a string in the form:

```
Sun Sep 16 01:03:52 1973\n\0
```

using the equivalent of the following algorithm.

```
char *asctime(const struct tm *timeptr)
{
  static const char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
  };
  static const char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
  };
  static char result [26];
  sprintf(result, "%.3s %.3s %.2d %.2d:%.2d:%.2d %d\n",
          wday_name[timeptr->tm_wday],
          mon_name[timeptr->tm_mon],
```

```
          timeptr->tm_mday, timeptr->tm_hour,
          timeptr->tm_min, timeptr->tm_sec,
          1900 + timeptr->tm_year);
  return result;
}
```

Returns

The *asctime* function returns a pointer to the string.

The ctime function

ctime

Synopsis

#include <time.h>

```
struct tm *ctime (timer);

const time *timer;
```

Description

The *ctime* function converts the calendar time pointed to by `timer` into a broken-down time expressed as local time.

Returns

The *ctime* function returns the pointer returned by the `asctime` function with that broken-down time as argument.

See Also

The localtime function [111]

The gmtime function

gmtime

Synopsis

#include <time.h>

```
struct tm *gmtime (timer);

const time *timer;
```

Description

The *gmtime* function converts the calendar time pointed to by `timer` into a broken-down time expressed as *Coordinated Universal Time* (UTC).

Returns

The *gmtime* function returns a pointer to that object, or a null pointer if UTC is not available.

The localtime function

localtime

Synopsis

#include <time.h>

```
struct tm *localtime (timer);

const time *timer;
```

Description

The *localtime* function converts the calendar time pointed to by `timer` into a broken-down time expressed as local time.

Returns

The *localtime* function returns a pointer to that object.

The strftime function

strftime

Synopsis

#include <time.h>

```
size_t strftime (s, maxsize, format,
timeptr);

char *s;
size_t maxsize;
const char *format;
const struct tm *timeptr;
```

Description

The *strftime* function places characters into the array pointed to by `s` as controlled by the string pointed to by `format`. The format shall be a multi-byte character sequence, beginning and ending in its initial shift state. The `format` string consists of zero or more conversion specifiers and ordinary multi-byte characters. A conversion specifier consists of a `%` character followed by a character that determines the behavior of the conversion specifier. All ordinary multi-byte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap, the behavior is undefined. No more than `maxsize` characters are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the `LC_TYPE` category of the current locale and by the values contained in the structure pointed to by `timeptr`.

`%a`	is replaced by the locale's abbreviated weekday name (Sun)
`%A`	is replaced by the locale's full weekday name (Sunday)
`%b`	is replaced by the locale's abbreviated month name (Dec)
`%B`	is replaced by the locale's full month name (December)
`%c`	is replaced by the locale's date and time (Dec 2 06:55:15 1979)
`%d`	is replaced by the day of the month (02)
`%H`	is replaced by the hour of the 24-hour day (06)
`%I`	is replaced by the hour of the 12-hour day (06)
`%j`	is replaced by the day of the year, from 001 (335)
`%m`	is replaced by the month of the year, from 01 (12)
`%M`	is replaced by the minutes after the hour (55)
`%p`	is replaced by the locale's AM/PM indicator (AM)

%S	is replaced by the seconds after the minute (15)
%U	is replaced by the Sunday week of the year, from 00 (48)
%w	is replaced by the day of the week, from 0 for Sunday (6)
%W	is replaced by the Monday week of the year, from 00 (47)
%x	is replaced by the locale's date (Dec 2 1979)
%X	is replaced by the locale's time (06:55:15)
%y	is replaced by the year of the century, from 00 (79)
%Y	is replaced by the year (1979)
%Z	is replaced by the time zone name, if any (EST)
%%	is replaced by the percent character %

If a conversion specifier is not one of the above, the behavior is undefined.

Returns

If the total number of resulting characters including the terminating null character is not more than `maxsize`, the `strftime` function returns the number of characters placed into the array pointed to by `s` not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

Chapter 13

Test Output <report.h>

The header `<report.h>` defines several functions that are useful in test programs. These are based on similar functions in the Ada Validation suite (the ACVC Tests) in the package Report.

Briefly, a test program begins by calling the function `test`. The first parameter gives the name of the test, the second parameter gives a brief description of the test. A test ends with a call to `result`. This will print the result of the the test in a standard (and machine readable) format, and will typically report PASSED or FAILED.

Between the calls of `test` and `result`, are the tests. Any test that fails should call `failed` with a parameter giving the reason for the failure. Comments may be output using the function `comment`. Both `failed` and `comment` take a format string parameter optionally followed by a number of values to print, and are compatible with `printf`.

Example 13.1. Test Program

```
#include <report.h>
int
main ()
{
  int ans = 1 + 2;
  test ("t1", "Example test program");
  if (ans != 3)
    failed ("error in arithmetic, got %d, expected 3", ans);
  result ();
}
```

If the test passes, then the output will be as follows:

Example 13.2. Output from Test Program

```
,.,. t1 GTS Version 0.1
---- t1 Example test program.
==== t1  PASSED ============================.
```

If the test fails (by changing “1 + 2” to “2 + 2” for example), then we get the following output:

Example 13.3. Output from Failed Test Program

```
,.,. t1 XTS Version 0.2
---- t1 Example test program.
   * t1 error in arithmetic, got 4, expected 3.
**** t1  FAILED ***************************.
```

13.1. Test Support Functions

The test function

test

Synopsis

#include <report.h>

```
void test (name, description);

const char *name;
const char *description;
```

Description

The *test* function is called to initialize the test. The arguments `name` and `description` are copied into static memory, and used in the reports written by the other functions.

The comment function

comment

Synopsis

#include <report.h>

```
void comment (s, );

const char *s;
... ;
```

Description

The *comment* function is use to write comments to the test output log. Comments have a distinctive prefix.

The failed function

failed

Synopsis

#include <report.h>

```
void failed (reason, );

const char *reason;
... ;
```

Description

The *failed* function is called to indicate that a test has failed, and gives the reason for the failure.

The result function

result

Synopsis

#include <report.h>

```
void result ();

;
```

Description

The *result* function is called at the end of a test program to print the test result.

Chapter 14

POSIX Threads

The XGC library supports a subset of POSIX Threads as defined in *IEEE 1003.1c-1995*. It provides the majority of the POSIX functions for threads, mutexes, and condition variables, as well as pthread_once, thread-specific data, and cleanup handlers. There is an additional function to make a thread wait for an interrupt.

The relevant Pthread attribute functions are provided, but there are no useful Mutex attributes or Condition Variable attributes and hence no associated functions; the *attr* parameter to `pthread_mutex_init` or to `pthread_cond_init` must be null.

There is currently no `pthread_key_delete` function, nor is the `asynchronous pthread_cancel` or the associated functions provided.

Programs using Pthreads must call `pthread_init` before calling any other Pthread function. Termination of the main program will cause the termination of all threads, unless exit is made via `pthread_exit`. (But there is no need for other threads to finish with `pthread_exit`).

Scheduling is always FIFO within priority.

The POSIX functions `sleep`, `nanosleep`, and `clock_gettime` are provided, with a resolution of 0.01 seconds. The timeout parameter

of `pthread_cond_timedwait` requires an absolute time that is based on `clock_gettime`.

The implementation of signals is the ANSI C one, with just `signal` and `raise`, not the POSIX one with thread-specific masks. Synchronous interrupts, such as overflow, are mapped onto the appropriate ANSI C signal, and raised in the context of the current thread, which is the one that contains the fault.

Asynchronous interrupts can be handled by user supplied interrupt functions, connected by the `handler` function, and will run outside the context of any thread. User's handlers must save and restore the global `errno` if there is any risk of them altering it.

Alternatively, a thread can wait for one or more interrupts by using the `intwait` function, which takes a mask saying which interrupts it is waiting for, and returns a code saying which one was received. At most one thread can wait for each interrupt at any one time.

14.1. Initialization Functions

The pthread_init function

pthread_init

Synopsis

#include <pthread.h>

```
void pthread_init (void);

void;
```

Description

The `pthread_init` function must be called before any other function in the POSIX Threads library.

Implementation Notes

None

14.2. Create and Destroy Functions

The pthread_create function

pthread_create

Synopsis

#include <pthread.h>

```
int pthread_create (thread, attr, , arg);

pthread_t *thread;
const pthread_attr_t *attr;
void * (*start_routine)(void *) ;
void *arg;
```

Description

The `pthread_create` function creates a thread with the attributes specified in `attr`. If `attr` is `NULL` then the default attributes are used. The new thread starts execution in `start_routine`, which is passed the single specified argument.

Returns

If the `pthread_create` function succeeds it returns 0 and puts the new thread id into `thread`, otherwise it returns -1 and sets an error number as follows:

`EAGAIN` if there is insufficient memory to create another thread
`ENOMEM` if there is insufficient memory for the thread's stack
`EINVAL` if a value specified by `attr` is invalid

Implementation Notes

The function `pthread_create` calls `calloc` to allocate memory for the thread's data, and calls `malloc` to allocates the thread's stack.

See Also

The pthread_exit function [123]
The pthread_join function [123]

The pthread_detach function

pthread_detach

Synopsis

#include <pthread.h>

```
int pthread_detach (thread_ptr);

pthread_t *thread_ptr;
```

Description

The `pthread_detach` function marks the threads's internal data structure for deletion.

Returns

The `pthread_detach` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_equal function

pthread_equal

Synopsis

#include <pthread.h>

```
int pthread_equal (t1, t2);

pthread_t t1;
pthread_t t2;
```

Description

The `pthread_equal` function compares the two threads `t1` and `t2`.

Returns

The `pthread_equal` function returns one if the two threads are the same thread, and zero otherwise.

Implementation Notes

None

The pthread_exit function

pthread_exit

Synopsis

#include <pthread.h>

```
void pthread_exit (status);

any_t status;
```

Description

The `pthread_exit` function terminates the calling thread returning the value given by `status` to any thread that has called `pthread_join` for the calling thread.

Returns

The `pthread_exit` function returns no value.

Implementation Notes

None

The pthread_join function

pthread_join

Synopsis

#include <pthread.h>

```
int pthread_join (thread, status);
```

```
pthread_t thread;
any_t *status;
```

Description

The `pthread_join` function causes the calling thread to wait for the given thread's termination. If the parameter `status` is not null then it receives the return value of the terminating thread.

Returns

The `pthread_join` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

14.3. Scheduling Functions

The pthread_getschedparam function

pthread_getschedparam

Synopsis

#include <pthread.h>

```
int pthread_getschedparam (thread, policy,
param);

pthread_t thread;
int *policy;
struct sched_param *param;
```

Description

The `pthread_setschedparam` and `pthread_getschedparam` functions allow the scheduling policy and scheduling priority parameters to be set and retrieved for individual threads.

The `pthread_getschedparam` function retrieves the scheduling policy and scheduling priority parameters for the thread ID given by

`thread`, and then stores the values in the `policy` and `sched_priority` member of `param`, respectively.

Returns

The `pthread_getschedparam` function returns -1 and sets `errno` if there is an error.

Implementation Notes

The value for `policy` must be `SCHED_FIFO`.

See Also

The pthread_setschedparam function [125]

The pthread_setschedparam function

pthread_setschedparam

Synopsis

#include <pthread.h>

```
int pthread_setschedparam (thread, policy,
param);

pthread_t thread;
int policy;
struct sched_param *param;
```

Description

The `pthread_setschedparam` and `pthread_getschedparam` functions allow the scheduling policy and scheduling priority parameters to be set and retrieved for individual threads.

The `pthread_setschedparam` function sets the scheduling policy and related scheduling priority for the thread ID given by `thread` to the policy and associated priority provided in `policy`, and the `sched_priority` member of `param`, respectively.

Returns

The `pthread_setschedparam` function returns 1 and sets `errno` in th event of an error.

Implementation Notes

The value for `policy` must be `SCHED_FIFO`.

See Also

The pthread_getschedparam function [124]

The sched_get_priority_max function

sched_get_priority_max

Synopsis

#include <pthread.h>

```
int sched_get_priority_max (policy);

int policy;
```

Description

The `sched_get_priority_max` function returns the first value in the range of priorities for the given policy.

Returns

The `sched_get_priority_max` function returns the priority.

Implementation Notes

The range of priority is 0 .. 100. The lowest priority is 0. The highest priority is 100.
The value for `policy` must be `SCHED_FIFO`.

The sched_get_priority_min function

sched_get_priority_min

Synopsis

#include <pthread.h>

```
int sched_get_priority_min (policy);
```

```
int policy;
```

Description

The `sched_get_priority_min` function returns the first value in the range of priorities for the given policy.

Returns

The `sched_get_priority_min` function returns the priority.

Implementation Notes

The range of priority is 0 .. 100. The lowest priority is 0. The highest priority is 100.
The value for `policy` must be `SCHED_FIFO`.

The sched_yield function

sched_yield

Synopsis

#include <pthread.h>

```
int sched_yield ();

;
```

Description

The `sched_yield` function yields the processor to another thread.

Returns

The `sched_yield` function returns `0`.

Implementation Notes

None

14.4. Timing Functions

The sleep function

sleep

Synopsis

#include <pthread.h>

```
int sleep (seconds);

unsigned int seconds;
```

Description

The `sleep` function delays the execution of the calling thread by at least the given number of seconds.

Returns

The `sleep` function returns zero if successful, or -1 in the event of an error.

Implementation Notes

On 16-bit targets the maximum sleep time is 32767 seconds, or approximately 9 hours.
32-bit targets use a 31-bit signed value for sleep time.

See Also

The nanosleep function [129]

The clock_gettime function

clock_gettime

Synopsis

#include <pthread.h>

```
int clock_gettime (clock_id, tp);
```

```
int clock_id;
struct timespec *tp;
```

Description

The `clock_gettime` function gets the time from the given clock.

Returns

The `clock_gettime` function returns zero if successful and `-1` otherwise.

Implementation Notes

The value of `clock_id` must be `CLOCK_REALTIME`.

The nanosleep function

nanosleep

Synopsis

#include <pthread.h>

```
int nanosleep (rqtp, rmtp);

const struct timespec *rqtp;
struct timespec *rmtp;
```

Description

The `nanosleep` function delays the execution of the calling thread until either the time interval given by `rtqp` has elapsed or a signal is handled by the thread.

Returns

The `nanosleep` function returns zero to indicate the given time interval has elapsed. Otherwise it returns -1 to indicate that the delay has been interrupted, and sets `rmtp` to the time interval remaining.

Implementation Notes

The resolution of `nanosleep` is determined by the interrupt period of the real time clock. This is set to 10 mSec in the file *crt0*.

The maximum time interval is 2147483647.999 seconds, or approximately 68 years.

See Also

The sleep function [128]

14.5. Pthread Attribute Functions

The pthread_attr_destroy function

pthread_attr_destroy

Synopsis

#include <pthread.h>

```
int pthread_attr_destroy (attr);

pthread_attr_t *attr;
```

Description

The `pthread_attr_destroy` function destroys the given thread attribute object.

Returns

The `pthread_attr_destroy` function returns `0`.

Implementation Notes

None

The pthread_attr_getdetachstate function

pthread_attr_getdetachstate

Synopsis

#include <pthread.h>

```
int pthread_attr_getdetachstate (attr,
detachstate);
```

```
pthread_attr_t *attr;
int *detachstate;
```

Description

The `pthread_attr_getdetachstate` function gets the value of the detachstate attribute from attr object.

Returns

The `pthread_attr_getdetachstate` function returns zero if successful and `-1` otherwise.

Implementation Notes

None

The pthread_attr_getinheritsched function

pthread_attr_getinheritsched

Synopsis

#include <pthread.h>

```
int pthread_attr_getinheritsched (attr,
inherit);

pthread_attr_t *attr;
int *inherit;
```

Description

The `pthread_attr_getinheritsched` function gets the value of the inheritsched attribute.

Returns

The `pthread_attr_getinheritsched` function returns zero if successful and `-1` otherwise.

Implementation Notes

None

The pthread_attr_getschedparam function

pthread_attr_getschedparam

Synopsis

#include <pthread.h>

```
int pthread_attr_getschedparam (attr, param);

pthread_attr_t *attr;
struct sched_param *param;
```

Description

The `pthread_attr_getschedparam` function gets the value of the scheduling parameter attribute.

Returns

The `pthread_attr_getschedparam` function returns zero if successful and `-1` otherwise.

Implementation Notes

The scheduling parameter attribute consists of the thread priority.

The pthread_attr_getschedpolicy function

pthread_attr_getschedpolicy

Synopsis

#include <pthread.h>

```
int pthread_attr_getschedpolicy (attr,
policy);

pthread_attr_t *attr;
int *policy;
```

Description

The `pthread_attr_getschedpolicy` function gets the policy for the given attribute.

Returns

The `pthread_attr_getschedpolicy` function returns zero if the given attribute is valid and `-1` otherwise.

Implementation Notes

None

The pthread_attr_init function

pthread_attr_init

Synopsis

#include <pthread.h>

```
int pthread_attr_init (attr);

pthread_attr_t *attr;
```

Description

The `pthread_attr_init` function initializes a thread attribute object with default values.

Returns

The `pthread_attr_init` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

The default priority is `0`.
The default stack size is 8192 bytes on 32-bit targetrs, and 1024 words (2048 bytes) on 16-bit targets.

The pthread_attr_setdetachstate function

pthread_attr_setdetachstate

Synopsis

#include <pthread.h>

```
int pthread_attr_setdetachstate (attr,
detachstate);

pthread_attr_t *attr;
int detachstate;
```

Description

The pthread_attr_setdetachstate function sets the detachstate attribute in referenced attr object.

Returns

The pthread_attr_setdetachstate function returns zero if the call is successful, otherwise it sets errno to EINVAL and returns -1.

Implementation Notes

None

The pthread_attr_setinheritsched function

pthread_attr_setinheritsched

Synopsis

#include <pthread.h>

```
int pthread_attr_setinheritsched (attr,
inherit);

pthread_attr_t *attr;
int inherit;
```

Description

The pthread_attr_setinheritsched function sets the inheritsched attribute in the referenced attr object.

Returns

The pthread_attr_setinheritsched function returns zero if the call is successful, otherwise it sets errno to EINVAL and returns -1.

Implementation Notes

None

The pthread_attr_setschedparam function

pthread_attr_setschedparam

Synopsis

#include <pthread.h>

```
int pthread_attr_setschedparam (attr, param);

pthread_attr_t *attr;
struct sched_param *param;
```

Description

The `pthread_attr_setschedparam` function sets the priority in the referenced attr object.

Returns

The `pthread_attr_setschedparam` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_attr_setschedpolicy function

pthread_attr_setschedpolicy

Synopsis

#include <pthread.h>

```
int pthread_attr_setschedpolicy (attr,
policy);

pthread_attr_t *attr;
int policy;
```

Description

The `pthread_attr_setschedpolicy` function sets the scheduling policy.

Returns

The `pthread_attr_setschedpolicy` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

The value for `policy` must be `SCHED_FIFO`.

The pthread_attr_setstacksize function

pthread_attr_setstacksize

Synopsis

#include <pthread.h>

```
int pthread_attr_setstacksize (attr,
stacksize);

pthread_attr_t *attr ;
size_t stacksize ;
```

Description

The `pthread_attr_setstacksize` function sets the stack size value on the given attribute.

Returns

The `pthread_attr_setstacksize` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

14.6. Pthread Cond Functions

The pthread_cond_broadcast function

pthread_cond_broadcast

Synopsis

#include <pthread.h>

```
int pthread_cond_broadcast (cond);

pthread_cond_t *cond ;
```

Description

The `pthread_cond_broadcast` function unblocks all threads that are waiting on the given condition variable.

Returns

The `pthread_cond_broadcast` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_cond_destroy function

pthread_cond_destroy

Synopsis

#include <pthread.h>

```
int pthread_cond_destroy (cond);

pthread_cond_t *cond ;
```

Description

The `pthread_cond_destroy` function destroys the given condition variable. If the variable has one or more waiting threads then `errno` is set to `EBUSY`.

Returns

The `pthread_cond_destroy` function returns zero if the call is successful, otherwise it sets `errno` and returns `-1`.

Implementation Notes

None

The pthread_cond_init function

pthread_cond_init

Synopsis

#include <pthread.h>

```
int pthread_cond_init (cond, attr);

pthread_cond_t *cond ;
pthread_condattr_t *attr ;
```

Description

The `pthread_cond_init` function initializes the given condition variable.

Returns

The `pthread_cond_init` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

The value of `attr` must be null.

The pthread_cond_signal function

pthread_cond_signal

Synopsis

#include <pthread.h>

```
int pthread_cond_signal (cond);

pthread_cond_t *cond ;
```

Description

The `pthread_cond_signal` function unblocks at least one thread waiting on a condition variable. The scheduling priority determines which thread is runs next.

Returns

The `pthread_cond_signal` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_cond_timedwait function

pthread_cond_timedwait

Synopsis

#include <pthread.h>

```
int pthread_cond_timedwait (cond, mutex,
timeout);

pthread_cond_t *cond ;
pthread_mutex_t *mutex ;
struct timespec *timeout ;
```

Description

The `pthread_cond_timedwait` function unlocks the given mutex and places the calling thread into blocked state. When the specified condition variable is signaled or broadcast, or the system time is greater than i or equal to `timeout`, this function re-locks the mutex and returns to the caller.

Returns

The `pthread_cond_timedwait` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_cond_wait function

pthread_cond_wait

Synopsis

#include <pthread.h>

```
int pthread_cond_wait (cond, mutex);

pthread_cond_t *cond ;
pthread_mutex_t *mutex ;
```

Description

The `pthread_cond_wait` function unlocks the given mutex and places the calling thread into a blocked state. When the specified condition variable is signaled or broadcast, this function re-locks the mutex and returns to the caller.

Returns

The `pthread_cond_wait` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

14.7. Pthread Mutex Functions

The pthread_mutex_init function

pthread_mutex_init

Synopsis

#include <pthread.h>

```
int pthread_mutex_init (mutex, attr);

pthread_mutex_t *mutex ;
pthread_mutexattr_t *attr;
```

Description

The `pthread_mutex_init` function initializes the given mutex with the given attributes. If `attr` is null, then the default attributes are used.

Returns

The `pthread_mutex_init` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

The argument `attr` must be null. The default attributes are always used.

The pthread_mutex_destroy function

pthread_mutex_destroy

Synopsis

#include <pthread.h>

```
int pthread_mutex_destroy (mutex);

pthread_mutex_t *mutex;
```

Description

The `pthread_mutex_destroy` function destroys the given mutex. If the mutex is already destroyed, then `errno` is set to EINVAL. If the mutex is locked, then `errno` is set to `EBUSY`.

Returns

The `pthread_mutex_destroy` function returns zero if the call is successful, otherwise it sets `errno` and returns `-1`.

Implementation Notes

None

The pthread_mutex_lock function

pthread_mutex_lock

Synopsis

```
#include <pthread.h>

int pthread_mutex_lock (mutex);

pthread_mutex_t *mutex;
```

Description

The `pthread_mutex_lock` function locks the given mutex. If the mutex is already locked, then the calling thread blocks until the thread that currently holds the mutex unlocks it.

Returns

The `pthread_mutex_lock` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_mutex_trylock function

pthread_mutex_trylock

Synopsis

```
#include <pthread.h>

int pthread_mutex_trylock (mutex);
```

```
pthread_mutex_t *mutex;
```

Description

The `pthread_mutex_trylock` function tries to lock the given mutex. If the mutex is already locked, the function returns without waiting for the mutex to be unlocked.

Returns

The `pthread_mutex_trylock` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_mutex_unlock function

pthread_mutex_unlock

Synopsis

#include <pthread.h>

```
int pthread_mutex_unlock (mutex);

pthread_mutex_t *mutex;
```

Description

The `pthread_mutex_unlock` function unlocks the given mutex.

Returns

The `pthread_mutex_unlock` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

14.8. Miscellaneous Functions

The pthread_once function

pthread_once

Synopsis

#include <pthread.h>

```
int pthread_once (once_control, );

pthread_once_t *once_control ;
void (*init_routine) (void) ;
```

Description

The *pthread_once* function ensure the function `init_routine` runs only once regardless of how many threads call it. The thread that makes the first call to *pthread_once* succeeds in the call. Subsequent calls from other threads fail.

Returns

The `pthread_once` function returns zero.

Implementation Notes

None

The pthread_self function

pthread_self

Synopsis

#include <pthread.h>

```
pthread_t pthread_self ();

;
```

Description

The `pthread_self` function returns a pointer to the calling thread.

Returns

The `pthread_self` function returns the pointer.

Implementation Notes

None

The pthread_key_create function

pthread_key_create

Synopsis

#include <pthread.h>

```
int pthread_key_create (key, );

pthread_key_t *key;
void (*destructor) ();
```

Description

The `pthread_key_create` function create a new key that is visible to all threads. This key can be used with `pthread_setspecific` and `pthread_getspecific` to save and retrieve data associated with the a thread. If the function `destructor` is not NULL, then when a thread terminates, the destructor function will be called with the value associated with the key as the argument.

Returns

The `pthread_key_create` function returns 0 if successful. Otherwise it will return `-1` and set `errno` to `ENOMEM`.

Implementation Notes

The value of POSIX_DATA KEYS_MAX is 8.

See Also

The pthread_getspecific function [146]

The pthread_getspecific function

pthread_getspecific

Synopsis

#include <pthread.h>

```
any_t pthread_getspecific (key);

pthread_key_t key;
```

Description

The `pthread_getspecific` function retrieves the value of a data key for the current thread. If `key` is not a valid key, then `errno` is set to `EINVAL`.

Returns

The `pthread_getspecific` function returns the value associated with `key`, or returns NULL if `key` is invalid.

Implementation Notes

None

The pthread_setspecific function

pthread_setspecific

Synopsis

#include <pthread.h>

```
int pthread_setspecific (key, value);

pthread_key_t key;
any_t value;
```

Description

The `pthread_setspecific` function associates a value with a data key for the calling thread.

Returns

The `pthread_setspecific` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_cleanup_push function

pthread_cleanup_push

Synopsis

#include <pthread.h>

```
int pthread_cleanup_push (new);

void (*fun) (), any_t arg, cleanup_t new;
```

Description

The `pthread_cleanup_push` function places the given function on the top of the thread's cleanup stack.

Returns

The `pthread_cleanup_push` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

The pthread_cleanup_pop function

pthread_cleanup_pop

Synopsis

#include <pthread.h>

```
int pthread_cleanup_pop (execute);

int execute;
```

Description

The `pthread_cleanup_pop` function pops a function off current thread's cleanup stack and if `execute` is non-zero, executes the function with the argument given in the corresponding call to `pthread_cleanup_push`.

Returns

The `pthread_cleanup_pop` function returns zero if the call is successful, otherwise it sets `errno` to `EINVAL` and returns `-1`.

Implementation Notes

None

Index

R

S

T

U

V

W